Rapid In-Memory Matrix Multiplication Using Associative Processor

Mohamed Ayoub Neggaz*, Hasan Erdem Yantır†, Smail Niar*, Ahmed Eltawil† and Fadi Kurdahi†
* LAMIH, University of Valenciennes, France
† CECS, University of California, Irvine, USA

***Abstract*—Memory hierarchy latency is one of the main problems that prevents processors from achieving high performance. To eliminate the need of loading/storing large sets of data, Resistive Associative Processors (ReAP) have been proposed as a solution to the von Neumann bottleneck. In ReAPs, logic and memory structures are combined together to allow in-memory computations. In this paper, we propose a new algorithm to compute the matrix multiplication inside the memory that exploits the benefits of ReAP. The proposed approach is based on the Cannon algorithm and uses a series of rotations without duplicating the data. It runs in $\mathcal{O}(n)$, where n is the dimension of the matrix. The method also applies to a large set of row by column matrix-based applications. Experimental results show several orders of magnitude increase in performance and reduction in energy and area when compared to the latest FPGA and CPU implementations.**

***Index Terms*—Resistive Associative Processor, Linear Algebra, Matrix Multiplication, FPGAs, Memristor.**

I. Introduction

With the recent decline in Moore's law, parallelism becomes the only resort for the increasing demand of computing power. With multiple compute cores, the memory needs to be faster than ever to provide the required data in time without augmenting the speed. This difference in speed between the memory and the computing unit is called the von Neumann Bottleneck.

Many applications suffer from this architectural limitation. Matrix multiplication (MM) is one of the most important building blocks in many applications like machine learning, image processing, etc. Although the logic of this operation is very simple (multiply and accumulate), the data needs to be massively transferred between the compute cores and memory.

In this work, we use in-memory computing (IMC) as well as the highly parallel structure of Associative Processors (APs) to solve the memory bottleneck for matrix multiplication. The resistive implementation of associative processors (ReAP) is used for its superior density. We propose a vectorized implementation of Cannon's algorithm [1] that suits the ReAP architecture. We also generalized the algorithm for other applications like All-Pairs Shortest Paths (APSP). Our approach is able to reduce the execution time by an order of magnitude when compared to one of the most efficient FPGA implementations for large size matrices. Area consumption reduction is even more important.

The paper is structured as follows: Section II presents related implementations of MM on different platforms. Our Resistive Associative Processor is described in Section III. Section IV present our MM implementation on ReAP. Experimental results are presented later in Section V.

II. Related Works

Matrix multiplication is used in a variety of fields like neural networks in machine learning, image and signal processing, etc. In most applications, matrix multiplication is the performance bottleneck. Numerous works targeted this problem. We classified these methods in two major categories:

A. Software Implementations

Many works tend to reduce the complexity of the naive i-j-k algorithm. Strassen's algorithm in [2] drops the overall complexity from $O(n^3)$ to $O(n^{2.807})$. Williams' algorithm of $O(n^{2.373})$ in [3] is the best current performance for MM. However, the complexity of the operations makes it very difficult to use and today's hardware is unable to benefit from its performance since it requires very large matrices to show a noticeable advantage. Also, Bshouty [4] proved the existence of a lower bound of $O(n^2)$. This lower bound limits the capabilities of software-only improvements.

B. Hardware Improvements

1) CPU: One of the major issues of computing a matrix product is the memory organization and the non-sequential access. For this reason, cache enhancements were proposed to cope with this issue. These improvements profit from cache spatial locality to reduce slow memory accesses. Optimized SW libraries such as Intel's MKL also proposes efficient implementations. However, due to its limited parallelism, CPUs cannot cope with the increasing size of the problem.

2) FPGA: Jang et. al. [5] proposed an energy-time efficient approach for MM by storing only one of the input matrices. In another work, Jiang et. al. [6] proposed a Scalable Macro-pipelined architecture (SMPA). This architecture uses the temporal parallelism to maximize the use of processing elements (PEs) in order to get better throughput.

These works and others ([7] and [8]) profit from the highly parallel architecture of reconfigurable devices to maximize the performance or the resource usage. The downside of these methods is resource utilization since input matrices need to be stored in the programmable logic (PL). In [5], Jang et. al. reported 78% energy dissipation on storage for a 12x12 MM. With the growing size of matrices, the energy dissipation increases proportionally.

