STREAM: Towards READ-based In-Memory Computing for Streaming based Data Processing

Muhammad Rashedul Haq Rashed*, Sven Thijssen*, Sumit Kumar Jha†, Fan Yao*, and Rickard Ewetz*
*Department of Electrical and Computer Engineering, University of Central Florida, Orlando, USA
†Department of Computer Science, University of Texas at San Antonio, San Antonio, USA
{rashed09, sven.thijssen}@knights.ucf.edu, sumit.jha@utsa.edu, {fan.yao, rickard.ewetz}@ucf.edu

2022 27th Asia and South Pacific Design Automation Conference (ASP-DAC) | 978-1-6654-2135-5/22/$31.00 ©2022 IEEE | DOI: 10.1109/ASP-DAC52403.2022.9712569

***Abstract*—Processing in-memory breaks von-Neumann based design principles to accelerate data-intensive applications. While analog in-memory computing is extremely energy-efficient, the low precision narrows the spectrum of viable applications. In contrast, digital in-memory computing has deterministic precision and can therefore be used to accelerate a broad range of high assurance applications. Unfortunately, the state-of-the-art digital in-memory computing paradigms rely on repeatedly switching the non-volatile memory devices using expensive WRITE operations. In this paper, we propose a framework called STREAM that performs READ-based in-memory computing for streaming-based data processing. The framework consists of a synthesis tool that decomposes high-level programs into in-memory compute kernels that are executed using non-volatile memory. The paper presents hardware/software co-design techniques to minimize the data movement between different nanoscale crossbars within the platform. The framework is evaluated using circuits from ISCAS85 benchmark suite and Suite-Sparse applications to scientific computing. Compared with WRITE-based in-memory computing, the READ-based in-memory computing improves latency and power consumption up to $139X$ and $14X$, respectively.**

I. Introduction

The amount of produced digital data is expected to reach 40 trillion gigabytes [1, 2] by 2025. This has powered the emergence of data-intensive applications such as computer-vision, digital twin, and system simulation. Unfortunately, today's high performance computing systems are ill-equipped to handle even petabytes of data. Mainly, due to the separation of computing and memory units within the von-Neumann architecture [3]. With continuously increasing computational demands, there is an increasing interest in emerging technologies and computing paradigms [4–6].

A promising solution strategy is based on performing in-memory computing using emerging non-volatile memory. The fabrication of non-volatile resistive devices has been pursued based on memristor [7, 8], phase change memory (PCM) [9], and spin-transfer torque magnetic random access memory (STT-RAM) technology [10]. By integrating the non-volatile memory into nanoscale crossbar, various in-memory compute kernels can be executed energy-efficiently with high-speed. In particular, analog matrix-vector multiplication can be performed using the natural multiply-accumulate feature of nanoscale crossbar arrays. The challenge of analog in-memory computing is that the resulting precision is low.

To overcome the precision challenge, digital in-memory computing has been proposed using logic families such as OR-plane [4] , MAGIC [5], IMPLY [6], Bit-wise-in-bulk [11], and FLOW [12]. The different logic styles encode the inputs and outputs of rudimentary Boolean functions (or gates) using the state of non-volatile memory devices and/or analog voltage levels. This results in that the different logic styles have distinctive performance differences in terms of power, latency, and area. Nevertheless, the precision of digital in-memory computing using any of the logic styles is deterministic because there is adequate noise margin between logic '0' and logic '1'. Many recent research efforts have focused on synthesis tools and platforms for MAGIC, FLOW, and Bitwise-in-bulk, which facilitate the evaluation of complex logic in a single crossbar. The drawback of those logic styles is that they require the non-volatile memory devices to be repeatedly programmed using WRITE operations, which are both slow and power-hungry [13]. In contrast, OR-plane logic is based on READ operations and can be performed with high speed and high energy-efficiency. However, in-memory computing based on OR-plane logic remains in the infancy stages and has mainly been used to implement look-up tables (LUTs) for FPGAs [14, 15].

In this paper, we propose a framework called STREAM that performs READ-based in-memory computing for streaming-based data processing. The framework is based on utilizing OR-plane logic to process high fan-in OR/NOR gates using READ operations. The challenge of OR-plane logic is that complex functions require inter-crossbar data transfer, which may introduce substantial performance and hardware overheads. The STREAM framework provides a hardware/software-centered solution to minimize these overheads. First, we design processing elements (PEs) that have multiple crossbars connected in serial using hardwired connections to minimize data transfer costs. Next, a synthesis tool is proposed to decompose complex Boolean functions into parts that fit into the PEs while minimizing the costly inter-PE communication. The STREAM framework is evaluated using circuits from the ISCAS85 benchmark suite and Suit-Sparse applications to scientific computing. Compared with WRITE-based in-memory computing, STREAM improves average power and latency up to $14X$ and $139X$, respectively.

