Ultra-Efficient Processing In-Memory for Data Intensive Applications

Mohsen Imani, Saransh Gupta and Tajana Rosing
CSE Department, UC San Diego, La Jolla, CA 92093, USA
{moimani, sgupta, tajana}@ucsd.edu

ABSTRACT

Recent years have witnessed a rapid growth in the domain of Internet of Things (*IoT*). This network of billions of devices generates and exchanges huge amount of data. The limited cache capacity and memory bandwidth make transferring and processing such data on traditional CPUs and GPUs highly inefficient, both in terms of energy consumption and delay. However, many *IoT* applications are statistical at heart and can accept a part of inaccuracy in their computation. This enables the designers to reduce complexity of processing by approximating the results for a desired accuracy. In this paper, we propose an ultra-efficient approximate processing in-memory architecture, called APIM, which exploits the analog characteristics of non-volatile memories to support addition and multiplication inside the crossbar memory, while storing the data. The proposed design eliminates the overhead involved in transferring data to processor by virtually bringing the processor inside memory. APIM dynamically configures the precision of computation for each application in order to tune the level of accuracy during runtime. Our experimental evaluation running six general OpenCL applications shows that the proposed design achieves up to 20× performance improvement and provides 480× improvement in energy-delay product, ensuring acceptable quality of service. In exact mode, it achieves 28× energy savings and 4.8× speed up compared to the state-of-the-art GPU cores.

CCS CONCEPTS

•**Hardware** → **Emerging architectures;** Non-volatile memory;

KEYWORDS

Processing in-memory, Non-volatile memory, Emerging computing

ACM Reference format:
Mohsen Imani, Saransh Gupta and Tajana Rosing. 2016. Ultra-Efficient Processing In-Memory for Data Intensive Applications. In *Proceedings of ACM conference, Austin, Texas USA, June 2017 (DAC '17),* 6 pages.
DOI: http://dx.doi.org/10.1145/3061639.3062337

1 INTRODUCTION

Today's IoT applications often analyze raw data by running machine learning algorithms such as classification or neural networks in data centers [1, 2]. Sending the entire data to cloud for processing is not scalable and cannot guarantee the required real-time response [3]. Running data intensive workloads with large datasets on traditional cores results in high energy consumption and slow processing speed, majorly due to the large amount of data movement between memory and processing units [4, 5]. The idea of near-data computing aims to address this problem. Near-data computing puts the processing units close to the main memory, which accelerates the computation by avoiding the memory/cache bottleneck [4, 6, 7]. Although this idea improves performance, it may consume more energy due to the extra computing units added to the memory. A more efficient way of addressing data movement issue is to process data within the memory, thus improving both performance and energy efficiency [5, 8].

On the other hand, most of the algorithms running on today's sensors' data are statistical in nature, and thus do not require exact answers [9, 10]. Similarly, in audio and video processing we have long exploited the fact that humans do not perceive all the colors or sounds equally well. Approximate computing is an effective way of improving the energy and performance by trading some accuracy. Much of the prior work seeks to exploit this fact in order to build faster and more energy efficient systems which are capable of responding to our needs with just good enough quality of response [11–13]. However, most of the existing techniques provide less energy or performance efficiency due to considerable data movement and lack of configurable accuracy.

Non-volatile memories (NVMs) are promising candidates for data intensive in-memory computations, where the an NVM cell can be used for both storing data and processing it [14–17]. Resistive random access memory (RRAM) is one such memory which stores data in form of its resistance. Owing to its low energy requirements, high switching speeds, and scalability, RRAM can be used to develop high capacity memories with great performance. These memories are compatible with the current CMOS fabrication process [18] which enables their easy integration with the existing technology.

In this paper, we propose a configurable approximate processing in-memory architecture, called APIM, which supports addition and multiplication operations inside the non-volatile RRAM-based memory. APIM exploits the analog characteristic of the memristor devices to enable basic bitwise computation and then, extend it to fast and configurable addition and multiplication within memory. This would make processing machine learning and data analysis algorithms more efficient since most of them use addition and multiplication aggressively. We propose a blocked crossbar memory which introduces flexibility in executing operations and facilitates shift operations in memory. Then, we introduce a novel approach for fast addition in memory. Finally, we design an in-memory multiplier using the proposed memory unit and fast adder. For each application, APIM can dynamically tune the level of approximation in order to trade the accuracy of computation while improving energy and performance. Our experimental evaluation over six general OpenCL applications shows that the proposed design improves the performance by 20× and provides 480× improvement in energy-delay product while ensuring less than 10% average relative error.