978-3-9819263-0-9/DATE18/©2018 EDAA

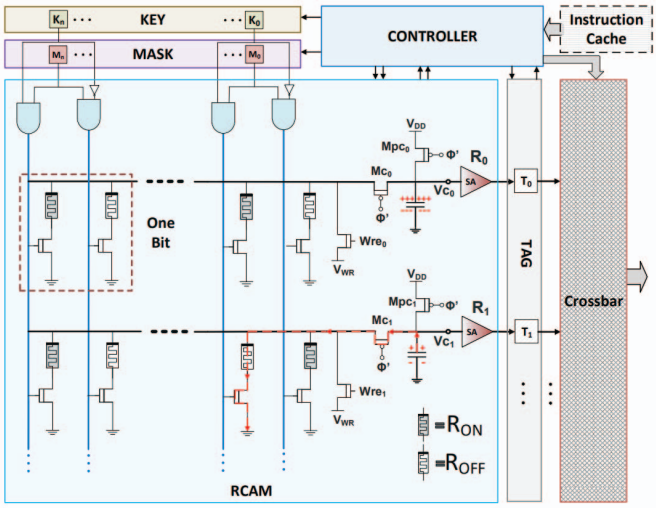


Fig. 1: Architecture of an associative processor.

3) Dedicated Hardware: Many application specific hardware (ASIC) was developed for matrix multiplication. They generally uses SUMMA [9] for its efficient memory management. However, a dedicated ASIC can only execute the targeted function. In the other side, our approach runs on an associative processor and can execute a variety of application compared to those solutions.

In a similar fashion, Google recently presented their Tensor Processing Unit (TPU) which uses systolic arrays to perform matrix multiplication. It is a CISC processor designed for machine learning applications. It targets data centers [10] and is not available for general public.

Haron et. al. [11] used the IMPLY logic to perform MM. In their 3D method, input matrices are duplicated and all the multiplications are done in a single stage. The additions are done using a binary adder tree which gives the overall algorithm an $O(\log n)$ complexity. As far as we know, this is the most efficient algorithm. However, duplicating matrices n times can be costly with the matrix size growth.

Morad et. al. [12] proposed a dense and sparse MM using a General Purpose-Single Instruction Multiple Data processor and a sequential processor. Associative memory array is used by the SIMD processor to perform the parallel computations. In a similar manner, Yavits et. al. [13] proposed an efficient implementation of sparse MM on an associative processor.

When compared to existing methods, our approach profits from IMC to perform the operation. This gives a huge leap in performance and area compared to FPGA implementations since no data storage or loading is required. For implementations based on AP, our storage policy is very efficient since we do not store indexes which is common for associative processors. We also implement all the operations inside the CAM, therefore, there is no need for a reduction tree.

III. Resistive Associative Processor

In this section, we describe the required knowledge on APs in order to understand our approach. We give details about the overall architecture and how to exploit it to perform logic operations.

TABLE I: LUT for Multiplication

Cr	R	B	A	Cr	R	Comment
0	0	1	1	0	1	2nd Pass
0	1	1	1	1	0	1st Pass
1	0	0	1	0	1	3rd Pass
1	1	0	1	1	0	4th Pass

A. Architecture

Figure 1 shows the architecture of an AP. It consists of a CAM, a controller, an instruction cache, an interconnection circuit and some specific registers (*Mask*, *Key* and *Tag*). The CAM holds the data in a column format. A program is a set of instructions to be performed on that data. An instruction on AP consists of consecutive compare and write phases. A Look-Up Table (LUT) is used for each instruction. The execution of an instruction is the application of the LUT on the CAM. Operations in an AP are performed on data inside the CAM in a column per column fashion. Therefore, the number of rows does not affect the execution time of an operation.

B. Logic execution

During the compare phase, columns to be compared against are marked with ”1” in the *Mask*. Only values inside these columns are compared with the *Key*. If the value inside the CAM matches the *Key*, the row will be tagged ”1” using the *Tag* register. For example, if the *Mask* is ”101” and the *Key* is ”100”. Only the first and the third columns are considered. These columns should be compared with the first (from right to left) and third columns of the *Key* register and should be ”0” and ”1” respectively.