This work was in part supported by NSF awards CNS-1908471, CNS-2008339, CCF-1822976, CCF-2113307, DARPA cooperative agreement #HR00112020002 and ONR grant #N000142112332.

II. Preliminaries

In this section, we review two logic styles for evaluating Boolean functions using digital in-memory computing. Next, we discuss the limitations of previous works and motivate the proposed framework.

A. Digital in-memory computing

In this section, we explain how Boolean gates (or rudimentary Boolean functions) can be evaluated using digital in-memory computing. The representative logic styles of MAGIC and OR-plane logic are illustrated in Figure 1.

We show how MAGIC can be used to evaluate a NOR gate in Figure 1(a). MAGIC is an example of a stateful logic style where both input and output operands are stored using the state of the memristor devices [5]. True and false are encoded using low resisitive state (LRS) and high resisitve state (HRS), respectively. To evaluate the NOR2 function $out = \overline{a+b}$, the input variables a and b are first programmed to the memristors along the left-most bitline, which is shown in Figure 1(a). The output memristor is also programmed to LRS. Next, the NOR2 function is evaluated using a special WRITE operation [5]. The output of the NOR2 function can be decoded from the state (LRS/HRS) of the output memristor. Using this scheme, multiple NOR/INV gates can be evaluated in parallel along adjacent bitlines.

We illustrate the concept of OR-plane logic in Figure 1(b). OR-plane logic is an example of a non-stateful logic style [4]. The input and output operands are encoded using input and output voltages, respectively. True (false) is encoded using a high (low) voltage. We show how an INV, a NOR2, and a NOR3 gate can be evaluated in parallel using a single READ operation to the crossbar. The memristors in the crossbar are first initialized once using WRITE operations based on the logic gates that are desired to be realized. Next, the Boolean variables a, b, and c are applied to the wordlines in the crossbar. The outputs of the INV/NOR2/NOR3 functions are decoded from the bitlines, respectively. High fan-in OR-gates are realized by attaching buffers instead of inverters at the end of the bitlines.

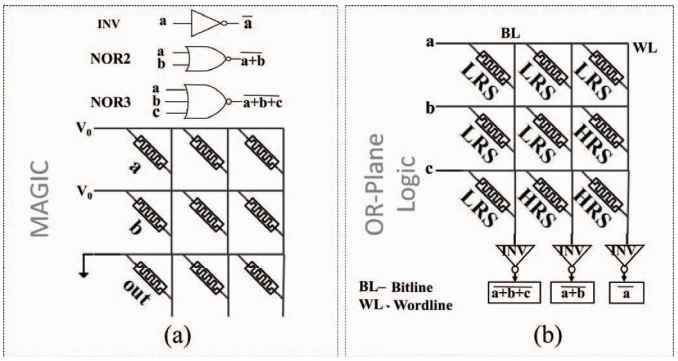

Fig. 1: Evaluation of Boolean gates using (a) WRITE-based MAGIC and (b) READ-based OR-plane logic.

B. Limitations of previous work

In this section, we compare different logic styles for digital in-memory computing. In principle, the paradigms perform computation using a one-time initialization phase and an evaluation phase. The initilization phase is performed with only the knowledge of the function to be evaluated. The evaluation phase is performed given specific instances of the Boolean input variables. The use of READ and WRITE operation in the respective phases are shown in Table I.

TABLE I: Comparison of READ/WRITE operations in the initialization and evaluation phase for different logic styles.

Logic style	Work in	Initialization phase	Evaluation phase
Flow-based computing	[12]	READ/WRITE	READ/WRITE
Bitwise-In-Bulk	[11]	READ/WRITE	READ/WRITE
MAGIC	[5]	READ/WRITE	READ/WRITE
IMPLY	[6]	READ/WRITE	READ/WRITE
OR-plane logic	(this work)	READ/WRITE	**READ**

The table shows that state-of-the-art digital in-memory computing paradigms rely on repeatedly performing expensive WRITE operations in the evaluation phase. WRITE operations to non-volatile memory devices are both slow and costly in energy. In contrast, the STREAM framework is based on OR-plane logic [4], which only uses READ operations in the evaluation phase. READ operations can be performed with high speed and energy-efficiency. For example, the energy and latency of a READ/WRITE operation is 1.08pJ/3.91nJ, and 29.31ns/50.88ns respectively [13]. An additional advantage of READ-based in-memory computing is that the platform will have a longer expected life-time, as the endurance of non-volatile devices is in the range of 10^3 to 10^9 WRITE operations [16].

