UC San Diego
UC San Diego Electronic Theses and Dissertations

Title
Processing in Memory using Emerging Memory Technologies

Permalink
https://escholarship.org/uc/item/95z3z84c

Author
Gupta, Saransh

Publication Date
2018

Peer reviewed|Thesis/dissertation

eScholarship.org Powered by the California Digital Library
University of California

UNIVERSITY OF CALIFORNIA SAN DIEGO

Processing in Memory using Emerging Memory Technologies

A Thesis submitted in partial satisfaction of the
requirements for the degree of Master of Science

in

Electrical and Computer Engineering (Electronic Circuits and Systems)

by

Saransh Gupta

Committee in charge:

Professor Tajana Šimunić Rosing, Chair
Professor Chung-Kuan Cheng
Professor Farinaz Koushanfar

2018

Copyright

Saransh Gupta, 2018

All rights reserved.

The Thesis of Saransh Gupta is approved and is acceptable in quality and form for publication on microfilm and electronically:

Chair

University of California San Diego

2018

ACKNOWLEDGEMENTS

I thank my advisor, Professor Tajana S. Rosing for her support throughout the course of my degree. I thank her for her guidance which helped me in solving numerous problems and made this thesis possible. Her enthusiasm and willingness to explore new domains have always encouraged me to look at a problem from different perspectives. I am grateful to her for giving me the opportunity to do research with her. I would also like to thank my thesis committee members, Professors Chung-Kuan Cheng and Farinaz Koushanfar, for their time to review my research and their insightful comments. Finally, I would like to thank my current and former labmates (Mohsen Imani, Yeseong Kim, Joonseop Sim, Michael Ostertag, Daniel Peroni, Minxuan Zhao, Sahand Salamat, Yunhui Guo, Ricardo Garcia, Samuel Bosch, Harveen Kaur, Sahil Sharma, Christine Chan) for their continuous support.

The material in this thesis is based on the following publications.

Chapter 3, in part, is a reprint of the material as it appears in M. Imani, S. Gupta, T. Rosing, "Ultra-Efficient Processing In-Memory for Data Intensive Applications," *Proc. IEEE/ACM Design Automation Conference*, 2017, pp. 1-6.

Chapter 4, in part, is a reprint of the material as it appears in M. Imani, S. Gupta, T. Rosing, "GenPIM: Generalized Processing In-Memory to Accelerate Data Intensive Applications," *Proc. IEEE/ACM Design Automation and Test in Europe Conference*, 2018, pp. 1155-1158.

Chapter 5, in part, has been submitted for publication of the material as it may appear in S. Gupta, M. Imani, H. Kaur, T. Rosing, "NNPIM: A Processing In-Memory Architecture for Neural Network Acceleration," *IEEE Transactions on Computers*, 2019.

Chapter 6, in part, is a reprint of the material as it appears in M. Imani, S. Gupta, A. Arredondo, T. Rosing, "Efficient Query Processing in Crossbar Memory," *Proc. IEEE/ACM International Symposium on Low Power Electronics and Design*, 2017, pp. 1-6 and M. Imani, S. Gupta, S. Sharma, T. Rosing, "NVQuery: Efficient Query Processing in Non-Volatile Memory," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2018.

My co-authors (Atl Arredondo, Mohsen Imani, Harveen Kaur, Prof. Tajana S. Rosing,

and Sahil Sharma, listed in alphabetical order) have all kindly approved the inclusion of the aforementioned publications in my thesis.

TABLE OF CONTENTS

Signature Page . . . iii

Acknowledgements . . . iv

Table of Contents . . . vi

List of Figures . . . viii

List of Tables . . . x

Vita . . . xi

Abstract of the Thesis . . . xii

Chapter 1 Introduction . . . 1

Chapter 2 Background and Related Work . . . 5
- 2.1 Memristive Memory . . . 5
- 2.2 Near-Data Computing . . . 7
- 2.3 Processing in-Memory . . . 7
  - 2.3.1 SRAM PIM . . . 7
  - 2.3.2 DRAM PIM . . . 8
  - 2.3.3 NVM PIM . . . 9
- 2.4 PIM Application Examples . . . 10

Chapter 3 PIM for Data Intensive Applications . . . 15
- 3.1 APIM Architecture . . . 15
  - 3.1.1 Fast Addition in APIM . . . 16
  - 3.1.2 Multiplication in APIM . . . 18
- 3.2 APIM Approximation . . . 22
- 3.3 Results . . . 24
  - 3.3.1 Experimental Setup . . . 24
  - 3.3.2 APIM vs state-of-the-art . . . 25
  - 3.3.3 Approximate APIM . . . 26
  - 3.3.4 APIM & Dataset Size . . . 27

Chapter 4 Data Management in a PIM Supported Heterogeneous System . . . 29
- 4.1 GenPIM: A Generalized PIM . . . 29
  - 4.1.1 GenPIM Architecture Overview . . . 31
  - 4.1.2 Functions Supported by GenPIM . . . 31
- 4.2 GenPIM Data Management . . . 32
- 4.3 Results . . . 34
  - 4.3.1 Experimental Setup . . . 34

4.3.2 Data size . . . 36

Chapter 5 Processing In-Memory Architecture for Neural Network Acceleration . . . 38
5.1 Neural Networks . . . 38
5.2 PIM for Neural Networks . . . 39
5.3 NNPIM Design . . . 41
5.3.1 NNPIM Overview . . . 41
5.3.2 Weight Clustering in NNPIM . . . 42
5.3.3 NNPIM Multiplication . . . 43
5.3.4 NNPIM Architecture . . . 47
5.3.5 In-Memory Parallelism . . . 50
5.4 Experimental Results . . . 52
5.4.1 Experimental Setup . . . 52
5.4.2 NNPIM & Weight Sharing . . . 53
5.4.3 Energy-performance Efficiency . . . 55
5.4.4 Area Overhead . . . 56

Chapter 6 Efficient Query Processing in NVM . . . 58
6.1 NVQuery Accelerator . . . 58
6.1.1 Exact Search . . . 60
6.1.2 Nearest Distance Search . . . 61
6.1.3 Join . . . 62
6.1.4 Bit-wise Operations and Addition . . . 63
6.2 Hardware Support . . . 64
6.2.1 Exact Search . . . 65
6.2.2 Nearest Distance Search . . . 66
6.2.3 Bit-wise Operation and Addition . . . 69
6.3 Approximation in NVQuery . . . 70
6.3.1 Bit Trimming . . . 71
6.3.2 Voltage Scaling . . . 72
6.4 Experimental Results . . . 73
6.4.1 Experimental setup . . . 73
6.4.2 NVQuery Efficiency . . . 74
6.4.3 NVQuery & Dataset Size . . . 75
6.4.4 NVQuery Approximation . . . 76
6.4.5 Area Overhead . . . 78

Chapter 7 Conclusion . . . 79

Bibliography . . . 80

LIST OF FIGURES

Figure 1.1. Increase in number of connected devices over the years. 1

Figure 1.2. The cost of various operations in a typical computer organization. Reading data from DRAM is the most energy consuming operation. 2

Figure 2.1. Working mechanism of memristor device when reading and writing. 6

Figure 3.1. (a) The overall structure of APIM consisting of several data and processing blocks, (b) APIM controller and parallel product generator, (c) Fast adder tree structure consisting of carry save adder and configurable interconnects. (d) Final product generator. 17

Figure 3.2. (a) Carry save addition (b) Tree structured addition of 9 n-bit numbers . . . 18

Figure 3.3. The circuit of (a) an interconnect (b) a sense amplifier. 21

Figure 3.4. Error and EDP comparison of the two approximation approaches. 23

Figure 3.5. Performance comparison of the proposed design with previous work for addition of N operands, each sized N bits 25

Figure 3.6. Energy consumption and speedup of exact APIM normalized to GPU vs different dataset sizes. 27

Figure 4.1. Architecture overview of the proposed GenPIM. 30

Figure 4.2. Execution time of conventional core and PIM addition/multiplication in different data sizes. 33

Figure 4.3. Speedup and energy efficiency improvement of proposed GenPIM as compared to traditional cores. 36

Figure 5.1. An example of MNIST classification accuracy during different retraining iterations when the NN weights are represented as eight cluster centers. . . 44

Figure 5.2. Example of Bernstein's Algorithm 45

Figure 5.3. Generating the partial products in latency-optimized NNPIM 46

Figure 5.4. Optimizing NNPIM by reducing the complexity of weights 48

Figure 5.5. Execution time and energy consumption of 32-bit NNPIM multiplication in energy and latency-optimized. 48

Figure 5.6. Architecture overview of the proposed NNPIM. (a) Overall view of neural network implementation in-memory; (b) in-memory implementation of neuron; (c) circuit for configurable interconnect; (d) functions used in NNPIM. 49

Figure 5.7. Operands and control vectors for two parallel NNPIM multiplications. . . . 51

Figure 5.8. Energy consumption and memory size requirement of NNPIM with and without weight sharing. 55

Figure 5.9. Comparing the energy consumption and execution time of NNPIM with state-of-the-art NN accelerators. 56

Figure 6.1. Proposed architecture with N banks and $k \times N$ blocks. The right part details the crossbar implementation of memory banks along with the supporting control logic. 59

Figure 6.2. Circuit level implementation of CAM SA, Memory SA, and Row Driver. . 64

Figure 6.3. Energy consumption and performance of running join operations with different table sizes on traditional cores and the proposed NVQuery. 66

Figure 6.4. NVQuery in nearest distance search configuration. 67

Figure 6.5. Timing characteristic of CAM block in nearest distance search configurations. 67

Figure 6.6. Energy consumption and performance of query processing running on traditional core and the proposed NVQuery. 74

Figure 6.7. Energy consumption and performance of the NVQuery at different approximation levels. 76

Figure 6.8. Area overhead as compared to conventional crossbar memory 78

LIST OF TABLES

Table 3.1. Quality of loss and EDP improvement of the proposed APIM compared to GPU in different level of approximation. 26

Table 4.1. Execution time of arithmetic functions using CMOS-based logic and different PIM architectures. 32

Table 4.2. Neural network configuration over CIFAR-10 dataset. 35

Table 5.1. NN models and baseline error rates for 6 applications (Input layer - *IN*, Fully connected layer - *FC*, Convolution layer - *C*, and Pooling layer - *PL*.) 53

Table 5.2. Quality loss of different NN applications due to weight sharing. 54

Table 6.1. NVQuery supported configurations 60

Table 6.2. NVQuery supported functionality 60

Table 6.3. Approximation in 16-Bit Addition 70

Table 6.4. NVQuery Approximation at Different Supply Voltages 72

Table 6.5. Energy Consumption and Performance Speedup of Queries in NVQuery Normalized to Digital Design over 1k Data 75

Table 6.6. Energy-Delay Product Improvement of SAQ, DAQ and Proposed NVQuery 75

VITA

2016	B.E.(Hons), Electrical and Electronics Engineering, Birla Institute of Technology and Science Pilani, Goa Campus, India
2016-2018	Graduate Student Researcher, University of California, San Diego
2018	M.S. in Electrical and Computer Engineering, University of California, San Diego

PUBLICATIONS

S. Gupta, M. Imani, T. Rosing, “FELIX: Fast and Energy-Efficient Logic in Memory,” *Proc. IEEE/ACM International Conference on Computer Aided Design*, 2018.

M. Imani, R. Garcia, **S. Gupta**, T. Rosing, “RMAC: Runtime Configurable Floating Point Multiplier for Approximate Computing,” *Proc. IEEE/ACM International Symposium on Low Power Electronics and Design*, 2018.

M. Zhou, M. Imani, **S. Gupta**, T. Rosing, “GAS: A Heterogeneous Memory Acceleration for Graph Processing,” *Proc. IEEE/ACM International Symposium on Low Power Electronics and Design*, 2018.

M. Imani, **S. Gupta**, S. Sharma, T. Rosing, “NVQuery: Efficient Query Processing in Non-Volatile Memory,” *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2018.

M. Imani, **S. Gupta**, T. Rosing, “GenPIM: Generalized Processing In-Memory to Accelerate Data Intensive Applications,” *Proc. IEEE/ACM Design Automation and Test in Europe Conference*, 2018, pp. 1155-1158.

M. Imani, **S. Gupta**, A. Arredondo, T. Rosing, “Efficient Query Processing in Crossbar Memory,” *Proc. IEEE/ACM International Symposium on Low Power Electronics and Design*, 2017, pp. 1-6.

M. Imani, **S. Gupta**, T. Rosing, “Ultra-Efficient Processing In-Memory for Data Intensive Applications,” *Proc. IEEE/ACM Design Automation Conference*, 2017, pp. 1-6.

ABSTRACT OF THE THESIS

Processing in Memory using Emerging Memory Technologies

by

Saransh Gupta

Master of Science in Electrical and Computer Engineering (Electronic Circuits and Systems)

University of California San Diego, 2018

Professor Tajana Šimunić Rosing, Chair

Recent years have witnessed a rapid growth in the amount of generated data, owing to the emergence of Internet of Things (IoT). Processing such huge data on traditional computing systems is highly inefficient, mainly due to the limited cache capacity and memory bandwidth. Processing in-memory (PIM) is an emerging paradigm which tries to address this issue. It uses memories as computing units, hence reducing the data transfers between memory and processing cores. However, the application of present PIM techniques is restricted by their limited functionality and inability to process large amounts of data efficiently. In this thesis, we propose novel techniques which exploit the analog properties of emerging memory technologies. Not only do these support more complex functions such as addition, multiplication, and search but

also manage and process large data more efficiently. We present a new blocked PIM architecture which uses inter-block interconnects to accelerate data intensive processing. We also introduce a heterogeneous architecture having general purpose cores and PIM-enabled memory and a data-dependent task allocation scheme for it. We apply application specific optimizations and approximation techniques to further design accelerators for neural networks and database query systems. While we design a multiplication-by-constant hardware for neural networks, query processing is accelerated by a novel in-memory nearest search technique. Our neural network accelerator achieves 113.9× higher energy efficiency and 56.3× speedup as compared to AMD GPU. The query accelerator provides 49.3× performance speedup and 32.9× energy savings as compared to recent Intel CPU.

Chapter 1

Introduction

The Internet of Things is expected to have billions of connected devices that will generate huge amount of raw data. This data amounted to 16.1 zettabytes (ZB) in 2016 and is expected to increase 10$\times$ till 2025 [1]. In recent years, the number of smart electronic devices has surpassed the number of humans in the world [2], as shown in Figure 1.1.

Several algorithms try to pre-process and compress big data [3]. These algorithms are traditionally run on conventional general purpose processors. However, conventional processing architectures have poor performance when processing big data. This inefficiency comes from large amount of data movement between the main memory and processing cores [4]. Figure 1.2 shows that DRAM read operation is the costliest operation in a system with a 32-bit read consuming ~170$\times$ more energy than a 32-bit floating point multiplication [5]. The limited on-chip

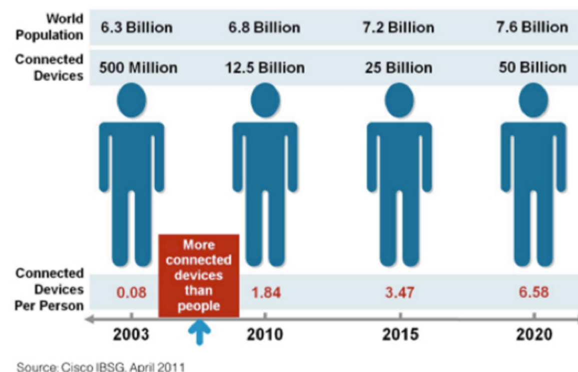
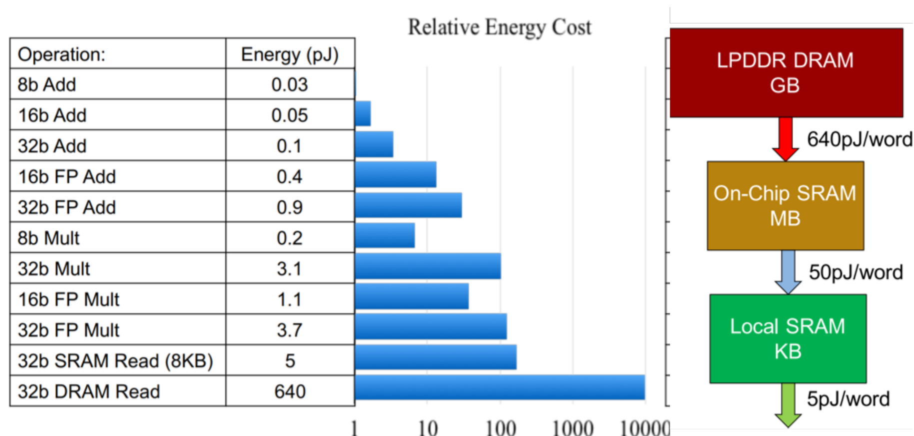


Figure 1.1. Increase in number of connected devices over the years.



Operation:	Energy (pJ)
8b Add	0.03
16b Add	0.05
32b Add	0.1
16b FP Add	0.4
32b FP Add	0.9
8b Mult	0.2
32b Mult	3.1
16b FP Mult	1.1
32b FP Mult	3.7
32b SRAM Read (8KB)	5
32b DRAM Read	640

Figure 1.2. The cost of various operations in a typical computer organization. Reading data from DRAM is the most energy consuming operation.

cache memories in traditional cores along with limited memory bandwidth of main memory, contribute to inefficiency. For example, running k-nearest neighbors algorithm (classification) requires calculating the distance of each data point with all existing points in the dataset [6]. To process 1 billion candidate points, one query needs 150 GFLOPs of computation and 500G of data communication [7]. This results in significant performance overhead when the data cannot fit in memory.

Near data computing (NDC) and processing in-memory (PIM) are two efficient techniques which aim to reduce the cost of data movement [8, 9, 10, 11, 12, 13]. NDC puts the computing units close to the main memory, in order to avoid data movement cost in computation [14]. On the other hand, PIM exploits the analog characteristics of emerging memory technologies to enable in-place computations. The goal of both these techniques is to perform computations on data near/where it is stored. Since they do not send the all data from memory all the way up to the processing cores, they can reduce the amount of data communication and the related high costs. These techniques have challenges of their own. While the application of NDC may be restricted by the difficult integration of memory and logic and the costs of large computing cores, PIM may be limited by the slow device latencies, low endurance, and limited computing capabilities. However, the emerging memory technologies have made it possible to overcome the

challenges faced by PIM and present feasible PIM implementations [15, 16, 17, 18, 19, 20, 21]. Resistive RAM (ReRAM) is one such memory, which enjoys the benefit of low energy, high switching speeds, high density, and scalability. Many solutions have been proposed which utilize ReRAM to realize efficient PIM architectures [8, 10, 22]. Some have proposed general purpose PIM implementations with limited capabilities, while others propose application-specific PIM accelerators with more advanced functionality. For example, the work in [23] only supports NOR directly in crossbar memory, while [24, 25] implements implication in memory and [26] implements majority operation. All other functions are implemented by repeated multiple cycles of the base operation. There is no exploration about how these can be used at application level and the related design constraints. On the other hand, some works propose accelerators targeted towards specific applications like neural networks [10, 27], query processing [28], graph processing [29], etc which cannot be generalized for other applications.

The goal of this thesis is to take steps to bridge the gap between these two extremes, i.e. the limited functionality of general PIM techniques and application-specific processors. We first propose a new PIM architecture which accelerates processing of data intensive workloads. It makes multi-input in-memory additions and multiplications faster. We introduce configurable interconnects to reduce the latency of in-memory shift operations and exploit the inherent parallelism in PIM operations. We observe that in-memory acceleration of applications is suitable only when these applications are data intensive. This is due to the slower switching of memory cells as compared to conventional CMOS devices.Hence, we design a heterogeneous architecture which combines a PIM accelerator with a general purpose processor. It includes a data management unit which controls the allocation of data intensive tasks to PIM, while others are executed by the general purpose cores. This allocation prevents PIM operations from becoming a bottleneck for small applications. We also propose two application specific accelerators. The neural network accelerator uses PIM techniques along with operation level optimizations to improve the performance and efficiency of inference tasks. We design a new in-memory multiplication technique customized for inference task in neural networks. It achieves

113.9× higher energy efficiency and 56.3× speedup as compared to AMD GPU. We also design a PIM accelerator for database queries. It uses a novel search technique to enable nearest distance search in-memory. We combine it with traditional content addressable memory (CAM) and PIM functionality to execute a wide range of query operations in memory. The query accelerator provides 49.3× performance speedup and 32.9× energy savings as compared to recent Intel CPU.

The thesis is organized as follows. Chapter 2 presents a background on memristors and near/in-memory processing. Chapter 3 introduces a new PIM architecture for data intensive applications. Chapter 4 proposes a heterogeneous architecture and a data management scheme for such systems. Chapter 5 & 6 present accelerators for neural networks and database queries respectively. The thesis is then concluded in Chapter 7.

Chapter 2

Background and Related Work

2.1 Memristive Memory

Recent years have witnessed the development of emerging non-volatile memory technologies, such as Conductive Bridging RAM (CBRAM), Resistive Random Access Memory (ReRAM or RRAM), Phase Change Memory (PCM), and Spin-Transfer Torque Magnetoresistive RAM (STT-RAM).Several industry leaders have demonstrated large capacity NVM products, including 16 Gb CBRAM [30], 32 Gb RRAM [31], 8 Gb PCM [32], 1 Gb STT-RAM [33], and 128 Gb 3D-Xpoint memory. These memories have been shown capable of both storing and processing data. This versatile nature of these technologies have increased interest in non von-Neumann architectures.

Memristive technologies are non-volatile and compatible with CMOS fabrication process [34]. Memristive devices are expected to have low switching energy and fast switching speed. The read and write times can be as fast as 120 *ps* [35, 36]. The switching energy is as low as 1 *pJ* [36]. The endurance limit of memristors is measured approximately as 10^{10} allowed write operations per cell [37] (except STT-RAM, where 10^{15} is achieved). This limit is likely to increase to 10^{15} [38]. Memristive devices are fabricated between two metals, which act as the top and bottom electrodes of a dielectric material [39]. Hence, memristors can be fabricated in the metal layers as part of a standard CMOS Back End of Line (BEOL) process. Memristive memories generally utilize a crossbar structure, with $4F^2$ density, where F is the feature size.