2 RELATED WORK AND BACKGROUND

Processing in-memory (PIM) can accelerate computation by reducing the overhead of data movement [5, 8]. Early PIM designs integrate high performance CMOS logic and memory on the same die. They predominantly designed a custom processing unit next to the sense amplifiers of main memory to support basic bitwise

DAC '17, Austin, Texas USA
 978-1-4503-4927-7/17/06…$15.00
DOI: http://dx.doi.org/10.1145/3061639.3062337

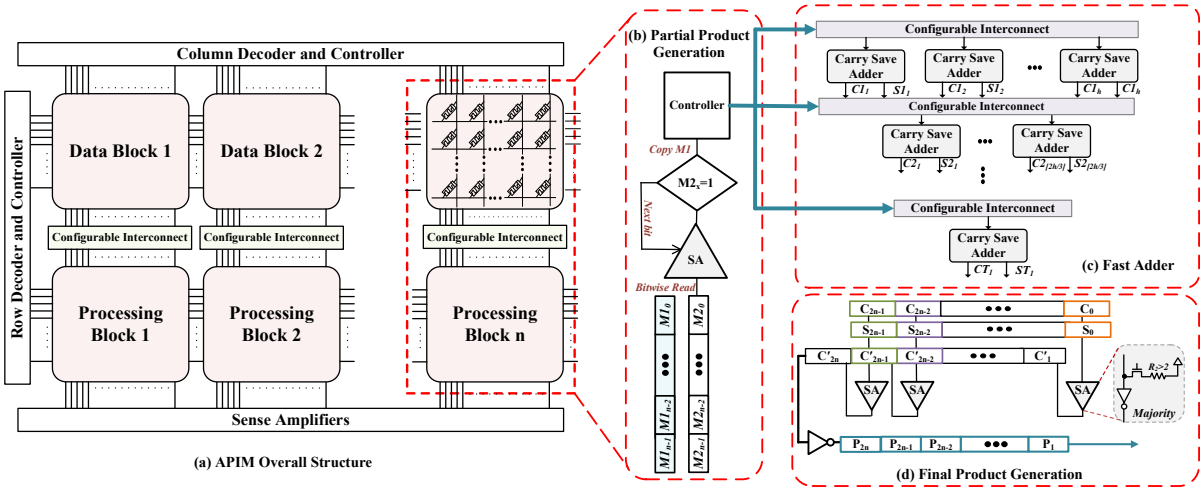


Figure 1: (a) The overall structure of APIM consisting of several data and processing blocks. (b) the APIM controller and parallel product generator. (c) Fast adder three structure consisting of Carry save adder and configurable interconnects. (d) Final product generator to ripple the carry bits of tree structure.

operations. However, to support other advance functionality (e.g. addition, multiplication), they added dedicated CMOS-based processing cores, making the manufacturing process complicated and costly.

High density, low-power consumption, and CMOS-compatibility of emerging non-volatile memories (NVMs), in particular memristor devices, make them appropriate candidates for both memory and computing unit [14, 15, 19, 20]. Many logic families have been proposed for computation inside memristive crossbar. Some of them implement logic purely in memory such as stateful implication logic [21, 22], and Memristor Aided loGIC (MAGIC) [23].The work in [24] presents schemes for addition in memristive crossbar memory. The direct application of these schemes in data intensive processing is limited largely due to the linear dependency of latency of execution on the size of data. While [25] presents a very fast adder, the area overhead involved in arrayed addition grows significantly for data intensive workloads.

We use `MAGIC NOR` [23] to execute logic functions in memory due to its simplicity and independence of execution from data in memory. An execution voltage, V_0, is applied to the bitlines of the inputs (in case of `NOR` in a row) or wordlines of the outputs (in case of `NOR` in a column) in order to evaluate `NOR`, while the bitlines of the outputs (`NOR` in a row) or wordlines of the inputs (`NOR` in a column) are grounded. The work in [24] extends this idea to implement adder in a crossbar. It executes a pattern of voltages in order to evaluate sum (S) and carry (C_{out}) bits of 1-bit full addition (inputs being A, B, C) given by

$$C_{out} = ((A+B)' + (B+C)' + (C+A)')'. \quad (1a)$$

$$S = (((A'+B'+C')' + ((A+B+C)' + C_{out})')')'. \quad (1b)$$

This design takes $12N+1$ cycles to add two N-bit numbers. We define a cycle time ($= 1.1ns$) as the time taken to implement one `MAGIC NOR` operation.

In contract, we propose a novel processing in-memory architecture which supports fast and efficient addition and multiplication operations. Our design supports approximation and configures it dynamically depending upon the requirements of different applications.

3 PROPOSED APIM

This section presents the design flow and techniques used to execute addition and multiplication in memory. Figure 1(a) shows the memory unit. It is a simple crossbar memory divided into different blocks further explained in Section 3.1. Section 3.2 proposes a fast adder, Figure 1(c), which uses a tree structure in the memory unit to add large number of operands with minimum delay. Section 3.3 describes a multiplier, Figure 1(b)-(d), which utilises the fast adder along with the modified sense amplifiers of the memory unit to obtain product in an efficient manner. We also propose approximation techniques in Section 3.4 which speed up multiplication while ensuring high accuracy.