A single crossbar for OR-plane logic implements OR/NOR operations. To evaluate arbitrary Boolean functions, multiple crossbars will have to be connected together in series, which we call a *staircase structure*. The primary inputs will be fed to the first crossbar and the final outputs are obtained from the last crossbar. The intermediate crossbars take inputs from the previous crossbar and provides outputs to the next crossbar. The challenge of the outlined approach is that data transfer between crossbars may introduce substantial performance overheads when performed using reconfigurable interconnects. On the other hand, if crossbars are hardwired together it becomes more difficult to maximize utilization and handle constraints imposed by the hardware. The trade-off between performance and ease of utilization is shown in Table II. The STREAM framework aims to enable streaming based processing by combining hardware/software co-design. The objective is to balance the overheads introduced by reconfigurability with the efficiency of hardwiring.

TABLE II: Hardwired vs. reconfigurable connections.

	Performance	Ease of utilization
Hardwired	high	difficult
Reconfigurable	low	smooth

III. The STREAM Framework

In this section, we introduce the STREAM framework. The framework consists of an in-memory computing platform and a synthesis tool capable of mapping computation to the platform, which is shown in Figure 2. The platform consists processing elements (PEs) connected together using high-speed interconnects. The PEs mainly consist of a staircase structure of connected crossbars. The details of the PEs are provided in Section V-B. The input to the synthesis tool is a specification of a Boolean function. The synthesis tool maps the computation into in-memory compute kernels and binds the kernels to the in-memory platform. Next, streaming-based processing is performed by providing input data to the reconfigured platform.

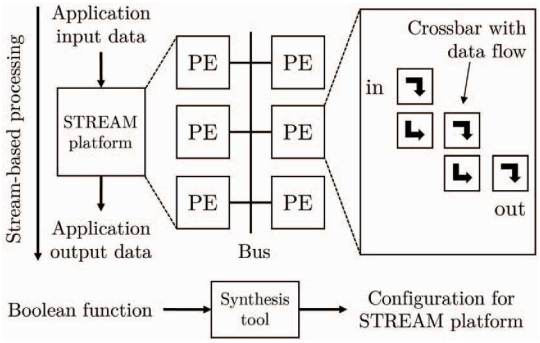

Fig. 2: Overview of the STREAM framework.

In the STREAM framework, we break the synthesis problem into two parts, as follows:

- **Problem I:** The first subproblem consists of mapping an arbitrary Boolean function to a PE with relaxed hardware constraints. Here, it is assumed that the crossbars are of arbitrary dimension and there are an arbitrary number of crossbars connected in series.
- **Problem II:** The second subproblem consists of decomposing the Boolean function into multiple parts. The objective is to satisfy the hardware constraints of the PEs when each part is mapped to a PE using the solution to Problem I.

A synthesis solution to the first subproblem is provided in Section IV. A synthesis solution to second subproblem focused on data-intensive applications based on matrix-vector multiplication is provided in Section V.

IV. Logic Synthesis for STREAM-based PEs

In this section, we provide a synthesis solution for mapping a Boolean function to a staircase structure of crossbars. An overview of the flow is shown in Figure 3.

The input of the framework is a Boolean function provided in a hardware descriptive language. The output consists of i) an assignment of the Boolean input variables to the first crossbar, ii) the state of all memristors in the platform, iii) an assignment of the Boolean output variables to the last crossbars.

The synthesis process consists of a technology independent optimization step, a technology mapping step, and a crossbar mapping step. In the technology independent optimization

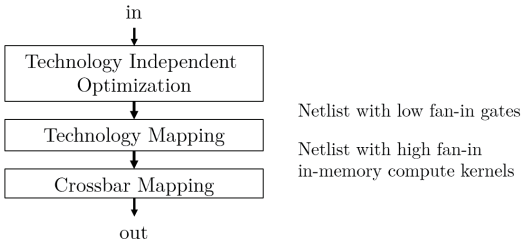

Fig. 3: Flow for the logic synthesis for STREAM-based PEs.

step, the input specification is mapped into a netlist with low fan-in gates using ABC [17]. This step is not described in further detail as it is performed directly using ABC and a library of INV and OR gates. In the technology mapping phase, the initial netlist is mapped into netlist of high fan-in OR/NOR in-memory compute kernels. In the crossbar mapping phase, the in-memory compute kernels are bound to the crossbar staircase structure. In principle, the in-memory compute kernels are sorted depth-wise and bound to the respective crossbar.

