GenPIM: Generalized Processing In-Memory to Accelerate Data Intensive Applications

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

***Abstract*—Big data has become a serious problem as data volumes have been skyrocketing for the past few years. Storage and CPU technologies are overwhelmed by the amount of data they have to handle. Traditional computer architectures show poor performance when processing such huge data. Processing in-memory is a promising technique to address data movement issue by locally processing data inside memory. However, there are two main issues with stand-alone PIM designs: (i) PIM is not always computationally faster than CMOS logic, (ii) PIM cannot process all operations in many applications. Thus, not many applications can benefit from PIM. To generalize the use of PIM, we designed GenPIM, a general processing in-memory architecture consisting of the conventional processor as well as the PIM accelerators. GenPIM supports basic PIM functionalities in specialized non-volatile memory including: bitwise operations, search operation, addition and multiplication. For each application, GenPIM identifies the part which uses PIM operations, and processes the rest of non-PIM operations or not data intensive part of applications in general purpose cores. GenPIM also enables configurable PIM approximation by relaxing in-memory computation. We test the efficiency of proposed design over two learning applications. Our experimental evaluation shows that our design can achieve 10.9× improvement in energy efficiency and 6.4× speedup as compared to processing data in conventional cores.**

I. Introduction

In recent years, the number of smart electronic devices has surpassed the number of humans in the world [1]. The paradigm of Internet of Things looks at all these electronic devices as a big network of connected devices which generate huge amount of raw data. Several algorithms try to pre-process, compress or secure such big data including machine learning, compression and security algorithms [2]. These algorithms are traditionally run on conventional general purpose processor. However, conventional processing architectures show poor performance when processing big data. This efficiency comes from large amount of data movement issue between the main memory and processing cores [3]. In particular, the limited on-chip cache memories in traditional cores along with limited memory bandwidth of main memory, makes the conventional cores inefficient in processing big data. For example, running k-nearest neighbors algorithm (classification) requires calculating the distance of each data point with all existing data in dataset [4]. To process 1 billion candidate points, one query needs 150 GFLOPs of computation and 500G of data communication [5]. Obviously, this results in significant performance overheads when the data cannot fit in memory.

Processing In-Memory (PIM) is one of the most efficient ways to avoid data movement issue [6], [7]. The goal of PIM is to perform computations on data locally where the memory stores it. Since PIM does not send the all data from memory to processing cores, it can address data movement issue by reducing the amount of data communication between main memory and processing cores. Although several designs tried to implement an efficient PIM structure, these designs (i) are mostly application specific [8], which not many algorithms can benefit from, (ii) support basic operations that not many applications can directly use.

On other hand, popular PIM implementations executing addition and multiplication [6], [9], [10] are usually much slower than CMOS-based cores in terms of computation. The only advantage of addition/multiplication PIMs comes from addressing the data movement issue for large sized data. Therefore, it is not useful to run small sized data on PIMs. Each application consists of different segments, with each segment using different types of operations and/or data. Hence, an efficient design needs to run only a certain part of the application on PIM and while the rest still runs on general purpose cores.

To generalize the application of PIM, we design a heterogeneous architecture, called GenPIM, consisting of general purpose processor and PIM accelerator. Design supports basic PIM functionalities in specialized non-volatile memory including: bitwise operations, search operation, addition and multiplication. For each application, GenPIM identifies the parts with large numbers of continuous PIM operations, while the rest of non-PIM operations or non data-intensive parts of applications are processed in general purpose cores. GenPIM also enables configurable PIM approximation by relaxing in-memory computation. We test the efficiency of proposed design over two machine learning algorithms. Our experimental evaluation shows that our design can achieve 10.9× improvement in energy efficiency and 6.4× speedup as compared to processing data in conventional cores.

II. Related work

Processing in-memory (PIM) architectures aim to address data movement issue of the convectional cores. Prior work exploits PIM functionality to design application specific accelerators [8], [11]. Work in used non-volatile memories to enhance CPU [12] or GPU architectures [13], [14], [15], [16]. For example, work in [17] used analog characteristic of non-volatile memories (NVMs) in order to accelerate neural network applications [18], [19]. Similarly, work in [3] used PIM to accelerate graph processing applications. Although, these design can achieve high efficiency, not many applications can benefit from them. In contrast, in this paper we propose a general purpose PIM accelerator framework which can accelerate a wide range of applications.