3.1 APIM Architectural Overview

A typical crossbar memory is an array of unit memory cells. In case of RRAM, these cells are made of resistive switching elements such as memristors. Each cell in the memory is accessed by activating the corresponding wordline and bitline. MAGIC makes execution of operations in a crossbar memory simple. It also allows easy copying of data provided the source and destination are in the same column/row. While being acceptable in many cases, this memory structure limits the performance of instructions which involve a lot of shifting and asymmetric movement of data. One such instruction is multiplication where the multiplicand is shifted and added. Multiple copy operations can emulate a shift operation. However, such an approach is impractical when the number to be shifted is large since it requires shifting each and every bit individually. The problem is aggravated when multiple such numbers are to be shifted.

We hence propose the use of a blocked memory structure as shown in Figure 1(a). The crossbar is divided into blocks. Any new data which is loaded into the memory is stored in the data block. Whenever there is a request to process data, it is copied to the processing block and computation is done. The two blocks are structurally the same and can be used interchangeably. These blocks are connected by configurable interconnects. The interconnects support shift operations inherently such that i^{th} bitline of one block can be connected to $(i+j)^{th}$ bitline of another block. The availability

of interconnects allows the memory to shift data while copying it from one block to another without introducing any latency overhead. This makes shifting an efficient operation since the entire string of data can be shifted at once, unlike shifting each bit individually.

3.2 Fast Addition in APIM

The design in [24] is good for small numbers but as the length of numbers increases, time taken increases linearly. A $N \times M$ multiplication requires addition of M partial products, each of size N bits, to generate a $(N+M)$-bit product. This takes $(M-1)\cdot(12(N-1)+1)$ cycles to obtain the final product.

In order to optimize latency of addition, we propose a fast adder for memristive memories. Our design is based on the idea of carry save addition (CSA) and adapts it for in-memory computation. Figure 2(a) shows carry save addition. Here, $S1[n]$ and $C1[n]$ are the sum and carry-out bits, respectively of 1-bit addition of $A1[n]$, $A2[n]$, and $A3[n]$. The 1-bit adders do not propagate the carry bit and generate two outputs. This makes the n additions independent of each other. The proposed adder exploits this property of CSA. Since, `MAGIC` execution scheme doesn't depend upon the operands of addition, multiple addition operations can execute in parallel if the inputs are mapped correctly. The design utilises the proposed memory unit, which supports shifting operations, to implement CSA like behaviour. The latency of this *3:2* reduction, 3 inputs to 2 outputs, is same as that of a 1-bit addition (*i.e.*, 13 cycles) irrespective of the size of operands. The two numbers can then be added serially, consuming $12N+1$ cycles. This totals to $12N+14$ cycles while the previous adder would take $24N-22$ cycles. The difference increases linearly with the size of inputs.

We use a Wallace-tree inspired structure leveraging the fast *3:2* reduction of our new adder design, as shown in Figure 2(b), to add multiple numbers (9 n-bit numbers in this case). At every stage of execution, the available addends are divided in groups of three. The addends are then added using a separate adder (as described above) for each group, generating two outputs per group. The additions in the same stage of execution are independent and can occur in parallel to each other. Our configurable interconnect arranges the outputs of this stage in groups of three for addition in the next stage. This structure takes a total of four stages for *9:2* reduction, having the same delay as that of four 1-bit additions. At the end of the tree structure we are left with two $(N+3)$-bit numbers which can then be added serially. The tree-structured addition reduces the delay substantially as carry propagation happens only in the last stage, unlike the conventional approach where carry is propagated at every step of addition. Although this speed up comes at the cost of increased energy consumption and number of writes in memory, it is acceptable because the latency is reduced by large margins as shown in Section 4.

3.3 Multiplication in APIM

The process of multiplication can be divided into three stages, partial product generation, fast addition, and final product generation as shown in Figure 1. The partial product generation stage creates partial products of a $N \times N$ multiplication. These partial products can then be added serially (inefficient) or by the fast adder introduced in Section 3.2. The fast addition reduces N numbers to 2. The final product generation stage adds two numbers generated by the previous stage and outputs the product of $N \times N$ multiplication.

A partial product is the result of `AND`ing the multiplicand (M_1) with a bit of the multiplier (M_2). Hence, $N \times N$ multiplication generates N (size of M_2) partial products of size N-bits (size of M_1). AND operation can be implemented as a series of three `NOR`

