Rochester Institute of Technology
RIT Digital Institutional Repository

Theses

6-2021

Securing in-memory processors against Row Hammering Attacks

Sahil K. Gogna
sxg4060@rit.edu

Follow this and additional works at: https://repository.rit.edu/theses

Recommended Citation
Gogna, Sahil K., "Securing in-memory processors against Row Hammering Attacks" (2021). Thesis. Rochester Institute of Technology. Accessed from

This Thesis is brought to you for free and open access by the RIT Libraries. For more information, please contact repository@rit.edu.

Securing in-memory processors against Row Hammering Attacks

Sahil K. Gogna

Securing in-memory processors against Row Hammering Attacks

Sahil K. Gogna

June 2021

A Thesis Submitted
in Partial Fulfillment
of the Requirements for the Degree of
Master of Science
in
Computer Engineering

RIT | Kate Gleason College of Engineering

Department of Computer Engineering

Securing in-memory processors against Row Hammering Attacks

Sahil K. Gogna

Committee Approval:

Dr. Amlan Ganguly *Advisor* — Date
Department of Computer Engineering

Dr. Cory Merkel *Co-Advisor* — Date
Department of Computer Engineering

Mr. Mark Indovina *Co-Advisor* — Date
Department of Electrical Engineering

Acknowledgments

I would like to take this opportunity to thank Dr. Amlan Ganguly for guiding me throughout my academic career; without his involvement, I would not have reached this point and I am ever grateful for the support and guidance he has given me to succeed in this environment. In addition, I also want to express my gratitude in always establishing a positive and joyous environment in which not only I thrive in, but other students can thrive in.

I would also like to thank my colleagues Purab Suthradhar, Mark Connolly, Prangon Das, and Sayed Ashraf for helping and brainstorming alongside me to grow and develop in the environment, allowing me to understand complex and unique problems that affect the modern world today.

In addition, I would like to thank my parents, Mr. Pardeep Kumar and Mrs. Alpana Gogna, who have always supported me and given me the tools to succeed. Without them, I would not have been able to garner and achieve a higher level of education. They have ultimately shaped who I am today and have lead me to a path of success up to this point.

Finally, I would like to thank all the students and faculty I have met along the way. They have all helped and inspired me to reach this point of my education. They gave me the confidence to believe in myself to succeed and they deserve the credit as well.

I dedicate this to my parents, Mr. Pardeep Kumar and Mrs. Alpana Gogna. Thank you for always guiding and helping me be a better me everyday.

Abstract

Modern applications on general purpose processors require both rapid and power-efficient computing and memory components. As applications continue to improve, the demand for high speed computation, fast-access memory, and a secure platform increases. Traditional Von Neumann Architectures split the computing and memory units, causing both latency and high power-consumption issues; henceforth, a hybrid memory processing system is proposed, known as in-memory processing. In-memory processing alleviates the delay of computation and minimizes power-consumption; such improvements saw a 14x speedup improvement, 87% fewer power consumption, and appropriate linear scalability versus performance. Several applications of in-memory processing include data-driven applications such as Artificial Intelligence (AI), Convolutional and Deep Neural Networks (CNNs/DNNs). However, processing-in-memory can also suffer from a security and reliability issue known as the Row Hammer Security Bug; this security exploit flips bits within memory without access, leading to error injection, system crashes, privilege separation, and total hijack of a system; the novel Row Hammer security bug can negatively impact the accuracies of CNNs and DNNs via flipping the bits of stored weight values without direct access. Weights of neural networks are stored in a variety of data patterns, resulting in either a solid (all 1s or all 0s), checkered (alternating 1s and 0s in both rows and columns), row-stripe (alternating 1s and 0s in rows), or column-striped (alternating 1s and 0s in columns) manner; the row-stripe data pattern exhibits the largest likelihood of a Row Hammer attack, resulting in the accuracies of neural networks dropping over 30%. A row-stripe avoidance coding scheme is proposed to reduce the probability of the Row Hammer Attack occurring within neural networks. The coding scheme encodes the binary portion of a weight in a CNN or DNN to reduce the chance of row-stripe data patterns, overall reducing the likelihood of a Row Hammer attack occurring while improving the overall security of the in-memory processing system.

Contents

Signature Sheet **i**

Acknowledgments **ii**

Dedication **iii**

Abstract **iv**

Table of Contents **v**

List of Figures **vii**

List of Tables **1**

1 Introduction **2**

1.1 Processing-in-Memory (PIM) 2
1.2 Row Hammering (RH) Security Attack 5
1.3 The Effects of Row Hammer Attack on In-Memory Processing 10

2 Background **14**

2.1 In-memory Processing 14
2.2 Row-Hammering 21

3 Vulnerability Analysis of a Programmable In-memory Architecture **30**

3.1 Programmable In-memory Processing core 31
3.2 Programmable In-memory Processing Cluster 32
3.3 Programmable In-memory Processing Router 33
3.4 Programmable In-memory Processing Function Word Generation . . 34
3.5 In-memory Processing Multiply and Accumulate Operations 34
3.6 In-memory Processing Activation Operations 35
3.7 Field Programmable Gate Array Implementation 36
3.7.1 Technology Resources 37
3.7.2 Machine Learning Model 37
3.7.3 In-memory processing Implementation 37
3.7.4 Communication 38
3.8 Results 39

3.8.1 pPIM characteristics 39
3.8.2 Performance Evaluation 40
3.8.3 Vulnerability Analysis of In-memory Processing Architectures 42

4 Row Hammer Reducing Encoding Scheme 45
4.1 In-memory Data Mapping 46
4.2 Encoding Connectivity 46
4.3 Encoding Scheme 47
4.3.1 Alternative Encoding Schemes 48
4.3.2 Comparisons with Error-Correction Codes 50
4.3.3 Comparisons to logic/counter based approaches 51
4.4 Operation Mapping 53
4.5 Results 54
4.5.1 Encoding and pPIM architecture characteristics 54
4.5.2 Encoding Functionality 56
4.5.3 Encoding/Decoding Performance Evaluation 57
4.5.4 Pipelined RH-encoding scheme 58
4.6 DRAM Design Analysis 60

5 Conclusion and Future Work 63
5.1 Future Work 65

Bibliography 67

List of Figures

1.1 In-memory Processing Architecture 4
1.2 3D Integration Stack [1] 5
1.3 Row Hammering Example 6
1.4 Row Hammering Data Pattern 8
1.5 Row Hammering Data Pattern Statistics [2] 8
1.6 Attack and Defense Model 12

2.1 Hybrid Memory Cube Integration Stack 17
2.2 Single Tesseract Core 18
2.3 Message Triggered Prefetching 20
2.4 Weight Sensitivity of MLP and LeNet with Iris and MNIST Datasets 26
2.5 TWiCe Architecture 28

3.1 Hierarchical view of the pPIM Architecture 30
3.2 Layout of the pPIM core architecture [3] 31
3.3 Layout of the pPIM cluster architecture [3] 32
3.4 Interconnect Router Architecture for n cores in a cluster [4] 33
3.5 MAC Operation Multiplication and Accumulation Stages 35
3.6 Dataflow model within a pPIM cluster for both (a) 8-bit full precision and (b) 4-bit half precision unsigned MAC operations [3] 35
3.7 Dataflow Model within a pPIM cluster for both (a) 16-bit precision and (b) 8-bit precision ReLU activation operations [3] 36
3.8 Dataflow Model within a pPIM cluster for both (a) 16-bit precision and (b) 8-bit precision Saturated ReLU activation operations [3] . . . 36
3.9 FPGA MAC and Classification Regions within FPGA Implementation of pPIM architecture 38
3.10 Communication Model between ZC702 Evaluation Board and End Device 39
3.11 Comparison of (a) throughput and (b) power consumption of various architectures 40
3.12 Comparison of (a) area and (b) efficiency/area of various architectures 41
3.13 Weight Vulnerability Heatmap on the MNIST dataset 43

4.1 Proposed Encoding Scheme for handling RH attacks 45

4.2 Model of the interface between the RH-encoding scheme with pPIM Clusters and memory elements 47
4.3 Original Encoding Scheme 49
4.4 Dual 3-to-4 Encoding Scheme 49
4.5 Uncorrectable multi-bit errors (in bold) [2] 51
4.6 Sequential model of both encoding and decoding operations on the pPIM architecture including (a) encoding sequential model and (b) decoding sequential model. The inputs **'a'**, **'b'**, and **'c'** indicate input to the cluster from memory with the high and low segments of memory dictated by the subscripts **'H'** and **'L'**, respectively. The input and the output of the encoding and decoding schemes are dictated by **'A'** for encoding and **'U'** for decoding. 55
4.7 Encoding Functionality Comparison of (a) AlexNet, and (b) VGG-16, with and without encoding 57
4.8 Comparison of (a) throughput and (b) power consumption of various CNN algorithms with and without RH-encoding scheme 58
4.9 Pipelined Sequential model of both encoding and decoding operations on the pPIM architecture including (a) encoding sequential model and (b) decoding sequential model. The inputs **'a'**, **'b'**, and **'c'** indicate input to the cluster from memory with the high and low segments of memory dictated by the subscripts **'H'** and **'L'**, respectively. The input and the output of the encoding and decoding schemes are dictated by **'A'** for encoding and **'U'** for decoding. 59
4.10 Comparison of throughput of various CNN algorithms with and without pipelined RH-encoding scheme 61
4.11 DRAM Coupling Capacitance vs. Number of Row Hammer Attacks . 62

List of Tables

3.1 Synthesis Results for pPIM Architecture 39

4.1 Row Hammer Encoding Truth Table Samples 48
4.2 Read Core Functionalities 54
4.3 Synthesis Results 56
4.4 Pipelined Synthesis Results 60

Chapter 1

Introduction

1.1 Processing-in-Memory (PIM)

Traditional von Neumann Architectures consist of a central processing unit (CPU) and memory unit separated via interconnects; data is transferred between these units in which one involves computation and the other storage. Interconnects are responsible for transporting data from one unit to the other to allow the architecture to function properly. As the technology node improves, interconnects have higher amounts of resistivity, capacitance, and inductance; in addition, the rate at which processing speed is improving is significantly higher than the rate at which memory speed is improving. The constant transfer of data over interconnects results in large latency and high power-consumption, leading to the proposal of an in-memory processing unit.

A limitation of the von Neumann Architecture involves the rates at which both processing speed and memory are improving. Computing power has sped up in the recent years whereas fetching data from memory has lagged behind, instead, focusing on memory density; processors are improving at a rapid pace but are trivial if data is not fetched at an appropriate rate. Moore's Law exhibits that a gap of about 50% between computing and memory power is growing per year, creating a memory wall scenario [5]. As applications continue to improve in parallel with the technology node, the demand for both higher computational and memory performance increases. Data-

driven applications such as artificial intelligence and brain-inspired computing utilize CNNs and DNNs which require intense and exhaustive computations; these processes include video or image processing purposes and utilize linear algebra multiplication of vectors and matrices. Data is continuously offloaded from and written to memory with CNN and DNN operations, incrementing latency.