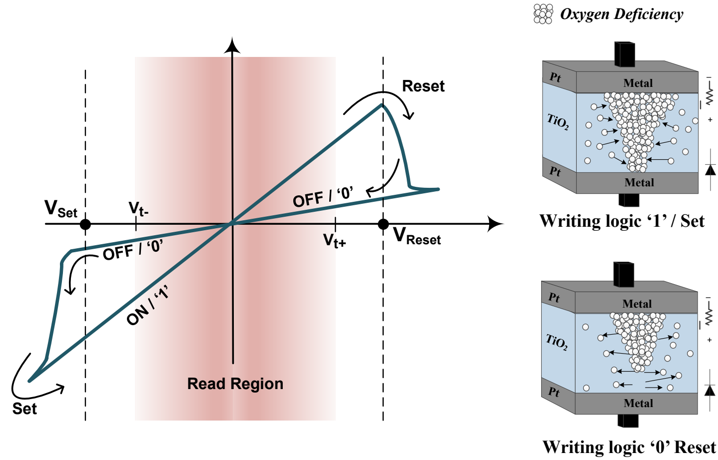


Figure 2.1. Working mechanism of memristor device when reading and writing.

Such a memory achieves significantly lower energy and higher scalability, while occupying negligible area unlike traditional memory technologies like SRAM and DRAM [40, 41]. Digital data is represented in terms of its resistance, where LRS (low resistance state, R_{ON}) is logical '1' and HRS (high resistance state, R_{OFF}) is logical '0.'

General structure of memristor is based on metal/oxide/metal. Two metal layers (e.g. Pt) sandwich an oxide layer which is based on Ta, Ti and Hf [40]. The metal/diode connection usually shows the Ohmic behavior (shown in Figure 2.1). The data is pre-stored based on the memristor resistance state. Figure 2.1 shows the functionality of the memristor device in low and high resistance states. The value is stored on a memristor device based on its hysteresis mechanism. By applying a large positive voltage across the device ($> V_{t+}$), the device changes its state from an LRS state to HRS, which is called *reset*. When the device is in the HRS, by applying a large negative voltage ($> |V_{t-}|$), it changes to the LRS. In order to read the memristor values, we apply a voltage that is less than a write threshold voltage across the device and sense the current passing through the device. In this way, we can identify the state of the device, without changing the memristor value [42].

2.2 Near-Data Computing

Near data computing (NDC) aims to address the issue of data movement by physically placing computing units closer to the memory chip [43, 44, 45, 46, 14, 47]. It leverages the high local data bandwidth to accelerate applications. These computing units may range from small ASICs, FPGAs all the way to GPUs and multi-core systems. These ”cores” are put close to memory to minimize the overhead of data transfers. Several works have proposed adding cores at different levels in the memory hierarchy. While many researchers propose to add computing capability to commodity DRAMs [48, 49, 50, 51], some also use SSDs [52, 53] and other emerging memory technologies [54, 55, 56, 57] to enable NDC.

Recent advances in die-stacking technology have been exploited by researchers to reduce the cost of NDC, improving its practicality [58, 59]. Moreover, introduction of new memory architectures like High Bandwidth Memory (HBM) [60] and Hybrid Memory Cube (HMC) [61] have resulted in an increased interest in using 3D-DRAMs for NDC. The through-silicon-vias (TSVs) optimized for 3D architectures enable stacking of multiple DRAM dies on top of a CMOS logic layer, customized to the requirements of targeted applications. The high memory-level parallelism in memories like HMC provides better random access performance than traditional DRAM [62]. In addition, the logic layer in HMC also allows memory operations like read-modify-write, locking, etc., paving way for accelerating operations in memory [49, 51, 63].

2.3 Processing in-Memory

Processing in-memory (PIM) accelerates computation by reducing the overhead of data movement and providing high parallelism in some cases [64, 65].

2.3.1 SRAM PIM

Some researches present PIM like techniques for SRAMs. The work in [66] proposes the concept of compute memory, where computation is deeply embedded into SRAM. It en-

ables multi-row read access and analog signal processing. Based on this compute memory, the authors in [67] proposes energy-efficient and high throughput implementation of sparse distributed memory which is used as an associative processor. The work in [68] introduces a new content addressable memory (CAM) based on the conventional 6T SRAM cells. They design a configurable SRAM with both CAM and logic functions. Further, the design in [69] presents approaches to improve the performance and energy efficiency of this configurable SRAM, called compute cache. Such caches are able to execute only simple bitwise functions on the data stored in SRAM cells using sense amplifiers. The work in [70] accelerates machine learning by doing computations in SRAM cell. However, it uses large DACs to implement it, introducing significant power and area overhead.

2.3.2 DRAM PIM

The work in [71] exploits the concept of 3D stacking to separate the logic and memory circuits into different DRAM dies, overcoming the cost challenges involved in integrating memory and logic. There have been other proposals to enable computing capabilities inside DRAM [72, 73, 74, 75, 76]. DRAM is inherently destructive in read operations, that is, the stored bits are invalidated after read operations. Thus, the original data should be backed up to another cell before any computations, causing undesired overhead in PIM operations [74]. The work in [76, 73, 72] enable processing by modifying the sense amplifiers (SAs) as well as the 1-transistor-1-capacitor (1T1C) cell structures. These approaches result in additional area overhead to the conventional design. The In-Memory Intelligence (IMI) architecture proposed in [75] is a standard DRAM in form and function with the ability for massive SIMD parallelism and a standard and familiar programming model. It attaches simple bit-serial computing elements to DRAM array's sense amplifiers. These elements are based on standard DRAM structures, for example, slow, dense transistors, and a drastically limited number of metal layers.

2.3.3 NVM PIM

High density, low-power consumption, and CMOS-compatibility of emerging non-volatile memories (NVMs), in particular memristor devices, make them appropriate candidates for both storage and computing purposes [54, 77, 78]. A class of PIM techniques in memristors utilize the inherent dot-product capability of the crossbar structure to implement logic [79, 80, 81, 82]. The architectures employing such techniques use various peripheral circuits to implement different functions. The other class of PIM techniques use the analog properties of memristive devices to realize logic in-memory. Many logic families have been proposed for computation inside memristive crossbar. Some of these logic implementations such as stateful implication logic [24, 25] and Memristor Aided loGIC (MAGIC) [23] are purely realized within memory. The work in [83] extends the bitwise operations in [23] to present schemes for addition in memristive crossbar memory. They also introduce PIM in transpose crossbar memory, allowing more flexibility in logic execution. On the other hand, Pinatubo [84] and MPIM [85] modified sensing circuits to implement fundamental bitwise operations, such as AND and OR. However, they can not support arithmetic operations, e.g. addition and multiplication, which are the key functions involved in many applications such as deep learning algorithms and image processing. The direct application of these schemes in data intensive applications such as DNNs is highly limited due to the linear dependency of execution latency on the size of the data. The work in [86] presents a very fast adder based on complementary resistive switches (CRS). However, CRS-based logic involves reading and sensing the intermediate data during execution, which makes execution dependent on inputs. Moreover, the area overhead involved in arrayed addition grows significantly for data intensive workloads.

Of the proposed memristor-based PIM techniques, `MAGIC NOR` [23] is the simplest to implement, with its execution being independent 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 [83] extends this idea to implement adder in a crossbar. It executes a pattern of voltages in order to evaluate sum and carry bits of 1-bit full addition (inputs being A,B,C) given by,

$$C_{out} = ((A+B)' + (B+C)' + (C+A)')'. \tag{2.1a}$$

$$S = (((A'+B'+C')' + ((A+B+C)' + C_{out})')')'. \tag{2.1b}$$

Here, S and C_{out} have been related to the inputs (A,B, and C) in terms of `NOR` operations. Here, C_{out} is realized as a series of 4 `NOR` operations while S is obtained by 3 `NOT` operations (evaluation of A',B', and C') followed by 5 `NOR` operations. A `NOT` operation is implemented as a `NOR` operation with 1 input. From this point onwards, a `NOR` operation by default implies a `MAGIC NOR` operation. This design takes $12N+1$ cycles to add two N-bit numbers.

2.4 PIM Application Examples

Neural Networks (NNs): Deep neural networks (NNs) demonstrate superior effectiveness for diverse classification problems, image processing, video segmentation, speech recognition, computer vision and gaming [87, 88, 89, 90]. Although many NN models are implemented on high-performance computing architectures, such as parallelizable GPGPUs, running neural networks on the general purpose processors is still slow, energy hungry, and prohibitively expensive. Attempts have been made to improve NNs' computation cost but the data movement between memory and processing cores remains the main bottleneck for NNs' energy consumption and execution time. Several recent research works reduce this bottleneck by in-memory acceleration.

The work in [91] presents an SRAM-embedded convolution architecture, which does not require reading the weights explicitly from the memory. They implement voltage averaging using ADCs and add local multiply-and-average circuits for computation. The authors in [92] accelerated on-chip training of always-on machine learning classifiers using analog computations

in the periphery of the SRAM bitcell array. The CIM-SRAM in [93] presents a cost-aware solution for DNN AI edge processors which is designed by co-optimizing the previously proposed SRAM PIM circuits and the system.

The authors in [94] present a novel process-in-memory architecture to process emerging binary CNN tests in 3D DRAMs. It conducts XNORs inside DRAM arrays, transfers XNOR results by through-silicon-vias (TSVs), and completes popcounts on the logic die. The work in [95] introduces a programmable and scalable digital architecture platform for computing neuro-inspired algorithms. It integrates a parallel compute layer within 3D HMC.

The authors in [10, 27, 96, 97, 98] proposed PIM architectures for implementing NNs in ReRAM. They utilize the dot-product computation inherently supported by ReRAM crossbars to implement matrix vector multiplication in memory. These designs utilize multi-level memristor cells to store data and perform NN computations.

The work in [27] divided a ReRAM bank into three types of subarrays: memory (Mem), buffer, and full function (FF). Mem subarrays only store data whereas FF subarrays can either store data or perform NN computations. They reused write drivers and SAs, with some modifications, to perform the function of DAC and ADC, respectively. Although the sharing of periphery between computation and memory lowers the area overhead, yet the overhead is significantly high. The work in [98] extends the work in [27] to provide support for back-propagation and weight-update, while leveraging peripheral circuitry for inference operation. The sense amplifier implements ReLU and max functions and precision control apart from analog-to-digital conversion.

A ReRAM-based tiled architecture for accelerating NNs was presented in [97]. Each tile in [97] has an MCA-based PE (processing engine) array, an eDRAM buffer, a register buffer and multiple DACs. The input feature maps of CNN are stored in the eDRAM buffer, which is cached by the register buffer for different convolution steps. Every PE has also an ADC, some logic elements and SH units. While the ADC works similar to the other works, this work requires less number of DACs since it uses DACs in only one column, reducing the area and

power overheads.

The CNN accelerator presented in [10] uses ReRAM dot-product computations for convolution and classifier layers. The system has multiple tiles each designed with ReRAM crossbars which store synaptic weights and computations on them. Since a crossbar cannot be efficiently reprogrammed at runtime, it assigns one crossbar for processing a group of neurons in any CNN layer. The strength of their design lies in pipelined architecture where different CNN layers are pipelined, reducing the buffering requirement and increasing the throughput.

The authors in [96] accelerated both training and testing of CNNs using ReRAM crossbar. They divide MCAs into two types: memory and morphable. The morphable MCAs perform both computation and data-storage and memory MCAs only store data. The design exploits both intra-layer and inter-layer parallelism. To exploit intra-layer parallelism, they map the kernels to multiple MCAs and, then, collect and add their outputs. The number of duplicate copies of MCAs storing the same weight presents a trade-off between hardware overhead and throughput. They also propose a pipelined training architecture where inputs inside a batch can be processed in pipelined manner. They eliminated the use of both DACs and ADCs by integrating the weighted spike-codes in a counter.

Query Processing: Data management systems (DMS) are the standard tools for collecting and serving large amounts of information for web applications and end users. Over the past decade, data generation has grown exponentially due the diversity of collection sources [99, 100, 101]. In addition, organizations collect large amounts of information for decision making and business analytics [102, 103, 104]. In the majority of scenarios, the execution time of DMS queries tends to increase linearly and sometimes exponentially as more records are stored in a single server instance. This has been one of the main challenges of DMS and its caused by the the hardware and software co-design limitations [105].

To increase the performance of query processing, the work in [28] presented a ReRAM-based PIM architecture for SQL queries. Their technique utilizes the dot-product computation scheme of ReRAM crossbar to process cells storing identical attribute in different tuples and

different attributes in the same tuple. They mapped data to ReRAM such that a tuple is stored in a ReRAM row and attributes of a tuple are stored in columns of a row. Their technique supports three query operations: restriction (selecting rows that fulfill a criterion), projection (selecting specific columns in a row) and aggregation (summarizing specific properties of multiple columns in a group of rows, e.g., adding the values). The multiplication of attributes happens in CPU, for which data is transferred from memory to CPU. An analog comparator is used for comparison operations whereas to process the output by equality or Boolean functions, an ADC is used at row-output for converting the result to digital domain. In a projection operation, '1' is applied at the column to be read and the remaining columns are supplied with '0' signal. Only sum is supported in aggregation, which is implemented using column-wise dot-product computation.

The design proposed in [106] further proposed a query optimization method to make the optimizer fully utilize the performance of the PIM structure and generate an execution plan specific to the SQL query unit hybrid database system proposed in [28]. They propose some heuristic rules and a cost-based optimizer for such systems. The heuristic rules are based on the PIM characteristics of the RRAM-based SQL unit. These rules are required since the RRAM-based SQL unit supports a subset of typical operations of a database. Hence, the rules try to utilize the RRAM-based SQL unit whenever possible during query processing to reduce system cost. The cost-based optimizer helps the system determine if it should choose an index or PIM to implement a restriction operation.

Graph Processing: Graph processing plays an important role in data processing because most of data collected from the real world can be represented as graphs like social network [107], road network [108], and human brain [109]. Since the current trends suggest an explosive growth of data in near future [110, 111, 112, 113, 114], processing large graphs in an efficient way, therefore, has become significantly important.

Inspired by this, the work in [4] designed a new programmable accelerator for in-memory graph processing that can effectively utilize PIM using 3D-stacked DRAM and provided the programming interface for the same. They introduced novel communication mechanism and

hardware prefetchers to fully utilize the available memory bandwidth. The work in [63] proposes a PIM architecture for graph processing on Hybrid Memory Cube (HMC) array. They integrate SRAM-based on-chip vertex buffers to eliminate local bandwidth degradation. They also introduce reconfigurable double-mesh connection to provide high global bandwidth.

The work in [29] investigates the use of PIM based ReRAM crossbar for graph processing. GraphR [29] is a ReRAM based accelerator which processes graph applications using matrix multiplication capability of ReRAM crossbar memory. It treats ReRAM crossbar as a replacement for conventional processing unit and dynamically transfers data from the memory to a multi-level memory cell based crossbar to finish computations for sub-graphs. Furthermore, GraphR utilizes ReRAM for matrix-vector multiplication, using ADC and DAC devices for conversions at the memory periphery.

Hyperdimensional Computing: Although deep learning algorithms work well for many applications, they are facing scalability and power issues when running on todays computers. This motivate designing brain-inspired hyperdimensional (HD) computing algorithms which have significantly lower power consumption, while providing excellent accuracy. Brain-inspired HD computing explores this idea by looking at computing with ultra-wide words high-dimensional vectors, or hypervectors [115, 116, 117]. Hyperdimensional computing has been shown to support a wide range of applications. For example, the design proposed in [118] enables general speech recognition in high dimensional spaces and then combines them to generate a unique vector for each output class. The work in [119] introduces a language recognition algorithm working with high dimensional vectors. Similarly, [120] uses hyperdimensional computing for DNA sequencing. Researchers have proposed new architectures for hyperdimensional associative memory that can facilitate energy-efficient, fast, and scalable implementation of HD in-memory [121]. These analog HD designs use ReRAM based memory to store vectors and perform HD operations. These designs linearly scale with the number of dimensions in the hypervectors, while providing orders of magnitude higher efficiency.

Chapter 3

PIM for Data Intensive Applications

A number of logic families have been proposed for computation inside memristive crossbar. They focus on logical and arithmetic operations between limited inputs and extend them linearly for large number of inputs. The direct application of these schemes in data intensive processing is limited largely due to the dependency of latency of execution on the size of data. In this thesis, 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. 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 also dynamically tune the level of approximation in order to trade the accuracy of computation while improving energy and performance.

3.1 APIM Architecture

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 3.1(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.

3.1.1 Fast Addition in APIM

The design in [83] 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 3.2(a) shows carry save addition. Here, $S1[n]$ and $C1[n]$ are the sum and

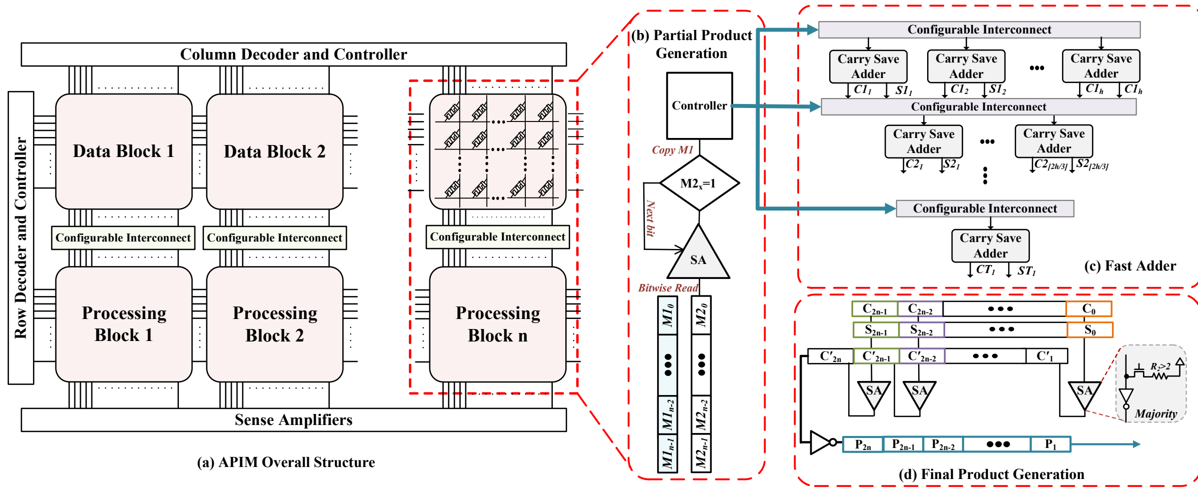


Figure 3.1. (a) The overall structure of APIM consisting of several data and processing blocks, (b) APIM controller and parallel product generator, (c) Fast adder tree structure consisting of carry save adder and configurable interconnects. (d) Final product generator.

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 3.2(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

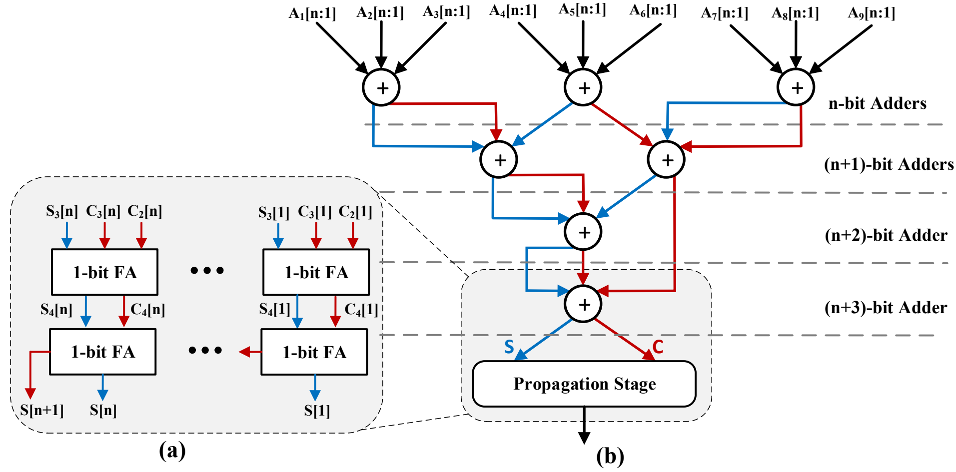


Figure 3.2. (a) Carry save addition (b) Tree structured addition of 9 n-bit numbers

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 3.3.

3.1.2 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 3.1. 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 3.1.1. 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 ANDing 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` operations as given by,

$$F = AND\ (A,B) = NOR\ (NOR(A),\ NOR(B)) \tag{3.1}$$

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 3.1(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.

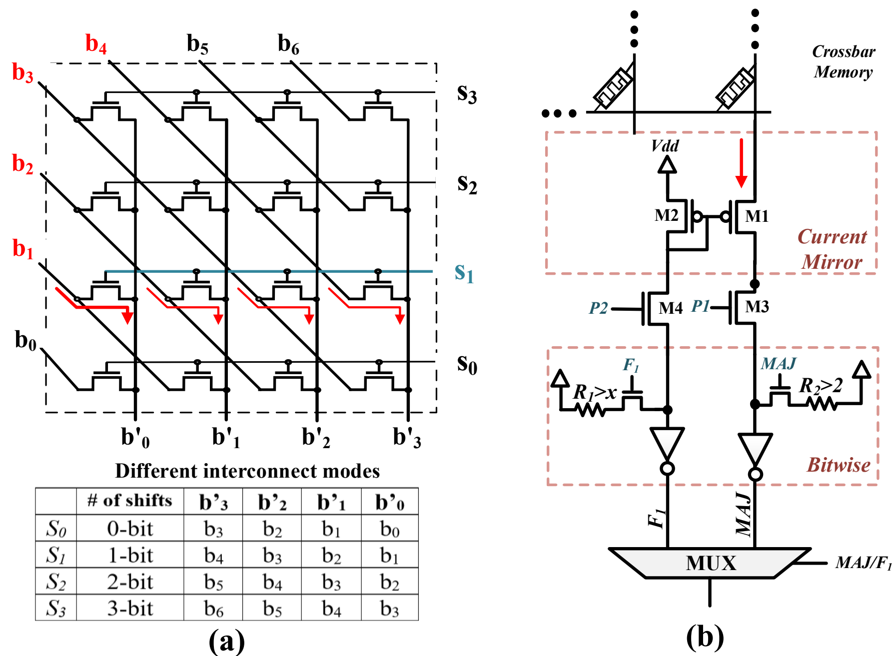
Although this is a huge improvement in latency from the initial design, we have not yet considered the cost of shifting the partial products for add operation. Shift operation in a normal memory crossbar can only be done bitwise, which would be quite large given the number and size of operands to be 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 3.1.1 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 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 utilizing 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 3.3(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



	# 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 3.3. The circuit of (a) an interconnect (b) a sense amplifier.

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.

3.2 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.