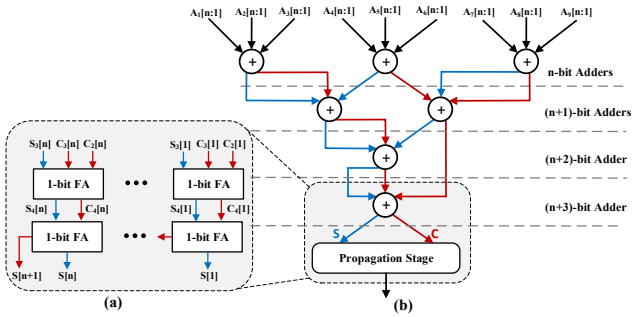


Figure 2: (a) Carry save addition (b) Tree structured addition of 9 n-bit numbers

operations as given by,

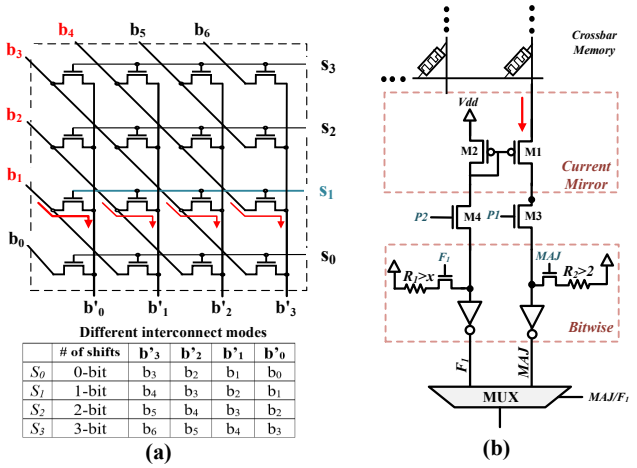
$$F = AND\ (A,B) = NOR\ (NOR(A),\ NOR(B)) \tag{2}$$

This requires three cycles given that the inputs A & B and output F are in the same row or column. In the case of in-memory computation, even if we assume that the numbers to be multiplied are located adjacent to each other, we would require an empty crossbar row/column of length N^N which is quite large even for $N = 16$. This would be expensive not only in terms of area but also in latency, requiring $3N^N$ cycles.

We propose the use of sense amplifiers to develop a faster partial product generator as shown in Figure 1(b). In order to avoid the time and area overhead involved in transposing and creating multiple copies of multiplier, we read-out the multiplier. The design exploits the fact that the partial product is the multiplicand itself if multiplier bit is '1' and 0 otherwise. M_2 is read bit-wise using the sense amplifier. If the read bit is '1', M_1 is copied, while nothing is done when the bit is '0'. We achieve this by modifying the multiplexer in the controller, incorporating the sensed bit in the select signals. In this way, we avoid writing data when the bit is zero, thus saving energy. A copy operation is equivalent to two successive *NOT* operations. This result is used for all successive copy operations, limiting the worst case delay of copying to $N+1$ cycles. The actual delay would vary depending upon the number of '1s' in M_2.

Although this is a huge improvement in latency from the initial design, we have not yet considered the cost of shifting the partial products for add operation. Shift operation in a normal memory crossbar can only be done bitwise, which would be quite large given the number and size of operands to be shifted. The blocked memory architecture introduced in Section 3.1 proves advantageous in this scenario. If the above operations are performed in a blocked memory crossbar, the latency of shifting would actually reduce to zero. Shift operation can be clubbed with copy operation and hence, shifted partial products can be obtained in the processing block at no extra delay.

The fast adder discussed in Section 3.2 reduces the generated partial products to 2, owing to its N:2 reduction. Since a step in the fast adder involves parallel additions, it requires that the three addends of a $3:2$ adder are present in the same columns (rows) and all such groups in a step are present in the same rows (columns). Interestingly, arranging the partial products in this manner involves no added latency as this arrangement can be done while shifting and copying the data in the partial product generation stage. Instead of writing the partial products one below the other, the interconnects are set such that the partial products are arranged in the required way. After the first step of $3:2$ reduction, we again need to arrange the intermediate results into groups of three. This can be done by moving these results to the data block, performing the next $3:2$



	# of shifts	b'3	b'2	b'1	b'0
S_0	0-bit	b_3	b_2	b_1	b_0
S_1	1-bit	b_4	b_3	b_2	b_1
S_2	2-bit	b_5	b_4	b_3	b_2
S_3	3-bit	b_6	b_5	b_4	b_3

Figure 3: The circuit of (a) an interconnect (b) a sense amplifier.

reduction there (blocks being functionally the same), and coming back to the current block for the following reduction. In this way, $N:2$ reduction can be efficiently executed by utilising only 2 blocks of the memory, toggling between them at every step. However, if the data block is specifically reserved for storing data and bars logic execution, a 3-level memory (with 2 processing blocks per data block) can be used. The reduction is done until only 2 addends remain.

The major advantage of reduction addition is that the time taken by this adder is independent of the size of the operands *i.e.*, $N \times 32$ multiplication takes the same time in this stage for any value of N. It varies only by the number of operands to be added. Moreover, since we only generate a partial product when the multiplier bits are 1, the actual number of operands to be added is quite small. For instance, with a random input data, there would be only 16 additions on average for 32×32 multiplication. The final product generation stage adds the two outputs of the previous stage to generate the required product.