In the write phase, only the rows that are tagged ”1” are considered. Depending on the LUT of the current instruction, a value should be written to columns with a mask of ”1”.

Table I shows an example of the LUT for unsigned multiplication. ”A”, ”B” and ”R” are the masked bits of the corresponding variable. ”Cr” is the carry bit and is always masked. In the compare phase, rows that correspond to the quadruple (A, B, R, Cr) in the left column of the table are tagged. If a line matches these values, the result ”R” and carry ”Cr” are updated using the right column of the LUT. Other quadruples are omitted since they do not affect the two outputs. The order of operations is important and is shown in the ”Comment” tab in the right column. The quadruples need to be processed in the given order to prevent a row from being processed twice.

C. Circuit Implementation

In [14], a 2-transistor-2-memristor CAM cell is presented. A memristor can change its resistance. We only consider two levels, Ron and Roff. In the experiments, Ron is set to 500 Ω and Roff to 10k Ω. Depending on their order, the two memristors inside a cell can store a value of ”1” (Ron - Roff) or ”0” (Roff - Ron). Writing consists of flipping the resistance of these two memristors to match the required order of the desired value.

Before each compare phase, a pre-charge step is required. A capacitor connected to each CAM row is charged. If the value

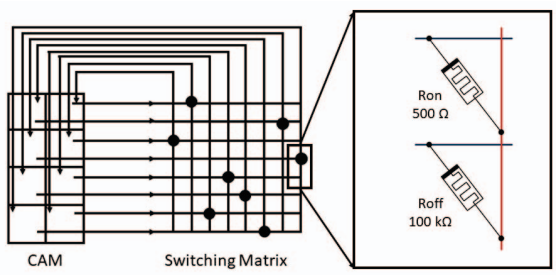


Fig. 2: Connection between the CAM and the SM. The black dots stands for a short circuit which means that the input line connected to this dot will be redirected to the output line connected to the CAM.

inside the *Key* does not correspond to the value inside the cell, a line connecting the capacitor to the ground can be found and a discharge will take effect (row 2 in Figure 1). Otherwise, the capacitor will maintain its charge if the searched word corresponds to the stored value in the row (row 1 in Figure 1). After that, a sense amplifier decides on the logical value of the tag register by comparing the voltage across the capacitor to the threshold voltage (V_{th}).

D. Data Management

Data inside the CAM can be moved using a Switching Matrix (SM). Each CAM cell is connected to other cells via a crossbar like in Figure 2. The SM can be implemented using memristors. We can consider the Ron as a short circuit and the Roff as an open one. With this in mind, only one memristor connecting a horizontal line with a vertical line in the SM can play the role of a simple switch like in Figure 2. Memristors also allow reconfigurability since we can change the values and hence the interconnection scheme.

IV. Matrix Multiplication Algorithm

In this section we present an efficient MM implementation of Cannon's algorithm [1] on ReAP. We first discuss how we store the data inside the CAM. The operations needed for the operation are then explained.

A. Cannon's Algorithm on ReAP

Given a general matrix multiplication $C = A * B$. Cannon's algorithm as shown in Algorithm 1 performs a series of rotations followed by a Multiply-and-Accumulate (MAC) to compute the output C. A, B and are square matrices of n rows. A grid of n by n processing elements (PEs) can efficiently performs the required operations in n compute stages. Each processing element PE_X holds a value of A: $PE_{X,A}$, a value of B: $PE_{X,B}$ and computes a value of C: $PE_{X,C}$. After each stage, elements of A and B are circularly transfered between processors to satisfy the matrix product formula for each value of C: $c_{i,j} = \sum_{k=1}^{k=n} a_{i,k} * b_{k,j}$

B. Cannon Algorithm on CAM

In a ReAP, the CAM cells are the compute units. Each row is able to perform one operation on values residing in its cells. For this reason, we store matrices in vectorized format. 3 columns are needed to store A, B and C. The execution consists of consecutive execute-and-rotate stages. At each stage, we multiply columns A and B and add the results to column C. Values of C are initially set to "0". Then, we rotate the data inside the CAM to match Cannon's circular shifts.