Another issue that deals with von Neumann Architectures includes the resistivity, capacitance, and inductance of interconnects [6]. Resistivity increases as a result of the length and thinness of the wire as the technology node scales; in addition, metal lines run over thick oxide covering the substrate, resulting in both higher levels of resistance and capacitance. Capacitance increases as results of area, lateral, and fringe capacitance within wire dimensions. Furthermore, Miller's Effect, which involves coupling capacitance creates additional capacitance based on signal values in adjacent wires. If signals do not switch in the same direction amongst adjacent wires, additional or parasitic capacitance is tacked onto to the base capacitance of interconnect, resulting in higher capacitance and overall higher delay and power-consumption. As the technology node improves, interconnects get thinner, resulting in fewer resistance but larger inductance, increasing frequency and the use of coppers in wires [1]. These three factors positively correlate with latency and power consumption; moreover, the usage of CNNs and DNNs involve millions of operations in which data is continuously transported over interconnects, causing a greater amount of latency and power-consumption added to the traditional von Neumann Architecture.

In-memory processing is a proposed solution to the memory and interconnect bottlenecks exhibited in contemporary Von Neumann Architectures, as operating within or near memory reduces both the distance data has to travel from and to memory along with quicker data fetching. Processing in or near memory utilizes accelerators, or small cores or processors on memory, typically DRAM, allowing logic to occur within the memory unit [5]. In the fabrication process, in-memory processors

and memory are closely coupled, in which a machine has several processors that share the same amount of memory, consequently improving the memory transfer rate and memory bandwidth while both latency and power-consumption can be minimized. Processing-in-memory (PIM) is a quite innovative idea that explores the union of both memory and processing units, decreasing latency and reducing power consumption. Figure 1.1 illustrates a type of architecture of in-memory processing.

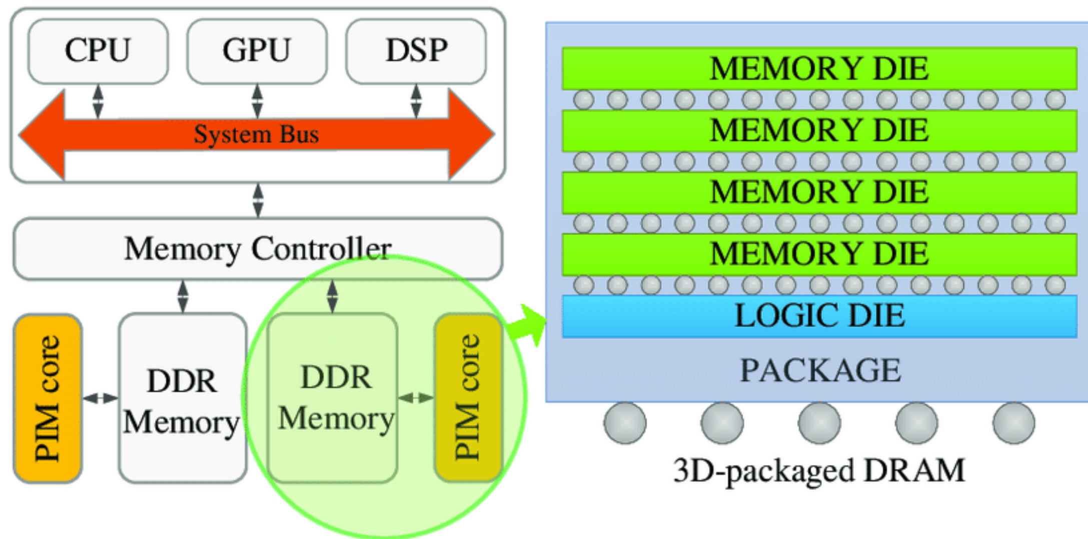

Figure 1.1: In-memory Processing Architecture

Some novel forms of interconnects may be used to serve a purpose within in-memory processing such as 3D, photonic, and on-chip wireless interconnects. 3D Integration is an emerging interconnect that involves stacking chips, creating multiple layers; this saves both space and power consumption within an integrated chip [7]. However, heat dissipation is a common problem within 3D Integration since chips are stacked closely and densely as possible. Figure 1.2 shows a 3D Integration stack example.

In addition to 3D integration, photonic networks utilize electronic packet-switched networks to operate network on chip (NOC) architectures, which is typically used and integrated via 3D integration, further reducing power-consumption. However, ongoing

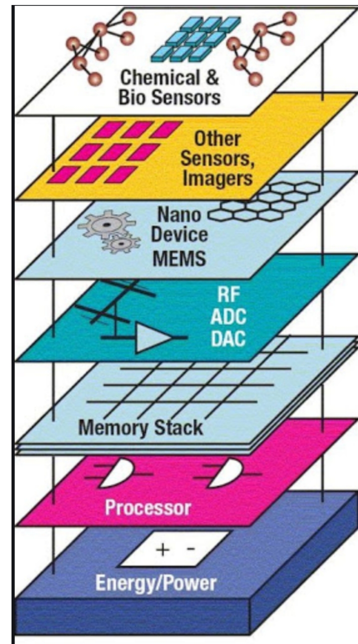

Figure 1.2: 3D Integration Stack [1]

research is required to investigate photonic NOCs [8]. On-chip wireless interconnects utilize wireless or RF signals to communicate parts of a processing unit without interconnects via transmission lines. Although interconnect-less, long transmission line routing may cause latency issues and signal drops through transmission. Several of these solutions can be implemented to form an in-memory processing unit.

1.2 Row Hammering (RH) Security Attack

As the technology node scales, the separation between memory cells decreases, forcing rows of memory to be closer to one other. As rows are continually accessed, the charge within cells of certain rows can leak due to continual pre-charge and activation of the memory cell capacitors, flipping bits within cells of adjacent rows without accessing them [2] .

Row-hammering (RH) occurs between two consecutive refreshes of DRAM; this is due to capacitive interference between two adjacent memory rows [9]. Whenever a row is switched or accessed frequently, the capacitive interference causes the cells in

adjacent rows to slightly charge or discharge. When repeatedly toggling/accessing a certain row, due to how close the rows are (typically below 35 nm) and how charge can easily jump due to electrical interference, adjacent rows can be activated without access due to charge affecting their wordlines (electromagnetic coupling) [2]. Using this, a complete attack model can be designed in which the attacker program continuously and repetitively activates and precharges a certain row(s) of DRAM. This repetitive action eventually alters the values stored in the cells of adjacent rows (upper or lower). Figure 1.3 exhibits a double-sided row hammer attack.

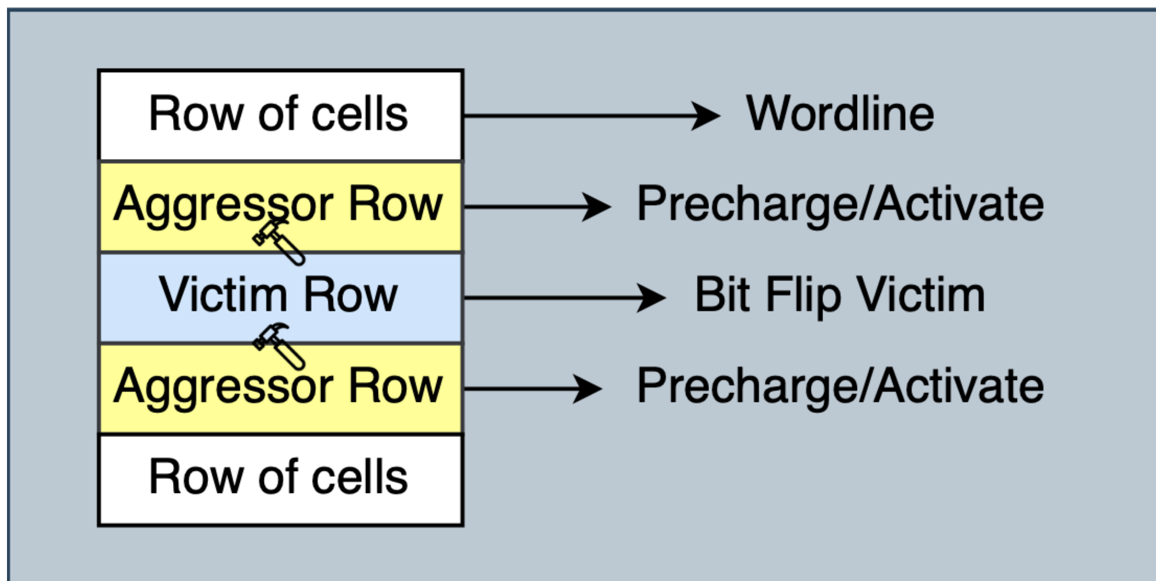

Figure 1.3: Row Hammering Example

In addition to Figure 1.3, a simple x86 Assembly Code, shown below also simulates the Row Hammer Attack as it can only occur in the same bank of memory [2].

```
code1a:
    mov (X), %eax   // read from address X
    mov (Y), %ebx   // read from address Y
    clflush (X)     // flush cache for address X
    clflush (Y)     // flush cache for address Y
    mfence
```

```
    jmp code1a
```

The code above alternates the accesses to two different rows (but on the same bank of memory); in addition, memory read is not from the same row and the content of the row cannot be used. Since an activation to both rows occur via the 'mov' commands are right after one another, a Row Hammer Attack can occur. However, alternating the code below such as below will not simulate a Row Hammer Attack [2].

```
code1b:
    mov (X), %eax  // read from address X
    clflush (X)    // flush cache for address X
    mfence
    jmp code1b
```

The code above does not simulate the Row Hammer attack because there is no access to another row in memory; the data will be served from the row buffer in memory since there is no need to reactivate the row; therefore, the Row Hammer attack only occurs via activation, not accessibility. In addition, the Row Hammer attack occurs within the refresh window, typically 64 ms, only in the same bank of memory but different rows [2].

In commodity DRAM chips, around 9600 row activations are required to simulate the RH attack [10]; the bug can attack via two methods: single-sided and double-sided; the effects of RH attacks are similar to those of cross-talk effects displayed between interconnects [11]. In the double-sided case, rowhammer occurs via continuous activation and pre-charge of the two outer rows, causing bits flips in the middle row or the inner or middle row is the aggressor row, causing bits to flip without being accessed in the above or below rows. In the single-sided case, rowhammer occurs via only one aggressor row and one victim row, with one aggressor row 'hammering' the victim row. Attackers can either have the information of the virtual to psychical

mapping of the system (Whitebox RH Attack), allowing for a less brute-force tactic and higher chance of successful data corruption or they may not have the mapping, which may also corrupt data in a more forceful manner (Blackbox RH Attack) [12].

Furthermore, the rowhammer security bug has a data pattern dependence, indicating that the organization of data in DRAM is critical to the security of the system. Four data patterns exist: Solid, ColStripe, RowStripe, and Checkered. Figure 1.4 below displays the four data patterns [2].

Solid	RowStripe	ColStripe	Checkered
111111	111111	101010	101010
111111	000000	101010	010101
111111	111111	101010	101010
111111	000000	101010	010101

Figure 1.4: Row Hammering Data Pattern

Amongst all data patterns, the RowStripe pattern exhibits the worst-case scenario for rowhammer attacks. In 2014, commodity DRAM were put under the rowhammer attack with all four data pattern dependencies to indicate which data pattern had the highest number of bit flips [2]. Figure 1.5 below shows the results of the rowhammer attack on all four data pattern dependencies on three manufacturers of DRAM [2].