A. Technology Mapping

In this step, we convert the netlist with low fan-in gates, obtained from ABC, into a netlist with high fan-in gates that can be executed using OR-plane logic. The technology mapping is needed to take advantage of that OR-plane logic executes n-input OR/NOR gates using a single bitline. In contrast, ABC only generates netlists with at most 5 inputs, as the tool targets CMOS technology [17].

The input to the technology mapping is a netlist with low fan-in gates that is converted into a directed acyclic graph (DAG) $G = (V,E)$, where nodes and edges correspond to gates and wire connections, respectively. The graph is called a subject graph. Next, technology independent optimization is performed to cover the subject graph with in-memory compute kernels. Lastly, a DAG representation with high fan-in gates is extracted from the cover.

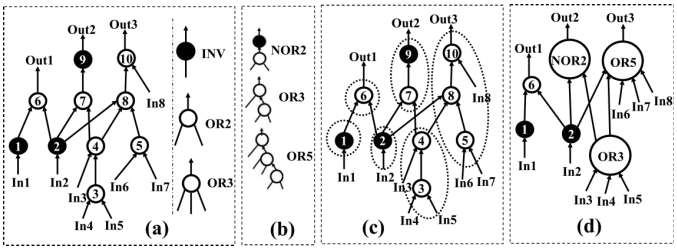

Fig. 4: Technology mapping within STREAM. (a) Initial netlist with gate encoding, (b) library of in-memory compute kernels, (c) cover of subject graph, (d) optimized netlist.

The technology mapping is illustrated with an example in Figure 4. The subject graph and the encoding of the gates is shown in Figure 4(a). A library with a subset of the in-memory compute kernels is shown in Figure 4(b). The subject graph covered with library gates is shown in Figure 4(c). The cover is obtained by first decomposing the subject graph into multiple trees by breaking edges. Next, the dynamic programming

formulation in DAGON is used to determine a cover for each tree [18]. The resulting netlist is shown in Figure 4(d). It can be observed that the initial netlist with 10 gates has been reduced to an in-memory compute kernel netlist with only 6 gates.

B. Crossbar Mapping

In this section, we bind the netlist of in-memory compute kernels to the crossbars within a PE. Any in-memory compute kernel can be executed in any crossbar. The challenge is that kernels that are adjacent in the netlist must be placed in adjacent crossbars. We solve this connection challenge by inserting dummy nodes and by utilizing graph algorithms to map the in-memory kernels to the crossbars.

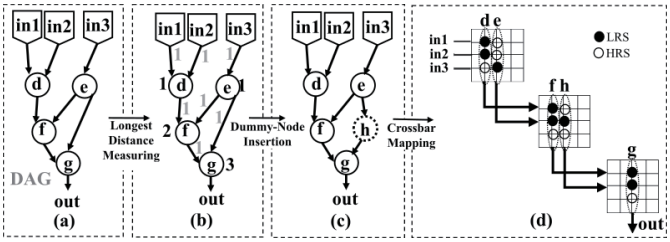

Fig. 5: Flow for binding in-memory kernels to crossbars. (a) Input netlist in DAG format, (b) longest distance to each node, (c) dummy node insertion and, (d) crossbar mapping.

The proposed algorithm is illustrated with an example in Figure 5. A DAG representation of the netlist is shown in Figure 5(a). Each node represents an in-memory compute kernel that can be evaluated in a crossbar. Next, we determine the longest path to each node in the graph, which is shown in Figure 5(b). The longest path to each node is determined using a topological sort followed by an in-order traversal. Let each crossbar in the staircase structure be labeled layer 1 to layer N. The longest path to a node corresponds to the crossbar that the node will be assigned. The outlined method ensures that all connections go from crossbars with lower layers to higher layers. To eliminate connections that skip layers, dummy nodes realized by buffers are inserted into the netlist. The insertion of a dummy node between 'e' and 'g' is shown in Figure 5(c). The height and width of the crossbar realize layer l is equal to the number of nodes in layer (l) and $(l+1)$, respectively. Finally, it is straightforward to assign the kernels to the crossbars in the staircase structure, which is shown in Figure 5(d).

V. STREAM Framework for MVM Applications

In this section, we leverage the STREAM framework to accelerate data-intensive applications that are dominated by high precision matrix-vector multiplication (MVM). This requires the computation to be broken into parts such that each part is mapped to a PE with specified hardware resources. This is Problem II outlined in Section III.