Algorithm 1 Cannon's Algorithm

procedure MM(**Mat**: A, **Mat**: B, **Mat**: C)
  for $i = 0 \dots n-1$ **do**
    $A[i,:] \leftarrow CircularShiftLeft(i)$
  end for
  for $i = 0 \dots n-1$ **do**
    $B[:,i] \leftarrow CircularShiftUp(i)$
  end for
  for $k = 0 \dots n-1$ **do**
    $PE_{X,C} + \leftarrow PE_{X,A} * PE_{X,B}$
    $A[k,:] \leftarrow CircularShiftLeft(1)$
    $B[:,k] \leftarrow CircularShiftUp(1)$
  end for
end procedure

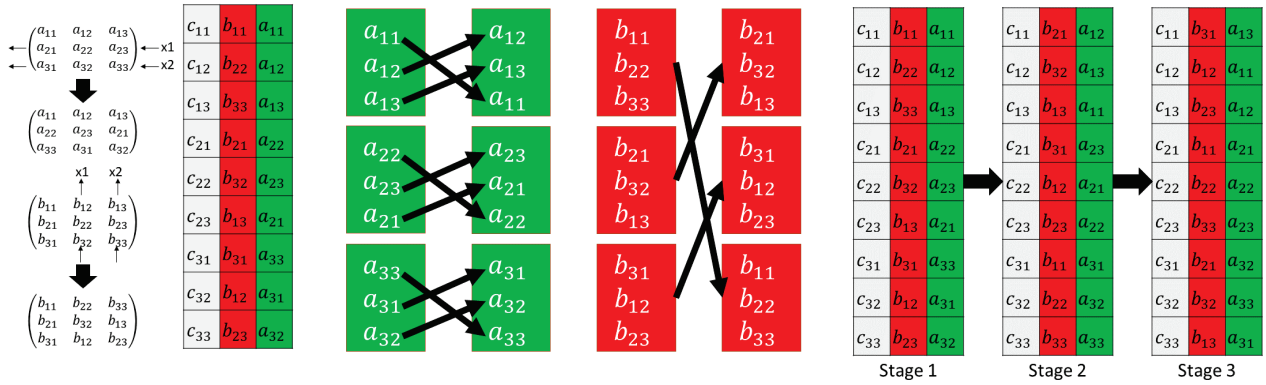
In Figure 3, we see the execution stages on ReAP. After storing the two matrices, we perform a first MAC operation. Once the MAC operation is finished, we should prepare the data for the next stage. We have computed one partial sum of each one of the C values. Since matrices are stored in a vector format (1D) inside the CAM, Cannon's circular shifts are replaced by inner and outer rotations as illustrated in Figure 3b. Algorithm 2 illustrate the multiplication of two matrices stored in columns A and B inside the CAM.

The execution is a loop of n stages, in each, we compute one multiplication for each element in C. The MAC operation performs a multiply-and-accumulate operation between columns A and B and stores the temporary result in the column C. The rotations in 3b are performed on the SM with the two instructions $InnerRot$ and $OuterRot$. These two operations run at the same time on the SM. The dots in Figure 2 corresponds to all the required rotations in the algorithm. The time needed to perform these rotations is constant. Since MAC operations run also in a constant time, the loop in Algorithm 2 performs a constant number of operation for n times, which proves the linear complexity of our algorithm.

In Figure 3c we can see the results of these rotations. The MAC operation is performed between each two rotations. Considering MAC-and-Rotate for the first row, the value computed is: $b_{11} * a_{11} + b_{21} * a_{12} + b_{31} * a_{13}$ which is actually c_{11}.

Algorithm 2 MM on ReAP Execution

procedure MM(**Column**: A, **Column**: B, **Column**: C)
  for $i = 1 \dots n$ **do**
    $C \leftarrow MAC(A,B)$ ▷ Perform MAC on the CAM
    { ▷ Parallel rotations inside the SM
      $A \leftarrow InnerRot(A)$ ▷ Rotations for column A
      $B \leftarrow OuterRot(B)$ ▷ Rotations for column B
    }
  end for
end procedure



(a) CAM state after loading input matrices.

(b) Inner and outer rotations.

(c) CAM state after each stage rotation.