Author in [7] proposed a PIM architecture which could support basic bitwise and search operations in-memory. Although the designs reported 2-3 orders of magnitude of speedup, these

978-3-9819263-0-9/DATE18/©2018 EDAA

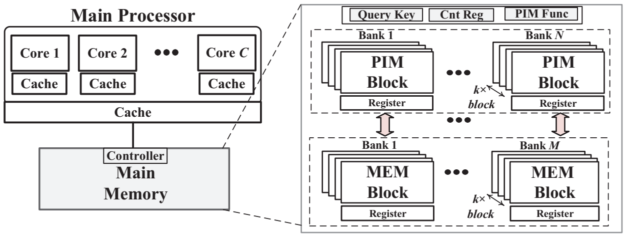


Fig. 1. Architecture overview of the proposed GenPIM.

operations are not commonly used in big data applications. For example, in graph processing less than 1% of graph processing application involve bitwise operations [7]. Work in [6], [9], [10] enabled PIM operations which support more popular operations such as addition and multiplication. However, the PIM operations in these work are much slower than traditional CMOS-based logic. Therefore, these PIM architectures cannot be used to accelerate general applications. In contrast, we propose a novel generalized PIM architecture which schedules different parts of each application to process on either PIM or general purpose core.

III. GenPIM: A Generalized PIM

The memory hierarchy in traditional computers have been designed to provide the maximum data locality for the processing cores. Caches are placed close to the processing cores to store the data which would probably be accessed in near future. However, in the domain of big data, the small cache memories do not have the capability to store this huge amount of data locally. This significantly degrades the efficiency of traditional computers. In addition, the limited memory bandwidth between DRAM and on-chip memories significantly slows down the computation when the size of data increases beyond the memory capacity. Processing in-memory (PIM) is an efficient way to address data movement issue. PIM processes data locally in memory, i.e. the place where it is stored. Therefore, our design avoids the cost of moving entire data to processing cores.

Although, the idea of PIM is general, there are several issues that limit the efficiency of these emerging PIM architectures. In this section, we talk about these limitations and explain how they can be addressed by the proposed design.

In order to design an efficient PIM architecture, PIM needs to support highly used operations in each program. For example, addition, multiplication and search operations are commonly used operations in many applications. In contrast, the operations such as exponential are not common operations, thus PIM does not need to support such less popular functionalities. As PIMs are usually realized by modification in memory sense amplifier, (i) PIMs cannot support many processing functionalities. (ii) Enabling more number of PIM operations significantly increases the memory cost in terms of area, energy and performance.

A. GenPIM Architecture Overview

In order to generalize PIM functionality, our design uses PIM along with a general purpose processor, e.g. CPU. Figure 1 shows the overview of the proposed GenPIM architecture consisting of conventional processing cores and PIM accelerators. In GenPIM, conventional cores are connected to the main memory, which has a PIM functionality. Our GenPIM replaces DRAM with non-volatile memory (NVM) since NVM can support both memory and processing functionalities. In memory functionality, PIM works similar to DRAM, but with lower efficiency. A memory controller accesses the NVM data and sends it through the data bus to caches in processing cores. During PIM functionality, GenPIM applies general PIM operations on the stored data in memory. Our PIM memory is divided into two blocks, a memory and a processing block. Processing block can either store the data or apply general PIM functionalities over the stored values. When the data is stored in memory block, GenPIM moves the data to processing block in order to apply PIM operations. Note that this internal data movement is performed with very low latency. Since NVMs are slower than DRAM, we expect to have lower performance efficiency as compared to using DRAM as main memory. However, this overhead is minor for big data applications, considering the advantage that PIM operations can provide.

TABLE I
Execution time of arithmetic functions using CMOS-based logic and different PIM architectures.

32-bit Addition			32-bit Multiplication	
CMOS Logic	**MAGIC** [10] & **APIM** [6]	**CRS [9]**	**CMOS Logic**	**APIM [6]**
4.8ns	458.0ns	74.8ns	24.6ns	1090.3ns

B. Functions Supported by GenPIM

These are the general operations that GenPIM supports:

Addition/Multiplication: These operations are the most important and popular PIM functionalities which many big data applications can benefit from. For example, emerging big data applications such as neural network or security algorithms are based on large sized matrix multiplication. PIM supports these operations, which enables us to significantly accelerate these operations in-memory. In this work, we used addition and multiplication proposed in [6] to enable PIM functionality for general purpose applications.

Search: The search operation is another key operation in several big data applications including: graph processing, machine learning and query processing. PIM can support exact search as well as can search for data with nearest distance to a value [7]. Examples of such applications include Breadth First Search (BFS) or Single Source Shortest Path (SSSP) in graph processing or nearest neighbor search in machine learning algorithms such as *k*-means.