The motivation for breaking the computation into parts is that the hardware requirements would otherwise be unacceptably high. For example, mapping a 128x128 matrix with 32 bit precision to a PE requires a staircase structure with 74.5 million crossbars. The crossbar with the largest required dimension would have 40,500 wordlines and 40,000 bitlines. In STREAM, we propose to reduce the hardware requirements using spatial and bit-wise partitioning.

A. Spatial Partitioning

In this section we propose to partition the matrix vector multiplication using blocks with dynamic size, which is illustrated in Figure 6.

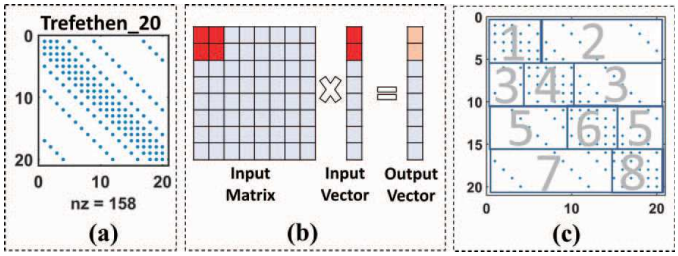

Fig. 6: (a) Sparse matrix of Trefethen-20 benchmark, (b) partitioning with fixed block size, and (c) partitioning with dynamic block size.

Many matrices within scientific computing applications are sparse, one of which is shown in Figure 6(a). The workload of a matrix block is largely dependent on the number of non-zero matrix elements. Therefore, it is easy to understand that partitioning using a fixed block size, which is shown in Figure 6(b) results in that some PEs are heavily under-utilized. Instead, we propose to utilize a dynamic partitioning scheme that uses blocks of dynamic size, which is shown in Figure 6(c). In our implementation, we dynamically expand the block size column-wise and row-wise until a threshold of non-zero elements have been covered.

B. Bit-wise Partitioning

In this section, we propose to utilize bit-slicing to partition the computation across multiple time steps.

The concept of bit-slicing for a 32-bit fixed point computation is shown in Figure 7(a). We aim to decompose the 32-bit element-wise multiplication into a series of multiplications with smaller bit-widths, as shown in Figure 7(a)-(i). The key idea is to bit-slice the input vector with unknown operands, as shown in Figure 7(a)-(ii). Next, the overall multiplication is performed in a series of multiplications and shift&add operations, as demonstrated in Figure 7(a)-(iii). All in all, the bit-slicing introduces a trade-off between time steps (latency) and hardware utilization. We show an updated architecture for the PE to support the proposed bit-slicing in Figure 7(b).

The PE units consists of several element-wise multiplication blocks (EMBs), an adder tree to accumulate the outputs of EMBs and a central eDRAM buffer to store the input, the output and the intermediate operands of the accelerator unit. Figure 7(b) illustrates the components of each EMB. Each EMB has 8 multiplication units (one for each of the eight bit-slices), seven shifters, an input register (IR) unit and an output register (OR) unit. The results of each multiplication unit is accumulated using the accelerator adder tree. The tree

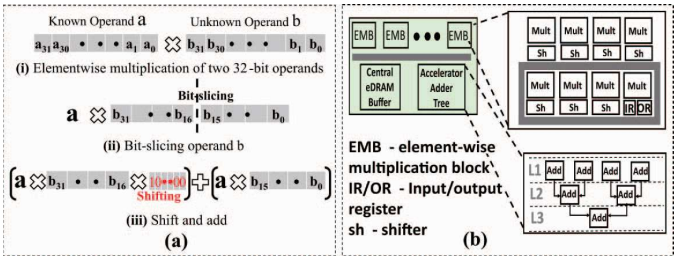

Fig. 7: (a) Bit-slicing element-wise multiplication and (b) architecture design of PEs for bit-slicing support.

contains 3 layers of adders to sum the results from the EMBs. The components utilize both parallelism and pipelining to maximize performance.

VI. Experimental Evaluation

In this section, we evaluate the performance of the STREAM framework. The framework is evaluated on two benchmarks suites, and is compared with two state-of-the-art in-memory computing paradigms. In Section VI-A, we compare the READ-based STREAM framework with the state-of-the-art WRITE-based in-memory computing paradigm CONTRA [19] on benchmarks of the ISCAS85 benchmark suite [20]. In Section VI-B, we evaluate STREAM with data-intensive MVM operations from a number of applications from the Suit-Sparse matrix collection of [21]. A comparison is made with SIMPLER [22], which is the state-of-the-art MAGIC mapping scheme for applications based on matrix-vector multiplication.