Fig. 3: The execution scheme of MM on ReAP with the loading phase in 3a and the rotations after each stage in 3b. Intermediate CAM states after rotations in 3c.

C. Generalization

All operations between two matrices, that can be performed in an ijk format (row-per-column) can be accelerated by this approach i.e. algorithms of the form:

$$c_{i,j} = \otimes_k(\oplus(a_{i,k}, b_{k,j}))$$

Where $\otimes_k$ is an operation over a set and $\oplus$ is an operation between two values $a_{i,k}, b_{k,j}$. For MM, $\otimes_k$ is the *sum over k elements* and $\oplus$ is a *multiplication*.

Algorithms of this class can be found in a variety of linear algebra computations. Dominance product [15] is one of these algorithms. We denote it as $C = A \circledS B$. It is defined as:

$$c_{i,j} = |\{k / a_{i,k} \leq b_{k,j}\}|$$

In this product, the two operators are $||$ (which denotes cardinal of a set) and $\leq$. The implementation of the dominance product using our method also yields a linear time and is performed as follows. LUTs for $||$ and $\leq$ are first implemented in the ReAP. Then A and B are stored inside the CAM as shown in Section 3a. The results are then computed in parallel inside the ReAP. For a given stage, we compare the value of A with the value of B. If the value of A is less than or equal to the value of B, we store "1" in a temporary column T. At the end of each stage, the cardinal can be implemented as an *Increment If* operation when the value in T is "1".

Another matrix processing that can be accelerated by our approach is the All-Pairs Shortest Paths (APSP) algorithm [16]. This algorithm also knows as the Floyd Warshall algorithm consists of computing the shortest paths between every two nodes in a graph.

For a given graph $G(V, E)$, we can define the adjacency matrix A which is a $|V|$ square matrix. Each element $a_{i,j}$ is the distance between vertices i and j in G. The APSP is defined as:

$$c_{i,j} = \min_k(a_{i,k} + a_{k,j})$$

In this case, operators are $\min_k$ and $+$. Similar to MM, an atomic operation, in this case, is the sum of the two elements residing on the same row. The aggregation operator is the min. This can be implemented using an in-place minimum operator.

The complexity of operations used in APSP and the dominance product are similar in ReAP. For this, and also for the sake of clearness, we will only be including APSP results in the experiments.

V. Experimental Results

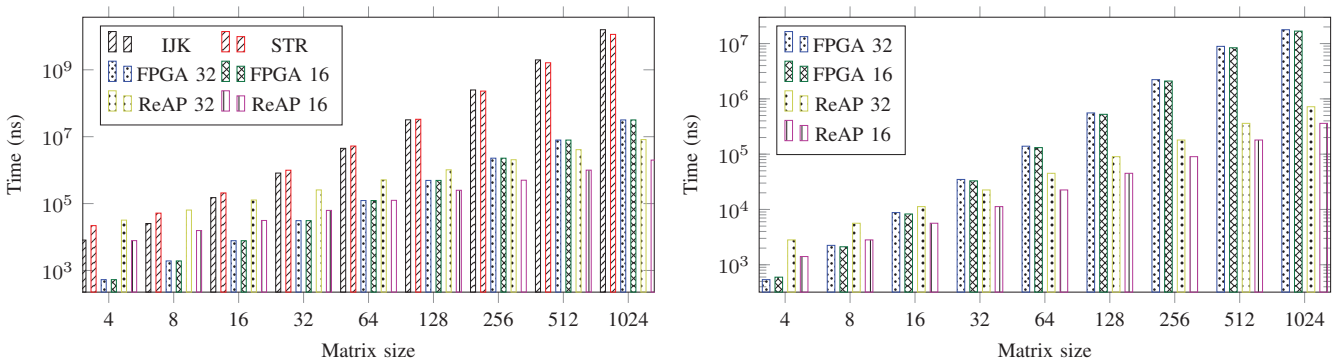
In this section, we present experimental values obtained during simulation. We will first explain the experimental environment. Results are discussed later in this section.

For performance evaluation, we have used the cycle-accurate associative processor simulator in [17]. For energy comparisons, HSpice is used to emulate the behavioral part of the ReAP. We generate the number of compares/writes for each row/cell, and using a ReAP netlist, we compute the energy consumption for a given cycle.