Module	TestBulk(*DP*) + TestBulk(∼*DP*)			
	Solid	RowStripe	ColStripe	Checkered
A_{23}	112,123	**1,318,603**	763,763	934,536
B_{11}	12,050	**320,095**	9,610	302,306
C_{19}	57	20,770	130	**29,283**

Figure 1.5: Row Hammering Data Pattern Statistics [2]

Row-hammering induces errors such as privilege separation, system crashes, a

hijack of control of the system, and error injection. Google has actually exploited the poor scaling of DRAM via the Row Hammer bug to gain kernel privileges by repeatedly accessing rows causing bit flips; such privileges can be given to the wrong user, potentially causing a reverse-engineering based security issue.

Inspired by [13], researchers from Google Project Zero demonstrated that row hammer attacks can be exploited at the user-level to gain kernel privileges on real systems. One exploit involved running a Native Client (NaCl) program and escalated privilege to escape the x86-64 sandbox environment. Another exploit runs as a normal x86-64 process on Linux and escalates privilege to gain access to all of the physical memory mapping and hijack the entire system. The attacker hammers a page table entry (PTE) in memory. This changes the PTE to point to a page table owned by the attacking process. This gives the attacking process full read-write access to its own page table and hence to all of physical memory, which enables the attacking process to take over the entire system.

Drammer investigates the vulnerability of mobile devices to the RH attack [14]. Drammer investigates the effects of row hammer attacks on ARM-based mobile devices. [14] takes advantage of the deterministic memory allocation patterns in the Android Linux Operating System. Via deterministic memory allocation patterns, a methodology is proposed for forcing a victim process to allocate its PTE in a RH-vulnerable region of memory. This involves the attacker process to allocate all possible memory regions for a page table allocation and then release the page table allocation that contains the RH-vulnerable DRAM cells at bit offsets that enable exploitation. [14] discovered 18 out of 27 phone models to be vulnerable to the RH attack and have since released a mobile application that tests memory for RH-vulnerable cells and aggregates statistics on how widespread the RH phenomenon is on mobile devices. This work shows that existing mobile systems are widely vulnerable to RH attacks [14].

[15] involves RH attacks that includes hijacking mobile systems by triggering the RH attack using the WebGL interface on a mobile GPU, takeover of a remote system by triggering the attack through the Remote Direct Memory Access (RDMA) protocol and various other attacks [15]. The RH attack has widespread and profound real implications on system security, as it breaks memory isolation on top of which modern system security principles are built.

Various solutions such as counter and probabilistic based approaches, along with new additional components, have been taken to address row-hammering issues. Counter-based approaches involve placing counters along each DRAM row to monitor the amount of activations until a threshold has been met [16]; probabilistic solutions involves auto-refresh techniques within DRAM such that the topmost entry is reset to its original value regularly. Pruning is also a technique in which the aggressor rows are recorded in a table and the least aggressive rows are pruned out continuously from the table to find the most aggressive row [17].

1.3 The Effects of Row Hammer Attack on In-Memory Processing

Security of data is essential to any architecture, network, or system of operations; recently, the novel row hammering security bug was discovered in which bits of data can be flipped without access [9]. The continuous pre-charge and activation of rows causes charge to propagate to adjacent rows and flip bits of data, causing host privilege escalation issues, incorrect qualitative/quantitative information, and possibilities of attacks on networks, including CNNs and DNNs [13].

Weights stored in both CNNs and DNNs are subject to security vulnerabilities such as the Row Hammer Attack, as they are stored within memory, typically DRAM; as the Row Hammer Attack flips bits of memory without access, the weights are vulner-

able to this, comprising memory information as well as manipulating the accuracy of a network.

An attack model composes of an attacker that either knows or does not know the virtual to physical memory mapping. If the attacker does not the know mapping, a brute-force tactic would occur, effectively pre-charging and activating random rows to damage the network. However, if the attacker does know virtual to physical memory mapping, with a knowledge of the data pattern dependency, the attack on the network can be severe as the structure in which the weights stored in memory can influence the manner in which the Row Hammer Attack can occur.

Weights can be stored in a plethora of ways in memory, either horizontally or vertically; however, the way they are stored can influence the likelihood of a Row Hammer Attack occurring on the network. The four data pattern dependencies as exhibited in Figure 1.4 shows all types of data patterns of weights. Out of the four data patterns, the RowStripe data pattern, which exhibits rows alternating in 1s and 0s, has a significantly highest probability of a Row Hammer Attack occurring as compared to other three data patterns [2]. Investigating the effects of the data patterns on the weights stored in the network can mitigate the effect of the Row Hammer Attack, particularly looking into the effects of the RowStripe data pattern. As the RowStripe data pattern has the highest likelihood of a Row Hammer Attack occurring on a network, minimizing the effects of this data pattern specifically can significantly reduce the chances of a Row Hammer attack affecting on a network.

Henceforth, we explore the the performance of a neural network along with the sensitivity of its layers. In addition to layer sensitivity, a weight vulnerability analysis is performed to determine which weights are subject to the RowStripe data pattern. A RowStripe Avoidance Coding (RAC) scheme is proposed to encode the weights in the network to remove the RowStripe data pattern in certain bit positions, specifically the most vulnerable positions of the weight. An attack and defense model is represented

below in Figure 1.6.

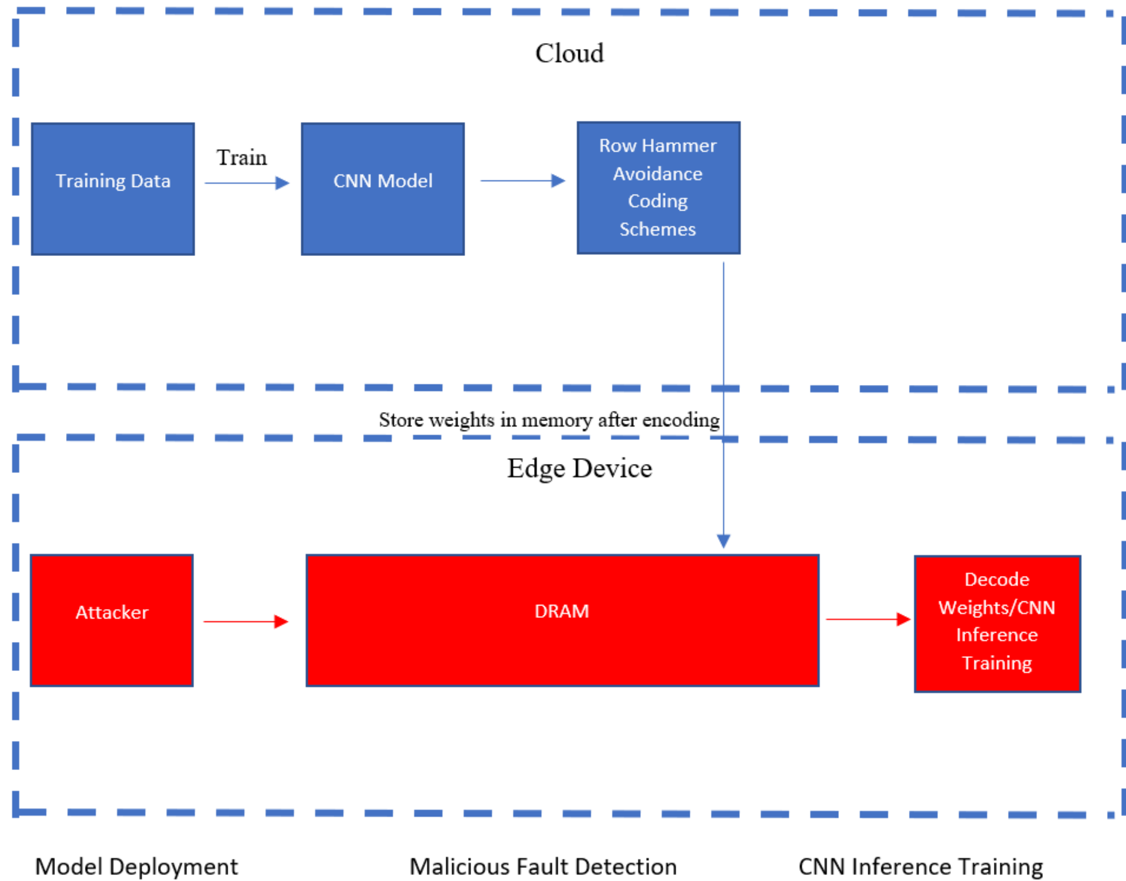

Figure 1.6: Attack and Defense Model

The attack and defense model shown in the figure below is to be implemented to avoid any RowStripe vulnerable data patterns in memory. The weights of a neural network are encoded in the cloud and then stored within memory such that the attacker cannot exploit the RowStripe vulnerable weights. Memory on the edge device will store the encoded weights with the RowStripe Avoidance Coding Schemes and decoding will be done within the cloud. The main contributions of this work are the design of an original and unique coding mechanism as well as a creation of a performance benchmark for the scheme to evaluate its effectiveness. In addition, an evaluation of the bit error rate as well as additional hardware is implemented to allow for proper data-flow of information securely, taking into consideration the

Rowhammer attack on networks.

Chapter 2

Background

2.1 In-memory Processing

In-memory processing is a potential solution to the von Neumann bottleneck, in which both speed and energy are compromised due to the use of interconnects. Recently, there has been an explorative effort on various PIM architectures.

LAcc is a look-up-table (LUT) based in-memory accelerator utilized to multiply data in a unique manner [18]; many designs of PIM for multiplication do exist yet many are not efficient as LAcc. LAcc utilizes multiplication decomposition as shown below in Equation 2.1.

$$A * B + C * D = (A * B_0 * 2^0 + A * B_1 * 2^1) + (C * D_0 * 2^0 + C * D_1 * 2^1) \qquad (2.1)$$

In the above equation, A and C are the addends whereas B and D are the select bits selected by the LUTs when performing multiplication decomposition; these calculations are used for CNN or DNN applications and are typically vector-based operations. The table size of the LUT depends on the bit-length of the values to be multiplied. To achieve proper multiplication decomposition and save LUT table space, LAcc utilizes horizontal and vertical partitioning to split the vectorized values to multiply. An

example of this is shown below in Equation 2.2.

$$\begin{pmatrix} 1 & 1 & 0 & 1 & 0 & 1 & 0 & 0 \end{pmatrix} \begin{pmatrix} 1 \\ 0 \\ 1 \\ 0 \\ 1 \\ 0 \\ 1 \\ 0 \end{pmatrix} = \begin{pmatrix} 1 & 1 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 \\ 0 \\ 1 \\ 0 \end{pmatrix} + \begin{pmatrix} 0 & 1 & 0 & 0 \end{pmatrix} \begin{pmatrix} 1 \\ 0 \\ 1 \\ 0 \end{pmatrix} \tag{2.2}$$

The LUT size for the equation above would be 2^8 ($2^4 * 2^4 = 2^8$) bits. Without horizontal or vertical partitioning, the LUT table size would have been 2^16 ($2^8 * 2^8 = 2^{16}$) bits, saving a significant amount of space. LAcc is a PIM DRAM accelerator that computes multiplicaton 6.3x fast than the current method of multiplication on traditional Von Neumann Architectures [18].