Figure 3(a) shows the configurable interconnect used in our design. It can be visualized as a collection of switches, similar to a barrel shifter, which connects the bitlines of the two blocks. b_n and b'_n are bitlines coming into and going out of the interconnect respectively. The select signals, s_n control the amount of shift. These interconnects can connect cells with different bitlines together. For example, they can connect $b_n, b_{n+1}, b_{n+2}, \ldots$ incoming bitlines to, say, $b'_{n+4}, b'_{n+5}, b'_{n+6}, \ldots$ outgoing bitlines, respectively, hence enabling the flow of current between the cells on different bitlines of blocks. This kind of a structure makes shifting operations energy efficient and fast, having the latency same as that of a normal copy operation. It also allows for inter-column `MAGIC NOR` operations between the two blocks. If inputs are stored on n^{th} bitline of one block, the output of `NOR` operation can be stored on, say, $(n+4)^{th}$ bitline of another block. This can be extended to multiple `NOR` operations in parallel. The shift select signals, s_n, are controlled by the memory controller present at the periphery of memory unit. It is important to note that all of these blocks still share the same row and column controllers and decoders. So, the area and logic overhead introduced by the proposed memory unit is restricted to the interconnect circuit and its control logic.

3.4 APIM Approximation

The individual additions in the final stage cannot occur in parallel since they require the propagation of carry in order to generate the final answer. The two addends in the final stage of APIM multiplication are of size $2N$ each. The conventional approach requires $13 \cdot 2N$ cycles to compute the result. This latency is dominant as compared to the previous stages of multiplication, making the last stage a bottleneck of the entire process.

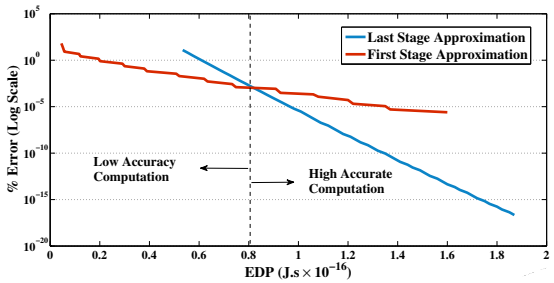
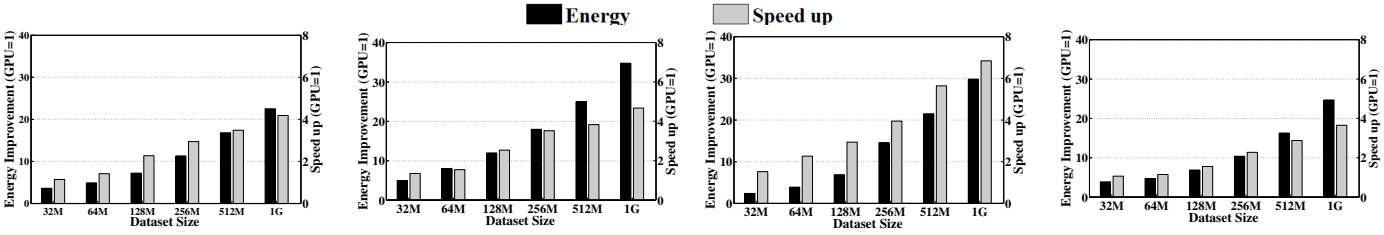


Figure 4: Error and EDP comparison of the two approximation approaches.

However, we can dramatically speed it up if a fully accurate result is not desired. This is the case with many highly data intensive applications which tolerate some inaccuracy as long as it is within the prescribed limits. One approach is to mask some of the LSBs of M_2, reducing the number of computations. The other approach approximates the sum bits in the last stage from the accurately generated carry bits and saves the delay involved in calculation of the bit. We use the second approach in the evaluation of our design.

In the first approach, the number of masked bits depends upon the amount of accuracy desired. Masking the bits of the multiplier effectively reduces the number of partial products to be added because we don't generate partial products when the multiplier bit is 0. For example, masking 8 LSBs of M_2 in the first stage reduces a 32×32 multiplication to 32×24. Hence, this method of approximation results in a direct reduction in delay and energy consumption of multiplication. It comes majorly due to the reduction in computation in the fast adder stage. However, since this approach masks the bits in the initial stage itself, the error propagates through the entire process, resulting in huge errors in some cases. This makes it unsuitable for an application demanding very high accuracy.