For the FPGA part, we used Vivado HLS to create an IP core for MM [18]. The generated design uses pipelining for the outer loop in the matrix product and unrolling/flattening for the inner loops. Input matrices are also partitioned along the correspondent dimension (1 for first matrix and 2 for the second). The partitioning factor is equal to the number of elements in each row/column. With these directives, the generated IP will create as much multiply and accumulators as it can to reach maximum parallel computations with a pipeline interval of 1. We tested the obtained IP for latency and energy results on a Virtex-7 XC7V485T.

A. Performance

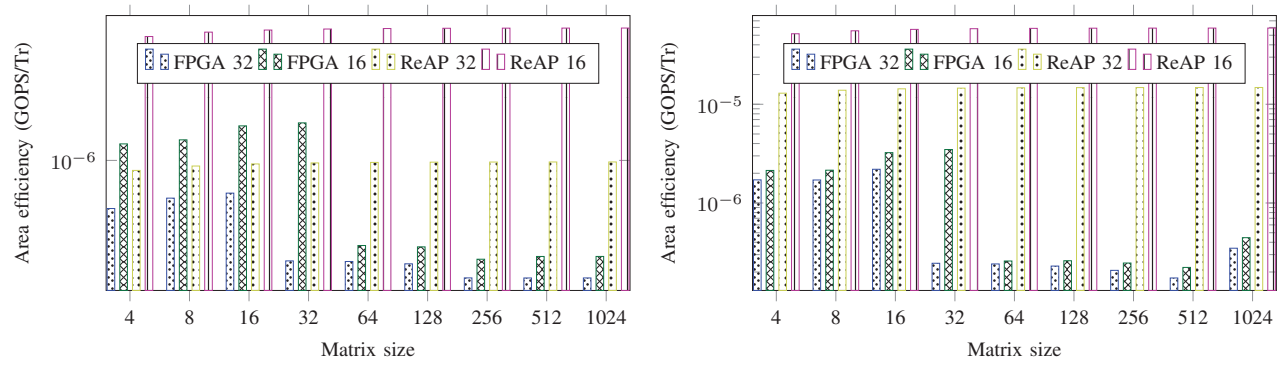
We compared the execution time of same instance sizes and compared the results in Figure 4a. This figure compares execution times for 6 implementations: on an i7 CPU using the conventional IJK algorithm and with the Strassen's (STR) implementation, on a FPGA using 16-bits and 32-bits implementation [18] and finally, using our approach on a ReAP



(a) Matrix multiplication execution time (ns).

(b) All pairs shortest paths execution time (ns).

Fig. 4: Execution time in nsec for MM (left) and APSP (right) on an i7 CPU (IJK, Strassen), an FPGA and a ReAP for 16-bit and 32-bit data width.



(a) Efficiency in area usage for matrix multiplication.

(b) Efficiency in area usage for APSP.

Fig. 5: Area efficiency in GOPS per transistor for MM (left) and APSP (right).

using 16-bits and 32-bits implementations. With relatively big matrices (256x256), the FPGA design provides very poor performance compared to the proposed ReAP method. We can also see that the time needed to compute 1024 matrix product is lower on ReAP than the 512 MM on FPGA.

As mentioned in Section IV-C, the proposed approach for MM can be generalized to any row per column application. For the APSP example, the execution time is dramatically reduced as shown in Figure 4b. The $+$ and min operations can be performed in a linear time depending on the word size on ReAP. For a word length of m=32-bits, one stage will only take 64 passes to complete. The final results are two orders of magnitude better than the FPGA design.

In addition, the ReAP is also able to store the two matrices for future usage and no time is needed to load them. Matrices can represent weights in a neural network, hence, the inference part can be done easily with no data transfers.

CPUs perform very badly against accelerators as shown in Figure 4a. For this, we decided to exclude it from area and energy comparisons.