TABLE III: Area-Power Cost of STREAM Components

Component	Parameter	Specs	Area	Power
Crossbar	dimension	128×128	25 μm^2	0.3 mW
Controller	# unit	1	400 μm^2	0.65 mW
Mult. (Total)	# crossbars	12	**0.005 mm^2**	**11.4 mW**
Shifter	# unit	1	60 μm^2	0.05 mW
IR	size	4 KB	4200 μm^2	2.48 mW
OR	size	512 B	1500 μm^2	0.46 mW
local bus	#wires	128	0.03 mm^2	2.33 mW
EMB (Total)	# Mult. # Shifter #IR/#OR	8 7 1/1	**0.077 mm^2**	**96.82 mW**
Adder Tree (Total)	# crossbars	198	**0.084 mm^2**	**188.1 mW**
eDRAM Buffer	size	128 KB	0.166 mm^2	41.4 mW
Bus	bandwidth	128-bits	15.7 mm^2	13 mW
PE (Total)	#EMBs #Adder tree #eDRAM Buffer	16 1 1	**17.18 mm^2**	**1791.67 mW**

The experiments are executed on a machine with an Intel Core i9 processor and NVIDIA GeForce RTX 2070S GPU. A SPICE-level simulation is performed with the fitting characteristics of the VTEAM model [23]. For STREAM, the crossbar dimensions are 128×128, and the resistance R_{LRS} and R_{HRS} of the memristors are $10k\Omega$ and $10M\Omega$, respectively. We use a read and write latency of 29.31ns and 50.88ns respectively, as reported in [13]. In Table III, the remaining parameters for the different components of the STREAM platform are provided. For each component, we provide the specifications, area, and power consumption. The parameters have been obtained from [23–25].

A. Evaluation on ISCAS85 benchmarks

In this section, we evaluate the performance of the READ-based STREAM framework with the state-of-the-art WRITE-based in-memory computing paradigm CONTRA [19]. Ten benchmarks of the ISCAS85 benchmarks suite [20] have been selected for evaluation. An overview of the function type and properties (number of inputs and outputs) of the benchmarks is provided in Table IV. Both frameworks are compared in terms of area, latency (number of cycles), and the power consumption. The numbers are provided in Table V.

TABLE IV: Overview of ten ISCAS85 benchmarks.

Benchmark	Function	Inputs	Outputs
c432	Priority Decoder	36	7
c499	ECAT	41	32
c880	ALU and control	60	126
c1355	ECAT	41	32
c1908	ECAT	33	25
c2670	ALU and control	233	140
c3540	ALU and control	50	22
c5315	ALU and selector	178	123
c6288	16-bit multiplier	32	32
c7552	ALU and control	207	108

In Table V, we observe that STREAM improves both speed and power, at an expense of the area usage. The latency for STREAM is improved by 139X compared with the latency for CONTRA due to the READ-based paradigm characteristics of STREAM. Indeed, the write latency is almost double the read latency and STREAM relies solely on READ operations. Above this, the number of cycles for STREAM is 80X smaller compared with the number of cycles for CONTRA. Next, the power consumption for STREAM is improved by 14X compared with CONTRA. Based on the same argument, this is due to the fact that STREAM relies on READ operations to evaluate in-memory compute kernels instead of WRITE operations. Whereas CONTRA relies on overwriting old data, STREAM relies on performing computations by pipelining the data. The intermediate evaluations are forwarded between crossbars connected in series. Hence, STREAM is a non-stateful logic style. However, area usage increases by 56X in contrast with CONTRA, as STREAM cannot reuse area by overwriting old data, resulting in the need of larger area and more crossbars. In conclusion, for READ-based computing, the area usage is inversely proportional to the latency and power consumption.

TABLE V: Comparison of area, number of cycles, and power consumption for CONTRA and STREAM on ten benchmarks of the ISCAS85 benchmarks suite.

	CONTRA [19]			STREAM		
Benchmark	Area (μm^2)	Latency (μs)	Power (W)	Area (μm^2)	Latency (μs)	Power (W)
c432	601	39.18	2.35	13222	0.64	0.35
c499	601	68.33	4.10	17429	0.73	0.41
c880	601	64.26	3.85	17429	0.85	0.47
c1355	601	68.38	4.10	14424	0.59	0.33
c1908	601	74.74	4.48	16227	0.79	0.43
c2670	601	104.81	6.28	28848	0.88	0.54
c3540	601	181.89	10.90	28247	1.35	0.74
c5315	601	245.80	14.73	37863	0.97	0.62
c6288	601	401	24.04	105175	3.31	2.00
c7552	601	356	21.46	59499	1.5	0.96
Norm. avg.	**0.018**	1.000	1.000	1.000	**0.0072**	**0.071**

TABLE VI: Overview of eleven matrices of the SuitSparse Matrix Collection in terms of application type, matrix dimensions, and number of non-zero elements.