In the second approach, our design exploits the fact that the sum bit (S) of an 1-bit addition is the complement of the generated carry bit (C_{out}) except for two combinations of inputs (*i.e.*,(A,B,C) = (0, 0, 0) and (1, 1, 1)) [26]. It evaluates C_{out} accurately (hence, preventing the propagation of error) and then approximates S. Our design uses a modified sense amplifier (SA). It supports basic memory operation along with `MAJ` (majority) function as shown in Figure 3(b). The carry generated (C_{out}) as a result of addition of three input bits (A,B,C_{in}) is MAJ(A,B,C) *i.e.*, $AB+BC+CA$. Our circuit level evaluation shows that just reading the inputs from memory takes about $0.3ns$, while our design needs $0.6ns$ to calculate majority and compute C_{out} resulting in an effective delay of less than 1 cycle. One additional cycle is needed to write the computed C_{out} to the memory. It acts as an input to the next 1-bit addition, the output of SA is written such that it is in the same column as that of the next two inputs, saving the trouble and cost of copying it. Since the carry bit is propagated, these 1-bit additions cannot occur in parallel to each other. The computation of C_{out} takes $2 \cdot 2N$ cycles. All S bits can then be approximated by just inverting the C_{out} bits, which costs only 1 cycle and can all be done at the same time. This technique reduces the latency from $13 \cdot 2N$ (time taken to add two $2N$-bit numbers) cycles to $2 \cdot 2N+1$ cycles. This improvement comes with a significant cost of 25% error (2 out of 8 cases) for a random input data.



(a) Sobel (b) Robert (c) FFT (d) DwtHaar1D

Figure 5: Energy consumption and speedup of exact APIM normalized to GPU vs different dataset sizes.

The accuracy can be improved substantially by approximating just a part of the final product while accurately calculating the rest of the product. The design improves accuracy by dividing the product into two groups of size k and m bits such that $k+m=2N$. The k bits are calculated using the conventional approach which consumes $13k$ cycles and produces k accurate bits in the product. On the other hand, m bits are approximated using the technique described above, which takes a total of $2m+1$ cycles. This increases the accuracy since the k accurate bits are generally the most significant bits and any error in the m least significant bits has less effect on the result, as shown in Section 4.3. The proposed technique implements the final stage with a latency of $13k+2m+1$ cycles. The appropriate values of k and m depend upon the application in hand. Section 4.3 talks about different applications and selecting these values in order to obtain acceptable results.

While approximation in the last stage reduces the latency, it is still slower that the first approach in which approximation is done in the first stage. The first approach is more energy efficient too since it reduces the size of multiplication and uses less resources. However, unlike the first approach, second approach introduces error only in the final stage of the process. This approach can thus achieve very low error rates making it suitable for applications requiring precise results. Figure 4 presents a comparison between the two approaches for 32×32 multiplication. While, approximation in the first stage is convenient for error tolerant applications, approximation in the last stage can guarantee very high accuracies for similar EDPs. For example, for 32×32 multiplication and an EDP of 1.4×10^{-16}, the error rate for the second approach is less by 5 orders of magnitude as compared to the first approach.

4 RESULTS

4.1 Experimental Setup

We compare the efficiency of the proposed APIM design with state-of-the-art processor AMD Radeon R9 390 GPU with 8GB memory. In order to avoid the disk communication in the comparison, all the data used in the experiments is preloaded into 64GB, 2.1GHz DDR4 DIMMs. We used Hioki 3334 power meter to measure the power consumption of GPU. We implement the APIM functionality by changing the multi2sim [27], cycle-accurate CPU-GPU processor. Performance and energy consumption of proposed hardware are obtained from circuit level simulations for a 45nm CMOS process technology using Cadence Virtuoso. We use VTEAM memristor model [28] for our memory design simulation with R_{ON} and R_{OFF} of $10k\Omega$ and $10M\Omega$ respectively.

We compare the efficiency of APIM and GPU by running six general OpenCL applications including: *Sobel, Robert, Fast Fourier transform (FFT), DwHaar1D, Sharpen* and *Quasi Random*. For image processing we use random images from *Caltech 101* [29] library, while for non-image processing applications inputs are generated randomly. Majority of these applications consists of additions and multiplications. The other common operations such as square root has been approximated by these two functions in OpenCL code. To evaluate the computation accuracy in approximate mode, our framework compares the approximate output file of each application with the golden output from calculating exactly. For image processing applications, we accept $30dB$ peak signal-to-noise ratio as an accuracy metric. For other applications, the acceptable accuracy is defined by having less than 10% average relative error. To find a proper level of accuracy, our framework computes APIM at the maximum level of approximation (32 relax bits). In case of large inaccuracy, it increases the level of accuracy in 4-bit steps until ensuring the acceptable quality of service.

4.2 Exact APIM & Dataset Size

Figure 5 shows the energy savings and performance improvements of running applications on APIM exact, normalized to GPU energy and performance. For each application, the size of input dataset increases from 1Kb to 1GB. In traditional cores, the energy and performance of computation consists of two terms: computation and data movement. In small dataset (˜KB), the computation cost is dominant, while running applications with large datasets (˜GB), the energy and performance of consumption are bound by the data movement rather than computation cost. This data movement is due to small cache size of transitional core which increases the number of cache miss. Consecutively, this degrades the energy consumption and performance of data movement between the memory and caches. In addition, large number of cache misses, significantly slows down the computation in traditional cores. In contrast, in proposed APIM architecture the dataset is already stored in the memory and computation is major cost. Therefore, regardless of dataset size (the dataset can fit on APIM), the APIM energy and performance of increases linearly by the dataset size. Although the memory-based computation in the APIM is slower than transitional CMOS-based computation (*i.e.* floating point units in GPU), in processing the large dataset, the APIM works significantly faster than GPU. In terms of energy, the memory-based operations in APIM is more energy efficient than GPU. Our evaluation shows that for most applications using datasets larger than 200MB (which is true for many IoT applications), APIM is much faster and more energy efficient than GPU. With 1GB dataset, the APIM design can achieve 28× energy savings, 4.8× performance improvement as compared to GPU architecture.