B. Area

The ReAP approach requires storing the two matrices in separate columns. One column is needed for the result matrix. For MM, and for a word size of m, we need $4m$ cells per line; each cell contains 2 transistors and the number of rows is equal to the matrix size (n^2). The overall number of transistors is calculated using the equation:

$$Tr_{ReAP} = 2 * (4m) * n^2$$

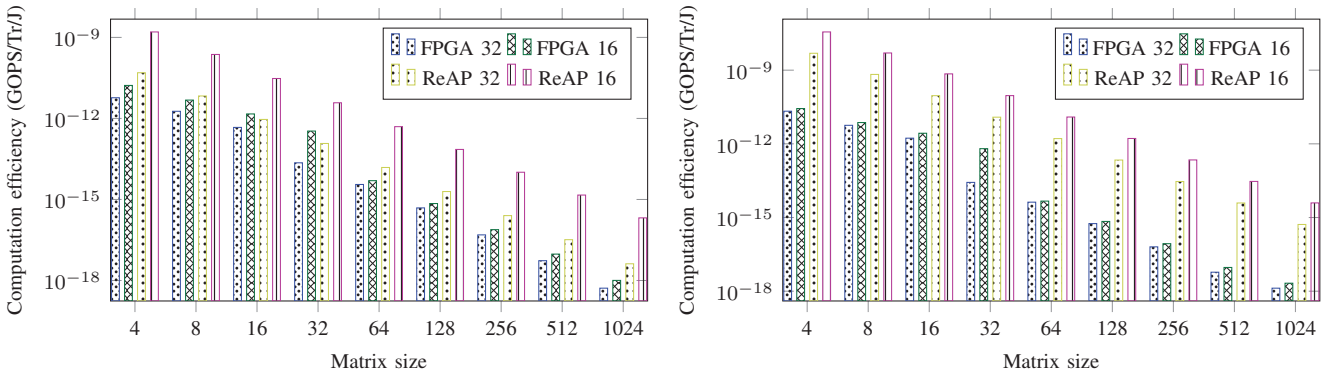
For FPGA, since the resource report does not contain the transistor count, we estimated this number using the formula:

$$Tr_{FPGA} = Tr_{BRAM} * N_{BRAM} + Tr_{FF} * N_{FF} + Tr_{LUT} * N_{LUT} + Tr_{DSP} * N_{DSP}$$

With $N_{Component}$ and $Tr_{Component}$ the number of instances of the component and the minimum number of transistors to realize it[1].

In Figure 5 we computed the number of transistors obtained by the previous formula with the corresponding estimated count on FPGA. Area efficiency is then calculated as performance per transistor. The results show an order of magnitude better efficiency for MM and two orders for the all-pairs shortest path. This proves a very small cost for the same

[1]For an 18K BRAM we need 18*1024*8*6 transistors. 18*1024 is the number of words, 8 is the bit-width and 6 is the minimum number of transistors for a memory cell.



(a) Matrix multiplication computation efficiency.

(b) All pairs shortest paths computation efficiency.

Fig. 6: Computation efficiency in GOPS per transistor per Joule for MM (left) and APSP (right).

operation. For FPGA, area depends on the type of resources used. For matrices bigger than 32, FPGA starts using BRAMs to store matrices which leads to area deficiency.

The huge difference in transistor count is mainly due to the simplicity of the 2T2R cell design [14], and the number of cells is exactly the size of the matrices times the data width. For FPGA, resources are spread between storage, logic, and communication which explains the efficiency of our approach compared to FPGA. The reduced number of transistor also affects indirectly the chip size. Since halving chip size reduces chip cost by roughly a factor of 8 (2^3) [10], [19], ReAPs will be relatively cheap compared to other similar embedded cores.

VI. Conclusions

In this paper, we presented a novel implementation of matrix multiplication on an associative processor. The method runs on a Resistive Associative Processor. The proposed technique shows better timing results compared to other efficient implementations on CPU and FPGA.

Since the memristor layer can be placed on top of the CMOS layer, resulting area is very efficient compared to FPGA based techniques. The number of transistors is an order of magnitude less in ReAP than in FPGA. This offers very high density and hence, very high computing power.

Although current memristors faces variability and short life cycles, it is very plausible to discuss the integration of memristor-based components in today's FPGA SoCs. The associative computing power combined with the non-volatile in-memory capabilities can be very useful in the hands of every SoCs developer.

References

[1] L. E. Cannon, "A cellular computer to implement the kalman filter algorithm," Ph.D. dissertation, Bozeman, MT, USA, 1969, aAI7010025.