Bitwise: Bitwise computation is another popular set of operations in applications such as graph or query processing. PIM supports bitwise computation over 8 rows for AND operation and up to 256 rows for OR operation at the same time [7].

C. GenPIM Data Management

In all cores, the computation efficiency is determined by two terms: processing and data movement costs. Running big data applications makes data movement a dominant factor in computation cost, while the processing part is fast and efficient. For example, graph processing workloads consist of millions of vertexes. When running popular BFS and SSSP applications over graphs, the cost of processing is minor as compared to the data movement cost, since it requires the system to bring whole data to the caches. This indicates that PIM may not

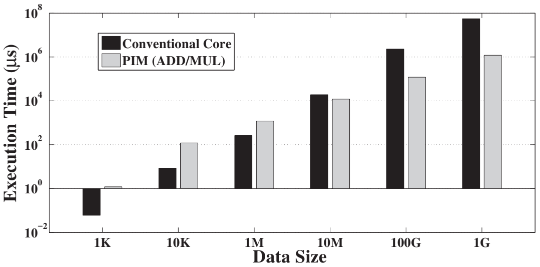


Fig. 2. Execution time of conventional core and PIM addition/multiplication in different data sizes.

always outperform CMOS-based processing speed/efficiency (e.g. CPU cores). Instead, it should be able to address the data movement issue as much as possible. Looking at recent prior work in this area, we observe that PIM processing speed for addition and multiplication is significantly slower than CMOS-based cores [6], [9], [10]. Table I compares the performance of 32-bit addition and multiplication over PIM and CMOS-based logic. The result shows that over addition and multiplication, the CMOS-based logic can achieve at least 15.5× and 44.3× higher performance as compared to PIM processing. However, as the data size increases, PIM becomes more efficient. This makes PIM a suitable choice to process data intensive applications, which mostly suffer from data movement.

GenPIM is supported by a data management unit, which decides whether to run a part of data on PIM or general purpose processor. GenPIM categorizes the application's operations to PIM compatible and incompatible operations. As we explained, GenPIM supports popular operations in memory. The part of application which is not supported by PIM, still needs to be processed on traditional core.

Looking at PIM operations, all operations cannot be accelerated by PIM. In particular, we observe from Table I that GenPIM is much slower than CMOS-logic in terms of processing performance. Therefore, PIM addition/multiplication is not efficient to be processed on (i) small data sizes since such data easily fits into caches of conventional core, (ii) non-continuous PIM operations which require frequent access to traditional cores. Let us consider the performance of data processing over small neural network. Although, neural network consists of several multiplications in each layer, the data can fit on cache for a small network. In addition, after each PIM multiplication/addition in neural network layer, PIM requires to access traditional cores to apply activation function over each neuron output. In this case, traditional cores show significantly higher performance than PIM to process data. PIM operations are beneficial when they need to be applied over large amount of data stored in-memory.

To show the impact of data size on PIM, we generate some random data and apply PIM addition/multiplication over it. Figure 2 compares the performance of PIM and traditional cores when processing data of different sizes. The result shows that PIM multiplication and addition can be beneficial when the size of data surpass 10MB. While working on smaller data size, the CMOS-based logic always outperforms the PIM operations.

Based on this observation, the programmer needs to annotate the code once, identifying the part of application which can be processed by PIM. Based on this annotation, the compiler can optimize different parts of code depending on the characteristics of PIM and conventional cores. Note that this data management is necessary only over addition/multiplication, since the rest of GenPIM operations, i.e., search and bitwise, have lower latency than CMOS-based logic. Therefore, they don't need such code annotation.

TABLE II
NEURAL NETWORK CONFIGURATION OVER CIFAR-10 DATASET.

Network Configurations	Accuracy
$Conv: 32\times32\times3, Conv: 32\times3\times3$, $Pooling: 2\times2, Conv: 64\times3\times3$, $Conv: 64\times3\times3$, Fully connected: 512, 1024 1024, 10	87.7%

IV. Experimental Results

A. Experimental Setup

To simulate the functionality of the proposed GenPIM, we wrote a Python-based cycle-accurate simulator which models the hardware functionality of the proposed GenPIM. We integrate this simulator with GEM5 [20], a cycle-accurate CPU-GPU simulator to completely simulate the functionality of GenPIM. We compare the efficiency of the proposed design with Intel i7 7600 CPU with 16GB memory which is placed next to AMD Radeon R9 390 GPU with 8GB memory. For the measurement of the system and processor power, we used Hioki 3334 power meter and AMD CodeXL [21]. The efficiency of PIM in the proposed GenPIM is evaluated using the HSPICE simulator. We use VTEAM memristor model [22] for our memory design simulation with R_{ON} and R_{OFF} of $10k\Omega$ and $10M\Omega$ respectively.