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)) [122]. 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

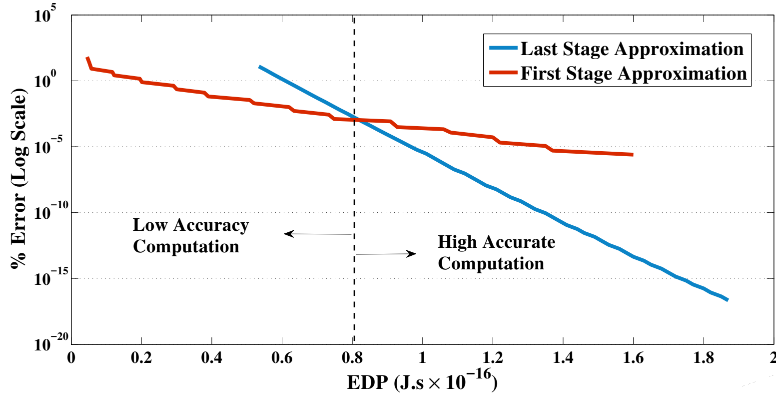


Figure 3.4. Error and EDP comparison of the two approximation approaches.

Figure 3.3(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.

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 3.3.3. 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 3.3.3 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 3.4 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.

3.3 Results

3.3.1 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 [123], 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 [124] for our memory design simulation with R_{ON} and R_{OFF} of $10k\Omega$ and

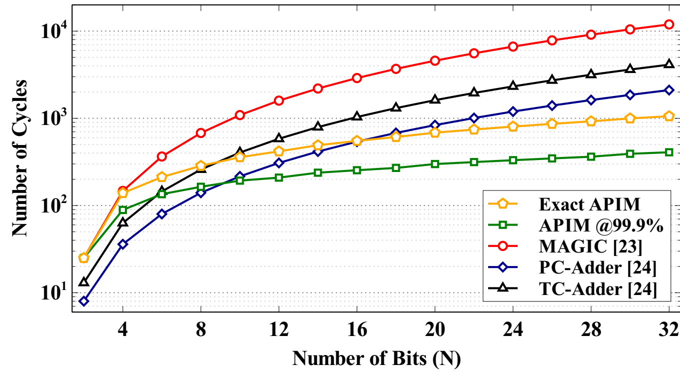


Figure 3.5. Performance comparison of the proposed design with previous work for addition of N operands, each sized N bits

$10M\Omega$ 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* [125] 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.

3.3.2 APIM vs state-of-the-art

Figure 3.5 compares the performance efficiency of the proposed design with the state-of-the-art prior work [83, 86]. The work in [83] computes addition in-memory using MAGIC

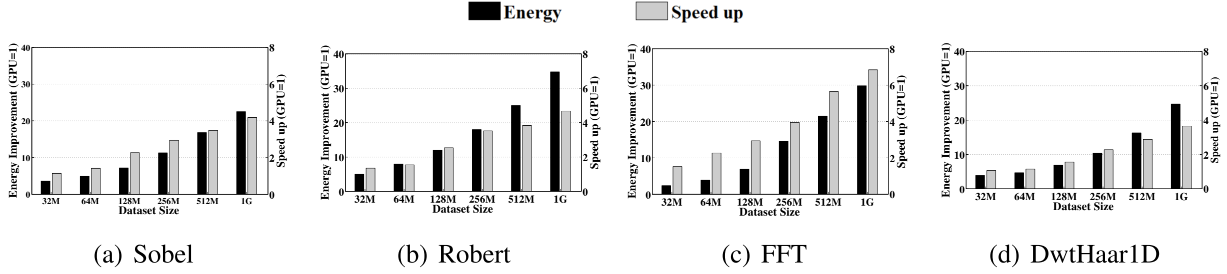
Table 3.1. Quality of loss and EDP improvement of the proposed APIM compared to GPU in different level of approximation.

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%

logic family, while the work in [86] 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 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 [86] 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.

3.3.3 Approximate APIM

Table 3.1 shows the energy consumption and performance of the applications running on APIM in different approximation level. As we explained in Section 3.2, 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 3.1 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



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

Figure 3.6. Energy consumption and speedup of exact APIM normalized to GPU vs different dataset sizes.

480× energy-delay product improvement as compared to APIM in the exact mode.

3.3.4 APIM & Dataset Size

Figure 3.6 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.

This observation shows the need for intelligent allocation of an application or a task to a PIM accelerator. Chapter 4 introduces a hybrid CPU-PIM architecture and then proposes a data size-aware management scheme to decide when to process data in CPU or PIM accelerator.

Chapter 3, in part, is a reprint of the material as it appears in M. Imani, S. Gupta, T. Rosing, "Ultra-Efficient Processing In-Memory for Data Intensive Applications," *Proc. IEEE/ACM Design Automation Conference*, 2017, pp. 1-6.

Chapter 4

Data Management in a PIM Supported Heterogeneous System

PIM implementation executing addition and multiplication introduced in Chapter 3 is much slower than CMOS-based cores in terms of computation. The major advantage of PIM comes from addressing the data movement issue for large data. 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 fraction of the application on PIM and while the rest still runs on general purpose cores. We design a heterogeneous architecture, called GenPIM, consisting of 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 numbers of continuous PIM operations, while the rest of non-PIM operations are processed on general purpose cores.

4.1 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

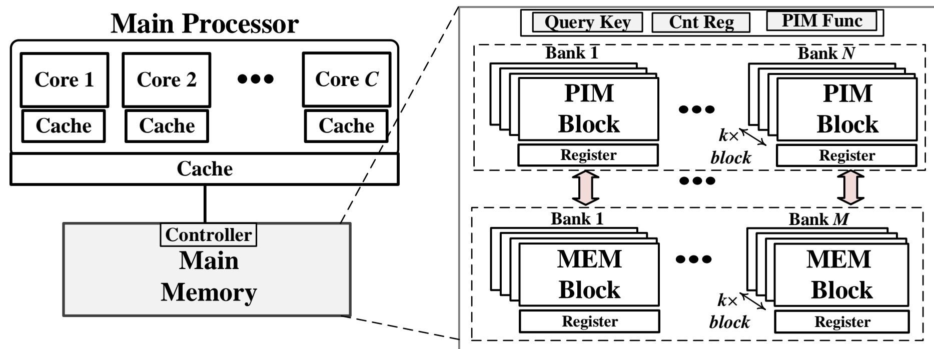


Figure 4.1. Architecture overview of the proposed GenPIM.

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.

4.1.1 GenPIM Architecture Overview

In order to generalize PIM functionality, our design uses PIM along with a general purpose processor, e.g. CPU. Figure 4.1 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.

4.1.2 Functions Supported by GenPIM

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 Chapter 3 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

Table 4.1. Execution time of arithmetic functions using CMOS-based logic and different PIM architectures.

32-bit Addition			32-bit Multiplication	
CMOS Logic	**MAGIC** [83] & **APIM** [126]	**CRS [127]**	**CMOS Logic**	**APIM [126]**
4.8ns	458.0ns	74.8ns	24.6ns	1090.3ns

search as well as can search for data with nearest distance to a value [85]. 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 [85].

4.2 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 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 [127, 83]. Table 4.1 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

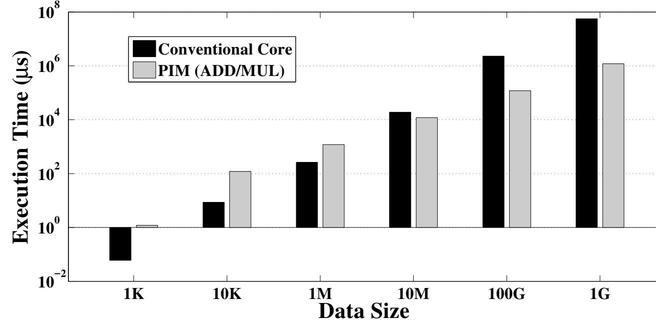


Figure 4.2. Execution time of conventional core and PIM addition/multiplication in different data sizes.

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 4.1 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 4.2 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.

4.3 Results

4.3.1 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 [128], 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 [129]. The efficiency of PIM in the proposed GenPIM is evaluated using the HSPICE simulator. We use VTEAM memristor model [124] 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 [130]. The goal is to classify an input image to the correct category, e.g., animals, airplane, automobile, ship, truck, etc. Table 4.2

Table 4.2. Neural network configuration over CIFAR-10 dataset.

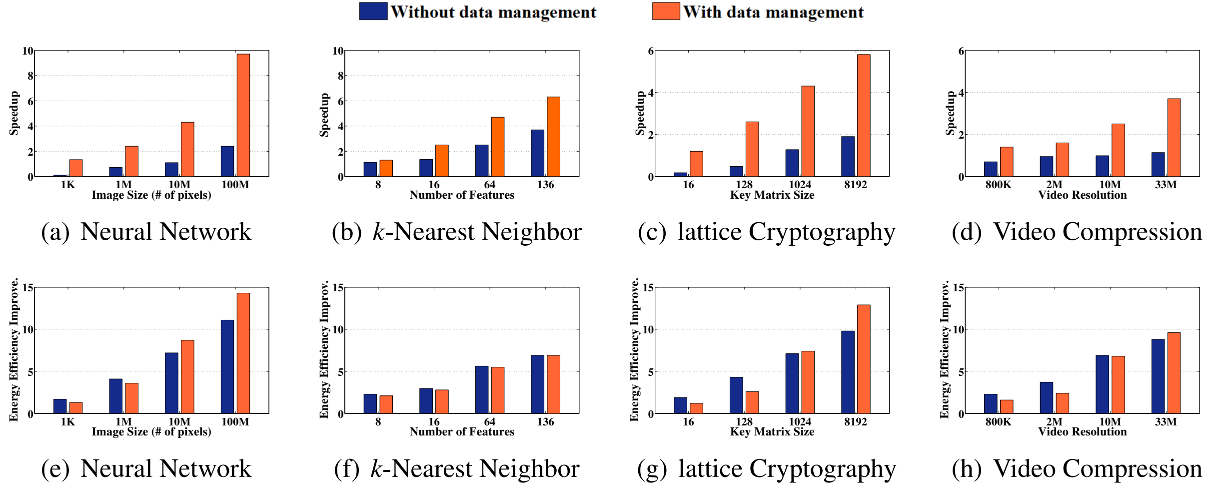
Network Configurations	Accuracy
$Conv: 32\times32\times3, Conv: 32\times3\times3$, $Pooling: 2\times2, Conv: 64\times3\times3$, $Conv: 64\times3\times3$, Fully connected: $512,\ 1024\ \ 1024,\ 10$	87.7%

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 [6], 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 [131]. 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.

Lattice-based Cryptography: In recent years, lattice-based cryptography has been used for strong provable security guarantees and apparent resistance to quantum attacks, flexibility for realizing powerful tools like fully homomorphic encryption, and high asymptotic efficiency [132, 133]. Here we look at the application of lattice-based cryptography in security. The majority of this application consists of matrix multiplications and additions. We change the size of generated matrix in the design from 16 to 8192 in order to explore the efficiency of the GenPIM over this application.

Video Compression: This algorithm involves several matrix multiplication, matrix transpose, and matrix inversion operations. This dataset is composed by 1296 double compressed video sequences, extending the dataset [134]. These sequences are obtained starting from 6 uncompressed well-known sequences (4 at CIF resolution and 2 at 4CIF resolution, measuring from 240 to 300 frames each). To explore the efficiency of the GenPIM over video compression,



(a) Neural Network (b) *k*-Nearest Neighbor (c) lattice Cryptography (d) Video Compression

(e) Neural Network (f) *k*-Nearest Neighbor (g) lattice Cryptography (h) Video Compression

Figure 4.3. Speedup and energy efficiency improvement of proposed GenPIM as compared to traditional cores.

we ran this algorithm over videos with four different qualities: 720pi, 1080pi, 4K and 8K.

4.3.2 Data size

Figure 4.3 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 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.

In recent chapters, we proposed a new PIM supported hybrid architecture for data intensive computations. We also introduced data management solutions to increase the performance of the hybrid system. In the following chapters, we show how these can be used to significantly accelerate neural networks and query processing.

Chapter 4, in part, is a reprint of the material as it appears in M. Imani, S. Gupta, T. Rosing, "GenPIM: Generalized Processing In-Memory to Accelerate Data Intensive Applications," *Proc. IEEE/ACM Design Automation and Test in Europe Conference*, 2018, pp. 1155-1158.

Chapter 5

Processing In-Memory Architecture for Neural Network Acceleration

Chapters 3 and 4 presented a novel architecture to implement data intensive computations efficiently. In this chapter, we show how it can be used to execute neural networks (NNs) efficiently in memory. We propose a novel NN accelerator, called neural network processing in-memory (NNPIM), which significantly reduces the overhead of data movements while supporting all the NN functionalities completely in memory. To realize such computation, our design first analyzes computation flows of a NN model and encodes key NN operations for a specialized PIM-enabled accelerator. The proposed NNPIM supports three layers popularly used for designing a NN model: fully-connected, convolution, and pooling layer. We divide the computation tasks of the networks into four operations, multiplication, addition, activation function, and pooling. Our accelerator supports all of these operations inside a crossbar memory.

5.1 Neural Networks

A NN model consists of multiple layers which have multiple neurons. These layers are stacked on top of each other in a hierarchical formation, so each layer takes the output of previous layer as input and forwards its output to the next layer. In this thesis, we focus on three types of layers that are most commonly utilized in practical neural network designs: (i) convolution layers, (ii) fully connected layers, and (iii) pooling layers. Figure 5.6a depicts one

neuron. In neural network, each neuron takes a vector of inputs from neurons of the preceding layer $X = \langle X_0, \cdots, X_n \rangle$, then computes its output as follows:

$$\varphi(\sum_{i=1}^{n} W_i X_i + b)$$

where W_i and X_i correspond to a weight and an input respectively, b is a bias parameter, and $\varphi(.)$ is a nonlinear activation function. Prior to the execution of NNs, parameters W_i and b are learned in a training process. For inference, the pre-trained parameters are used to compute the outputs of each neuron, called *activation units*. A neuron produces one activation unit based on two main operations, the weighted accumulation, i.e. $\sum W_i X_i$, and the activation function, i.e. $\varphi(.)$. By processing all the computation through the layers, also known as *feed-forward* procedure, it produces multiple outputs which are used for the final prediction.

In inference, neural networks use a combination of convolution, pooling, and fully connected layers to process or classify the data. There are two types of data in neural networks: **(i)** a large number of trained weights, which we call them network model and **(ii)** the input data which is processed by the network. The main computation in neural network involves processing the input data over network using the trained weights. It leads to several computations between weights and inputs.

5.2 PIM for Neural Networks

Processing in-memory supports essential functionalities among different memory rows. These operations should be general enough to benefit many applications. Neural network computation is based on a few basic operations, so executing them in-memory can allow us to run whole application inside a memory. This would reduce data movement issue and accelerate any network locally in memory. The goal of PIM is to locally perform operations between these inputs and weights inside a memory block, such that there is no need to send data up to processor. To support all the required operations in memory, we design a PIM architecture which

can perform addition, multiplication, activation function, and pooling locally in memory. These operations are managed inside a memory using simple controllers.

The memory architecture used in this work supports the following functions on the same hardware:

Addition/Multiplication: Our design can execute the addition of three data values, in memory, by activating their corresponding rows. If more values have to be added at the same time, our design implements addition in a tree structure. The multiplication inside memory is performed in a similar way, by generating all possible partial products and adding them in parallel in memory. Their hardware implementation has been discussed in Section 3.1.

Activation Function: Traditionally, *Sigmoid* function has been used as an activation function [135]. This function is defined as: $S(x) = 1/1+e^{-x}$. Implementing this functionality in memory requires modeling exponential operations. Our design can handle this operation by using the Taylor expansion of the *Sigmoid* function and considering the first few terms to approximate the *Sigmoid* function. The Taylor expansion only consists of addition and multiplication. We can easily implement any function in memory as long as it is representable by Taylor expansion and the more terms we consider in Taylor expansion, the better the model is for activation functions. Prior work showed that it is not necessary to use *Sigmoid* as an activation function. Instead, using simple ”Rectified Linear Unit” clamped at a certain point (e.g. X=a could provide similar or better accuracy than *Sigmoid*. In that case, the activation function can be implemented using a single comparator which checks if input X surpasses a value a. Note that in case of rectified linear unit, activation function can be processed simply inside a controller.

Pooling: Our hardware implements in-memory pooling using nearest search operation. PIM stores the output of convolution layer inside a memory block with nearest search capability. Then to find the maximum value, our design searches for a row with the closest distance (maximum similarity) to inf value. This inf value in hardware is the maximum value which can be represented by hardware. Using this block, we can search for the MAX value among the selected rows inside the memory. Similarly, the MIN pooling can be implemented by searching

for the smallest value in lookup table ($-\inf$).

5.3 NNPIM Design

5.3.1 NNPIM Overview

As described before, an inference task in neural network involves multiplying inputs with the weights, which are calculated during the training phase. Once a network is trained, the weights remain constant and do not change over different inference tasks. The previously proposed hardware designs to accelerate neural networks do not exploit this property of neural networks. In such cases, multiplication with fixed weights is computationally as expensive as that with variable weights.

NNPIM uses this fixed nature of weights to reduce the complexity of in-memory neural network multiplications. Instead of using the weights directly, NNPIM breaks down the weights into simpler factors. These factors are chosen such that multiplying a number with them just requires a shift and add/subtract operation. Hence, instead of exhaustively generating all the partial products and adding them, we rely on the fixed nature of weights to pre-process them and calculate their "multiplication-friendly" factors. All these computations utilize the PIM operations proposed in Section 5.2.

A neural network usually involves a large number of weights. Using this large number of weights restricts the enhancements which in-memory processing can provide. We realize that the memory requirement and energy consumption of NNPIM depend on the number of weights. Hence, we use weight sharing to reduce the number of unique weights in each neuron. Since all the computations in NNPIM happen in-memory, we design NNPIM such that this reduction in weights directly results in a decrease in the number of memory blocks required for computations.

5.3.2 Weight Clustering in NNPIM

The conventional NN requires a large number of multiplications. We leverage shared weights to reduce number of operations, *i.e.* multiple inputs of each neuron share the same value, however, a naive implementation of weight sharing can result in undesirable loss of accuracy. We devise a greedy algorithm to select the near optimized shared weights that reduce the loss of accuracy; instead of applying shared weights to the already trained NN, we train the NN in a way that weight sharing does not impose much loss of accuracy.

The weights of each layer are fixed in the inference phase; in order to share the weights, the clustering algorithm is applied on the fixed weights. Assuming that a fully-connected layer maps N neurons into M outputs, the corresponding matrix $W_{M\times N}$ is clustered once and a single set of weights are generated for the whole matrix. For convolution layers, the weights corresponding to different output channels are clustered separately: a convolution layer mapping N channels into M channels using a weight tensor $W_{h\times h\times N\times M}$ is divided into M different tensors and each tensor is clustered separately, resulting in M different weights.

After clustering, each weight replaces by their closest centroids. The objective of clustering is to minimize the within cluster sum of squares (WCSS):

$$\min_{c_{i1},\dots,c_{iN_{clusters}}} (WCSS = \sum_{k=1}^{N_{clusters}} \sum_{W_{ij}^{l}\in c_{ik}} ||W_{ij}^{l} - c_{ik}||^2) \tag{5.1}$$

where $C = \{c_{i1}, c_{i2}, \dots, c_{iN_{clusters}}\}$ are the cluster centroids. We use K-means algorithm for clustering. Weight clustering essentially finds the best matches that can represent this distribution, and replaces all parameters with their closest centroids. Weight clustering is often accompanied by some degree of additive error, $\Delta e = e_{clustered} - e_{baseline}$.

To compensate for this error, our algorithm retrains the neural network based on the new weight constraints. After each retraining, our design again clusters the weights and estimates the quality of the classification using the new cluster centers. The procedure of weight clustering and

retraining continues until the estimated error becomes smaller than a desired level. Otherwise the retraining procedure stops after a pre-specified number of epochs. Figure 5.1 shows the accuracy of neural network for MNIST dataset during different retraining iterations. The result shows that retraining improves the classification accuracy by finding a suitable clusters for each neuron weights.

One major advantage of weight sharing is that it can significantly reduce the number of required multiplications. Each neuron in neural network multiplies several input data, say n, with pre-stored weights. Therefore, each neuron requires to multiply n input-weight pairs. Using weight clustering, the number of distinct weights in each neuron can be reduced to k, where $k << n$. Instead of multiplying all input-weight pairs, we can simply add all inputs which share the same weight and finally multiply the result of addition with the weight value. This method reduces the number of multiplications in each neuron from n to k. This significantly accelerates the NNPIM computation, since in PIM the multiplication performs much slower than addition. Moreover, our hardware enables fast addition of multiple input vectors in-memory. Hence, the inputs corresponding to the same weight can first be added together using carry save addition. Then, the result can be multiplied with the weight. In other words, multiple multiplications are broken down into a large addition and a multiplication. In this way, we reduce the number of computations required as well as the complexity of operations involved.

5.3.3 NNPIM Multiplication

The multiplier in Section 3.1 performs exhaustive binary multiplication. It generates a partial product for each '1' present in the multiplier and performs addition. Although this approach is general and works for all applications but it can lead to unnecessary latency overheads in certain cases. For example, multiplication by 255 (b11111111) would require generation of 8 partial products, corresponding to each '1', and their subsequent addition. The same operation can executed by multiplying by 256, i.e. shifting by 8 bits, and then subtracting the multiplicand from the obtained result.

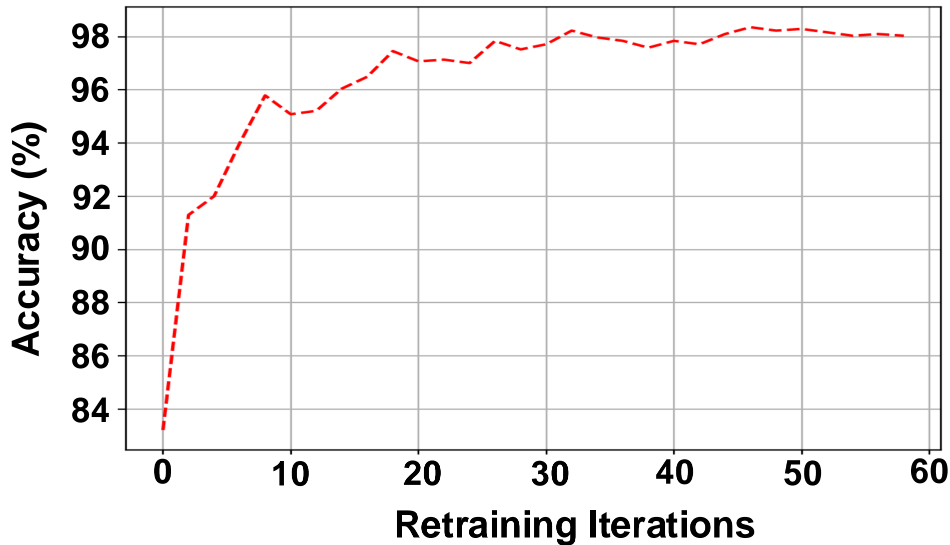


Figure 5.1. An example of MNIST classification accuracy during different retraining iterations when the NN weights are represented as eight cluster centers.