Applications	Systems	Matrix Dimensions	#Non-zeros
Trefethen-20	Combinatorial	20×20	158
mesh3em5	Structural	289×289	1377
Trefethen-150	Combinatorial	150×150	2040
Trefethen-200b	Combinatorial	199×199	2873
Trefethen-200	Combinatorial	200×200	2890
bcsstk02	Structural	66×66	4356
Trefethen-300	Combinatorial	300×300	4678
Chem97ZtZ	Statistical/Mathematical	2541×2541	7361
Trefethen-500	Combinatorial	500×500	8478
Journals	Undirected Weighted Graph	124×124	12068
Trefethen-700	Combinatorial	700×700	12654

B. Evaluation on SuitSparse Matrix Applications

In this section, we evaluate the performance of the STREAM framework for MVM applications. Eleven sparse matrices have been selected from the SuitSparse Matrix Collection [21]. An overview of the matrices is provided in Table VI. We compare STREAM with SIMPLER [22], a state-of-the-art MAGIC mapping scheme for applications with high order of parallel computation. SIMPLER can perform each matrix-row$\times$input-vector operation in parallel using a row-mapping fashion, which greatly improves the performance of the accelerator. For the STREAM framework, we use a dynamic block-wise partitioning that covers at most 16 element-wise multiplications. For each element-wise multiplication, we use a bit-slicing of 4-bit width. Each PE has an area of $17.18mm^2$ and power of $1791.67mW$.

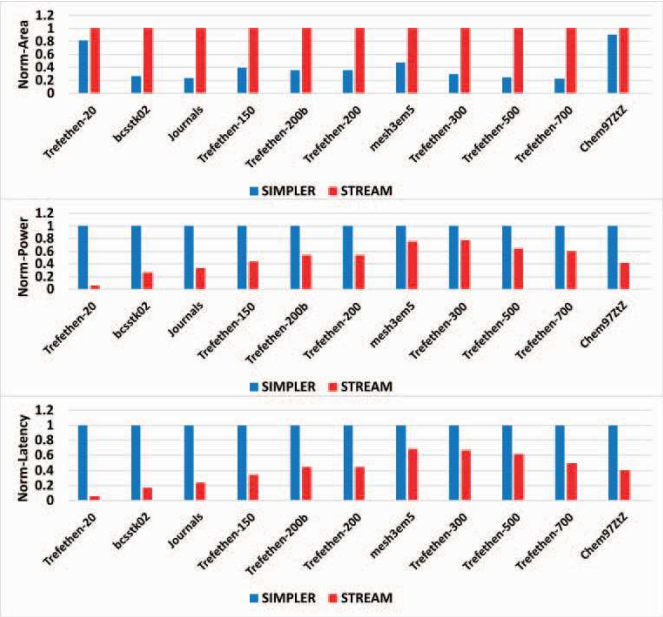

Fig. 8: Comparison of area, power, and latency for SIMPLER and STREAM on eleven benchmarks of the SuitSparse Matrix Collection.

In Figure 8, we compare the performance of STREAM with SIMPLER on the eleven sparse matrices in terms of area, power consumption, and latency. The experimental result shows that the STREAM framework requires 2.2X more area usage than SIMPLER. However, STREAM achieves 2.04X power efficiency, and improves the latency by 2.42X compared with the latency of SIMPLER. Also here, the area usage increases, and both latency and power consumption decrease for STREAM due to the READ-based in-memory computing style.

VII. Summary and Future Work

In this paper, we have introduced a framework, STREAM, for the synthesis of digital circuits onto nanoscale crossbars. The framework is tailored for READ-based in-memory computing based on OR-plane logic. The experimental evaluation illustrates the effectiveness of READ-based computing compared with WRITE-based in-memory computing paradigms. In the future, we plan to use STREAM to accelerate genome sequencing applications. We also plan to augment STREAM with analog in-memory accelerators.