[2] V. Strassen, "Gaussian elimination is not optimal," *Numer. Math.*, vol. 13, no. 4, pp. 354–356, Aug. 1969.

[3] V. V. Williams, "Multiplying matrices faster than coppersmith-winograd," in *Proceedings of the Forty-fourth Annual ACM Symposium on Theory of Computing*, ser. STOC '12. New York, NY, USA: ACM, 2012, pp. 887–898.

[4] N. H. Bshouty, "A lower bound for matrix multiplication," *SIAM Journal on Computing*, vol. 18, no. 4, pp. 759–765, 1989.

[5] J.-W. Jang, S. B. Choi, and V. K. Prasanna, "Energy- and time-efficient matrix multiplication on fpgas," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 13, no. 11, pp. 1305–1319, Nov 2005.

[6] J. Jiang, V. Mirian, K. P. Tang, P. Chow, and Z. Xing, "Matrix multiplication based on scalable macro-pipelined fpga accelerator architecture," in *2009 International Conference on Reconfigurable Computing and FPGAs*, Dec 2009, pp. 48–53.

[7] Z. Jovanovic and V. Milutinovic, "Fpga accelerator for floating-point matrix multiplication," *IET Computers Digital Techniques*, vol. 6, no. 4, pp. 249–256, July 2012.

[8] T. Zhang, C. Xu, T. Li, Y. Qin, and M. Nie, "An optimized floating-point matrix multiplication on fpga," *Information Technology Journal*, vol. 12, no. 9, p. 1832, 2013.

[9] R. A. Van De Geijn and J. Watts, "Summa: Scalable universal matrix multiplication algorithm," *Concurrency-Practice and Experience*, vol. 9, no. 4, pp. 255–274, 1997.

[10] S. Kaz, Y. Cliff, and P. David, "An in-depth look at googles first tensor processing unit (tpu)," https://cloud.google.com/blog/big-data/2017/05/an-in-depth-look-at-googles-first-tensor-processing-unit-tpu, 2017.

[11] A. Haron, J. Yu, R. Nane, M. Taouil, S. Hamdioui, and K. Bertels, "Parallel matrix multiplication on memristor-based computation-in-memory architecture," in *2016 International Conference on High Performance Computing Simulation (HPCS)*, July 2016, pp. 759–766.

[12] A. Morad, L. Yavits, and R. Ginosar, "Efficient dense and sparse matrix multiplication on GP-SIMD," in *2014 24th International Workshop on Power and Timing Modeling, Optimization and Simulation, PATMOS 2014*. IEEE, sep 2014, pp. 1–8.

[13] L. Yavits, A. Morad, and R. Ginosar, "Sparse Matrix Multiplication On An Associative Processor," *IEEE Transactions on Parallel and Distributed Systems*, vol. 26, no. 11, pp. 3175–3183, nov 2015.

[14] J. Li, R. Montoye, M. Ishii, K. Stawiasz, T. Nishida, K. Maloney, G. Ditlow, S. Lewis, T. Maffitt, R. Jordan, L. Chang, and P. Song, "1mb 0.41 m2 2t-2r cell nonvolatile tcam with two-bit encoding and clocked self-referenced sensing," in *2013 Symposium on VLSI Technology*, June 2013, pp. C104–C105.

[15] V. Vassilevska, R. Williams, and R. Yuster, "All-pairs bottleneck paths for general graphs in truly sub-cubic time," in *Proceedings of the thirty-ninth annual ACM symposium on Theory of computing*. ACM, 2007, pp. 585–589.

[16] R. W. Floyd, "Algorithm 97: Shortest path," *Commun. ACM*, vol. 5, no. 6, pp. 345–, Jun. 1962.

[17] H. E. Yantr, M. E. Fouda, A. M. Eltawil, and F. J. Kurdahi, "Process variations-aware resistive associative processor design," in *2016 IEEE 34th International Conference on Computer Design (ICCD)*, Oct 2016, pp. 49–55.

[18] D. Bagni, A. D. Fresco, J. Noguera, and F. M. Vallina, "A zynq accelerator for floating point vivado hls," Jan. 2016.

[19] J. L. Hennessy and D. A. Patterson, *Computer architecture: a quantitative approach*. Elsevier, 2011.