Bernstein algorithm [136] factorizes the constant multiplier into factors which are a power of 2 or a power of 2 ± 1. It uses branch and bound based search pruning and finds the factors based on a formulation for their costs. Figure 5.2 gives an example of how the algorithm can reduce the number of operations. In this case, binary multiplication takes 6 instructions whereas the factor-based multiplication takes only 4 instructions. The binary method is the worst case factorization which can be obtained using the algorithm.

Using this algorithm involves finding suitable factors. It can be time consuming and may add unwanted latency if the operands change frequently. However, such an algorithm can be useful if one of the operands is constant. In that case, the constant operand can be factorized once and these factors can be referenced every time the constant is involved in multiplication. This makes such factorization suitable for neural networks, where the weights are always constant and only the inputs are variable. NNPIM exploits this property by storing the factors of the weights and using these factors for computations. We now discuss two ways in which we use Bernstein algorithm to improve computations in neural networks. One approach aims to minimize the energy consumption of the design while the other approach presents a latency-optimized

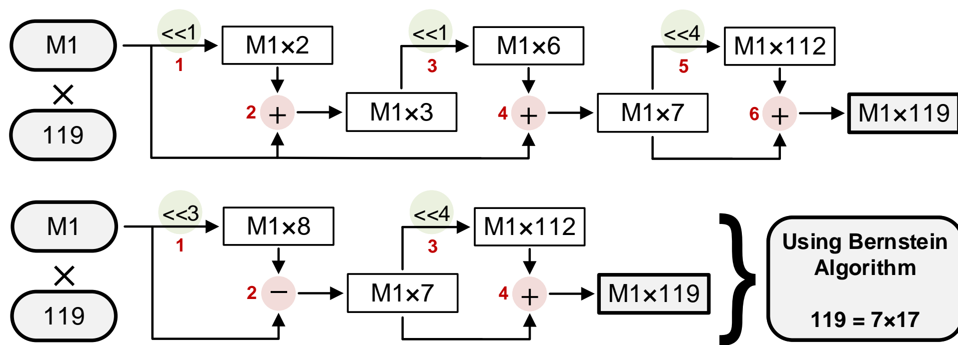


Figure 5.2. Example of Bernstein's Algorithm

technique.

Energy-Optimized NNPIM: The hardware in Section 3.1 utilizes carry save addition to reduce the latency of multiplication. However, in order to minimize the propagation of carry and reduce the latency, it implements a large number of partially redundant parallel operations. This consumes significant amount of energy. A naive energy-efficient design would process all partial products serially, adding two at a time. Such a design is intuitive but does not exploit the constant operands in neural networks. The inference phase of neural networks involves multiplication of many input vectors with weights obtained from the training phase and fixed during inference. This phase is defined by multiplication of variables, i.e. input vectors, with constants, i.e. weights, making it a suitable application for Bernstein algorithm. We can accelerate the testing phase by factorizing the weights and using these factors instead of actual weights for computation. For the example discussed before, binary implementation requires 6 serial shift or add operations, while NNPIM only requires 4 serial shift, add, or subtract operations.

Latency-Optimized NNPIM: The above approach based on Bernstein algorithm is perfect when the total energy consumption of the design is the major concern. Bernstein algorithm reduces the number of operations required but does not necessarily accelerate the overall in-memory processing. In carry save addition, carry is propagated only in the end to minimize the time taken to compute the final product. Breaking the weights into smaller factors requires the computation of multiple intermediate products to achieve the final output.

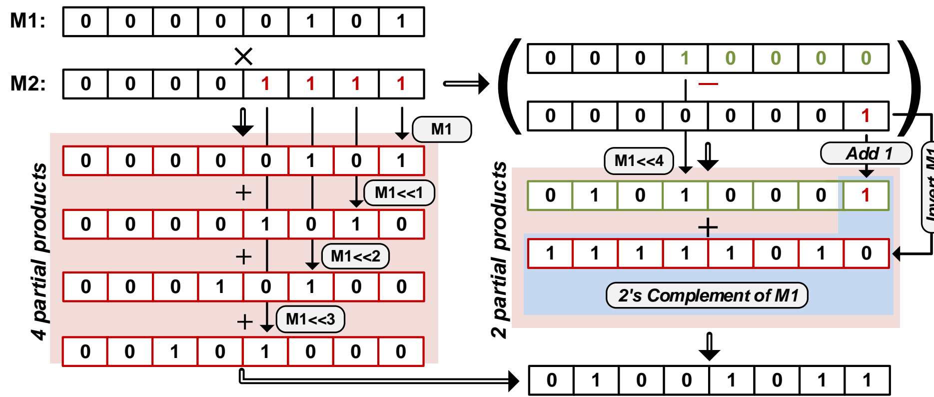


Figure 5.3. Generating the partial products in latency-optimized NNPIM

Factorizing 119 into 7 and 17 leads to two carry propagation stages instead of one. Since carry propagation is the bottleneck in the multiplication process, many such operations make it impossible to gain time from the reduced number of instructions.

In order to reduce latency, NNPIM uses an adder structure similar to that in Section 3.1 while taking into consideration the constant operand in neural networks. It exploits the fact that in binary representation, a sequence of 1s, for example $b00011111$, can be written as a difference of two shifted 1s, i.e. $b00011111 = b00100000 - b00000001$. Instead of generating multiple shifted partial products, NNPIM generates only two partial products. It is similar to Booth's recoding but differs in the way it is implemented in memory. Instead of applying the operations serially as in the case of Booth's recoding, we modify subtraction to make it suitable for parallel execution. To maintain uniformity by executing only addition instructions, NNPIM simplifies subtraction as shown in Figure 5.3. In the figure, generation of 2's complement of M1 involves inversion of M1 and addition of 1. Inversion is a single MAGIC NOR step, where all the bits can be inverted in parallel. Moreover, 1 is added to the shifted version of M1. The LSB of the shifted M1 is always 0, converting the addition of 1 ($Add1$ in Figure 5.3) to a simple SET operation on LSB. The two partial products can then be added normally as in case of a conventional multiplication.

The above technique may not be applicable directly since it is highly unlikely for the

weights to always be a sequence of 1's. Hence, we propose a modified version of Bernstein algorithm which is suitable for carry save addition. Instead of breaking down the constant operands into smaller factors, we break them down into chunks of continuous 1s as shown in Figure 5.4. These smaller parts of constants are then reduced using the same concept as discussed above. Since this approach generates two partial products for a series of 1s, reduction is done only when there are more than 2 consecutive 1's. In the example shown in Figure 5.4, the binary execution would require 11 partial products, but the optimized one generates just 6 partial products. Unlike the factors obtained by Bernstein algorithm, these partial products are added in parallel using carry save addition. This reduces the latency of NNPIM significantly.

Figure 5.5 compares the energy-optimized and latency-optimized approaches for 32-bit multiplication. The result shows that energy-optimized approach can provide 2.3× higher energy efficiency as compared to latency-optimized approach, while the latency-optimized is 1.8× faster.

5.3.4 NNPIM Architecture

Figure 5.6 details the architecture of the proposed NNPIM. Figure 5.6a shows the overview of the architecture of NNPIM. Each neuron in NN has a corresponding computation unit. Each of this unit is made up of several computation sub-units, one for every weight corresponding to the inputs of the neuron. Every unit has an additional computation sub-unit which is responsible for accumulation of all the multiplication results for a neuron and implementing the activation function, which we call activation unit. The outputs from all the activation units are sent to the pooling unit. In case pooling is not required, the output of activation units is used directly for the next layer.

NNPIM is entirely based on crossbar memory. The crossbar structure is divided into smaller blocks, upper blocks and lower blocks as shown in Figure 5.6b. All these blocks are architecturally and functionally the same as described in [126]. Each computation sub-unit as well as activation unit is one such block pair (pair of one upper and one lower block). All the

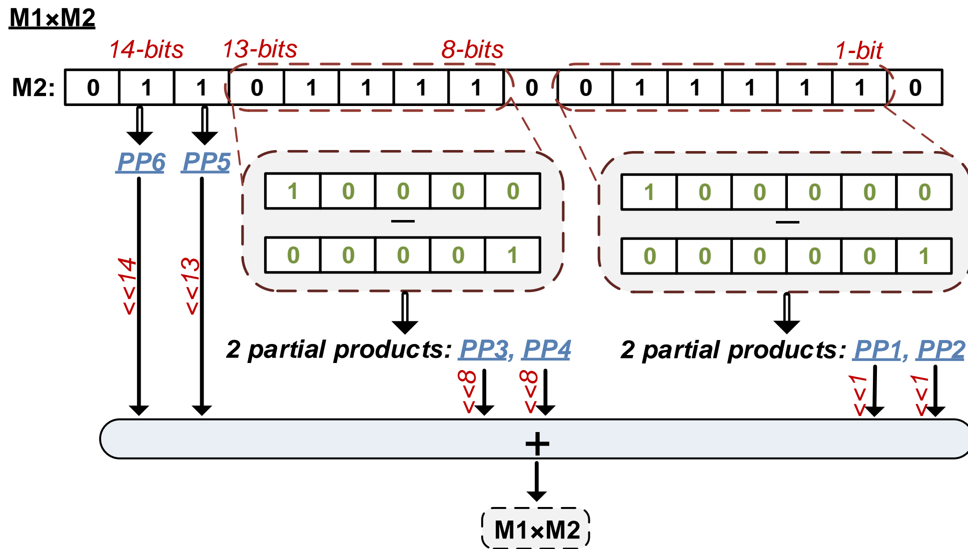


Figure 5.4. Optimizing NNPIM by reducing the complexity of weights

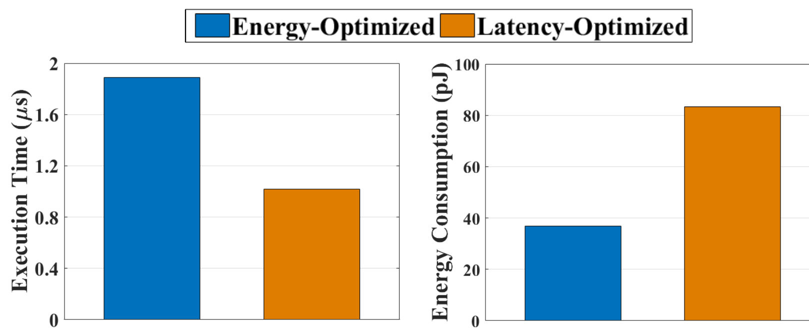


Figure 5.5. Execution time and energy consumption of 32-bit NNPIM multiplication in energy and latency-optimized.

computations for a weight are executed in the corresponding block pair. Hence, a neuron with N weights will have N computation sub-units which implies N block pairs. The major peripheral circuitry including the bitline and wordline controllers, sense amplifiers, row/column decoders, etc. are shared by all these pairs.

Each upper block is connected to the corresponding lower block via configurable interconnects as shown in Figure 5.6c. These interconnects are 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

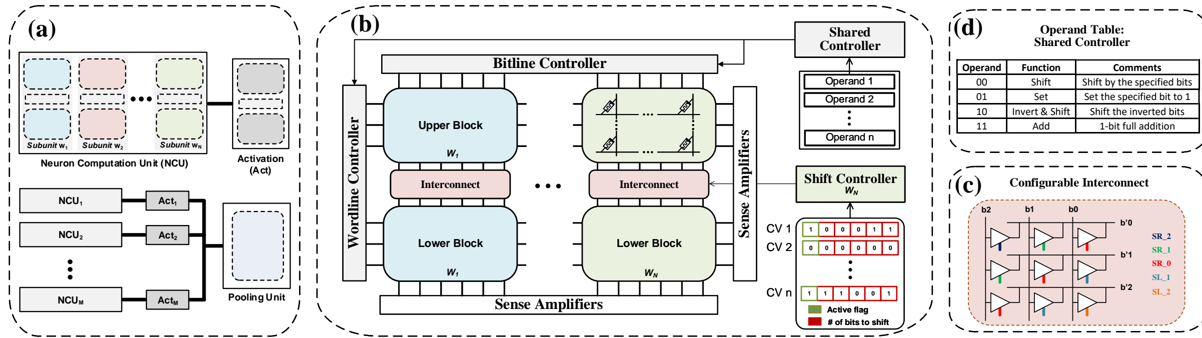


Figure 5.6. Architecture overview of the proposed NNPIM. (a) Overall view of neural network implementation in-memory; (b) in-memory implementation of neuron; (c) circuit for configurable interconnect; (d) functions used in NNPIM.

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 the otherwise slow shifting operations energy efficient and fast, having the latency same as that of a normal copy operation. It is important because neural networks involve large number of shift operations (mainly due to multiplication), which could be a bottleneck if not dealt at the hardware level.

All the outputs of multiplication for a neuron are accumulated and Taylor expanded activation function is implemented in the activation unit, which is made up of the same sub-unit as described above. The outputs of all these units are sent to the pooling unit. This pooling unit is a usual crossbar memory which doesn't require splitting the memory into multiple blocks. The pooling unit works on the in-memory search operations. The outputs from all the activation units are written and the outputs closest to $+inf/-inf$ are selected for MAX/MIN pooling.

In a general purpose implementation, the weights would be stored in memory and the inputs would get multiplied with the stored weights in parallel in different blocks. However, such an architecture will not be able to take advantage of the optimizations proposed in the previous sections. NNPIM uses a control-store architecture, where a control word for a block is defined by a shared operand and the corresponding local control vector (CV). Instead of storing the actual

fixed weights in the memory, we pre-program the control words in the memory. These control words are optimized based on the techniques proposed before. The memory unit loads a control word and implements the operation without worrying about the actual weights.

The shared controller for the bitline and wordline, takes in 2-bit operands as shown in Figure 5.6b. Each operand, detailed in Figure 5.6d, corresponds to a specific function required by NNPIM for computations. Each pair of upper and lower blocks in our architecture has an independent shift controller which governs the bit shifts between the two blocks. The shift controller is a simple circuit which activates a particular select line depending upon the control vector sent to it. The control vector has two fields: (i) active flag which indicates whether the shift controller is active in that cycle and (ii) a 5-bit field indicating the amount of shift. A computational unit has a common shared operand list, while each sub-unit (i.e. each block pair) has its own CV list. A memory with N block pairs has N configurable interconnects and hence, N shift controllers. Each operand sent to the shared controller has a corresponding control vector for each shift controller. Our architecture enables independent shifts among different pairs of blocks while introducing very little overhead as shown in Section 5.4.

Example: Figure 5.7 shows sample execution of two NNPIM multiplications in parallel, $In1 \times W1$ and $In2 \times W2$. After applying the optimization described in Section 5.3.3, the first multiplication results in 5 partial products while the second multiplication results in 6 partial products. The partial products generation by a shift and subtraction (i and ii in Figure 5.7) take three operations each. Here, in order to reduce the number of operations, the shifts before and after the subtraction are combined together. Also, the last operation in the example is not required by $W1$. So, the enable bit in the control vector for $W1$ is set to zero.

5.3.5 In-Memory Parallelism

NNPIM uses a blocked memory structure as shown in Figure 5.6b. Here, each block processes computation corresponding to one weight. Since each block pair in NNPIM has a shift controller, all these blocks can independently implement multiplication in parallel and

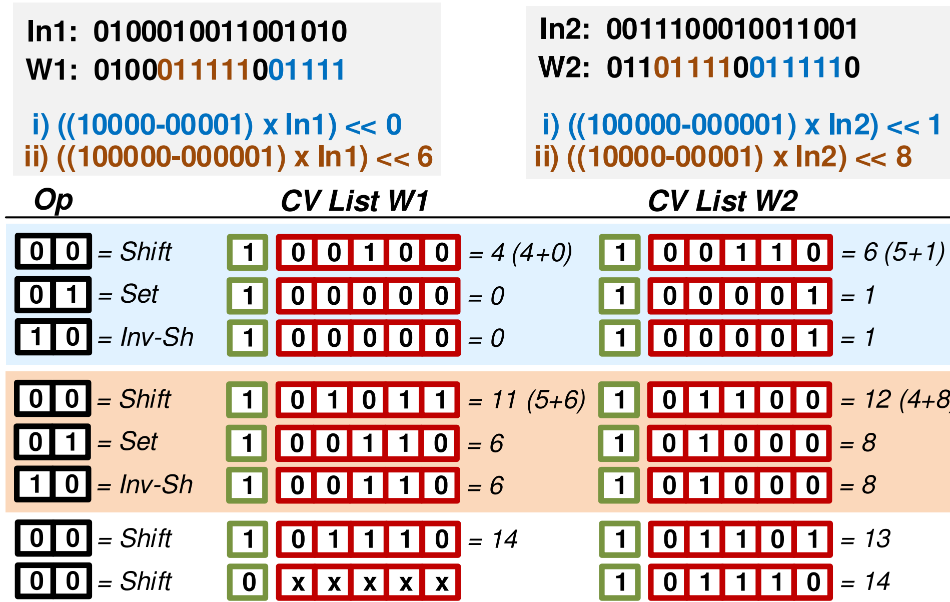


Figure 5.7. Operands and control vectors for two parallel NNPIM multiplications.

computation for multiple weights can happen simultaneously. The number of computations possible in parallel directly effect the number of neurons that can be processed in parallel. This is limited by the size of the memory available. Assume that our memory allows for 2k block pairs. In a network where each neuron has 512 weights corresponding to 512 inputs, our memory can implement just 4 (=2k/512) neurons in parallel. This can be a bottleneck in large networks.

Weight sharing turns out to be useful in such cases as it restricts the number of unique weights for each neurons, thereby enabling the execution of more neurons in parallel. For the case discussed above, the number of neurons possible to be executed in parallel increases from 4 to 32 when the number of weights are restricted to 64. This further increases to 64, 128, and 256 when the number of weights are restricted to 32, 16, and 8 respectively. More the number of neurons executed in parallel, lesser is the overall latency of the network. Hence, weight sharing not only reduces the number of computations but also increases the overall performance of the network as further verified in Section 5.4.

5.4 Experimental Results

5.4.1 Experimental Setup

We designed the NNPIM framework support, which retrains NN models for the accelerator configuration, in C++ while exploiting two back-ends, Scikit-learn library [137] for clustering and Tensorflow [138] for the model training and verification. For the accelerator design, we use Cadence Virtuoso tool for circuit-level simulations and calculate energy consumption and performance of all the NNPIM memory blocks. The controller has been designed using System Verilog and synthesized using Synopsys Design Compiler in 45nm TSMC technology. We use VTEAM memristor model [124] for our memory design simulations with R_{ON} and R_{OFF} of $10k\Omega$ and $10M\Omega$ respectively.

We evaluate the efficiency of the proposed NNPIM over six popular neural network applications:

Handwriting classification (MNIST) [139]: MNIST includes images of handwritten digits. The objective is to classify an input image to one of ten digits, $0 \ldots 9$.

Voice Recognition (ISOLET) [140]: ISOLET consists of speech signals collected from 150 speakers. The goal is to classify the vocal signal into one of 26 English letters.

Indoor Localization (INDOOR) [141]: We designed a NN model for the indoor localization dataset. This NN localizes into one of 13 places where there is high loss in GPS signals.

Activity Recognition (HAR) [142]: The dataset includes signals collected from motion sensors for 8 subjects performing 19 different activities. The objective is to recognize the class of human activities.

Object Recognition (CIFAR) [143]: CIFAR-10 and CIFAR-100 are two datasets which include 50000 training and 10000 testing images belonging to 10 and 100 classes, respectively. The goal is to classify an input image to the correct category, e.g., animals, airplane, automobile, ship, truck, etc.

Table 5.1 presents the NN topologies and baseline error rates for the original models

Table 5.1. NN models and baseline error rates for 6 applications (Input layer - *IN*, Fully connected layer - *FC*, Convolution layer - *C*, and Pooling layer - *PL*.)

Dataset	Network Topology	Error
MNIST	$IN:784,\ FC:512,\ FC:512,\ FC:10$	1.5%
ISOLET	$IN:617,\ FC:512,\ FC:512,\ FC:26$	3.6%
INDOOR	$IN:520,\ FC:512,\ FC:512,\ FC:13$	4.2%
HAR	$IN:561,\ FC:512,\ FC:512,\ FC:19$	1.7%
CIFAR-10	$IN:32\times32\times3, CV:32\times3\times3, PL:2\times2,$	14.4%
CIFAR-100	$CV:64\times3\times3, CV:64\times3\times3, FC:512,\ FC:10\ (100)$	42.3%

before weight sharing. The error rate is defined by the ratio of the number of misclassified data to the total number of a testing dataset. Each NN model is trained using stochastic gradient descent with momentum [144]. In order to avoid overfitting, Dropout [145] is applied to fully-connected layers with a drop rate of 0.5. In all the NN topologies, the activation functions are set to "Rectified Linear Unit" (ReLU), and a "Softmax" function is applied to the output layer.

5.4.2 NNPIM & Weight Sharing

We compare the efficiency and accuracy of the NNPIM over different application with and without weight sharing. Table 5.2 shows the impact of weight sharing on the classification accuracy of NNPIM. Table 5.2 shows the NNPIM quality loss (QL) for different applications when the number of shared weights in each neuron changes from 8 to 64. The QL is defined as the difference between NNPIM accuracy with and without weight sharing. Our evaluation shows that a network with 64 shared weights can provide the same accuracy as a design without weight sharing. Further reducing the number of weight to 8 reduces the classification accuracy of applications. For instance, CIFAR-10 and CIFAR-100 lose 1.2% and 2.8% quality respectively when the number of shared weights decreases to 8.