The computational ideas shown in [18] resonate with that of approximate computing, in which a dynamic bit-width is used to compute to the accuracy of a machine learning program (MLP) [19]. Using a shorter, or dynamic bit width, may aid in improving power-consumption and limit accuracy loss. In-memory processors can leverage the idea of having dynamic bit-widths to compute to further increase power-consumption savings as well as limit latency in calculations all while minimizing accuracy loss. Data can be split into segments, allowing for faster computation and minimal power-consumption, used in multiple in-memory processing architectures.

pPIM is a programmable PIM architecture that combines processing of machine learning networks such as CNNs and DNNs. CNNs and DNNs, such as AlexNet, use millions of multiplication operations to compute image and video processing [4]. To simplify the operations, partial products are made out of the multiplication op-

erations and all possible solutions are construed within a Lookup Table (LUT). For example, two four-bit values are to be multiplied; therefore, the total number of possible outcomes of the multiplication operations is 256, or $(2^4)^2$, since each value is of four bits in binary. The LUT would be populated with all potential outcomes, ranging from 0 to 255. The partial products of the two inputs would be created via multiple addition operations, reducing the cycles needed to complete each addition operation. Albeit there may be multiple additional operations within the core which may cause further delay, the reduction of interconnects and use of LUTs will not only compensate for the millions of operations, but it will also improve the performance as compared to previous Von Neumann Architectures.

The Hybrid Memory Cube (HMC) is a hybrid Dynamic Random Access Memory (DRAM) memory/logic structure created via 3D Integration in Very-large-scale integration (VLSI) [20] [1]. 3D Integration is an emerging interconnect that involves stacking chips, creating multiple layers; this saves both space and power consumption within an integrated chip [7]. However, heat dissipation is a common problem within 3D Integration since chips are stacked closely and densely as possible.

The Hybrid Memory Cube utilizes this architecture to communicate between memory and logic in a revolutionary, faster, and power-efficient manner; furthermore, the HMC is scalable, allowing for better bandwidth consumption as the system scales [1].

The Hybrid Memory Cube utilizes crossbar switching to communicate between cores, or typically called vaults, within its own architecture. Each vault, or partition, contains the logic implementation of the HMC, allowing for processing-in-memory to occur, Figure 2.2 shows the Hybrid Memory Cube Integration Stack.

Computation occurs within each partition, which is within each bank of DRAM chip. DRAM structure is composed of ranks, which is also known as a group of chips. Within each chip, there are multiple banks; each bank has rows and columns of memory cells, which are composed of a pass transistor and capacitor to store charge

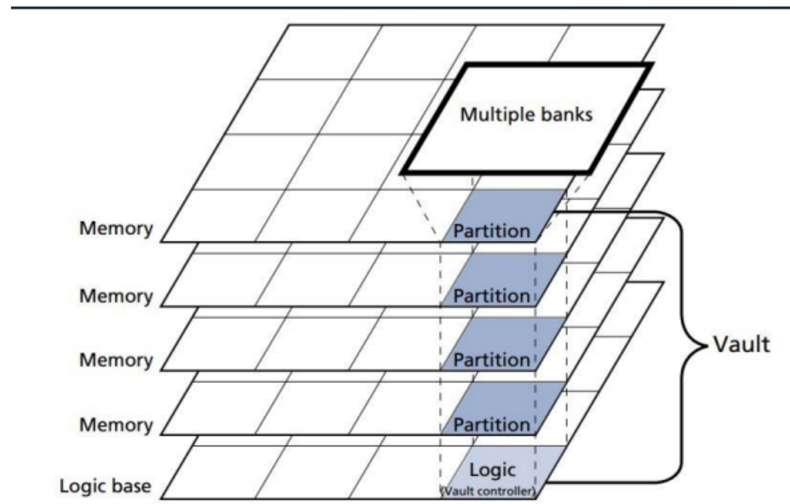

Figure 2.1: Hybrid Memory Cube Integration Stack

and access bit-/word-lines via charge stored in capacitors. Within the HMC's banks, there exist partitions where each processing unit lies, underneath the memory cells; this completes the layering of the HMC.

Tesseract is a programmable PIM architecture that combines processing of machine learning networks such as CNNs and DNNs within graphs, such as traversals and data collection of edges and vertices [21]. Tesseract utilizes the Hybrid Memory Cube Architecture via 3D Integration and its memory unit; however, it begins to differ when it comes to its logic unit. Unlike the HMC, which utilizes a simple logic system within its vault via multiple multiplexers over crossbar communication, Tesseract's processing unit contains a single-issue in-order core using message buffers and prefetching techniques in order to utilize memory bandwidth in an efficient manner. Furthermore, the programming interface allows for greater control over each core in a vault [1].

Tesseract is composed of 8 GB DRAM layers, eight 40 Gb/s high speed serial links, 32 slices (or vaults), and a single-issue in-order cube [21]. Figure 2.3 below shows a Tesseract core within an HMC vault.

Tesseract has two ways of handling function calls: blocking function call and

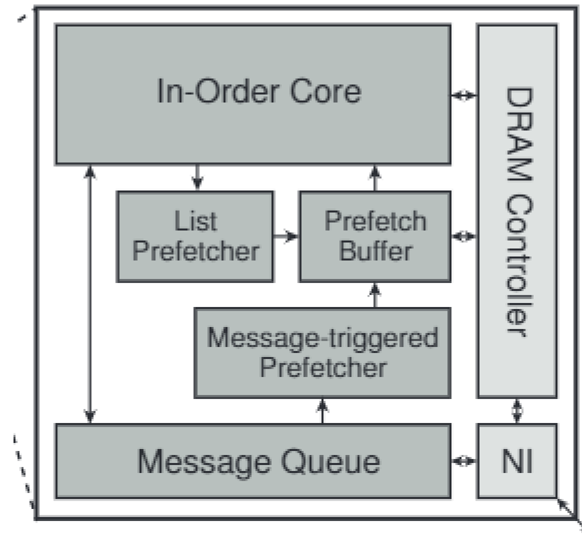

Figure 2.2: Single Tesseract Core

non-blocking function call.

Blocking function calls involve the core retrieving function arguments via the network interface; upon retrieving the arguments, the core goes into interrupt mode to isolate that process and computes its result in a special register. Once the result has been computed within the special register, the core exits interrupt mode and transmits the output to the remote core that requested the computation. The blocking function call is typically used to check for global state changes; for example, a check on two variables such as "$diff > a$" [21]. A drawback to the blocking function call is that each interrupt executed has latency since it is per one core at one time, not multiple functions for multiple cores; furthermore, local cores are blocked until responses arrive from the remote core [1].

Non-blocking function calls involve the core retrieving function arguments via the network interface; however, the core will use its message queue feature to successfully complete not only one task at a time, but multiple tasks at a time. Function arguments will continue to accumulate at the message queue until it is full; once the queue is full, a “batch” interrupt will be executed to complete all the tasks in the

queue, as well as taking into account dependence of tasks. The non-blocking function has no return value and is typically used to update values within registers only [21].

Tesseract offers two methods of prefetching to retrieve function arguments (or instructions): list prefetching and message triggered prefetching.

List prefetching involves traversing strides of a graph; a stride is a chronological sequence of vertices or edges [21]. A stride prefetcher is used based on a reference prediction table (RPT) that prefetches multiple cache blocks ahead to utilize high memory bandwidth. The prefetcher operates on a loop-based mechanism that allows storage of the address. Within the loop, the address is put into a list; it is then put into the RPT as an entry if it conforms to a hint for the processor. This entry may be removed if the processor is at the end of memory or stride and then removed from the general list [1].

Message triggered prefetching is typically used over list prefetching; realistically, stride access pattern is not the commonly used traversal [21]. Random access pattern is used in more common-day graph processing. Furthermore, this works well with non-blocking function calls because data is literally being prefetched to be accessed by the function call. Initially, the network interface retrieves a message. This message is then enqueued into the message queue and is requested by the message queue for a prefetch; the message is then marked as "ready" and is then to be processed within the core. Figure 2.4 shows the action of the message triggered prefetching [1].

A performance and energy comparison between conventional architectures (non Von Neumann) and Tesseract was conducted, with Tesseract having a speedup of 9x without prefetching techniques and 14x with prefetching techniques [21]. It has an 87% fewer energy consumption as compared to [20] and a bandwidth usage (performance scalability) of 17.2 GB/s and 512 cores from 8.5 GB/s and 128 cores, respectively [21]. The scalability of Tesseract gives PIM the ability to expand and scale easily for larger environments [1].

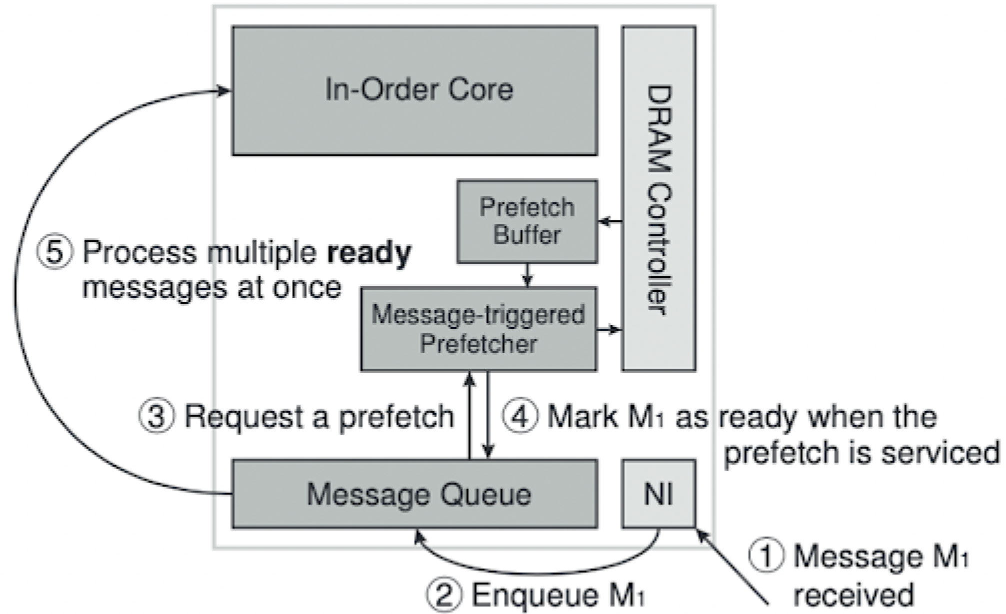

Figure 2.3: Message Triggered Prefetching

PIMSim is a flexible and detailed in-memory processing simulator that allows users to evaluate and test actual processing-in-memory applications [22]. The advent of heterogeneous aspects of processors have led to the rise of PIM simulators as processing in data has two main advantages. In-memory computing logic has a much higher internal bandwidth as compared to off-chip bandwidth and data movement between the CPU and memory is reduced significantly, saving both power consumption and latency issues. PIMSim allows users to track instructions per cycle (IPC) dynamically to compare how fast an instruction is computed in-memory as compared to a traditional Von Neumann Architecture; in addition, the in-memory logic can also be very flexible such that either logic gates or a full processor core can be used to simulate in-memory computing.

2.2 Row-Hammering

As the technology node improves, memory components gradually become denser and closer to one another, leading to issues such as charge leakage and movement. When repeatedly toggling/accessing a certain row of memory, due to how close the rows are and how charge can easily jump, adjacent rows can be activated without access due to charge affecting their wordline (electromagnetic coupling) on the same banks of memory but different rows, known as the rowhammer security bug [2]. This novel security bug can attack either one row or two rows of memory, with the worst case as one row, the middle row as the victim, and the two adjacent rows surrounding it as the aggressors, known as a double-sided rowhammer attack; other attacks can occur where only one aggressor and one victim row, known as a one-sided rowhammer attack.