Figure 6 compares the performance efficiency of the proposed design with the state-of-the-art prior work [24, 25]. The work in [24] computes addition in-memory using MAGIC logic family, while the work in [25] uses the complementary resistive switching to perform addition inside the crossbar memory. Our evaluation comparing the energy and performance of addition of N operands of length N bits each shows that the APIM can achieve at least 2× speed

Table 1: Quality of loss and EDP improvement of the proposed APIM compared to GPU in different level of approximation.

Applications	0 bit		4 bits		8 bits		16 bits		24 bits		32 bits	
	EDP Imp.	QoL	EDP Imp.	QoL	EDP Imp.	QoL	EDP Imp.	QoL	EDP Imp.	QoL	EDP Imp.	QoL
Sobel	94×	0%	164×	1.3%	235×	3.1%	305×	**6.9%**	376×	11.4%	446×	15.6%
Robert	177×	0%	311×	1.2%	444×	2.9%	577×	4.8%	711×	6.8%	844×	**9.1%**
FFT	203×	0%	356×	2.2%	509×	3.7%	662×	5.8%	815×	**8.6%**	968×	13.5%
DwHaar1D	90×	0%	157×	0.9%	225×	2.6%	293×	5.7%	361×	**7.9%**	428×	10.6%
Sharpen	104×	0%	149×	3.4%	206×	5.1%	273×	**8.1%**	340×	12.5%	410×	18.4%
QuasiR	69×	0%	127×	2.1%	198×	3.5%	258×	5.8%	310×	**9.3%**	386×	15.7%

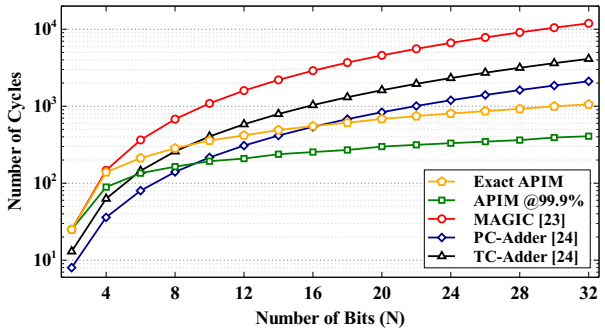


Figure 6: Performance comparison of the proposed design with previous work for addition of N operands, each sized N bits

up compared to previous designs in exact mode. APIM can be at least 6× faster with 99.9% accuracy. The proposed design is even better since the calculations for previous work do not include the latency involved in shift operations. This improvement comes at the expense of the overhead of interconnect circuitry and its control logic. However, the next best adder, *i.e.,* the PC-Adder [25] uses multiple arrays each having different wordline and bitline controllers, introducing a lot of area overhead. This overhead is not present in our design since all the blocks share the same controllers.

4.3 Approximate APIM

Table 1 shows the energy consumption and performance of the applications running on APIM in different approximation level. As we explained in Section 3.4, APIM approximates m least significant bits of the product in the final product generation stage. The value of m has an important impact on the computation accuracy and performance of multiplication in APIM design. As Table 1 shows, large number of approximate LSBs (m) significantly improve energy-delay product of APIM, at the expense of increased computation inaccuracy. Similarly, small m makes the computation more accurate with small benefit. We observed that different applications satisfy the required accuracy for different values of m. Therefore, our design detects the application at runtime and then sets the pre-calculated value of m that is obtained offline to ensure the acceptable quality of computation. Using this adaptive design, our design can achieve 480× energy-delay product improvement as compared to APIM in the exact mode.

5 CONCLUSION

In this paper, we propose a configurable processing in-memory architecture which enables addition and multiplication operations inside the memory, while storing data. The design addresses the inefficiency involved in moving data by enabling computations within memory. The performance of the design is further enhanced by approximating results with high accuracies. Our evaluation shows that increasing the dataset size significantly improves the energy efficiency of APIM. In addition, by enabling tunable approximation during the runtime, our design can tune the level of approximation to trade energy/performance with accuracy. Our evaluation running four OpenCL application shows that in approximate mode (exact mode), the APIM can achieve up to 480× (123×) energy-delay product improvement compared to recent GPU architecture ensuring the acceptable quality of service.

6 ACKNOWLEDGMENT

This work was supported by NSF grant #1527034 and Jacobs School of Engineering UCSD Powell Fellowship.