NNPIM exploits this weight sharing in order to accelerate neural network computation by reducing the multiplication cost. Figure 5.8 shows the energy consumption and memory

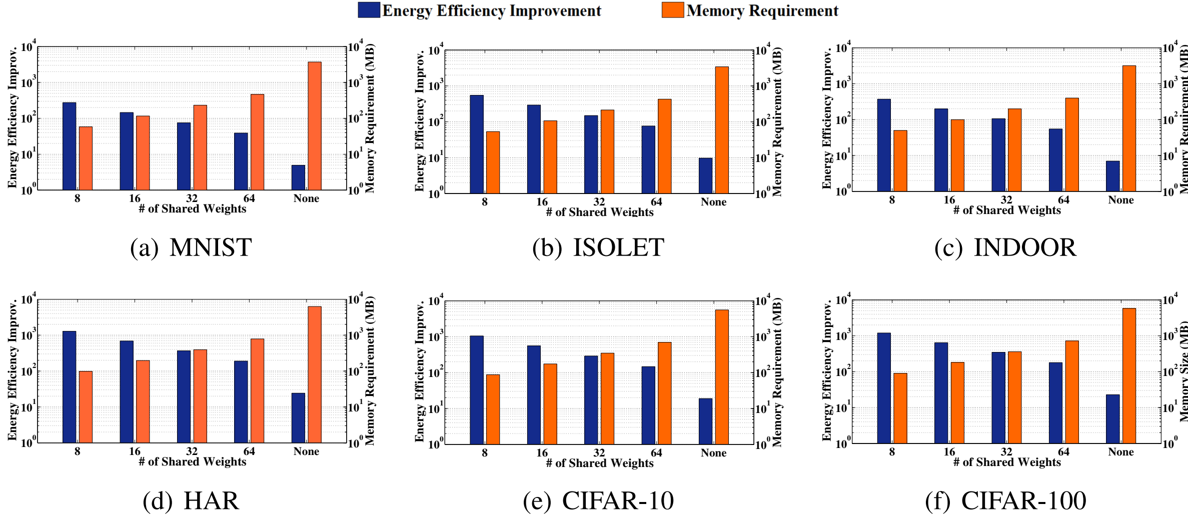
Table 5.2. Quality loss of different NN applications due to weight sharing.

Dataset	8 weights	16 weights	32 weights	64 weights
MNIST	1.1%	0.26%	0%	0%
ISOLET	0.33%	0.12%	0%	0%
INDOOR	0.38%	0.24%	0.13%	0%
HAR	2.1%	0.32%	0.14%	0%
CIFAR-10	1.2%	0.29%	0.09%	0%
CIFAR-100	2.4%	1.2%	0.8%	0%

requirement of NNPIM running different applications with different weight sharing. The reported improvements are compared to energy consumption of the same applications running on AMD Southern Island GPU. The energy efficiency of NNPIM significantly improves as the number of shared weights reduce. Our evaluation shows that NNPIM without weight sharing provides 14.6× energy efficiency improvement as compared to GPU architecture. We observe that NNPIM gets energy efficiency improvements from removing the data movement cost and efficient in-memory computation. However, in terms of performance the NNPIM advantage comes mostly from addressing the data movement issue.

The NNPIM advantages are more obvious on large networks such as CIFAR-10 and CIFAR-100, since these networks have more data movement. Weight sharing can significantly improve the NNPIM efficiency by reducing the computation cost. The result shows that using 64 shared weights provides 113.9× energy efficiency improvement and 14.8× speedup as compared to GPU architecture while ensuring 0% quality loss. Similarly, accepting 1% and 2% quality loss, the average energy efficiency improvements of NNPIM increase to 370.4× and 786.3× respectively. Weight sharing does not impact the performance of NNPIM since all neurons in a layer are implemented in parallel and consecutive layers still need to be processed serially.

Figure 5.8 also shows the required NNPIM memory size for different amounts of weight sharing. NNPIM requires significantly lower memory size for PIM operation as compared to NNPIM without weight sharing. As our results show, decreasing the number of weights by half,



(a) MNIST (b) ISOLET (c) INDOOR

(d) HAR (e) CIFAR-10 (f) CIFAR-100

Figure 5.8. Energy consumption and memory size requirement of NNPIM with and without weight sharing.

reduces the number of required multiplications by half. Our evaluation over all applications indicates that by reducing the number of weights to 64, NNPIM will provide maximum quality while using 7.8× less memory as compared to NNPIM without weight sharing. Similarly, ensuring less than 1% and 2% quality loss, NNPIM uses 12.4× and 15.6× lower memory size as compared to NNPIM without weight sharing.

5.4.3 Energy-performance Efficiency

In this section we compare the energy consumption and execution time of NNPIM with DaDianNao [146] and ISAAC [10], the state-of-the-art NN accelerators. All designs have been tested over six different applications. For NN accelerators, we select the best configuration reported in the papers [10, 146]. For instance, ISAAC design works at 1.2GHz and uses 8-bits ADC, 1-bit DAC, 128×128 array size where each memristor cell stores 2 bits. DaDianNao works at 600MHz, with 36MB eDRAM size (4 per tile), 16 neural functional units, and 128-bit global bus. We see that of the previously proposed designs, ISAAC performs better over all datasets. Figure 5.9 shows the energy efficiency improvement and speedup of NNPIM, DaDianNao and ISAAC as compared to AMD GPU architecture. Our evaluation shows that NNPIM outper-

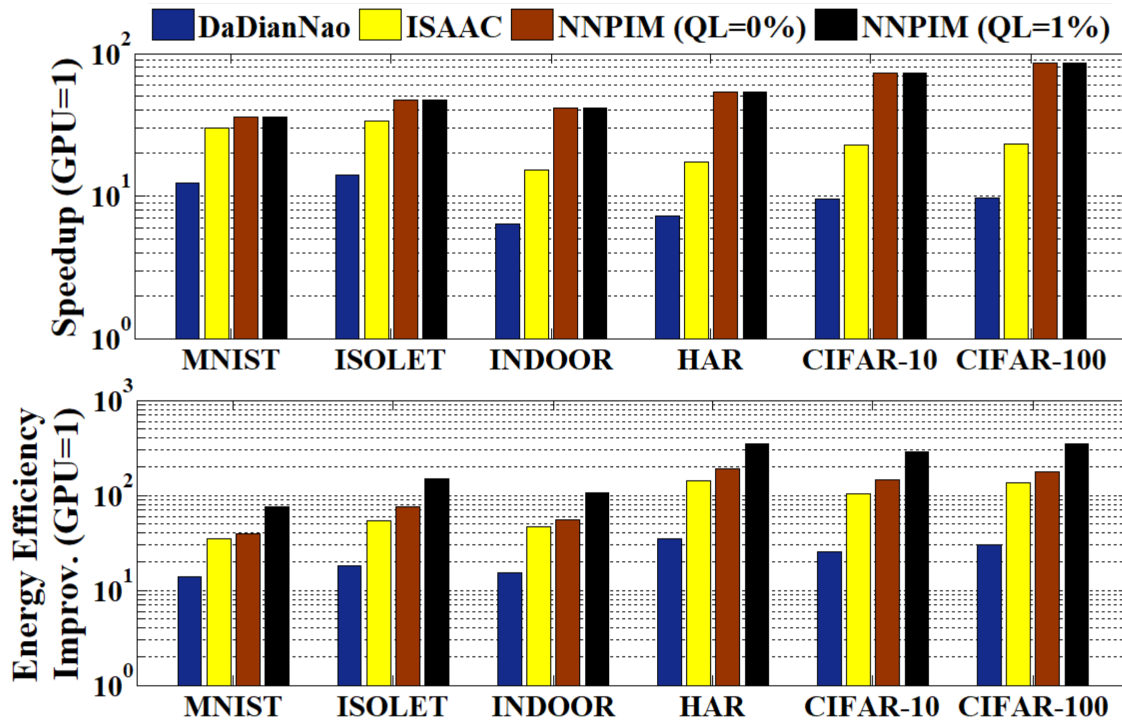


Figure 5.9. Comparing the energy consumption and execution time of NNPIM with state-of-the-art NN accelerators.

forms both DaDianNao and ISAAC over all applications. For example, benchmarking with MNIST, proposed NNPIM can provide 2.8× energy efficiency improvement and 2.9× speedup as compare to DaDianNao. These improvements are higher over CIFAR-10 and CIFAR-100, as NNPIM provides higher computational efficiency over large networks. Considering the average improvements over all applications, our design can achieve 4.9× (1.3×) energy efficiency improvement and 5.7× (2.4×) speedup as compared to DaDianNao (ISAAC) while providing the same classification accuracy. While accepting 1% quality loss, the NNPIM energy efficiency improves to 9.5× and 2.5× as compared to DaDianNao and ISAAC respectively.

5.4.4 Area Overhead

Comparing the area overhead of NNPIM to conventional crossbar memory shows that the NNPIM takes 3.8% area of the chip. This area corresponds to 2.9% for the registers storing the network weights and 0.9% for the barrel shifter used for multiplication. In addition, the weight

sharing significantly reduces the NN model size and the number of required hardware to process the weights. Our result shows that the NNPIM area overhead is negligible compared to prior PIM-based DNN accelerators [10, 146] which use large ADCs and DACs to covert the data from digital to analog and analog to digital.

This chapter showed how complex arithmetic computations can be done over a large amount of data in-memory. This efficient PIM technique was then used to design a PIM architecture for neural networks. In the following chapter, we extend the functionality of to non-arithmetic operations like search and show how it can accelerate query processing.

Chapter 5, in part, has been submitted for publication of the material as it may appear in S. Gupta, M. Imani, H. Kaur, T. Rosing, “NNPIM: A Processing In-Memory Architecture for Neural Network Acceleration,” *IEEE Transactions on Computers*, 2019.

Chapter 6

Efficient Query Processing in NVM

Previous chapters proposed architectures for traditional arithmetic computations. However, applications like query processing use a wide range of functions like aggregation and prediction functions, bit-wise operations, addition, joins, exact and nearest distance search operations. Conventionally these operations have been implemented using sequential arithmetic operations over large chunks of data. In this chapter, we exploit the analog characteristic of non-volatile memory to enable these operations in-memory. We design a novel non-volatile, memory-based query processing accelerator, called NVQuery. It is an efficient PIM-based query processor which supports a wide range of query functions. The configurable crossbar memory structure of our design supports these functions inside the memory.

6.1 NVQuery Accelerator

Fig. 6.1 shows the general architecture of the proposed NVQuery. The proposed NVQuery integrates with DRAM and enables the main processor to accelerate query processing. NVQuery can also be used as a secondary storage to improve the effective DRAM capacity. NVQuery consists of N banks, where each has k memory blocks. Each memory block can be configured as memory or query accelerator.

Our design is a heterogeneous architecture, where the NVQuery co-operates with main processor (CPU) in order to find the query result. In NVQuery, each memory block returns a

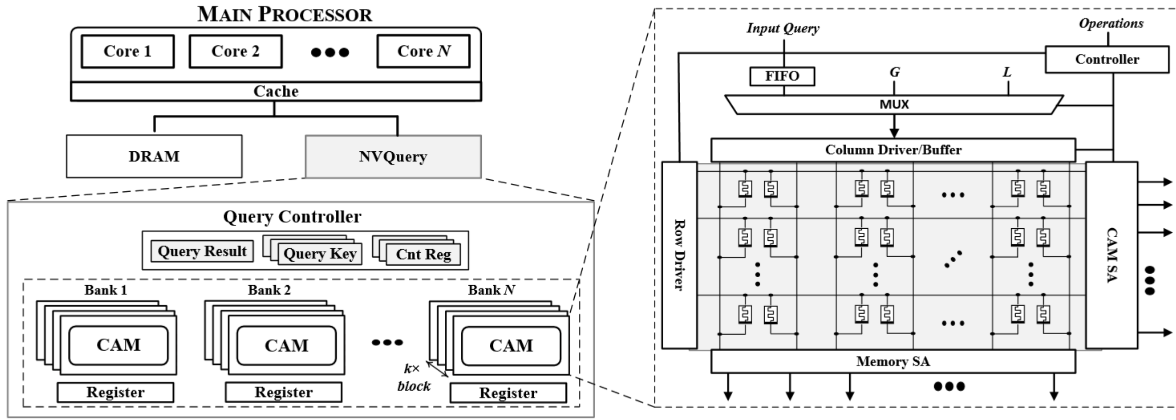


Figure 6.1. Proposed architecture with N banks and $k \times N$ blocks. The right part details the crossbar implementation of memory banks along with the supporting control logic.

result of the query, independent from other blocks. Therefore, to find the result of a query from the whole data set, the main processor receives output response of each memory block (a total of $N \times k$ values instead of the entire data). Finally, it processes data to find the result of query over the entire data set. In this way, the load on memory bandwidth due to query processing and its related costs are significantly reduced. Table 6.1 lists different configurations that NVQuery can take including: nearest search, search, and memory. For each of the configurations, we show the status of different memory peripherals for some example functions. In this section, we describe the functions supported by our proposed non-volatile query processor, NVQuery. Table 6.2 lists the NVQuery support functionalities. NVQuery supports a large number of essential functions including aggregation (`MIN`, `MAX`, `Average`, `SUM`, and `Count`), boolean functions (such as `AND`, `OR`), addition, comparison (equality or non-equality), and different types of Join. In addition, NVQuery can also process prediction functions such as `Exist`, `Search Condition`, `Like`, `Group`, `Between`, and `Top` in memory.

We map all query functionalities explained in Table 6.1 to NVQuery which can work in three main configurations: (i) look-up table (LUT) with capability of exact search, (ii) nearest distance search, and (iii) memory. We propose a new memory architecture which can process data locally without reading it. In each of these configurations, our design shown in Fig. 6.1

Table 6.1. NVQuery supported configurations

Configuration	Example Functions	CAM Input	CAM SA	Memory SA	Comment
Nearest Search	MIN	Least	Nearest	NA	L: Least possible value
	MAX	Greatest	Nearest	NA	G: Greatest possible value
	TOP K	IQ (FIFO)	Nearest	NA	Requires k iterations
Search	Exact Search	IQ (FIFO)	Exact	NA	IQ: Input query
Memory	Bitwise	NA	NA	AND/OR	CAM input: Bitline driver
	Memory	NA	NA	MEM	CAM input: Bitline driver
	Addition	NA	NA	MAJ	CAM input: Bitline driver

Table 6.2. NVQuery supported functionality

	Notation	Functions
Aggregation	$F(S_I) \rightarrow S_O$	MIN, MAX, Average, Count
Bit-wise Operations	$F(S_I) \rightarrow S_O$	AND, OR, XOR (Combination of AND, OR)
Addition	$F(S_I) \rightarrow S_O$	In-memory addition
Comparison	$= \leq \geq$	Bit-wise and value-wise comparison
Predict	p	Exist, Search condition, Top, Like, Group, Between
Join	$\bowtie, , , \ldots$	Inner, Left, Right, Outer, Semi joins

processes query operations without approximating the result. In the following subsections, we explain how each query operation can be supported in memory.

6.1.1 Exact Search

The most common operation in many query processors is looking up for a set of data which matches with input query. A typical search query involves a brute-force search through a LUT till the data is located. This is usually implemented in one of the two ways, (i) word-by-word search and (ii) bit-by-bit search. A word-by-word search looks through every stored word in the LUT sequentially and finds a match. In the worst case, it involves processing each and every element present in the LUT. The bit-by-bit search scans through one bit (but same index) for multiple words at a time. The first iteration analyses a particular bit index of every word in the LUT, looking for a match with the corresponding entry in the input query. The following iterations are performed only on the words filtered by previous iterations. This approach does not analyze all the elements since the size of candidate pool decreases after each iteration. The

exact search operation supports the following functions in NVQuery:

`Exist`: It is used to test the existence of some specific data in the LUT. The exact search can be directly used to implement this function.

`Count`: It is used to get the number of rows that match a certain criteria. The `Count` output can be obtained by counting the number of hits for an exact search query. Our design adds a counter block to NVQuery in order to support this query.

`Like`: It is used to find the existence of a specific data or pattern of data in the rows. It involves searching for occurrence at (i) a particular position and (ii) any position. The first case can be easily implemented using the exact search mode in the same way as the `Exist` function. The second case requires repeated use of exact search mode for all the occurrence patterns possible. This comes with an inherent latency overhead due to multiple serial exact searches.

`Group by`: It is used to group the rows on the basis of one or more columns. The grouping is usually based on the output of some operation applied to the data in the column. Multiple serial exact searches are used to find the rows belonging to different groups.

6.1.2 Nearest Distance Search

NVQuery can be configured to perform the closest distance search operation inside the memory. The bit-by-bit search described above can be used to implement this functionality. Here, the nearest data is the one which remains selected for the maximum number of iterations. Our design exploits this functionality to support aggregation functions like `MIN` and `MAX` and prediction functions like `Top k`. Running these queries on traditional core has a time complexity of $O(\log n)$. However, our hardware can find `MIN, MAX` queries in a single cycle and `Top k` in k cycles.

`MIN`: This query runs on a set of stored data to find the minimum value. To perform this query in LUT, NVQuery block adopts the nearest distance search configuration and searches for the data which has the closest distance to the minimum possible value. In case of unsigned

numbers, our design searches for an entry which has the closest distance to zero. In the case of signed values, this number is the largest possible negative number (single one followed by a chain of zeros, i.e., 1000...0).

`MAX`: To find the data with the maximum value, we search for the entry which has the least distance from the largest positive number. For unsigned values, the largest value is a chain of ones (1111...1), while in the case of signed numbers, this value is represented by a zero followed by a chain of ones (0111...1).

`Top k`: To search for k values closest to the input data, we perform the nearest distance search for k iterations. After each iteration, our design deactivates the selected word and repeats the nearest distance search on the remaining words. This approach gives a set of k nearest values arranged in the order of their proximity to the input. Our design also supports bit-wise/value-wise comparison by searching for the exact and nearest values.

`Between`: This operator takes in two inputs, the lower and upper limits, and outputs those values from the stored data which are equal to or between these limits. Traditional implementations of this function involve a lot of computational overhead, comparing each value with the limits. The nearest distance search proposed above enables efficient implementation of this operator. Instead of finding the values nearest to the upper and lower limits, we find the values nearest to the midpoint of the total range. Then the values are sorted as explained in Section 6.2.2. NVQuery compares the values with the input limits and selects the entries which lie between them. Instead of naively searching through the data, NVQuery uses binary search to find the corner cases and reduce the computational overhead.

6.1.3 Join

NVQuery supports different types of joins namely, `inner, left, and right joins`. Our implementation is similar to in-memory hash joins, but more efficient due to NVM-based PIM. Ideal implementation of join would involve fetching the data from memory to the core and searching through the involved tables. Although, optimizations like hash join reduce the amount

of data to be transferred yet the cost of data movement is a lot. NVQuery reduces this overhead by reducing searching for keys inside the memory itself. The exact search discussed earlier is used to implement joins.

Equi joins involve searching for exact match of the join key through the tables. Exact search mode can be easily extended to implement different kinds of *equi* joins, enabling the records that are needed for the final join computation. Memory read bus along the columns of a table is used to read the desired columns of the rows with matching data. The read data is copied to the memory location pertaining to the final join output. Limited 4K row capacity forces the implementation to break the table into multiple 4K slices or blocks, this does not affect the computational complexity of the implementation and the impact on execution time is also minimal. A choice between a block or slice is made based on the query and size of the input.

NVQuery is flexible enough to cater to any combination of columns to implement different types of joins like inner, left, and right joins. Multi joins are implemented by saving the temporary result of two table joins and iteratively applying joins to that.

Different SQL implementations use a combination of nested loop, merge and hash joins. Execution complexity of these methods vary based on availability of index on the join property. The worst case complexities are $O(NM), O(MlogN + NlogM)$ and $O(N + M)$ respectively, where *N,M* are the table sizes. NVQuery's worst case execution complexity is equivalent to hash joins, though it does not require explicit hashing of the join key like hash joins do.

6.1.4 Bit-wise Operations and Addition

A traditional processor implements bit-wise logic operations in the main core. The operands are fetched from the main memory and brought through the memory hierarchy all the way up to the core. The core then performs the required computations. On the other hand, our design implements these operations in the memory, avoiding the need to transfer data from memory. For executing these operations, NVQuery is set into memory configuration

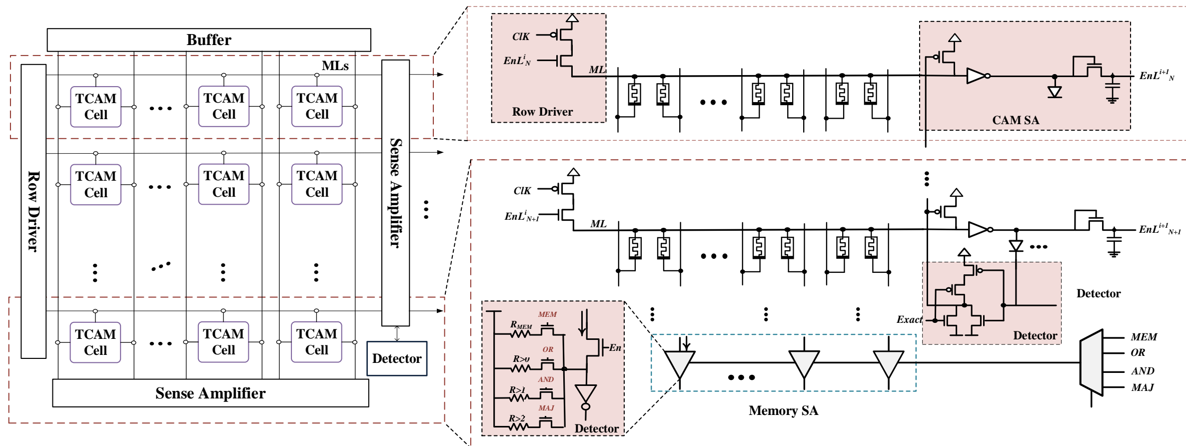


Figure 6.2. Circuit level implementation of CAM SA, Memory SA, and Row Driver.

and the output is obtained from memory SA. This operation can support the following queries: `AND, OR, XOR` and `Average`. Our design supports average query by using a counter and sending the data to main processor.

6.2 Hardware Support

This section describes the hardware implementation of NVQuery and the way in which it supports the functions described in Section 6.1. NVQuery is designed using a crossbar non-volatile memory architecture. The crossbar is configured in such a way that a set of two storage elements in the crossbar corresponds to one bit data. Data 0 is stored as $\{R_{HIGH}\ R_{LOW}\}$, while 1 is stored as $\{R_{LOW}\ R_{HIGH}\}$. However, our architecture does not use any access transistors for these elements, hence it is called 0T-2R. Implementations like 2T-2R require access transistors. This makes the design unsuitable for a crossbar memory, reducing the area density benefit of non-volatile memories. Moreover, the presence of transistors introduces non-linearity to the system. On the other hand, 0T-2R doesn't need access transistors and can be implemented on a conventional crossbar memory, making it more area efficient.