The rowhammer attack is a large problem that affects memory devices, typically newer DRAM chips are far more susceptible to RH attacks since fewer activations on rows are required to induce the attack, typically around 9600 activations [10]. [10] investigates three types of DRAM to analyze how the rowhammer attack affects each one: DDR3, DDR4, and LPDDR4. Between the three types of memory, DDR4, the newest of the three, suffers the most from rowhammer issues because of feature scaling via technology nodes. However, DDR3 did not suffer as much as it an performing on a significantly older technology node; therefore, some measurement values do not include DDR3 as some cases did not exhibit RH attacks at all. Initially, data pattern dependency is first checked with four types of data pattern exhibited in memory: solid (either all 0's or all 1's), row-stripe (alternating rows of 1's and 0's), col-stripe (alternating columns of 1's and 0's), and checkered (alternating rows and columns of 1's and 0's). The row-stripe data pattern exhibits the worst case data pattern (highest vulnerability to the rowhammer attack) [10]. This is mainly due to opposite

magnitudes of charge in rows as compared to other ones which exhibit more similar magnitudes of charge. Another vulnerability aspect looked into was hammer count (HC). [10] shows a linear relationship between the number of attempts of RH attacks along with HC, indicating that as there are a larger number of potential RH attempts as the HC count increases. In addition to HC count, RH spatial effects were looked into as well with DDR3 data values only affected in the victim row location, DDR4 mainly affected in the victim row location (20% on two rows away), and LPDDR4 mainly affected in rows further away from victim rows (20% two rows away, 10% four rows away, and 60% six rows away). DRAM cells that fail with the least amount of accesses were also looked into with newer chips, such as DDR4, experiencing only 20,000 HC counts for the first bit flip as compared to DDR3 and LPDDR4, which had bit flips at around 65,000 and 25,000, respectively [10]. The effects of error correcting codes (ECC) were also analyzed on the three different types of memory, with HC_{first} counts decreasing by 2.78x in DDR4-old and DDR4-new DRAM chips, and 1.65x in DDR3-new DRAM chips [10].

Attackers can exploit the rowhammer security bug, leading to privilege escalation, shell injection, memory and disk corruption, and advanced Denial of Service (DoS) attacks. If the specific state of a privileged software module is vulnerable to a rowhammer attack, an attacker can repeatedly try to corrupt that state via 'hammering' the cells which store that target state in memory. The Rowhammer Attack Injection (RAI) serves as a framework created to identify, validate, and evaluate rowhammer attacks target states [12]. The rowhammer attack can be classified into two categories: Whitebox RH attacks and Blackbox RH attacks. Whitebox RH attacks consist of attackers having prior knowledge of the virtual to physical mapping of the system and allowing for less brute force attacks, typically leading to a higher chance of successful corruption of data. Blackbox RH attacks involve attackers not having prior knowledge of virtual to physical mapping information, creating more

brute forced attacks, typically leading to lower chances of successful corruption of data [12].

The rowhammer attack injection (RAI) methodology consists of multiple steps beginning with the attacker identifying the memory blocks, having at least one aggressor row and victim row. The attacker can then pressure that state of memory by allocating nearly all of its available physical memory space such that no one else can allocate the aggressor and victim rows. Once the aggressor and victim rows are selected and allocated, a fork process on that target state of memory is introduced to store data structures of targeted states where the victim cell is. The rowhammer attack is then introduced onto the memory, exploiting and corrupting the state of memory. Finally, the attacker ends with concealing and cleaning up the data after tampering with the data [12].

Security aspects of machine learning algorithms are also overlooked in terms of the RH attacks. Attackers with access to CNN or DNN weight values stored in memory can cause significant system malfunction or disruption such as accuracy drops, damaging the effectiveness of the classifier. Both CNNs and DNNs store weights in memory, which are subject to the RH attacks. A vulnerability analysis of neural networks using weight sensitivity was pursued to study the effects of the rowhammer security bug on the memory. Both a weight sensitivity analysis and weight replacement attack were implemented to simulate a rowhammer attack on a CNN's or DNN's stored weights [23]. The weight sensitivity analysis allows attackers to analyze the vulnerablity of weights of trained neural networks with respect to accuracy and misclassification. The weight sensitivity algorithm as shown below indicates an amount value which is used to scale the weights in a certain layer of the neural network and verify the accuracy of the network based on that scaling. By going through each layer via the vulnerability algorithm, the layer with the largest accuracy drop indicates it has the most sensitive weights.

```
Weight Sensitivity Analysis
    amount = 0.5;
    layer_i = FC_1;
    for W in layer_i do
        W' = W * amount
        model' = save(W');
        accuracy' = test(model',test_set)
        return(accuracy', W)
    end
```

The weight replacement attack below replaces few weights based on their sensitivity during run-time for misclassification in neural networks. The algorithm below shows the Weight Replacement Attack as described in [23].

```
Weight Replacement Attack
    threshold = 0.1; amount= 0.5 ;Configurable by the user
    input = (accuracy', W)
    sort(accuracy', W);
    for W, i in layer _i do
        if accuracy'[i] < threshold then
            break;
        else
            W' [i] = W[i] * amount;
            model' = save( model', W' [i]);
            cascaded_accuracy' = test(model', test_set);
            if cascaded_accuracy < threshold then
                W'[i] = W[i] * amount;
```

```
                model' [i] = save( model', W' [i]);
            else
                if cascaded_accuracy > threshold then
                    continue;
                else
                    return model'
                end
            end
        end
    end
```

In the algorithm above, a random preset value 'amount' is used to scale weights in the most vulnerable layer to see accuracy per layer. The weights are then sorted in terms of significance with the loop going across each weight in every layer to find which specific weight decreases accuracy to pre-selected accuracy chosen via the user. The algorithm ends when the accuracy, based on the scaled weight, matches the pre-selected accuracy value. A cascaded array of accuracy values is used to store the accuracies calculated via each weight modification. If the cascaded accuracy value is less than the threshold, the algorithm stops and returns the most recent model with the weight modifications.

In [23], both algorithms were tested upon two DNN's, LeNet and a simple Multi-layer Perceptron (MLP), with Iris and MNIST as training datasets. Both algorithms were used to see how manipulating weights, such as simulating a row hammer attack, on both neural networks would affect the accuracies of both of these networks. Scaling weights with values such as 0.0 - 1.0 on both networks resulted in accuracy drops over 30% for the MLP network with Iris dataset for both input and output layers, but very little accuracy drop for LeNet's convolutional layer. A collection of plots from [23] below show the accuracy drops based on weight scaling in Figure 2.5.

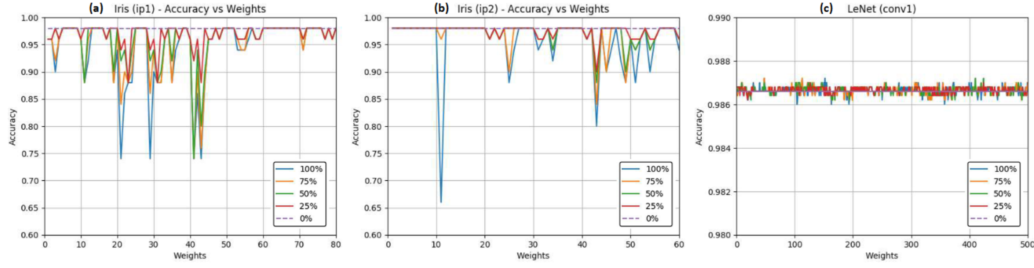

Figure 2.4: Weight Sensitivity of MLP and LeNet with Iris and MNIST Datasets

Input and output layers seemed to be the most vulnerable layers in neural networks as compared to the inner layers such as convolutional or pooling layers. Moreover, accuracy drops in neural networks increase linearly with the number of weights modified with more complex networks taking longer times to attack to achieve a desired accuracy drop.

Various solutions to correcting damages done by the row hammer attack have been implemented, ranging from counter-based techniques, register clock drivers, row shuffling, and encoding [9].

Counter and probabilistic based approaches have been taken to address row-hammering issues. A counter-based approach involves placing a counter along each DRAM row and incrementing each activation until it reaches the threshold value; at this moment, the row would then be refreshed, preserving the values of the bits in that row. Although this method is logical, the placement of counters along each row would increase the area overhead, causing a larger die and higher power consumption [24] [9].

Another counter based approach dealt with assigning counters to the top and bottom rows, respectively [16]. Based on the row activations, rows are marked as either “warm” or “cold”. Rows marked as “warm” will then be further investigated via child counters spawned by the parent counter within adjacent rows. These counters will continually monitor the row activations for that row and those adjacent around it; once a row reaches the row that has hit the row-hammering threshold, all child

and parent counter-monitored “warm” rows will be refreshed, allowing for a reduction of power and fewer counters as compared to counter per row based method in [24]. However, both of these methods require using a row-hammering threshold value which can be reverse-engineered by an adversary [9].

In addition, disturbance bin counters (DBCs) are another counter-based mechanism used to aid in row hammer attacks [25]. DBCs recieve copies of every row activation or auto-refresh command issued via DRAM. The disturbance counters maintain the number of ’disturbances’ of every row in the bin (a set of rows) since the row’s last refresh. DBCs have a 1,024-entry DBC table that can be installed into a memory controller to track the detection of DRAM disturbance errors or row hammering errors. A simple hashing function, using the rank index, bank index, and generated row address, are used to create the bin address in the DBC table. If the DBC table accumulates the number of disturbances in the bin to be over a certain threshold, an alarm is set and the row is auto-refreshed such that no values are harmed due to any disturbances. Grabbing a larger set of rows, compared to looking at one at a time, allows for a quicker time to identify if a row hammer attack occurs [9].

TWiCe, an alternative solution, utilizes fewer counters and proposes a method to counter row-hammering using a register clock driver. A table is used to store entries of rows which are to be pruned if these rows are not accessed that much; furthermore, a buffer and control logic scheme are also implemented within the register clock driver. Figure 2.6 displays the architecture of TWiCe [17] [9].