References

[1] R. Taylor, D. Baron, and D. Schmidt, "The world in 2025-predictions for the next ten years," in *IMPACT*, pp. 192–195, IEEE, 2015.
[2] J. Gantz and D. Reinsel, "The digital universe in 2020: Big data, bigger digital shadows, and biggest growth in the far east," *IDC iView: IDC Analyze the future*, vol. 2007, no. 2012, pp. 1–16, 2012.
[3] J. Backus, "Can programming be liberated from the von neumann style?: A functional style and its algebra of programs," *CACM*, vol. 21, no. 8, pp. 613–641, 1978.
[4] A. Dehon, "Nanowire-based programmable architectures," *JETC*, vol. 1, no. 2, pp. 109–162, 2005.
[5] S. Kvatinsky *et al.*, "Magic—memristor-aided logic," *TCAS-II: Express Briefs*, vol. 61, no. 11, pp. 895–899, 2014.
[6] J. Borghetti *et al.*, "'memristive' switches enable 'stateful' logic operations via material implication," *Nature*, vol. 464, no. 7290, pp. 873–876, 2010.
[7] D. B. Strukov, G. S. Snider, D. R. Stewart, and R. S. Williams, "The missing memristor found," *Nature*, vol. 453, no. 7191, pp. 80–83, 2008.
[8] L. Chua, "Memristor-the missing circuit element," *IEEE Transactions on circuit theory*, vol. 18, no. 5, pp. 507–519, 1971.
[9] G. W. Burr *et al.*, "Phase change memory technology," *JVSTB*, vol. 28, no. 2, pp. 223–262, 2010.
[10] Y. Huai *et al.*, "Spin-transfer torque mram (stt-mram): Challenges and prospects," *AAPPS bulletin*, vol. 18, no. 6, pp. 33–40, 2008.
[11] S. Li *et al.*, "Pinatubo: A processing-in-memory architecture for bulk bitwise operations in emerging non-volatile memories," in *DAC*, pp. 1–6, IEEE, 2016.
[12] S. K. Jha, D. E. Rodriguez, J. E. Van Nostrand, and A. Velasquez, "Computation of boolean formulas using sneak paths in crossbar computing," 2016. US Patent 9,319,047.
[13] L. Song, X. Qian, H. Li, and Y. Chen, "Pipelayer: A pipelined reram-based accelerator for deep learning," in *HPCA*, pp. 541–552, IEEE, 2017.
[14] Y. Zha and J. Li, "Reconfigurable in-memory computing with resistive memory crossbar," in *ICCAD*, pp. 1–8, 2016.
[15] Y. Zha and J. Li, "Rram-based reconfigurable in-memory computing architecture with hybrid routing," in *ICCAD*, pp. 527–532, IEEE, 2017.
[16] H. Wu *et al.*, "Resistive random access memory for future information processing system," *Proceedings of the IEEE*, vol. 105, no. 9, pp. 1770–1789, 2017.
[17] A. Mishchenko *et al.*, "Abc: A system for sequential synthesis and verification." "http://www.eecs.berkeley.edu/alanmi/abc".
[18] K. Keutzer, "Dagon: Technology binding and local optimization by dag matching," in *DAC*, pp. 341–347, 1987.
[19] D. Bhattacharjee *et al.*, "Contra: area-constrained technology mapping framework for memristive memory processing unit," in *ICCAD*, pp. 1–9, 2020.
[20] M. C. Hansen, H. Yalcin, and J. P. Hayes, "Unveiling the iscas-85 benchmarks: A case study in reverse engineering," *IEEE Design & Test of Computers*, vol. 16, no. 3, pp. 72–80, 1999.
[21] T. A. Davis and Y. Hu, "The university of florida sparse matrix collection," *TOMS*, vol. 38, no. 1, pp. 1–25, 2011.
[22] R. Ben-Hur *et al.*, "Simpler magic: Synthesis and mapping of in-memory logic executed in a single row to improve throughput," *TCAD*, vol. 39, no. 10, pp. 2434–2447, 2019.
[23] S. Kvatinsky, M. Ramadan, E. G. Friedman, and A. Kolodny, "Vteam: A general model for voltage-controlled memristors," *TCAS-II: Express Briefss*, vol. 62, no. 8, pp. 786–790, 2015.
[24] A. Shafiee *et al.*, "Isaac: A convolutional neural network accelerator with in-situ analog arithmetic in crossbars," *SIGARCH*, vol. 44, no. 3, pp. 14–26, 2016.
[25] M. Imani, S. Gupta, Y. Kim, and T. Rosing, "Floatpim: In-memory acceleration of deep neural network training with high precision," in *ISCA*, pp. 802–815, IEEE, 2019.