As shown in Fig. 6.1, the crossbar memory in NVQuery is supported by peripheral components. The controller receives the input query and generates the appropriate control

signals. It is also responsible for collecting the output of the block and forwarding it for further processing. The multiplexer managed by the controller, selects the input which drives the bitlines of the crossbar memory. This input can either be the input query (in case of search operations) or greatest positive value (corresponding to `MAX`) or least representable value (corresponding to `MIN`). The column driver drives the bitlines of the crossbar. It not only applies the execution voltages for different operations but also maps the input query to the required bitline voltage levels. Row driver is responsible for charging the wordlines (also called match-lines due to the nature of operations). It is also responsible for selecting/activating different words (rows) in the memory. It also provides a limited set of voltage options essential to the working of crossbar. The crossbar is equipped with sense amplifiers (SAs) on both the wordlines (CAM SA) and the bitlines (memory SA). Fig. 6.2 shows these SAs. The CAM SAs are responsible for detecting charging and discharging behavior of wordlines. The nMOS-capacitor circuit acts as a latch. The inverter-diode-NOR circuit deactivates the wordlines as soon as the first edge is detected or the sampling signal for *Exact* is set. As a result, the latch is set only for the wordlines which discharge before this deactivation. The memory SAs are buffers with special resistors to support bit-wise and memory operations as described in Section 6.2.3. We next discuss how NVQuery enables different functions discussed in Section 6.1.

6.2.1 Exact Search

To implement the LUTs discussed in Section 6.1.1, NVQuery uses content addressable memory (CAM) configuration of crossbar. Fig. 6.2 shows the structure of non-volatile crossbar CAM, capable of searching for stored data which exactly matches the input query. During search operation, all the match-lines (MLs) pre-charge to V_{dd}. The input buffer (column driver) distributes the query point to all CAM rows using vertical bitline. Any cell with the same stored data as input query discharges the ML. The sense amplifier, connected to the horizontal ML, determines the equality of the input and stored data by sampling the ML voltage [147].

Consider a data set which contains the name, age, height, and income of people in

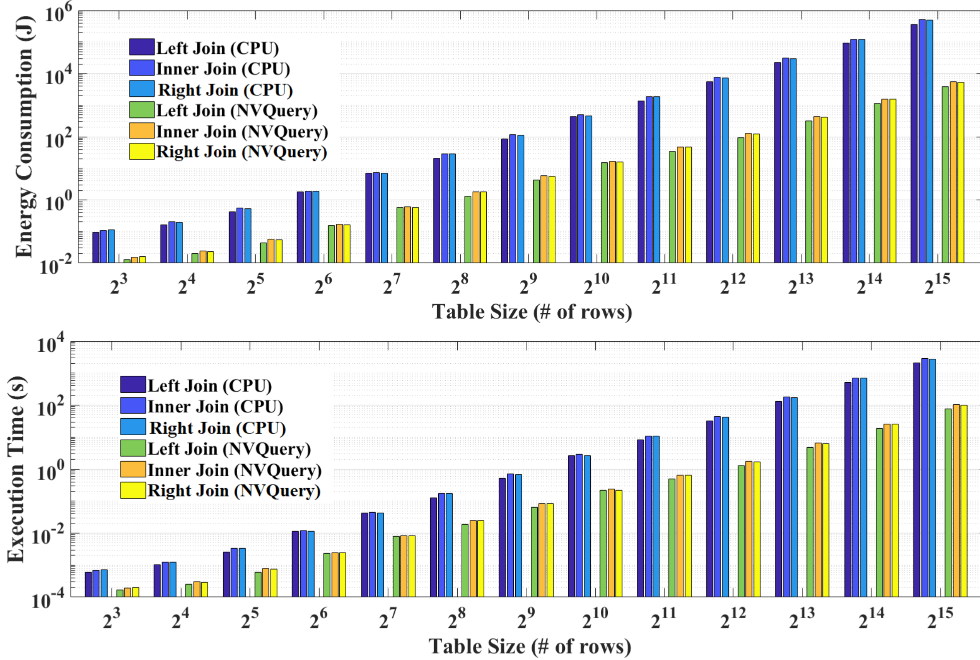


Figure 6.3. Energy consumption and performance of running join operations with different table sizes on traditional cores and the proposed NVQuery.

different companies. The query `SELECT F(income) FROM COMPANY1` is an example of SQL query. For this query, a query processor first selects all people in the list which are working for the `COMPANY1`. Then it applies another query function, `F`, on the `income` of all selected people. NVQuery eliminates the need for multiple sequential searches. It can perform a single step search by activating the bitlines corresponding to `COMPANY1` and `income` simultaneously. The output of the query is given by the rows with fastest discharging MLs. This not only saves time by eliminating multiple searches but also the power involved in repeated charging and discharging of MLs. Each memory block/LUT returns an output to the controller. Finally, the output data from each block is processed by the main processor which evaluates the final query result.

6.2.2 Nearest Distance Search

CAM has been extensively used to implement search operations. Different versions of CAM implementations (*e.g.* TCAM) on different types of hardware (crossbar, 2T-2R, 3T-1R,

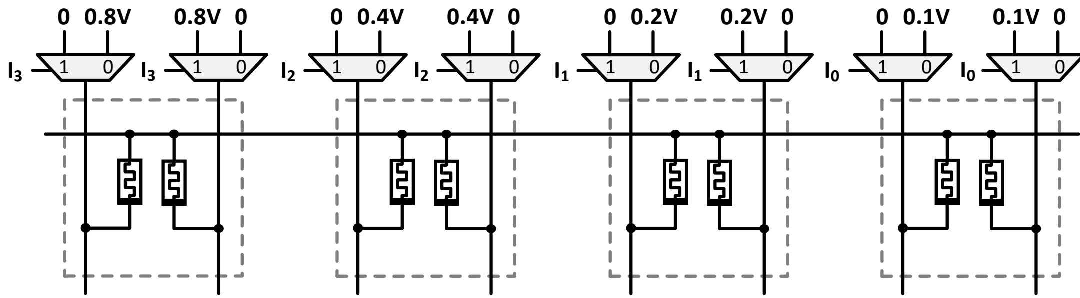


Figure 6.4. NVQuery in nearest distance search configuration.

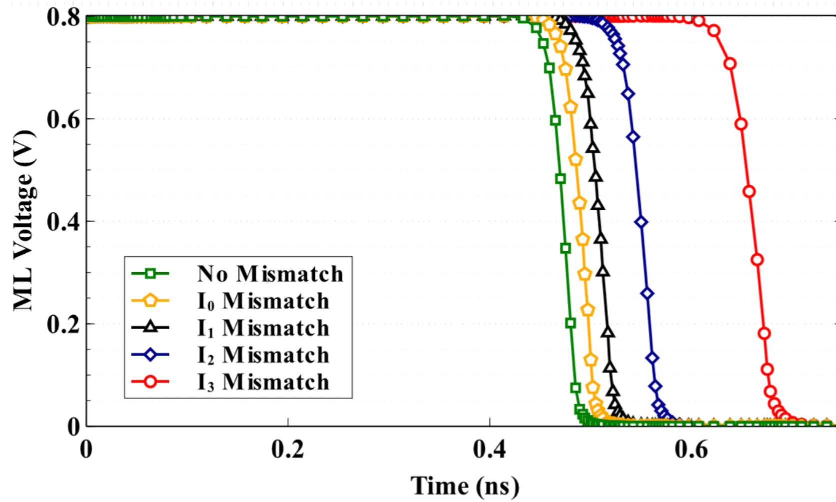


Figure 6.5. Timing characteristic of CAM block in nearest distance search configurations.

etc.) have been active topics of research recently. However, majority of the previous work revolves around exact and nearest hamming distance search operations. Hamming distance is a good criterion when considering hyper-dimensional vectors where the index of a bit does not matter. Only the face value of a bit and the total number of mismatches between the stored data and the input query are considered. Such a comparison is not practical for many real life applications where a query to the processor is dependent on the binary weighted values of the stored data.

To support such queries, some researchers have proposed the division of a memory block into stages [148]. In such an architecture, the first m most significant bits of data are stored in the first stage, the next m significant bits in the second stage and so on. Then, a search is performed

sequentially, starting from the first stage. The output of a stage selects the rows to be activated in the following stage. This increases the weight of the initial stages with respect to the later stages. However, the *m* bits in a stage are treated as having the same binary weight. This leads to inaccurate results in many cases. In this work, we address this issue by introducing a new method to assign binary weights to the bits within a stage.

For a search in conventional CAM, the match-lines (MLs) are pre-charged to V_{dd} and then bitlines are driven with V_{dd} or 0 depending upon the input query. The MLs of rows with more number of matches discharge earlier. The line to discharge first is the one with minimum mismatch with the input query. To give binary weight to the bits, we modify the bitline driving voltage. Suppose a stage contains *m* bits $(m-1:0)$. The bitlines which were earlier driven with V_{dd} and now driven with a voltage $V_i = V_{dd}/2^{(m-1-i)}$ where i denotes the index of a bit in the stage. Fig. 6.4 shows CAM in nearest search configuration for a stage size of 4 bits. As shown in Fig. 6.5, a match in the most significant bit results in faster ML discharging current than lower indices. We exploit this difference and design a CAM which can find the binary value nearest to the input query.

The different discharging currents also allow us to sort the data, with the nearest data discharging first. This sorting is easy for a smaller number of records. However, as the number of records increase, it becomes difficult to differentiate data depending upon the discharging currents. In such a case, nearest search is implemented in groups with limited rows selected at a time.

Now, as the number of bits increases, the bitline voltage V_i becomes very small. We limit the minimum available voltage source output to $100mV$. Moreover, the maximum voltage that can be applied is limited by the threshold voltage of the non-volatile elements. This ensures that the data in the memory is preserved. This upper bound is set to $1.8V$. Hence, the allowable voltage levels include $0.1V, 0.2V, 0.4V, 0.8V$ and $1.6V$, restricting the stage size to 5 bits. In this work, we split the CAM into multiple stages of 4-bits each for simplicity and then search for the nearest distance row in a serial manner, starting with the stage containing the most significant

bits.

6.2.3 Bit-wise Operation and Addition

Although a search based CAM can accelerate several functionalities in NVQuery, it cannot support a major part of queries such as addition, average, and all bit-wise operations. In order to make NVQuery a general design for query processing accelerator, we modify the sense amplifiers in the vertical bitlines to support bit-wise operations. Fig. 6.2 shows the sense amplifier in a single NVQuery bitline to support bit-wise operations. In this mode, each block works as memory instead of CAM, where one of the vertical bitlines in each CAM cell is activated. The tail of the shared bit-line is connected to a sense amplifier. Since our design supports AND and OR functions, the sense amplifier has two main parts: one for AND operation and a simple sense amplifier to support OR. These circuits work on the basis of the leakage current through the vertical bitline. When several rows in memory are active, each row leaks current through vertical bitlines depending upon the resistance value. If the stored bit is 1 (low resistance), this current is large, while in the case of 0, leakage is significantly small. The goal of OR operation is to identify the presence of at least one high (1) bit in all activated rows. Therefore, we use a sense resistor, $R_{>}0$, such that in the case of at least single high bit, it turns the output signal to one. However, for AND operation the goal is to find a case such that at least one input is not 1. In that case, the AND circuitry uses an appropriate sense resistance.

Interestingly, prior work shows that crossbar memory can further support addition within the memory [83, 126]. This approach breaks down an operation into a series of `NOR` operations. The logic family used in the paper executes `NOR` in crossbar memory with a latency of just 1 cycle. This functionality is supported by NVQuery due to its regular structure (unlike CAMs with access transistors), enabling it to perform data computations within memory. In the case when approximate results are acceptable, the sense amplifier at the bitlines can be used to improve the performance of NVQuery. The truth table for 1-bit full adder shows that the sum bit (S) can be obtained by inversion of the carry bit (C) in 75% of the cases. The sense amplifier calculates C

Table 6.3. Approximation in 16-Bit Addition

Approximated Bits	4	8	12	14	16
Error (%)	0.006	0.098	1.56	6.25	25
Energy (pJ)	3.52	2.41	1.3	0.75	0.197
Latency (ns)	182	133	84.7	60.5	36.3

(majority) in one step by simply using an appropriate sense resistance. S is obtained by inverting C. This introduces a worst case error of 25%. However, this error is reduced significantly by approximating only some LSBs depending upon the level of accuracy desired. The MSBs are calculated accurately using the techniques described in [83]. Table 6.3 shows the error corresponding to different number of approximated bits for an 8-bit addition. By calculating the carry bit correctly, the proposed approximation approach limits the effect of an error to one bit and does not propagate it.

Addition is extended to implement average function. The output of successive additions is sent to the processor, where the average is obtained by bit-shifting or simple division.

6.3 Approximation in NVQuery

In most cases, a query does not require a unique or completely precise answer. Instead, it requires a fast result with good enough accuracy. 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 [148, 149, 150]. However, most of the existing techniques provide less energy or performance efficiency due to considerable data movement and lack of configurable accuracy.

NVQuery can work in both exact and approximate mode. Approximate mode provides the advantage of better metrics, both in terms of latency and power consumption. However, this comes at the cost of loss in accuracy. Here, we investigate two ways of approximation: (i) bit trimming and (ii) voltage scaling.

6.3.1 Bit Trimming

One common way to apply approximation in query search is trimming or neglecting bits. Our design neglects few least significant bits of input data in order to accelerate the query functionality. For other bits, NVQuery performs the search serially on the blocks, starting from the most significant bits. The level of approximation is tuned by determining the number of neglected blocks. The upper and lower computation bounds are defined by the number of cut bits. For each input in query, the lower bound is defined by all trimmed bits being zero while the upper bound by all trimmed bits being one.

$$L_V < V < U_V$$

$$U_V - L_V = 2^K - 1$$

Where V is the exact value of V, and L_V and U_V are the lower and upper bounds respectively when the last k bits are trimmed. Therefore, our design guarantees that the NVQuery error rate on aggregation functions, (Minimum, Maximum, Average, Mean, etc.) is

$$Error_{Query} < 2^{M-K} - 1$$

where M is the total number of bits.

For a 5-bit CAM stage with a nominal V_{dd} of $1.6V$, $V_i = \{0.1V, 0.2V, 0.4V, 0.8V, 1.6V\}$ for $i = \{0,1,2,3,4\}$. This leads to an effective difference of $\{1.5V, 1.4V, 1.2V, 0.8V, 0V\}$ between ML and the bitline. If the lower bits in a block are approximated to have the same weight, then the required number of voltage levels can be reduced. However, the voltage levels for the

Table 6.4. NVQuery Approximation at Different Supply Voltages

Voltage	1V	0.87V	0.8V	0.74V	0.7V	0.67V
Errors bits	0	1	2	3	4	5
Norm. Energy	1	0.68	0.39	0.22	0.17	0.11

non-approximated bits should be chosen such that

$$V_i = \begin{cases} (k+1) \times 0.1V, & i = k \\ 2 \times V_i, & i > k \end{cases} \tag{6.1}$$

where k is the number of approximated bits. This ensures that the effective weight of the approximated bits is at least 1 LSB ($0.1V$) less than the first accurate significant bit. For example, if the lower 2 bits are approximated to have the same weight, then the required voltages are $\{0.1V, 0.1V, 0.3V, 0.6V, 1.2V\}$. This further reduces the required V_{dd} for ML, reducing the total energy requirement of the computation.

6.3.2 Voltage Scaling

In NVQuery, approximation is done by applying voltage overscaling (VoS) on selective CAM blocks [151]. While CAM works without any error with nominal V_{dd}, lower supply voltages increase the possibility of error on CAM matching and memory functionality. Table 6.4 lists the possible errors for each CAM block at different supply voltages. For instance, a 6-bit CAM block at 870mV supply voltage can match the input query with stored data with a single bit mismatch. Similarly, at 800mV and 740mV, the CAM block can search for data with 2-bit and 3-bit Hamming distance respectively from the input key.

Our design puts the blocks in different approximation levels based on their impact on approximation. For instance, if the ith block is configured with h-bit error, the $i-1^{th}$ block needs to have $h/2$-bit error. Generally, when the goal is to allow K bit error, we can estimate the error

distance of each bit as follows:

$$h = K/(1 + 1/2 + 1/4 + \ ... \ + 1/2^N)$$

Comparing these two ways of applying approximation shows that voltage overscaling can provide much higher advantage as compared to bit ignoring. In bit ignoring, the energy saving and speedup limits to a few bits which we neglected processing them. For example, in 6-bit CAM, trimming 2-bit, will give us $2/6 = 33\%$ energy savings.

6.4 Experimental Results

6.4.1 Experimental setup

For detailed evaluation of the proposed NVQuery, we run circuit-level simulations in HSPICE with 45nm TSMC technology. We use VTEAM [124] model of memristors with I_{ON}/I_{OFF} ratio of 10^3 for non-volatile memory crossbar design. We develop software-based cycle-accurate simulator (based on C++) which emulates the functionality of the designed NVQuery. This allows us to speed up the simulation time significantly and verify the proposed design with diverse practical data sets. The simulator has accurate models of the hardware, e.g., time and power extracted from the circuit-level simulation to evaluate the efficiency of the proposed design. We compare NVQuery performance and energy efficiency with state-of-the-art query processing approaches running on the same technology node. We evaluate two popular approaches, sampling-based approximate querying (SAQ) [152] and deterministic approximate querying (DAQ) [105] on Intel i7 7600 CPU with 8GB memory. For measurement of the processor power, we use Hioki 3334 power meter. We use a dataset consisting a table of `Census` of 10 million tuples using 32-bit unsigned integers to compare the efficiency of different techniques. This data is popularly used to model populations of various types ranging from cities and organizations to word frequencies in natural language corpora. The SQL server contains a single table with one 10GB column of randomly generated records. In the rest of the paper, power

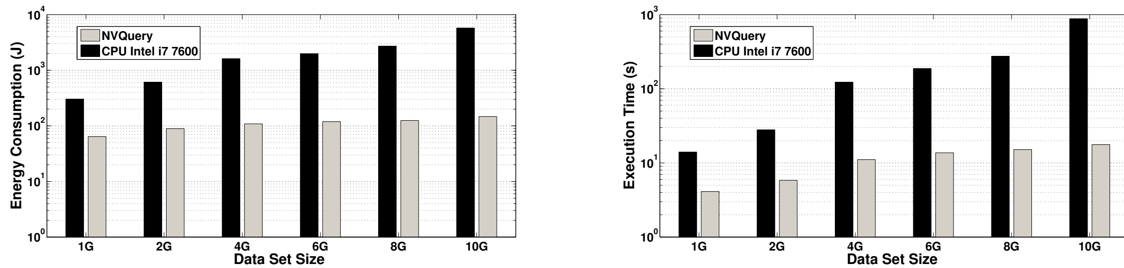


Figure 6.6. Energy consumption and performance of query processing running on traditional core and the proposed NVQuery.

and performance results have been reported for 1000 queries from aggregation and prediction functions over five randomly generated datasets. Join operation uses a different dataset, as the previously described dataset is not ideal for join based operations (need more than one column for join). Dataset includes 6 columned tables, randomly populated. Size of the table ranges from 2^2 to 2^{17}. The upper limit on the size is a function of the maximum datasheet size in MS Excel (2^{20}) and realistic join compute times.

6.4.2 NVQuery Efficiency

Here we highlight the advantage that NVQuery can provide in computing each query function. Table 6.5 compares the energy savings and performance speedup of running different queries on proposed NVQuery as compared to a digital ASIC design. Each energy is reported when 10 queries run on 1k dataset. The selected dataset is small so that the reported values compare the computation energy without data movement cost. The digital system is designed using System Verilog in 45nm ASIC flow. The result shows that NVQuery improves the computation cost of all queries significantly. Specifically, queries such as `MAX, MIN` and/or `TOP k` can be processed in a single cycle, instead of processing in $O(n)$ or $O(logn)$ time. Our evaluation shows that our design can provide 11.8× energy improvement and 26.85× performance speedup on average compared to digital approach for nearest distance search-based queries. Similarly, our design can achieve on average 13.7× and 92.1× (5.8× and 0.9×) energy savings and performance speedup over exact search (memory functionalities, e.g. addition).

Table 6.5. Energy Consumption and Performance Speedup of Queries in NVQuery Normalized to Digital Design over 1k Data

	Nearest search		**Search**	**Memory**	
Queries	*MAX/ MIN*	*Top 1*	*Search/ Count*	*Addition/ Average*	*Bit-wise*
Energy Improv.	9.5×	14.1×	13×	5.8×	46.7×
Speedup	24.2×	29.5×	92.1×	0.9×	122.6×

Table 6.6. Energy-Delay Product Improvement of SAQ, DAQ and Proposed NVQuery

Query Accelerators		**0%**	**1%**	**2%**	**4%**	**6%**	**8%**	**10%**
SAQ [152]	Error bound	0%	1.5%	3.1%	5.2%	7.4%	8.5%	10.9%
	EDP Improv.	4.1×	6.7×	8.3×	11.8×	17.2×	24.4×	39.8×
DAQ [105]	trimmed bits	0-bit	1-bit	2-bit	4-bit	6-bit	7-bit	9-bit
	EDP Improv.	16.4×	24.2×	36.9×	52.1×	69.2×	85.5×	104.5×
NVQuery	Relaxed bits	0-bit	2-bit	4-bit	6-bit	8-bit	11-bit	15-bit
	EDP Improv.	431×	505×	807×	1,515×	2,288×	2,587×	3,154×

NVQuery is also efficient in executing different join operations. For a small table with 2^2 rows, NVQuery provides 2.3x speedup and 5.9x energy savings on average as compared to conventional systems. However, the performance and energy efficiency of NVQuery increases with table size. For example, for a table with 2^{15} rows, NVQuery provides speedup and energy efficiency improvement of 21x and 83x respectively. Although, the performance of in-memory addition is less than that of digital-based design, but considering the cost of data movement, it makes sense to process data locally in-memory. In large size query processing, the data movement dominates the computation cost, which motivates us to perform in-memory computations to avoid data movement issue.