DRAM would receive a command and address from the memory controller such as an activation or access command. If the address has never been seen before, a new table entry is allocated within the TWiCe table for the entry (typically 6 bytes) [17]. Pruning is then done to ensure only a certain number of addresses are stored to avoid table over-sizing. If the row activation count, which is checked via the TWiCe counter, exceeds the row-hammering threshold, adjacent rows are refreshed

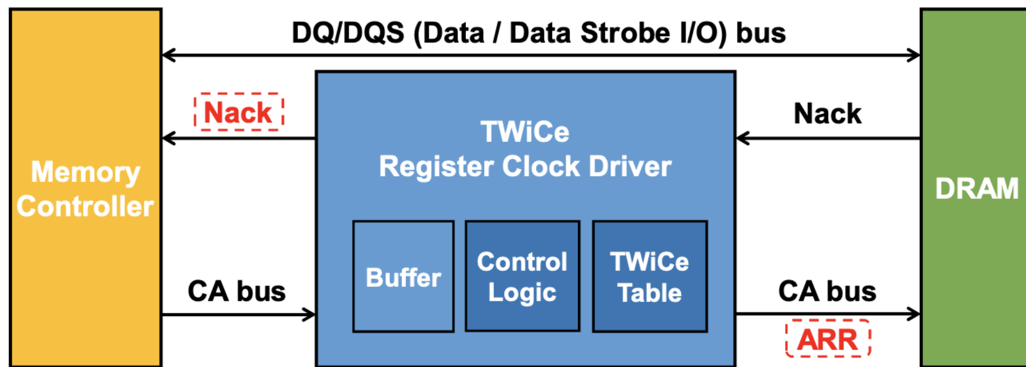

Figure 2.5: TWiCe Architecture

and the entry is deallocated. For each entry in the table, the row is considered to be an aggressor candidate if the number of activations exceed the pruning interval threshold; otherwise, the row is not considered a valid entry within the table. The number of counters required for TWiCe is limited as compared to other counter based approaches since row-hammering only occurs during the refresh window. Electrons are reset during the refresh window; therefore, electrons could be piled up due to row-hammering attack. TWiCe uses the register clock driver to avoid table over-sizing, per-DIMM protection, and is scalable for multiple types of environments. The addition of TWiCe between the memory controller and memory component requires a minimal additional energy of 82 pJ per count and 663 pJ per table update and there is a negligible amount of performance overhead [17] [1].

DRAM address remapping improves both hardware efficiency of row and column decoders as well as hides DRAM address spaces better, improving overall security. [26] discusses a novel solution to the row hammer attack which involves a two-level DRAM address remapping to reduce the rowhammer errors. It involves remapping address in the chip-level and remapping addresses inside a chip via two remapping matrices which keep track and shuffle the bits via splitting the input address into most significant bits (MSBs) and least significant bits (LSBs), making every but in

a word to be originated from a different address. Finally, it uses an error correcting code (ECC) to correct any errors detected. Eleven memory modules were tested on to verify if the swapping of arrays was sufficient enough to detect and reduce the number of row hammer attacks.

Bit-Flip Attack (BFA) detection framework is proposed via weight encoding by leveraging spatial locality of bit flipping as well as fast encoding for vulnerable weights [27]. Initially, a weight sensitivity analysis is done via gradient information within the weights of the neural network. A detection secret-key, or encoding of the weight, is generated to encode sensitive weights and a detection code is generated to calculate the hamming distance between the current detection code and original detection code on the cloud compared to the edge device. The Hamming distance tells the number of positions at which corresponding symbols are different, allowing for a false bit to be detected during transmission, indicating a row hammer attack. If the model is under a BFA, then retraining the DNN model for a few epochs aids in recovering the original accuracy. Three networks, ResNet-20, Resnet-34, and MobileNet were run under the BFA framework with transmission between the cloud and edge device, taking into account the hamming distance and number of bit-flips [27]. Figure 2.7 below shows the plot of the three networks, ten models of each, under the BFA detection framework with both CIFAR-10 and ImageNet as datasets. The Hamming Distance and number of bit-flips did not differ by much, indicating that accuracy of detection of BFA could be higher for malicious networks (more vulnerable weights) as compared to benign networks (less vulnerable weights) which exhibited little to few bit-flips, indicating the gradient information may be a plausible method in determining sensitivity of weights and layers. Accuracy recovery was also measured in which the three networks were modeled under significant bit-flip attacks and then retrained for a few epochs to recover back up from around 0.1%-10% to 67.68%-88.58% [27].

Chapter 3

Vulnerability Analysis of a Programmable In-memory Architecture

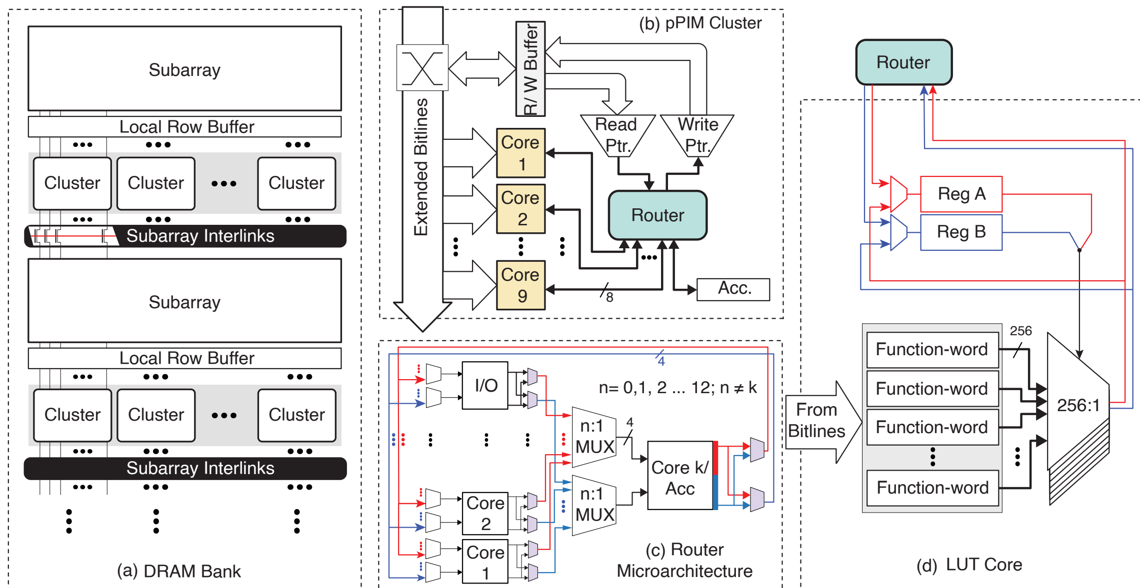

Figure 3.1: Hierarchical view of the pPIM Architecture

pPIM is the proposed PIM architecture used for for computing data-intensive applications. It operates via reprogrammable LUTs to perform a wide variety of applications with very little overhead. The hierarchy of the proposed architecture is depicted in Figure 3.1. The pPIM architecture is composed of a pPIM cluster at the top level. Each cluster contains multiple pPIM cores that can be reprogrammed within the architecture.

3.1 Programmable In-memory Processing core

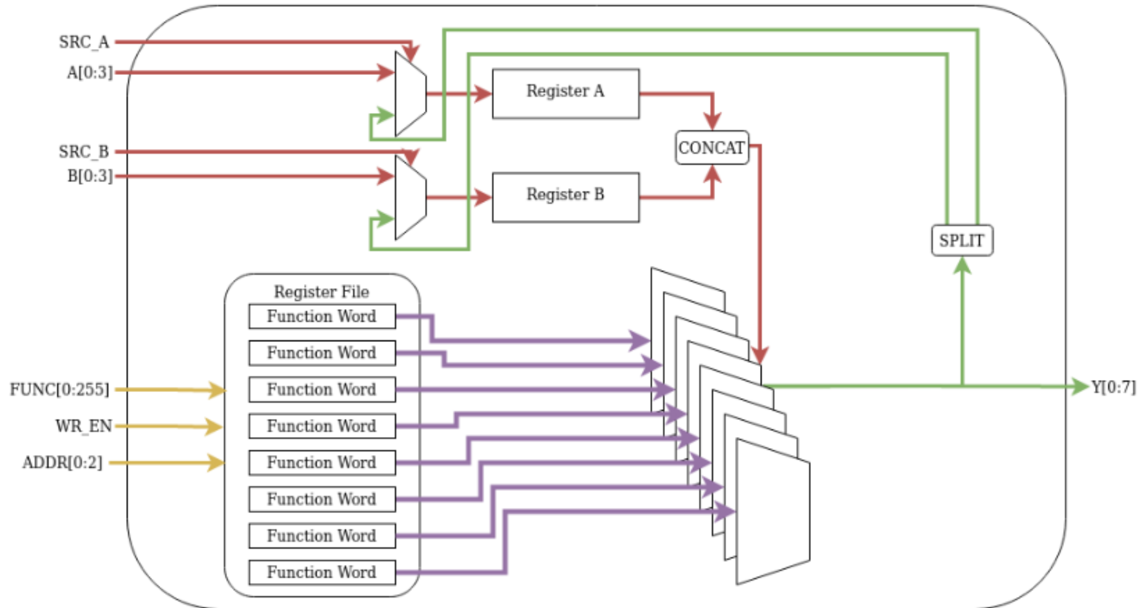

Figure 3.2: Layout of the pPIM core architecture [3]

The pPIM core is the base computing element of the pPIM architecture. The pPIM core design is shown in Figure 3.2. The pPIM core handles a majority of the computational power within the pPIM architecture. Processing within the core is handled through the LUTs into the multiplexers. Repgrommability is also handled within the pPIM core via a register file that feeds into the LUTs.

A plethora of functions can be used via the pPIM core(s). A function of either two 4-bit inputs or a single 8-bit input can be programmed into the pPIM core. Inputs can be viewed as either the high or low segments if containing a single operand. The usage of an array of LUTs that form eight 256-to-11 multiplexers allows for this functionality. The register file contains the data to be served as the input to the multiplexers. However, the select lines of the multiplexers are determined by the two input vectors to the pPIM core.

Eight function words exist within the pPIM core's register file. Each of the func-

tion words are eight bits in length to fit the input to the 256-to-1 multiplexers. Each of function words are then associated with a single bit position within the output of the pPIM core.

3.2 Programmable In-memory Processing Cluster

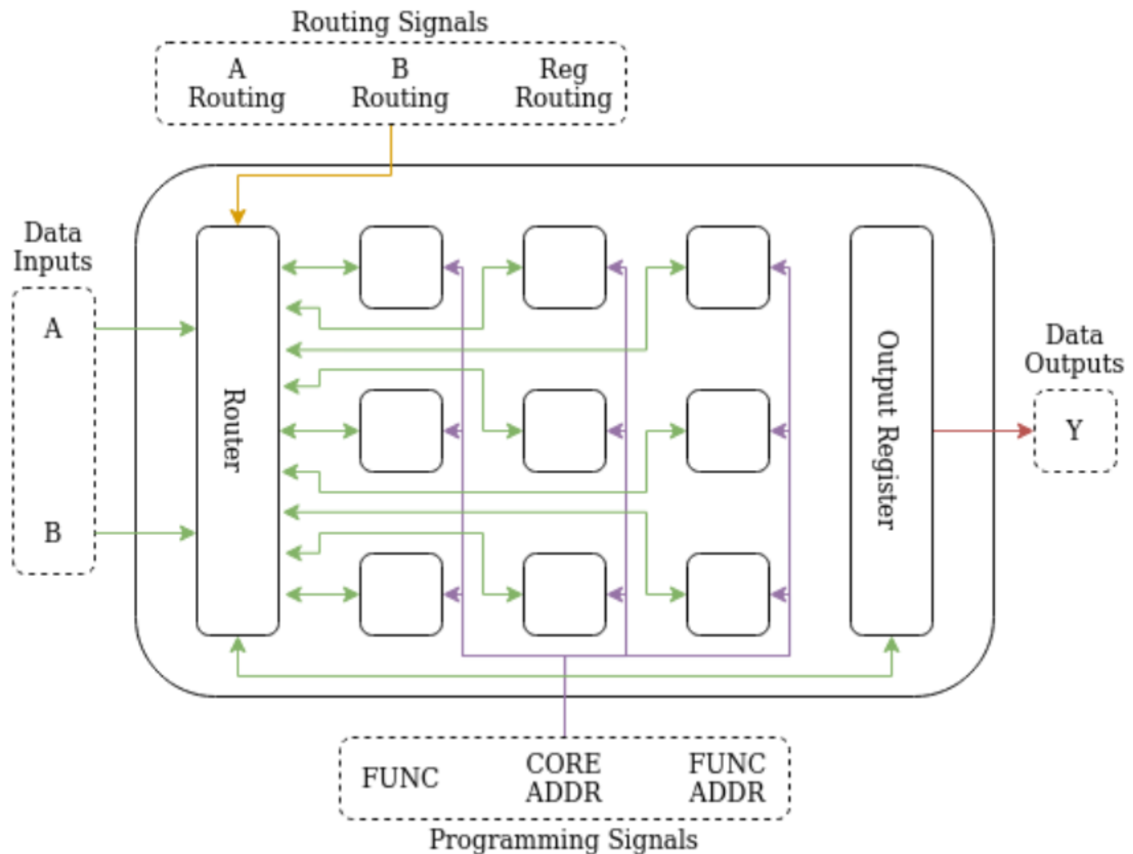

Figure 3.3: Layout of the pPIM cluster architecture [3]

The pPIM cluster serves as the top-level element in the pPIM architecture. Figure 3.3 shows the layout of the pPIM cluster architecture. The pPIM cluster contains nine pPIM cores, interconnected via a router. The router is implemented via a group of multiplexers that replicates a SPIN interconnect model. The pPIM cluster can perform more complex operations over a single or multiple stages via multiple pPIM cores operating in tandem. Operations that require greater resolution can exist via

the usage of multiple pPIM cores operating together.