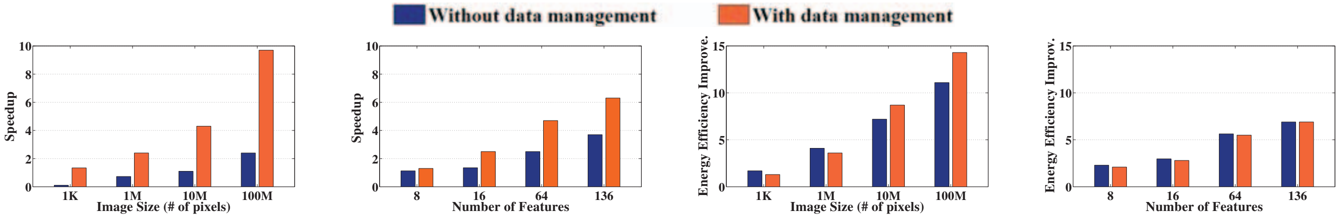
We test the efficiency of the proposed GenPIM over two machine learning applications. Here are the details of the tested applications:

Neural Network: We apply neural network on CIFAR-10 dataset which includes 50000 training and 10000 testing images belonging to 10 classes [23]. The goal is to classify an input image to the correct category, e.g., animals, airplane, automobile, ship, truck, etc. Table II shows the configuration and the baseline accuracy of the neural network over CIFAR-10 dataset. This network consists of four convolutional, one pooling and four fully connected layers.

K-nearest neighbor (*k*-NN): It is a popular machine learning classification algorithm. *k*-NN works based on similarity search, thus includes several nearest distance search operations. The original algorithm uses Euclidean distance as accuracy metric [4], but we change this metric to absolute distance, to make it appropriate for underlying hardware. We test the efficiency of the proposed design on physical activity monitoring dataset, PAMAP2 [24]. This dataset includes logs of 8 users and three 3D accelerometers positioned on arm, chest and ankle. They were collected over different human activities such as lying, walking and ascending stairs. In total, we extracted 16-136 features for the three accelerates, and then further applied the principal component analysis (PCA) to select most significant features with 0.1% of variance.

B. Data size

Figure 3 shows the impact of data size on the efficiency of the proposed PIM-based architecture. For each dataset, the x-axis shows a parameter for each application which changes the



(a) Neural Network (b) *k*-Nearest Neighbor (c) Neural Network (d) *k*-Nearest Neighbor

Fig. 3. Speedup and energy efficiency improvement of proposed GenPIM as compared to traditional cores.

number of computations. For neural network, this parameter is image size. We change the number of input features for *k*-NN. The results show the speedup and energy efficiency improvement of the GenPIM as compared to conventional cores with both systems running the same applications. The results of energy and performance have been reported for two cases. The first bar in the figure corresponds to the proposed GenPIM architecture which assigns all PIM-compatible operations to PIM to process and the second bar shows our adaptive GenPIM which assigns a job to main processor or PIM depending on the size of data. All results are normalized to the energy consumption and execution time of CPU core. Our evaluation shows that over applications with small data size, conventional cores outperform the GenPIM without data management. It happens because while running applications with small data size, the on-chip caches can provide proper data locality, which results in high performance on conventional architectures. However, using GenPIM supported by data management technique, it automatically assigns most of the PIM-compatible operations to traditional cores if the data does not satisfy the GenPIM policy.

Increasing the data size, we observe that GenPIM can achieve significant speedup and energy efficiency improvement as compared to GPU with or without data movement policy. This efficiency comes from the ability of PIM to process huge amount of data. Our evaluation shows that GenPIM can achieve 2.3× and 6.4× speedup as compared to GenPIM without and with data management policy.

In terms of energy efficiency, we observe that the proposed design always outperforms the conventional cores with or without using data management policy. This is because PIM operations are always more efficient than conventional operations on CMOS-based logic. Our evaluation shows that PIM using proposed data management policy can provide 10.9× energy efficiency in average over all tested learing applications. PIM using no data management policy provides lower energy efficiency, i.e. 9.1×. This happens because PIM in this mode degrades the cache locality and increases data movement between the processor and main memory as compared to PIM using data management policy.