REFERENCES

[1] J. Gubbi *et al.*, "Internet of things (IoT): A vision, architectural elements, and future directions," *Future Generation Computer Systems*, vol. 29, no. 7, pp. 1645–1660, 2013.
[2] M. Samragh *et al.*, "Looknn: Neural network with no multiplication," in *IEEE/ACM DATE*, 2017.
[3] K. Hwang *et al.*, *Distributed and cloud computing: from parallel processing to the internet of things*. Morgan Kaufmann, 2013.
[4] R. Balasubramonian *et al.*, "Near-data processing: Insights from a micro-46 workshop," *Microarchitecture*, vol. 34, no. 4, pp. 36–42, 2014.
[5] G. Loh *et al.*, "A processing-in-memory taxonomy and a case for studying fixed-function pim," in *WoNDP*, 2013.
[6] M. Imani *et al.*, "Mpim: Multi-purpose in-memory processing using configurable resistive memory," in *IEEE ASP-DAC*, pp. 757–763, IEEE, 2017.
[7] S. Pugsley *et al.*, "Comparing implementations of near-data computing with in-memory mapreduce workloads," *Microarchitecture*, vol. 34, no. 4, pp. 44–52, 2014.
[8] A. M. Aly *et al.*, "M3: Stream processing on main-memory mapreduce," in *ICDE*, pp. 1253–1256, IEEE, 2012.
[9] J. Han *et al.*, "Approximate computing: An emerging paradigm for energy-efficient design," in *ETS*, pp. 1–6, IEEE, 2013.
[10] M. Imani *et al.*, "Efficient neural network acceleration on gpgpu using content addressable memory," in *IEEE/ACM DATE*, 2017.
[11] M. Imani *et al.*, "Resistive configurable associative memory for approximate computing," in *DATE*, pp. 1327–1332, IEEE, 2016.
[12] V. Gupta *et al.*, "Impact: imprecise adders for low-power approximate computing," in *ISLPED*, pp. 409–414, IEEE, 2011.
[13] M. Imani *et al.*, "Masc: Ultra-low energy multiple-access single-charge tcam for approximate computing," in *IEEE/ACM DATE*, pp. 373–378, IEEE, 2016.
[14] Q. Guo *et al.*, "Ac-dimm: associative computing with stt-mram," in *ISCA*, vol. 41, pp. 189–200, ACM, 2013.
[15] Q. Guo *et al.*, "A resistive tcam accelerator for data-intensive computing," in *Microarchitecture*, pp. 339–350, ACM, 2011.
[16] M. Imani *et al.*, "Exploring hyperdimensional associative memory," in *IEEE HPCA*, IEEE, 2017.
[17] X. Yin *et al.*, "Design and benchmarking of ferroelectric fet based tcam," in *IEEE/ACM DATE*, IEEE, 2017.
[18] J. Borghetti *et al.*, "A hybrid nanomemristor/transistor logic circuit capable of self-programming," *PNAS*, vol. 106, no. 6, pp. 1699–1703, 2009.
[19] M. Imani *et al.*, "Acam: Approximate computing based on adaptive associative memory with online learning," in *IEEE/ACM ISLPED*, pp. 162–167, 2016.
[20] L. Yavits *et al.*, "Resistive associative processor," *IEEE Computer Architecture Letters*, vol. 14, no. 2, pp. 148–151, 2015.
[21] J. Borghetti *et al.*, "Memristive switches enable stateful logic operations via material implication," *Nature*, vol. 464, no. 7290, pp. 873–876, 2010.
[22] S. Kvatinsky, G. Satat, N. Wald, E. G. Friedman, A. Kolodny, and U. C. Weiser, "Memristor-based material implication (IMPLY) logic: design principles and methodologies," *TVLSI*, vol. 22, no. 10, pp. 2054–2066, 2014.
[23] S. Kvatinsky *et al.*, "MAGIC – memristor-aided logic," *TCAS II*, vol. 61, no. 11, pp. 895–899, 2014.
[24] N. Talati *et al.*, "Logic design within memristive memories using memristor-aided loGIC (MAGIC)," *IEEE TNano*, vol. 15, pp. 635–650, jul 2016.
[25] A. Siemon *et al.*, "A complementary resistive switch-based crossbar array adder," *JETCAS*, vol. 5, no. 1, pp. 64–74, 2015.
[26] V. Gupta *et al.*, "Low-power digital signal processing using approximate adders," *TCAD*, vol. 32, no. 1, pp. 124–137, 2013.
[27] R. Ubal *et al.*, "Multi2sim: a simulation framework for cpu-gpu computing," in *PACT*, pp. 335–344, ACM, 2012.
[28] S. Kvatinsky *et al.*, "Vteam: a general model for voltage-controlled memristors," *TCAS II*, vol. 62, no. 8, pp. 786–790, 2015.
[29] "Caltech Library." http://www.vision.caltech.edu/Image_Datasets/Caltech101/.