Each pPIM cluster contains nine pPIM cores used for computation. The pPIM cores within a pPIM cluster are separately programmed, creating a flexible design for various applications. An all-to-all communication network with minimal overhead is used to regulate the dataflow in the pPIM cluster via the usage. A crossbar switch architecture is used to accomplish this task. The crossbar switch routing architecture allows each pPIM core to receive input(s) from output(s) of any other pPIM cores within the pPIM cluster. To match the size of the inputs and output of the pPIM cores within the routing architecture, the 8-bit outputs are treated as two 4-bit segments.

3.3 Programmable In-memory Processing Router

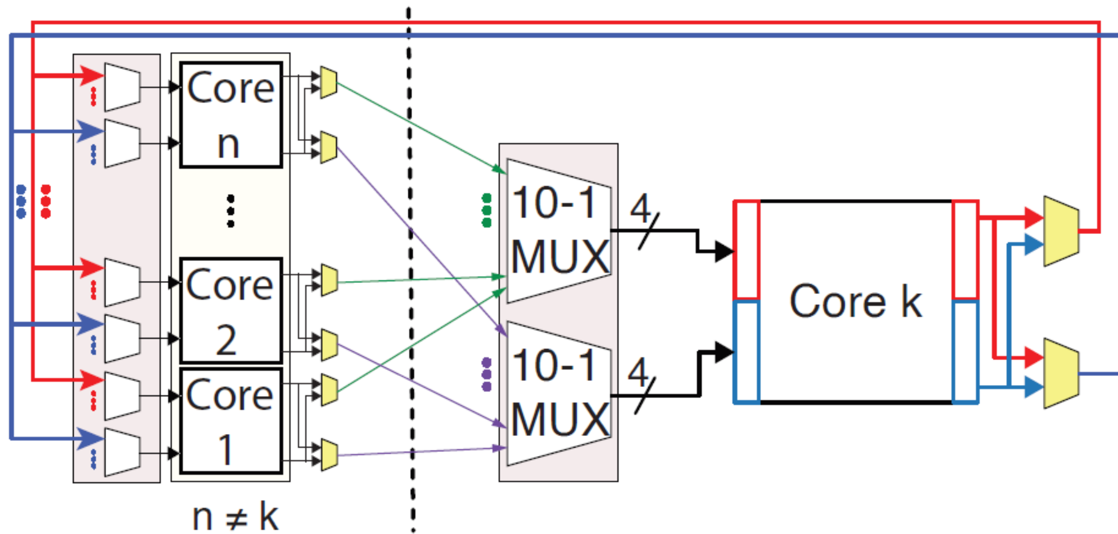

Figure 3.4: Interconnect Router Architecture for n cores in a cluster [4]

The pPIM router handles the data-flow between nine cores in the pPIM architecture. Figure 3.4 shows the interconnect router architecture for n cores in a cluster. The routing mechanism aids in data communication amongst the cores. It is implemented with 10:1 4-bit MUXes which form a SPIN interconnection fabric.

3.4 Programmable In-memory Processing Function Word Generation

The functionality of the pPIM core is generated via eight function words. The output of the pPIM core is based on the eight function words. The process must pass over 256 possible input pairings to determine an entire function word. During each pass, the bits of the output are split up and set in their respective function words at the index of the current iteration. Although some operations require a greater resolution and may not be achievable via one set of function words, the usage of smaller operations can achieve the desired output resolution.

3.5 In-memory Processing Multiply and Accumulate Operations

The pPIM architecture is responsible for data-intensive operations such as CNN inferences. CNN inferences consist of MAC (Multiply and Accumulate) operations. Both functionality and flexibility can be observed in the pPIM architecture through MAC operations.

MAC operations consist of both multiplication and accumulation. Multiplication is computed through the usage of partial products. Accumulation is done via addition of the partial products. Figure 3.5 shows the calculation of the MAC operation.

All upper and lower segments of the inputs are multiplied with each other to form partial products. These partial products are then aggregated to determine the result. Each partial product is placed into the accumulator to perform the MAC operation. This can be done for both 8-bit full precision and 4-bit half precision. A dataflow model of the MAC operation is shown in Figure 3.6 for both 8-bit full precision and 4-bit half precision.

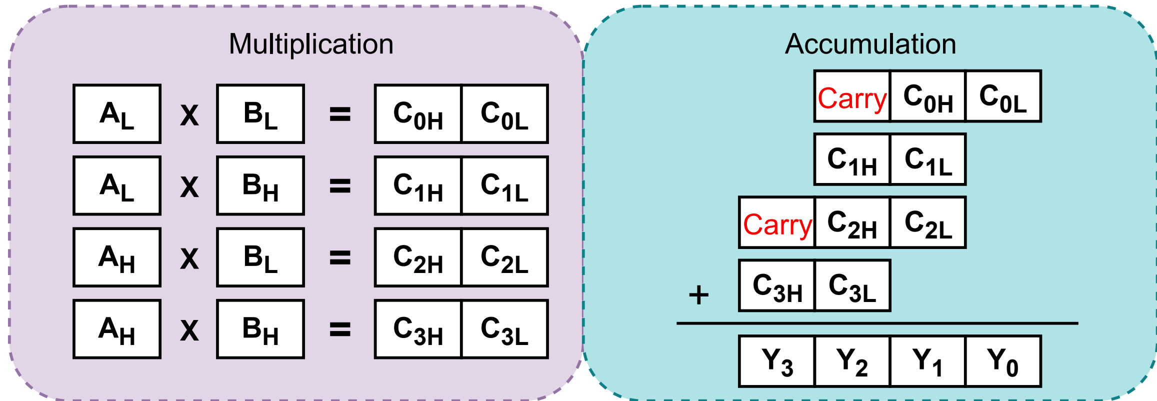

Figure 3.5: MAC Operation Multiplication and Accumulation Stages

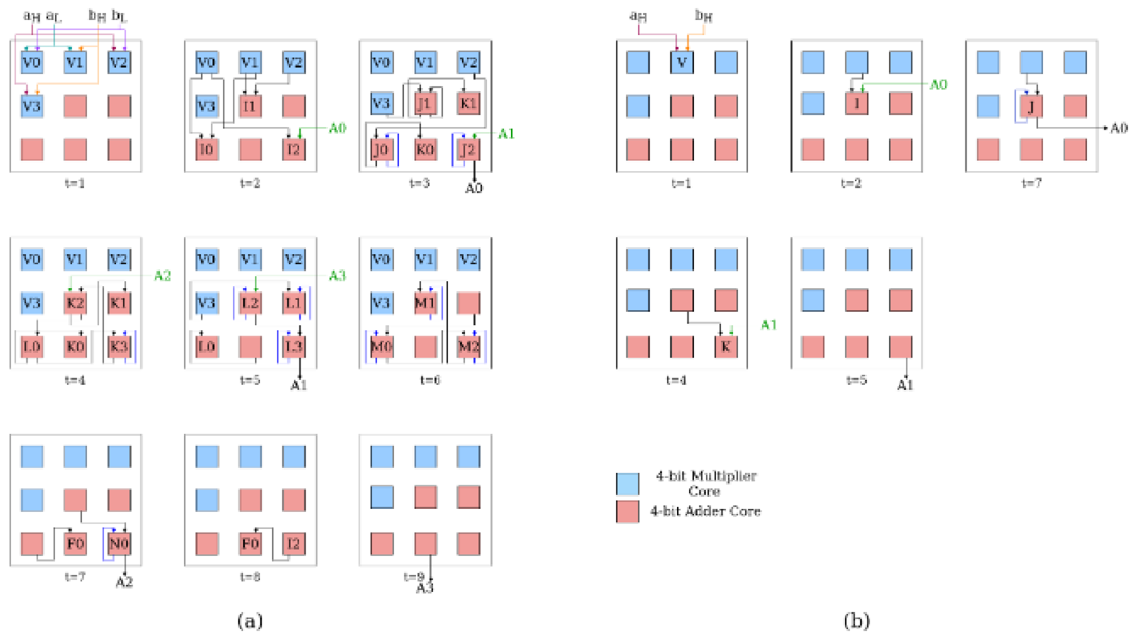

Figure 3.6: Dataflow model within a pPIM cluster for both (a) 8-bit full precision and (b) 4-bit half precision unsigned MAC operations [3]

3.6 In-memory Processing Activation Operations

CNN inferences also require the usage of activation functions to successfully accomplish inference. The mapped operations include ReLU, Linear, sigmoid, and tanh.

The ReLU activation operation can be accomplished within a single pPIM core. Figures 3.7-3.8 depict the dataflow for the ReLU and saturated ReLU activation functions.