6.4.3 NVQuery & Dataset Size

While running real dataset, the main advantage of NVQuery comes from addressing the data movement issue. Fig. 6.6 shows the average energy consumption and performance of running query processing on traditional core and NVQuery when the data set size changes from 1GB to 10GB. Our evaluation shows that the NVQuery has an advantage in processing

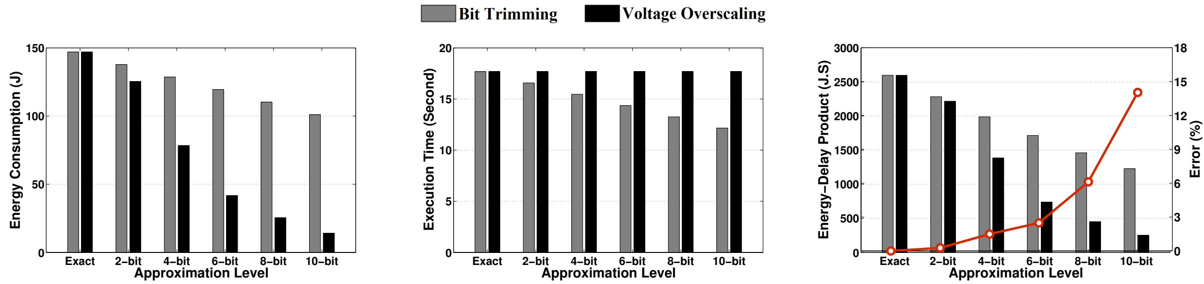


Figure 6.7. Energy consumption and performance of the NVQuery at different approximation levels.

the nearest distance search and related functions such as `MIN`, `MAX` or `Top` queries. However, to see the average NVQuery improvement, we generate the same number of queries running on the dataset. Our evaluation shows that increasing the data size significantly increases the energy and execution time of traditional cores. However, this increment is minor in NVQuery as it can locally process the data. As our result in Table 6.5 shows, NVQuery not only avoids the overhead of data movement, but also provides much cheaper computation than traditional cores. This difference is more prominent when the size of the dataset passes 8GB, which is the available main memory size in our tested platform. In such case, the traditional cores require to bring data up from the hard disk, which significantly slows down the computation. Comparing the energy and performance of NVQuery for 10G data shows that, our design can achieve 34.7× energy savings and 49.3× performance speedup as compared to traditional processor running the same query tasks.

6.4.4 NVQuery Approximation

Fig. 6.7 shows the energy, performance and energy-delay product, when NVQuery has been approximated using bit trimming and voltage scaling. The x-axis in the graph shows the number of relaxed bits. In addition, the red line in EDP graph shows the average relative error of query processing at different levels of approximation. Although the latency remains constant in the case of approximation by voltage scaling, it can achieve much higher efficiency than bit trimming. Our evaluation shows that NVQuery approximation using bit trimming and voltage

scaling can provide 490.7× and 507.9× EDP improvement as compared to NVQuery in exact mode while ensuring less than 0.2% average relative error. The efficiency of the voltage scaling approximation becomes more significant in deep approximation. For instance, while accepting 2% error, approximation by voltage scaling can achieve 45.0× and 17.6× energy savings and speedup (807× EDP improvement).

We also compare the efficiency of the proposed NVQuery with the state-of-the-art approximate query accelerators SAQ [152] and DAQ [105] using 8G dataset size. The NVQuery and DAQ approximation is defined based on the number of blocks under voltage overscaling and the number of least significant bits neglected respectively. In SAQ the error rate is determined based on the requested error bound. Table 6.6 shows the energy-delay product (EDP) improvement of the different query accelerators as compared to traditional CPU core when the level of approximation changes from 0% to 10%. For each error rate, we select those configurations of SAQ and DAQ which result in the best EDP improvement. As Table 6.6 shows, increasing the number of relaxed bits improves the energy consumption of our design. Our experimental evaluation shows that, NVQuery can achieve 105.0× and 26.2× EDP improvement as compared to SAQ and DAQ designs in exact mode. The main advantage of NVQuery comes from addressing data movement issue. The NVQuery can provide higher efficiency when it works in approximate mode, since our memory-based design put a larger portion of memory under voltage overscaling in order to achieve the same error rate as DAQ design. In other words, when DAQ neglects m-bits for accelerating query processing, our memory-based design can get the same accuracy by putting larger portion of memory blocks under voltage overscaling (shown in Table V for 4-bit stage size). Our evaluation shows that in approximate mode, NVQuery can achieve 79.2× and 30.1× EDP improvement as compared to SAQ and DAQ respectively, while providing similar error rate.

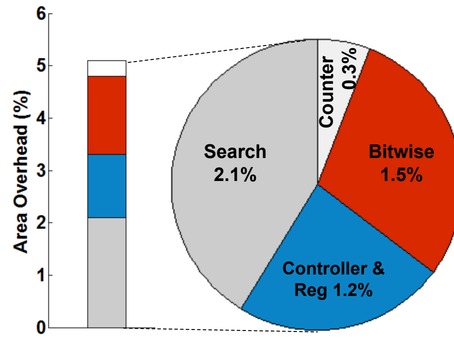


Figure 6.8. Area overhead as compared to conventional crossbar memory

6.4.5 Area Overhead

NVQuery has both memory and query processing functionalities. We added peripheral circuitry to crossbar memory to support nearest distance exact search operation, bit-wise/addition operations, counter and controller. Fig. 6.8 shows that proposed NVQuery has up to 5.1% area overhead compared to the conventional crossbar. The search circuitry takes 2.1% extra area. Counter and bit-wise circuits add 0.3% and 1.5% area overhead to design. Finally, the controller and registers take the rest 1.2% area overhead.

Chapter 6, in part, is a reprint of the material as it appears in M. Imani, S. Gupta, A. Arredondo, T. Rosing, "Efficient Query Processing in Crossbar Memory," *Proc. IEEE/ACM International Symposium on Low Power Electronics and Design*, 2017, pp. 1-6 and M. Imani, S. Gupta, S. Sharma, T. Rosing, "NVQuery: Efficient Query Processing in Non-Volatile Memory," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2018.

Chapter 7

Conclusion

This thesis explored the hardware aspect of PIM at various levels. It began with a basic implementation of PIM focused on individual operations such as bitwise computations, addition, and multiplication. Chapter 3 proposed a novel architecture to optimize these basic operations for data intensive workloads. It exploits the inherent parallelism in PIM operations while introducing simple hardware changes which significantly enhance the performance and efficiency of in-memory processing of large amount of data. Chapter 4 then integrated the architecture proposed in Chapter 3 with a general purpose core to design a hybrid system. It also introduces data management schemes to ensure maximum performance. Chapters 5 and 6 use this hybrid system to design application-specific accelerators. The proposed architectures achieve orders of magnitude of improvement in both performance and energy efficiency.

However, there are opportunities to further improve PIM systems. On one side, there is a need for appropriate software interface (including a PIM optimized programming model, execution model, and mapping schemes) to provide support for automated porting of code to new PIM architectures. On the other side, there is scope for improving the performance and reliability of individual PIM operations and the overall architecture. Research in these directions would help in emergence of PIM as the next big computing paradigm.

Bibliography

[1] D. Reinsel, J. Gantz, and J. Rydning, "Data age 2025: The evolution of data to life-critical dont focus on big data; focus on data thats big," *IDC, Seagate, April*, 2017.

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

[3] I. H. Witten, E. Frank, M. A. Hall, and C. J. Pal, *Data Mining: Practical machine learning tools and techniques*. Morgan Kaufmann, 2016.

[4] J. Ahn, S. Hong, S. Yoo, O. Mutlu, and K. Choi, "A scalable processing-in-memory accelerator for parallel graph processing," in *Computer Architecture (ISCA), 2015 ACM/IEEE 42nd Annual International Symposium on*, pp. 105–117, IEEE, 2015.

[5] W. Dally, "High-performance hardware for machine learning," *NIPS Tutorial*, 2015.

[6] K. Q. Weinberger, J. Blitzer, and L. K. Saul, "Distance metric learning for large margin nearest neighbor classification," in *Advances in neural information processing systems*, pp. 1473–1480, 2006.

[7] Y. Deng and B. Manjunath, "Content-based search of video using color, texture, and motion," in *Image Processing, 1997. Proceedings., International Conference on*, vol. 2, pp. 534–537, IEEE, 1997.

[8] X. Yin, A. Aziz, J. Nahas, S. Datta, S. Gupta, M. Niemier, and X. S. Hu, "Exploiting ferroelectric fets for low-power non-volatile logic-in-memory circuits," in *Proceedings of the 35th International Conference on Computer-Aided Design*, p. 121, ACM, 2016.

[9] J. Sim, M. Imani, W. Choi, Y. Kim, and T. Rosing, "Lupis: Latch-up based ultra efficient processing in-memory system," in *Quality Electronic Design (ISQED), 2018 19th International Symposium on*, pp. 55–60, IEEE, 2018.

[10] A. Shafiee, A. Nag, N. Muralimanohar, R. Balasubramonian, J. P. Strachan, M. Hu, R. S. Williams, and V. Srikumar, "Isaac: A convolutional neural network accelerator with in-situ analog arithmetic in crossbars," in *Proceedings of the 43rd International Symposium on Computer Architecture*, pp. 14–26, IEEE Press, 2016.

[11] S. H. Pugsley, J. Jestes, R. Balasubramonian, V. Srinivasan, A. Buyuktosunoglu, A. Davis, and F. Li, "Comparing implementations of near-data computing with in-memory mapreduce workloads," *IEEE Micro*, vol. 34, no. 4, pp. 44–52, 2014.

[12] M. Imani, Y. Kim, and T. Rosing, "Nngine: Ultra-efficient nearest neighbor accelerator based on in-memory computing," in *Rebooting Computing (ICRC), 2017 IEEE International Conference on*, pp. 1–8, IEEE, 2017.

[13] Y. Kim, M. Imani, and T. Rosing, "Orchard: Visual object recognition accelerator based on approximate in-memory processing," in *Computer-Aided Design (ICCAD), 2017 IEEE/ACM International Conference on*, pp. 25–32, IEEE, 2017.

[14] R. Balasubramonian, J. Chang, T. Manning, J. H. Moreno, R. Murphy, R. Nair, and S. Swanson, "Near-data processing: Insights from a micro-46 workshop," *IEEE Micro*, vol. 34, no. 4, pp. 36–42, 2014.

[15] M. Saremi, S. Rajabi, H. J. Barnaby, and M. N. Kozicki, "The effects of process variation on the parametric model of the static impedance behavior of programmable metallization cell (pmc)," *MRS Online Proceedings Library Archive*, vol. 1692, 2014.

[16] S. N. Mozaffari, S. Tragoudas, and T. Haniotakis, "More efficient testing of metal-oxide memristor–based memory," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 36, no. 6, pp. 1018–1029, 2017.

[17] S. Salehi and R. F. DeMara, "Process variation immune and energy aware sense amplifiers for resistive non-volatile memories," in *Circuits and Systems (ISCAS), 2017 IEEE International Symposium on*, pp. 1–4, IEEE, 2017.

[18] M. Saremi, "A physical-based simulation for the dynamic behavior of photodoping mechanism in chalcogenide materials used in the lateral programmable metallization cells," *Solid State Ionics*, vol. 290, pp. 1–5, 2016.

[19] B. Pourshirazi and Z. Zhu, "Refree: A refresh-free hybrid dram/pcm main memory system," in *Parallel and Distributed Processing Symposium, 2016 IEEE International*, pp. 566–575, IEEE, 2016.

[20] M. K. Tavana, A. K. Ziabari, and D. Kaeli, "Live together or die alone: block cooperation to extend lifetime of resistive memories," in *Proceedings of the Conference on Design, Automation & Test in Europe*, pp. 1098–1103, European Design and Automation Association, 2017.

[21] S. Salehi, N. Khoshavi, and R. F. Demara, "Mitigating process variability for non-volatile cache resilience and yield," *IEEE Transactions on Emerging Topics in Computing*, 2018.

[22] C. Liu, Q. Yang, C. Zhang, H. Jiang, Q. Wu, and H. H. Li, "A memristor-based neuromorphic engine with a current sensing scheme for artificial neural network applications," in *Design Automation Conference (ASP-DAC), 2017 22nd Asia and South Pacific*, pp. 647–652, IEEE, 2017.

[23] S. Kvatinsky, D. Belousov, S. Liman, G. Satat, N. Wald, E. G. Friedman, A. Kolodny, and U. C. Weiser, "Magicmemristor-aided logic," *IEEE Transactions on Circuits and Systems II: Express Briefs*, vol. 61, no. 11, pp. 895–899, 2014.

[24] J. Borghetti, G. S. Snider, P. J. Kuekes, J. J. Yang, D. R. Stewart, and R. S. Williams, "Memristive switches enable stateful logic operations via material implication," *Nature*, vol. 464, no. 7290, pp. 873–876, 2010.

[25] 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.

[26] S. Shirinzadeh, M. Soeken, P.-E. Gaillardon, and R. Drechsler, "Fast logic synthesis for rram-based in-memory computing using majority-inverter graphs," in *Proceedings of the 2016 Conference on Design, Automation & Test in Europe*, pp. 948–953, EDA Consortium, 2016.

[27] P. Chi, S. Li, C. Xu, T. Zhang, J. Zhao, Y. Liu, Y. Wang, and Y. Xie, "Prime: A novel processing-in-memory architecture for neural network computation in reram-based main memory," in *Proceedings of the 43rd International Symposium on Computer Architecture*, pp. 27–39, IEEE Press, 2016.

[28] Y. Sun, Y. Wang, and H. Yang, "Energy-efficient sql query exploiting rram-based process-in-memory structure," in *Non-Volatile Memory Systems and Applications Symposium (NVMSA), 2017 IEEE 6th*, pp. 1–6, IEEE, 2017.

[29] L. Song, Y. Zhuo, X. Qian, H. Li, and Y. Chen, "Graphr: Accelerating graph processing using reram," 2018.

[30] R. Fackenthal, M. Kitagawa, W. Otsuka, K. Prall, D. Mills, K. Tsutsui, J. Javanifard, K. Tedrow, T. Tsushima, Y. Shibahara, and G. Hush, "19.7 a 16gb reram with 200mb/s write and 1gb/s read in 27nm technology," in *Solid-State Circuits Conference Digest of Technical Papers (ISSCC), 2014 IEEE International*, pp. 338–339, IEEE, 2014.

[31] T. y. Liu, T. H. Yan, R. Scheuerlein, Y. Chen, J. K. Lee, G. Balakrishnan, G. Yee, H. Zhang, A. Yap, J. Ouyang, T. Sasaki, A. Al-Shamma, C. Chen, M. Gupta, G. Hilton, A. Kathuria, V. Lai, M. Matsumoto, A. Nigam, A. Pai, J. Pakhale, C. H. Siau, X. Wu, Y. Yin, N. Nagel, Y. Tanaka, M. Higashitani, T. Minvielle, C. Gorla, T. Tsukamoto, T. Yamaguchi, M. Okajima, T. Okamura, S. Takase, H. Inoue, and L. Fasoli, "A 130.7-$hboxmm^2$ 2-layer 32-gb reram memory device in 24-nm technology," *IEEE Journal of Solid-State Circuits*, vol. 49, no. 1, pp. 140–153, 2014.

[32] Y. Choi, I. Song, M. H. Park, H. Chung, S. Chang, B. Cho, J. Kim, Y. Oh, D. Kwon, J. Sunwoo, J. Shin, Y. Rho, C. Lee, M. G. Kang, J. Lee, Y. Kwon, S. Kim, J. Kim, Y. J. Lee, Q. Wang, S. Cha, S. Ahn, H. Horii, J. Lee, K. Kim, H. Joo, K. Lee, Y. T. Lee,

J. Yoo, and G. Jeong, "A 20nm 1.8 v 8gb pram with 40mb/s program bandwidth," in *Solid-State Circuits Conference Digest of Technical Papers (ISSCC), 2012 IEEE International*, pp. 46–48, IEEE, 2012.

[33] "Everspin announces sampling of the worlds first 1-gigabit mram product." https://www.everspin.com/sites/default/files/pressdocs/Everspin_Announces_Customer_Sampling_1Gb.pdf.

[34] J. Borghetti, Z. Li, J. Straznicky, X. Li, D. A. Ohlberg, W. Wu, D. R. Stewart, and R. S. Williams, "A hybrid nanomemristor/transistor logic circuit capable of self-programming," *Proceedings of the National Academy of Sciences*, vol. 106, no. 6, pp. 1699–1703, 2009.

[35] A. C. Torrezan, J. P. Strachan, G. Medeiros-Ribeiro, and R. S. Williams, "Sub-nanosecond switching of a tantalum oxide memristor," *Nanotechnology*, vol. 22, no. 48, p. 485203, 2011.

[36] J. J. Yang, D. B. Strukov, and D. R. Stewart, "Memristive devices for computing," *Nature nanotechnology*, vol. 8, no. 1, p. 13, 2013.

[37] J. J. Yang, M.-X. Zhang, J. P. Strachan, F. Miao, M. D. Pickett, R. D. Kelley, G. Medeiros-Ribeiro, and R. S. Williams, "High switching endurance in tao x memristive devices," *Applied Physics Letters*, vol. 97, no. 23, p. 232102, 2010.

[38] J. Nickel, "Memristor materials engineering: From flash replacement towards a universal memory," in *Proceedings of the IEEE IEDM Advanced Memory Technology Workshop*, pp. 1–3, 2011.

[39] A. Flocke and T. G. Noll, "Fundamental analysis of resistive nano-crossbars for the use in hybrid nano/cmos-memory," in *Solid State Circuits Conference, 2007. ESSCIRC 2007. 33rd European*, pp. 328–331, IEEE, 2007.

[40] K.-H. Kim, S. Gaba, D. Wheeler, J. M. Cruz-Albrecht, T. Hussain, N. Srinivasa, and W. Lu, "A functional hybrid memristor crossbar-array/cmos system for data storage and neuromorphic applications," *Nano letters*, vol. 12, no. 1, pp. 389–395, 2011.

[41] Y. Yang, P. Sheridan, and W. Lu, "Complementary resistive switching in tantalum oxide-based resistive memory devices," *Applied Physics Letters*, vol. 100, no. 20, p. 203112, 2012.

[42] M. A. Lastras-Montaño, A. Ghofrani, and K.-T. Cheng, "A low-power hybrid reconfigurable architecture for resistive random-access memories," in *High Performance Computer Architecture (HPCA), 2016 IEEE International Symposium on*, pp. 102–113, IEEE, 2016.

[43] D. E. Culler, J. P. Singh, and A. Gupta, *Parallel computer architecture: a hardware/software approach*. Gulf Professional Publishing, 1999.

[44] D. Patterson, T. Anderson, N. Cardwell, R. Fromm, K. Keeton, C. Kozyrakis, R. Thomas, and K. Yelick, "Intelligent ram (iram): Chips that remember and compute," in *Solid-State Circuits Conference, 1997. Digest of Technical Papers. 43rd ISSCC., 1997 IEEE International*, pp. 224–225, IEEE, 1997.

[45] J. Torrellas, "Flexram: Toward an advanced intelligent memory system: A retrospective paper," in *Computer Design (ICCD), 2012 IEEE 30th International Conference on*, pp. 3–4, IEEE, 2012.

[46] K. Mai, T. Paaske, N. Jayasena, R. Ho, W. J. Dally, and M. Horowitz, "Smart memories: A modular reconfigurable architecture," *ACM SIGARCH Computer Architecture News*, vol. 28, no. 2, pp. 161–171, 2000.

[47] S. Khoram, Y. Zha, J. Zhang, and J. Li, "Challenges and opportunities: From near-memory computing to in-memory computing," in *Proceedings of the 2017 ACM on International Symposium on Physical Design*, pp. 43–46, ACM, 2017.

[48] Q. Zhu, B. Akin, H. E. Sumbul, F. Sadi, J. C. Hoe, L. Pileggi, and F. Franchetti, "A 3d-stacked logic-in-memory accelerator for application-specific data intensive computing," in *3D Systems Integration Conference (3DIC), 2013 IEEE International*, pp. 1–7, IEEE, 2013.

[49] S. H. Pugsley, J. Jestes, H. Zhang, R. Balasubramonian, V. Srinivasan, A. Buyuktosunoglu, A. Davis, and F. Li, "Ndc: Analyzing the impact of 3d-stacked memory+ logic devices on mapreduce workloads," in *Performance Analysis of Systems and Software (ISPASS), 2014 IEEE International Symposium on*, pp. 190–200, IEEE, 2014.

[50] S. Kumar, A. Shriraman, V. Srinivasan, D. Lin, and J. Phillips, "Sqrl: Hardware accelerator for collecting software data structures," in *Parallel Architecture and Compilation Techniques (PACT), 2014 23rd International Conference on*, pp. 475–476, IEEE, 2014.

[51] A. Farmahini-Farahani, J. H. Ahn, K. Morrow, and N. S. Kim, "Nda: Near-dram acceleration architecture leveraging commodity dram devices and standard memory modules," in *High Performance Computer Architecture (HPCA), 2015 IEEE 21st International Symposium on*, pp. 283–295, IEEE, 2015.

[52] B. Y. Cho, W. S. Jeong, D. Oh, and W. W. Ro, "Xsd: Accelerating mapreduce by harnessing the gpu inside an ssd," in *Proceedings of the 1st Workshop on Near-Data Processing*, 2013.

[53] A. De, M. Gokhale, R. Gupta, and S. Swanson, "Minerva: Accelerating data analysis in next-generation ssds," in *Field-Programmable Custom Computing Machines (FCCM), 2013 IEEE 21st Annual International Symposium on*, pp. 9–16, IEEE, 2013.

[54] Q. Guo, X. Guo, Y. Bai, and E. Ipek, "A resistive tcam accelerator for data-intensive computing," in *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, pp. 339–350, ACM, 2011.

[55] J. Chang, P. Ranganathan, T. Mudge, D. Roberts, M. A. Shah, and K. T. Lim, "A limits study of benefits from nanostore-based future data-centric system architectures," in *Proceedings of the 9th conference on Computing Frontiers*, pp. 33–42, ACM, 2012.

[56] Q. Guo, X. Guo, R. Patel, E. Ipek, and E. G. Friedman, "Ac-dimm: associative computing with stt-mram," in *ACM SIGARCH Computer Architecture News*, vol. 41, pp. 189–200, ACM, 2013.

[57] Y. Wang, Y. Han, L. Zhang, H. Li, and X. Li, "Propram: exploiting the transparent logic resources in non-volatile memory for near data computing," in *Proceedings of the 52nd Annual Design Automation Conference*, p. 47, ACM, 2015.

[58] M. G. Farooq, T. L. Graves-Abe, W. F. Landers, C. Kothandaraman, B. A. Himmel, P. S. Andry, C. K. Tsang, E. Sprogis, R. P. Volant, K. S. Petrarca, K. R. Winstel, J. M. Safran, T. D. Sullivan, F. Chen, M. J. Shapiro, R. Hannon, R. Liptak, D. Berger, and S. S. Iyer, "3d copper tsv integration, testing and reliability," in *Electron Devices Meeting (IEDM), 2011 IEEE International*, pp. 7–1, IEEE, 2011.