V. Conclusion

In this paper we designed a generalized processing in-memory architecture, called GenPIM, consisting of a general purpose processor and PIM accelerator. GenPIM supports basic PIM functionalities in specialized non-volatile memory, including: bitwise operations, search operation, addition and multiplication. For each application, GenPIM identifies the parts with large number of continuous PIM operations and processes the rest of non-PIM operations or non-data intensive part of applications in general purpose cores. GenPIM also enables configurable PIM approximation by relaxing in-memory computation. Our experimental evaluation shows that our design can achieve 10.9× energy efficiency improvement and 6.4× speedup as compared to processing data in conventional cores.

VI. Acknowledgment

This work was supported by NSF grants 1730158 and 1527034.

References

[1] S. Poslad, *Ubiquitous computing: smart devices, environments and interactions*. John Wiley & Sons, 2011.

[2] I. H. Witten *et al.*, *Data Mining: Practical machine learning tools and techniques*. Morgan Kaufmann, 2016.

[3] J. Ahn *et al.*, "A scalable processing-in-memory accelerator for parallel graph processing," in *ACM/IEEE ISCA*, pp. 105–117, IEEE, 2015.

[4] K. Q. Weinberger *et al.*, "Distance metric learning for large margin nearest neighbor classification," in *NIPS*, pp. 1473–1480, 2006.

[5] Y. Deng *et al.*, "Content-based search of video using color, texture, and motion," in *IEEE ICIP*, vol. 2, pp. 534–537, IEEE, 1997.

[6] M. Imani *et al.*, "Ultra-efficient processing in-memory for data intensive applications," in *ACM/IEEE DAC*, p. 6, ACM, 2017.

[7] M. Imani *et al.*, "Mpim: Multi-purpose in-memory processing using configurable resistive memory," in *ASP-DAC*, pp. 757–763, IEEE, 2017.

[8] M. Imani *et al.*, "Nngine: Ultra-efficient nearest neighbor accelerator based on in-memory computing,"

[9] A. Siemon *et al.*, "A complementary resistive switch-based crossbar array adder," *IEEE JETCAS*, vol. 5, no. 1, pp. 64–74, 2015.

[10] N. Talati *et al.*, "Logic design within memristive memories using memristor-aided logic (magic)," *IEEE TNANO*, vol. 15, no. 4, pp. 635–650, 2016.

[11] M. Imani *et al.*, "Efficient query processing in crossbar memory," in *ISLPED*, pp. 1–6, IEEE, 2017.

[12] M. Imani *et al.*, "Resistive cam acceleration for tunable approximate computing," *TETC*, 2016.

[13] M. Imani *et al.*, "Masc: Ultra-low energy multiple-access single-charge tcam for approximate computing," in *DATE*, pp. 373–378, IEEE, 2016.

[14] M. Imani, A. Rahimi, and T. S. Rosing, "Resistive configurable associative memory for approximate computing," in *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2016*, pp. 1327–1332, IEEE, 2016.

[15] M. S. Razlighi *et al.*, "Looknn: Neural network with no multiplication," in *DATE*, pp. 1775–1780, IEEE, 2017.

[16] M. Imani *et al.*, "Approximate computing using multiple-access single-charge associative memory," *TETC*, 2016.

[17] A. Shafiee *et al.*, "Isaac: A convolutional neural network accelerator with in-situ analog arithmetic in crossbars," in *IEEE ISCA*, pp. 14–26, IEEE Press, 2016.

[18] S. M. Seyedzadeh *et al.*, "Pres: Pseudo-random encoding scheme to increase the bit flip reduction in the memory," in *DAC*, pp. 1–6, IEEE, 2015.

[19] Y. Kim *et al.*, "Cause: critical application usage-aware memory system using non-volatile memory for mobile devices," in *ICCAD*, pp. 690–696, IEEE, 2015.

[20] N. Binkert *et al.*, "The gem5 simulator," *ACM SIGARCH Computer Architecture News*, vol. 39, no. 2, pp. 1–7, 2011.

[21] "AMD CodeXL." AMD Inc., http://developer.amd.com/tools-and-sdks/opencl-zone/codexl/.

[22] S. Kvatinsky *et al.*, "Vteam: A general model for voltage-controlled memristors," *IEEE TCAS II*, vol. 62, no. 8, pp. 786–790, 2015.

[23] A. Krizhevsky *et al.*, "Convolutional deep belief networks on cifar-10," *Unpublished manuscript*, vol. 40, 2010.

[24] A. Reiss *et al.*, "Creating and benchmarking a new dataset for physical activity monitoring," in *PETRA*, p. 40, ACM, 2012.