[59] Y. Liu, W. Luk, and D. Friedman, "A compact low-power 3d i/o in 45nm cmos," in *Solid-State Circuits Conference Digest of Technical Papers (ISSCC), 2012 IEEE International*, pp. 142–144, IEEE, 2012.

[60] D. U. Lee, K. W. Kim, K. W. Kim, H. Kim, J. Y. Kim, Y. J. Park, J. H. Kim, D. S. Kim, H. B. Park, J. W. Shin, J. H. Cho, K. H. Kwon, M. J. Kim, J. Lee, K. W. Park, B. Chung, and S. Hong, "25.2 a 1.2 v 8gb 8-channel 128gb/s high-bandwidth memory (hbm) stacked dram with effective microbump i/o test methods using 29nm process and tsv," in *Solid-State Circuits Conference Digest of Technical Papers (ISSCC), 2014 IEEE International*, pp. 432–433, IEEE, 2014.

[61] J. Jeddeloh and B. Keeth, "Hybrid memory cube new dram architecture increases density and performance," in *VLSI Technology (VLSIT), 2012 Symposium on*, pp. 87–88, IEEE, 2012.

[62] W. A. Wulf and S. A. McKee, "Hitting the memory wall: implications of the obvious," *ACM SIGARCH computer architecture news*, vol. 23, no. 1, pp. 20–24, 1995.

[63] G. Dai, T. Huang, Y. Chi, J. Zhao, G. Sun, Y. Liu, Y. Wang, Y. Xie, and H. Yang, "Graphh: A processing-in-memory architecture for large-scale graph processing," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2018.

[64] G. H. Loh, N. Jayasena, M. Oskin, M. Nutter, D. Roberts, M. Meswani, D. P. Zhang, and M. Ignatowski, "A processing in memory taxonomy and a case for studying fixed-function pim," in *Workshop on Near-Data Processing (WoNDP)*, 2013.

[65] A. M. Aly, A. Sallam, B. M. Gnanasekaran, L.-V. Nguyen-Dinh, W. G. Aref, M. Ouzzani, and A. Ghafoor, "M3: Stream processing on main-memory mapreduce," in *2012 IEEE 28th International Conference on Data Engineering*, pp. 1253–1256, IEEE, 2012.

[66] M. Kang, M.-S. Keel, N. R. Shanbhag, S. Eilert, and K. Curewitz, “An energy-efficient vlsi architecture for pattern recognition via deep embedding of computation in sram,” in *Acoustics, Speech and Signal Processing (ICASSP), 2014 IEEE International Conference on*, pp. 8326–8330, IEEE, 2014.

[67] M. Kang, E. P. Kim, M.-s. Keel, and N. R. Shanbhag, “Energy-efficient and high throughput sparse distributed memory architecture,” in *Circuits and Systems (ISCAS), 2015 IEEE International Symposium on*, pp. 2505–2508, IEEE, 2015.

[68] S. Jeloka, N. B. Akesh, D. Sylvester, and D. Blaauw, “A 28 nm configurable memory (tcam/bcam/sram) using push-rule 6t bit cell enabling logic-in-memory,” *IEEE Journal of Solid-State Circuits*, vol. 51, no. 4, pp. 1009–1021, 2016.

[69] S. Aga, S. Jeloka, A. Subramaniyan, S. Narayanasamy, D. Blaauw, and R. Das, “Compute caches,” in *High Performance Computer Architecture (HPCA), 2017 IEEE International Symposium on*, pp. 481–492, IEEE, 2017.

[70] J. Zhang, Z. Wang, and N. Verma, “In-memory computation of a machine-learning classifier in a standard 6t sram array.,” *J. Solid-State Circuits*, vol. 52, no. 4, pp. 915–924, 2017.

[71] R. Nair, S. F. Antao, C. Bertolli, P. Bose, J. R. Brunheroto, T. Chen, C. Y. Cher, C. H. A. Costa, J. Doi, C. Evangelinos, B. M. Fleischer, T. W. Fox, D. S. Gallo, L. Grinberg, J. A. Gunnels, A. C. Jacob, P. Jacob, H. M. Jacobson, T. Karkhanis, C. Kim, J. H. Moreno, J. K. O’Brien, M. Ohmacht, Y. Park, D. A. Prener, B. S. Rosenburg, K. D. Ryu, O. Sallenave, M. J. Serrano, P. D. M. Siegl, K. Sugavanam, and Z. Sura, “Active memory cube: A processing-in-memory architecture for exascale systems,” *IBM Journal of Research and Development*, vol. 59, no. 2/3, pp. 17–1, 2015.

[72] S. Li, D. Niu, K. T. Malladi, H. Zheng, B. Brennan, and Y. Xie, “Drisa: A dram-based reconfigurable in-situ accelerator,” in *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*, pp. 288–301, ACM, 2017.

[73] V. Seshadri, D. Lee, T. Mullins, H. Hassan, A. Boroumand, J. Kim, M. A. Kozuch, O. Mutlu, P. B. Gibbons, and T. C. Mowry, “Ambit: In-memory accelerator for bulk bitwise operations using commodity dram technology,” in *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*, pp. 273–287, ACM, 2017.

[74] V. Seshadri, K. Hsieh, A. Boroum, D. Lee, M. A. Kozuch, O. Mutlu, P. B. Gibbons, and T. C. Mowry, “Fast bulk bitwise and and or in dram,” *IEEE Computer Architecture Letters*, vol. 14, no. 2, pp. 127–131, 2015.

[75] T. Finkbeiner, G. Hush, T. Larsen, P. Lea, J. Leidel, and T. Manning, “In-memory intelligence,” *IEEE Micro*, vol. 37, no. 4, pp. 30–38, 2017.

[76] V. Seshadri, D. Lee, T. Mullins, H. Hassan, A. Boroumand, J. Kim, M. A. Kozuch, O. Mutlu, P. B. Gibbons, and T. C. Mowry, “Buddy-ram: Improving the performance and efficiency of bulk bitwise operations using dram,” *arXiv preprint arXiv:1611.09988*, 2016.

[77] L. Yavits, S. Kvatinsky, A. Morad, and R. Ginosar, "Resistive associative processor," *IEEE Computer Architecture Letters*, vol. 14, no. 2, pp. 148–151, 2015.

[78] M. N. Bojnordi and E. Ipek, "Memristive boltzmann machine: A hardware accelerator for combinatorial optimization and deep learning," in *High Performance Computer Architecture (HPCA), 2016 IEEE International Symposium on*, pp. 1–13, IEEE, 2016.

[79] M. Hu, H. Li, Q. Wu, and G. S. Rose, "Hardware realization of bsb recall function using memristor crossbar arrays," in *Proceedings of the 49th Annual Design Automation Conference*, pp. 498–503, ACM, 2012.

[80] B. Li, Y. Shan, M. Hu, Y. Wang, Y. Chen, and H. Yang, "Memristor-based approximated computation," in *Proceedings of the 2013 International Symposium on Low Power Electronics and Design*, pp. 242–247, IEEE Press, 2013.

[81] Y. Kim, Y. Zhang, and P. Li, "A reconfigurable digital neuromorphic processor with memristive synaptic crossbar for cognitive computing," *ACM Journal on Emerging Technologies in Computing Systems (JETC)*, vol. 11, no. 4, p. 38, 2015.

[82] M. Prezioso, F. Merrikh-Bayat, B. Hoskins, G. Adam, K. K. Likharev, and D. B. Strukov, "Training and operation of an integrated neuromorphic network based on metal-oxide memristors," *Nature*, vol. 521, no. 7550, p. 61, 2015.

[83] N. Talati, S. Gupta, P. Mane, and S. Kvatinsky, "Logic design within memristive memories using memristor-aided logic (magic)," *IEEE Transactions on Nanotechnology*, vol. 15, no. 4, pp. 635–650, 2016.

[84] S. Li, C. Xu, Q. Zou, J. Zhao, Y. Lu, and Y. Xie, "Pinatubo: A processing-in-memory architecture for bulk bitwise operations in emerging non-volatile memories," in *Design Automation Conference (DAC), 2016 53nd ACM/EDAC/IEEE*, pp. 1–6, IEEE, 2016.

[85] M. Imani, Y. Kim, and T. Rosing, "Mpim: Multi-purpose in-memory processing using configurable resistive memory," in *Design Automation Conference (ASP-DAC), 2017 22nd Asia and South Pacific*, pp. 757–763, IEEE, 2017.

[86] A. Siemon, S. Menzel, R. Waser, and E. Linn, "A complementary resistive switch-based crossbar array adder," *IEEE journal on emerging and selected topics in circuits and systems*, vol. 5, no. 1, pp. 64–74, 2015.

[87] L. Cavigelli, D. Bernath, M. Magno, and L. Benini, "Computationally efficient target classification in multispectral image data with deep neural networks," in *Target and Background Signatures II*, vol. 9997, p. 99970L, International Society for Optics and Photonics, 2016.

[88] C. Clark and A. Storkey, "Training deep convolutional neural networks to play go," in *International Conference on Machine Learning*, pp. 1766–1774, 2015.

[89] K. Srinivas, B. K. Rani, and A. Govrdhan, "Applications of data mining techniques in healthcare and prediction of heart attacks," *International Journal on Computer Science and Engineering (IJCSE)*, vol. 2, no. 02, pp. 250–255, 2010.

[90] T. Mikolov, M. Karafiát, L. Burget, J. Černockỳ, and S. Khudanpur, "Recurrent neural network based language model," in *Eleventh Annual Conference of the International Speech Communication Association*, 2010.

[91] A. Biswas and A. P. Chandrakasan, "Conv-ram: An energy-efficient sram with embedded convolution computation for low-power cnn-based machine learning applications," in *Solid-State Circuits Conference-(ISSCC), 2018 IEEE International*, pp. 488–490, IEEE, 2018.

[92] S. K. Gonugondla, M. Kang, and N. Shanbhag, "A 42pj/decision 3.12 tops/w robust in-memory machine learning classifier with on-chip training," in *Solid-State Circuits Conference-(ISSCC), 2018 IEEE International*, pp. 490–492, IEEE, 2018.

[93] W. S. Khwa, J. J. Chen, J. F. Li, X. Si, E. Y. Yang, X. Sun, R. Liu, P. Y. Chen, Q. Li, S. Yu, and M. F. Chang, "A 65nm 4kb algorithm-dependent computing-in-memory sram unit-macro with 2.3 ns and 55.8 tops/w fully parallel product-sum operation for binary dnn edge processors," in *Solid-State Circuits Conference-(ISSCC), 2018 IEEE International*, pp. 496–498, IEEE, 2018.

[94] L. Jiang, M. Kim, W. Wen, and D. Wang, "Xnor-pop: A processing-in-memory architecture for binary convolutional neural networks in wide-io2 drams," in *Low Power Electronics and Design (ISLPED, 2017 IEEE/ACM International Symposium on*, pp. 1–6, IEEE, 2017.

[95] D. Kim, J. Kung, S. Chai, S. Yalamanchili, and S. Mukhopadhyay, "Neurocube: A programmable digital neuromorphic architecture with high-density 3d memory," in *Computer Architecture (ISCA), 2016 ACM/IEEE 43rd Annual International Symposium on*, pp. 380–392, IEEE, 2016.

[96] L. Song, X. Qian, H. Li, and Y. Chen, "Pipelayer: A pipelined reram-based accelerator for deep learning," HPCA, 2017.

[97] S. Tang, S. Yin, S. Zheng, P. Ouyang, F. Tu, L. Yao, J. Wu, W. Cheng, L. Liu, and S. Wei, "Aepe: An area and power efficient rram crossbar-based accelerator for deep cnns," in *Non-Volatile Memory Systems and Applications Symposium (NVMSA), 2017 IEEE 6th*, pp. 1–6, IEEE, 2017.

[98] M. Cheng, L. Xia, Z. Zhu, Y. Cai, Y. Xie, Y. Wang, and H. Yang, "Time: A training-in-memory architecture for memristor-based deep neural networks," in *Design Automation Conference (DAC), 2017 54th ACM/EDAC/IEEE*, pp. 1–6, IEEE, 2017.

[99] J. Gubbi, R. Buyya, S. Marusic, and M. Palaniswami, "Internet of things (iot): A vision, architectural elements, and future directions," *Future generation computer systems*, vol. 29, no. 7, pp. 1645–1660, 2013.

[100] M. Chen, S. Mao, and Y. Liu, "Big data: A survey," *Mobile Networks and Applications*, vol. 19, no. 2, pp. 171–209, 2014.

[101] F. Imani, B. Yao, R. Chen, P. Rao, and H. Yang, "Factal pattern recognition of image profiles for manufacturing process monitoring and control," in *International Manufacturing Science and Engineering Conference*, p. 1, ASME, 2018.

[102] H. Chen, R. H. Chiang, and V. C. Storey, "Business intelligence and analytics: From big data to big impact.," *MIS quarterly*, vol. 36, no. 4, 2012.

[103] Y. Chen, S. Alspaugh, and R. Katz, "Interactive analytical processing in big data systems: A cross-industry study of mapreduce workloads," *Proceedings of the VLDB Endowment*, vol. 5, no. 12, pp. 1802–1813, 2012.

[104] B. Yao, F. Imani, A. S. Sakpal, E. Reutzel, and H. Yang, "Multifractal analysis of image profiles for the characterization and detection of defects in additive manufacturing," *Journal of Manufacturing Science and Engineering*, vol. 140, no. 3, p. 031014, 2018.

[105] N. Potti and J. M. Patel, "Daq: a new paradigm for approximate query processing," *Proceedings of the VLDB Endowment*, vol. 8, no. 9, pp. 898–909, 2015.

[106] Y. Sun, Y. Wang, and H. Yang, "Bidirectional database storage and sql query exploiting rram-based process-in-memory structure," *ACM Transactions on Storage (TOS)*, vol. 14, no. 1, p. 8, 2018.

[107] M. Imran, C. Castillo, F. Diaz, and S. Vieweg, "Processing social media messages in mass emergency: A survey," *ACM Computing Surveys (CSUR)*, vol. 47, no. 4, p. 67, 2015.

[108] C. Unsalan and B. Sirmacek, "Road network detection using probabilistic and graph theoretical methods," *IEEE Transactions on Geoscience and Remote Sensing*, vol. 50, no. 11, pp. 4441–4453, 2012.

[109] E. Bullmore and O. Sporns, "Complex brain networks: graph theoretical analysis of structural and functional systems," *Nature reviews. Neuroscience*, vol. 10, no. 3, p. 186, 2009.

[110] A. Ching, S. Edunov, M. Kabiljo, D. Logothetis, and S. Muthukrishnan, "One trillion edges: Graph processing at facebook-scale," *Proceedings of the VLDB Endowment*, vol. 8, no. 12, pp. 1804–1815, 2015.

[111] S. Salihoglu and J. Widom, "Gps: A graph processing system," in *Proceedings of the 25th International Conference on Scientific and Statistical Database Management*, p. 22, ACM, 2013.

[112] J. E. Gonzalez, R. S. Xin, A. Dave, D. Crankshaw, M. J. Franklin, and I. Stoica, "Graphx: Graph processing in a distributed dataflow framework.," in *OSDI*, vol. 14, pp. 599–613, 2014.

[113] A. Lumsdaine, D. Gregor, B. Hendrickson, and J. Berry, "Challenges in parallel graph processing," *Parallel Processing Letters*, vol. 17, no. 01, pp. 5–20, 2007.

[114] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski, "Pregel: a system for large-scale graph processing," in *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, pp. 135–146, ACM, 2010.

[115] P. Kanerva, "Hyperdimensional computing: An introduction to computing in distributed representation with high-dimensional random vectors," *Cognitive Computation*, vol. 1, no. 2, pp. 139–159, 2009.

[116] O. Rasanen and S. Kakouros, "Modeling dependencies in multiple parallel data streams with hyperdimensional computing," *IEEE Signal Processing Letters*, vol. 21, no. 7, pp. 899–903, 2014.

[117] A. Rahimi, P. Kanerva, and J. M. Rabaey, "A robust and energy-efficient classifier using brain-inspired hyperdimensional computing," in *Proceedings of the 2016 International Symposium on Low Power Electronics and Design*, pp. 64–69, ACM, 2016.

[118] M. Imani, D. Kong, A. Rahimi, and T. Rosing, "Voicehd: Hyperdimensional computing for efficient speech recognition," in *Rebooting Computing (ICRC), 2017 IEEE International Conference on*, pp. 1–8, IEEE, 2017.

[119] M. Imani, J. Hwang, T. Rosing, A. Rahimi, and J. M. Rabaey, "Low-power sparse hyperdimensional encoder for language recognition," *IEEE Design & Test*, vol. 34, no. 6, pp. 94–101, 2017.

[120] M. Imani, T. Nassar, A. Rahimi, and T. Rosing, "Hdna: Energy-efficient dna sequencing using hyperdimensional computing," in *Biomedical & Health Informatics (BHI), 2018 IEEE EMBS International Conference on*, pp. 271–274, IEEE, 2018.

[121] M. Imani, A. Rahimi, D. Kong, T. Rosing, and J. M. Rabaey, "Exploring hyperdimensional associative memory," in *2017 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pp. 445–456, IEEE, 2017.

[122] V. Gupta, D. Mohapatra, A. Raghunathan, and K. Roy, "Low-power digital signal processing using approximate adders," *TCAD*, vol. 32, no. 1, pp. 124–137, 2013.

[123] R. Ubal, B. Jang, P. Mistry, D. Schaa, and D. Kaeli, "Multi2sim: a simulation framework for cpu-gpu computing," in *Proceedings of the 21st international conference on Parallel architectures and compilation techniques*, pp. 335–344, ACM, 2012.

[124] S. Kvatinsky, M. Ramadan, E. G. Friedman, and A. Kolodny, "Vteam: A general model for voltage-controlled memristors," *IEEE Transactions on Circuits and Systems II: Express Briefs*, vol. 62, no. 8, pp. 786–790, 2015.

[125] "Caltech Library." http://www.vision.caltech.edu/Image_Datasets/Caltech101/.

[126] M. Imani, S. Gupta, and T. Rosing, "Ultra-efficient processing in-memory for data intensive applications," in *Proceedings of the 54th Annual Design Automation Conference 2017*, p. 6, ACM, 2017.

[127] A. Siemon, S. Menzel, R. Waser, and E. Linn, "A complementary resistive switch-based crossbar array adder," *IEEE journal on emerging and selected topics in circuits and systems*, vol. 5, no. 1, pp. 64–74, 2015.

[128] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood, "The gem5 simulator," *ACM SIGARCH Computer Architecture News*, vol. 39, no. 2, pp. 1–7, 2011.

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

[130] A. Krizhevsky and G. Hinton, "Convolutional deep belief networks on cifar-10," *Unpublished manuscript*, vol. 40, p. 7, 2010.

[131] A. Reiss and D. Stricker, "Creating and benchmarking a new dataset for physical activity monitoring," in *Proceedings of the 5th International Conference on PErvasive Technologies Related to Assistive Environments*, p. 40, ACM, 2012.

[132] C. Gentry, *A fully homomorphic encryption scheme*. Stanford University, 2009.

[133] C. Peikert, "Lattice cryptography for the internet," in *International Workshop on Post-Quantum Cryptography*, pp. 197–219, Springer, 2014.

[134] P. Bestagini, A. Allam, S. Milani, M. Tagliasacchi, and S. Tubaro, "Video codec identification," in *Acoustics, Speech and Signal Processing (ICASSP), 2012 IEEE International Conference on*, pp. 2257–2260, IEEE, 2012.

[135] G. Cybenko, "Approximation by superpositions of a sigmoidal function," *Mathematics of control, signals and systems*, vol. 2, no. 4, pp. 303–314, 1989.

[136] R. Bernstein, "Multiplication by integer constants," *Software: practice and experience*, vol. 16, no. 7, pp. 641–652, 1986.

[137] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay, "Scikit-learn: Machine learning in python," *Journal of Machine Learning Research*, vol. 12, no. Oct, pp. 2825–2830, 2011.

[138] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. J. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Józefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, R. Monga, S. Moore, D. G. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. A.

Tucker, V. Vanhoucke, V. Vasudevan, F. B. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng, "Tensorflow: Large-scale machine learning on heterogeneous distributed systems," *arXiv preprint arXiv:1603.04467*, 2016.

[139] Y. LeCun, C. Cortes, and C. J. Burges, "The mnist database of handwritten digits," 1998.

[140] "Uci machine learning repository." http://archive.ics.uci.edu/ml/datasets/ISOLET.

[141] "Uci machine learning repository." https://archive.ics.uci.edu/ml/datasets/UJIIndoorLoc.

[142] "Uci machine learning repository." https://archive.ics.uci.edu/ml/datasets/Daily+and+Sports+Activities.

[143] "The cifar dataset." https://www.cs.toronto.edu/~kriz/cifar.html.

[144] I. Sutskever, J. Martens, G. E. Dahl, and G. E. Hinton, "On the importance of initialization and momentum in deep learning.," *ICML (3)*, vol. 28, pp. 1139–1147, 2013.

[145] N. Srivastava, G. E. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, "Dropout: a simple way to prevent neural networks from overfitting.," *Journal of Machine Learning Research*, vol. 15, no. 1, pp. 1929–1958, 2014.

[146] Y. Chen, T. Luo, S. Liu, S. Zhang, L. He, J. Wang, L. Li, T. Chen, Z. Xu, N. Sun, and O. Temam, "Dadiannao: A machine-learning supercomputer," in *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*, pp. 609–622, IEEE Computer Society, 2014.

[147] X. Yin, M. Niemier, and X. S. Hu, "Design and benchmarking of ferroelectric fet based tcam," in *DATE*, pp. 1444–1449, IEEE, 2017.

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

[149] V. Gupta, D. Mohapatra, S. P. Park, A. Raghunathan, and K. Roy, "Impact: imprecise adders for low-power approximate computing," in *Proceedings of the 17th IEEE/ACM international symposium on Low-power electronics and design*, pp. 409–414, IEEE Press, 2011.

[150] M. Imani, S. Patil, and T. S. Rosing, "Masc: Ultra-low energy multiple-access single-charge tcam for approximate computing," in *2016 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pp. 373–378, IEEE, 2016.

[151] M. Imani, A. Rahimi, P. Mercati, and T. S. Rosing, "Multi-stage tunable approximate search in resistive associative memory," *IEEE Transactions on Multi-Scale Computing Systems*, 2017.

[152] S. Agarwal, B. Mozafari, A. Panda, H. Milner, S. Madden, and I. Stoica, “Blinkdb: queries with bounded errors and bounded response times on very large data,” in *Proceedings of the 8th ACM European Conference on Computer Systems*, pp. 29–42, ACM, 2013.