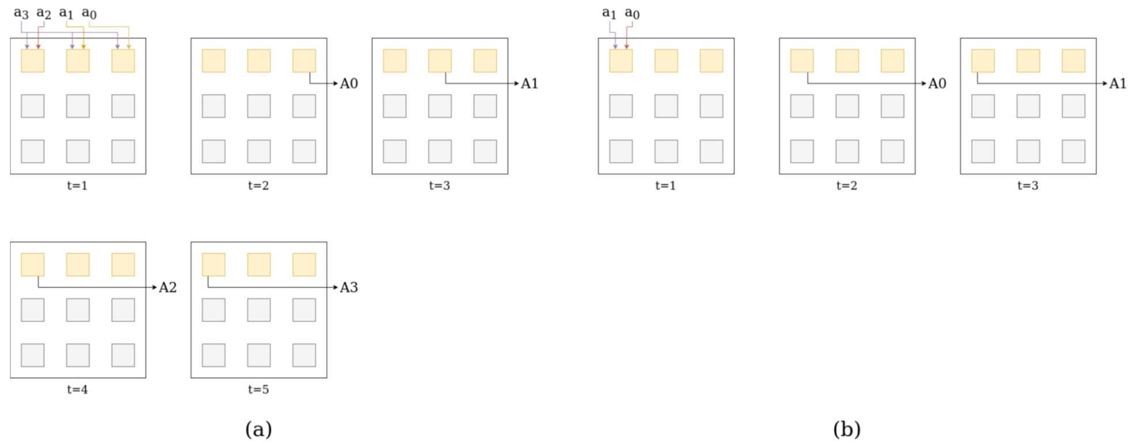

Figure 3.7: Dataflow Model within a pPIM cluster for both (a) 16-bit precision and (b) 8-bit precision ReLU activation operations [3]

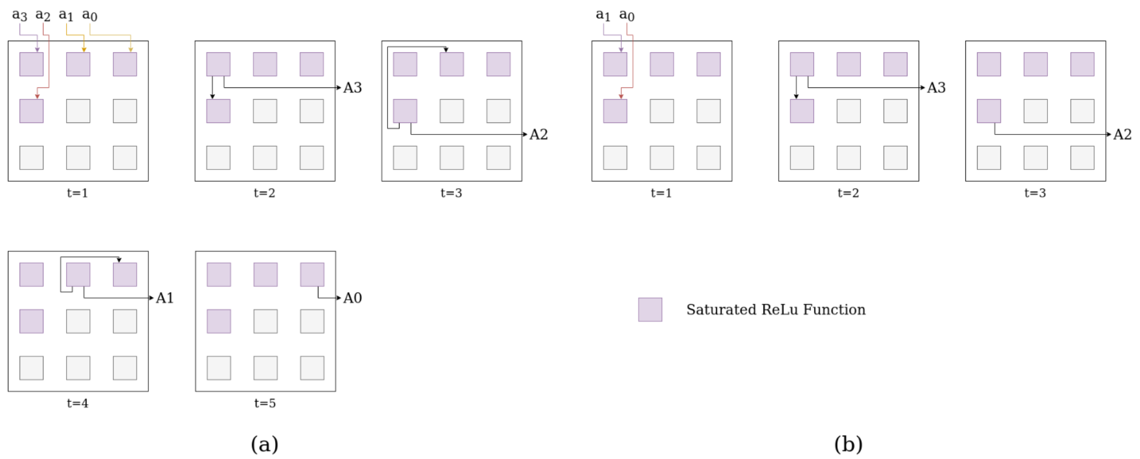

Figure 3.8: Dataflow Model within a pPIM cluster for both (a) 16-bit precision and (b) 8-bit precision Saturated ReLU activation operations [3]

The pPIM architecture can handle CNN inferences completely with both MAC operations and activation functions. Other functions such as sigmoid and tanh are capable on the pPIM architecture as well.

3.7 Field Programmable Gate Array Implementation

An FPGA implementation of the pPIM architecture is proposed for validation. The processing elements and dataflow through simulation of unsigned 8-bit mAC operations multiple pPIM cluster is used for validation. Classification is also handled by

a pPIM cluster with a max-index dataflow. The FPGA implementation is used to predict numbers within images with a single layer NN.

3.7.1 Technology Resources

The ZC702 base board is used to validate the pPIM architecture. The ZC702 is an FPGA composed of both a programmable logic (PL) core and a processing system (PS).

The Modified Standards and Technology (MNIST) dataset is used to train and test the NN. There are 70,000 handwritten digit images, with 60,000 training images and 10,000 testing images. Each image is of dimension 28 pixels by 28 pixels.

Google Keras is used to handle the training of the NN before moving on to the FPGA. The Keras library provides access to the MNIST dataset for training to model before using the FPGA for simulation and verification purposes.

3.7.2 Machine Learning Model

A single layer NN is used to train the MNIST dataset to be used on the pPIM architecture. The NN consists of a fully connected layer with a softmax activation function. The input image size is a 28x28 image. The output layer is a vector of ten 8-bit unsigned integers, with each index corresponding to the range of possible prediction outputs. The total number of parameters is 7,850, with 7,840 dedicated to the single dense layer and 10 dedicated to the activation function. The NN is trained for 5 epochs with a batch size of 128. A max-index operation is used to determine the output of the NN.

3.7.3 In-memory processing Implementation

The FPGA implementation of the pPIM architecture is composed of two distinct operations: MAC operations and classification (max-index) operations. The two

regions within the FPGA implementation are show in Figure 3.9. Each operation is allocated a separate region within the FPGA with no reconfiguration. This focuses on the validation of how the pPIM architecture processes data. Each layer acts as a finite state machine. Each layer tracks the input and output to the region and beginning execution. When the cluster has its input ready, the state machine sends a 'start signal' to begin execution. After the inputs are processed, the state machine sends the output to the next region or memory.

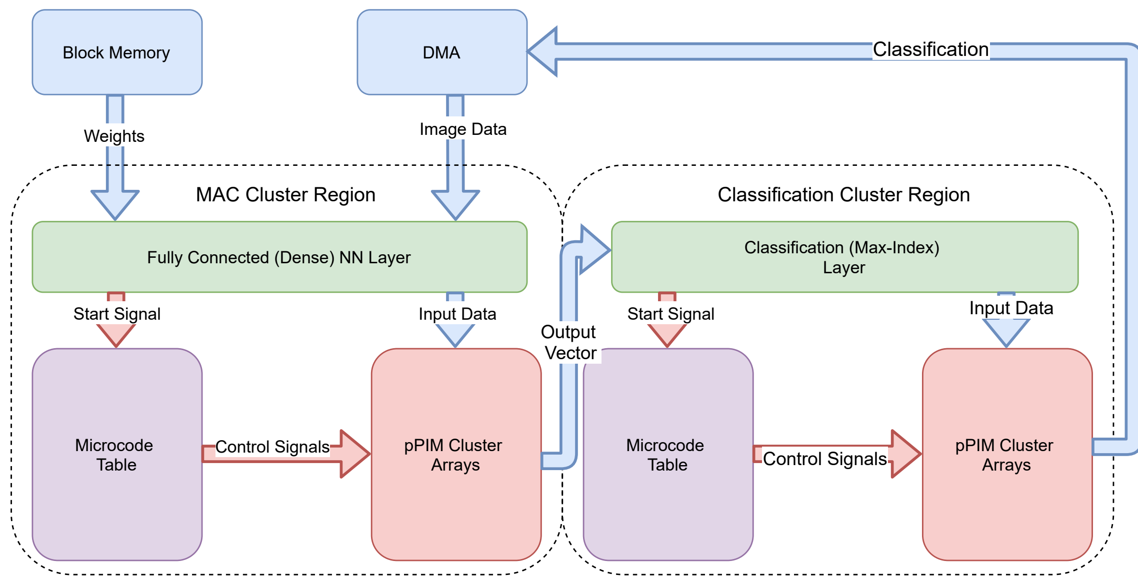

Figure 3.9: FPGA MAC and Classification Regions within FPGA Implementation of pPIM architecture

3.7.4 Communication

Communications between the host computer and elements on the ZC702 FPGA board requires the use of two communication protocols: direct memory access (DMA) and universal asynchronous receiver-transmitter (UART). The communication used between each of the elements is shown in Figure 3.10.

The PS on the board provides hardware for UART communication. Over the UART communication, the end device is capable of communicating with the PS.

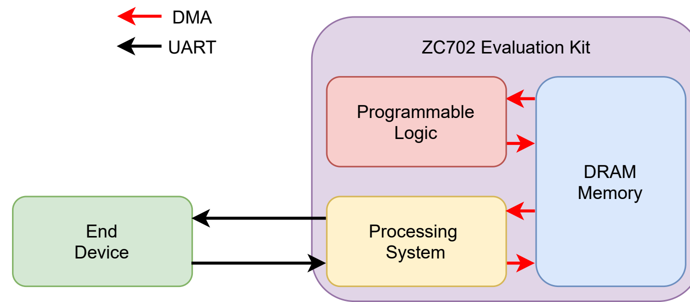

Figure 3.10: Communication Model between ZC702 Evaluation Board and End Device

Image data is passed from the end device to the PS. The PS also handles sending classification results back to the end device.

Both the PS and PL on the ZC702 board are capable of DMA. DMA on the PL requires introducing IP provided by Xilinx that communicates the data to the pPIM architecture. DMA is used to the transfer the image data to the pPIM architecture and for the pPIM architecture to return a classification result.

3.8 Results

3.8.1 pPIM characteristics

Table 3.1: Synthesis Results for pPIM Architecture

Component	Delay (ns)	Power (mW)	Active Area (μm^2)
PIM Core	0.8	2.7	4616.85
PIM Cluster (MAC Operation)	7.2	5.2	41551.66

The pPIM architecture is characterized using post-synthesis models using the Synopsys Design Compiler at the 28 nm technology node. The pPIM characteristic results are shown below in Table 3.1. The model includes both a synthesized model of the pPIM core and the pPIM cluster. The LUT multiplexers are modelled using

transmission gates, improving the area overhead of the design. In addition, an 8-bit unsigned MAC operation is chosen to record the power and delay figures for the pPIM cluster. Table 3.1 gives an accurate estimate for deep learning applications that contain MAC operations.

3.8.2 Performance Evaluation

We compare the pPIM with both von Neumann architectures alongside other contemporary PIM architectures in terms of throughput (frames per second) and power consumption for AlexNet inference in Figure 3.11. The representative von Neumann devices to be compared includes a high-end CPU and GPU such as Intel Knights Landing (KNL) [28] and Nvidia Tesla P100 [29], respectively. The PIM architectures used for comparison include DRISA [30] and DrAcc [31], SRAM-implemented Neural Cache [32], another LUT-based PIM implemented on the DRAM platform: LAcc [18] and the pPIM itself [4].

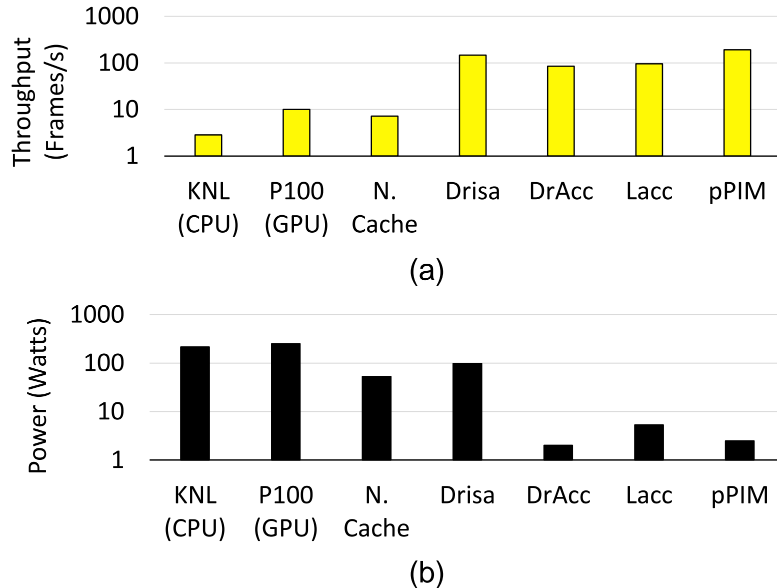

Figure 3.11: Comparison of (a) throughput and (b) power consumption of various architectures

A comparison of area and efficiency/area were also investigated for several archi-

tectures for the AlexNet CNN in Figure 3.12.

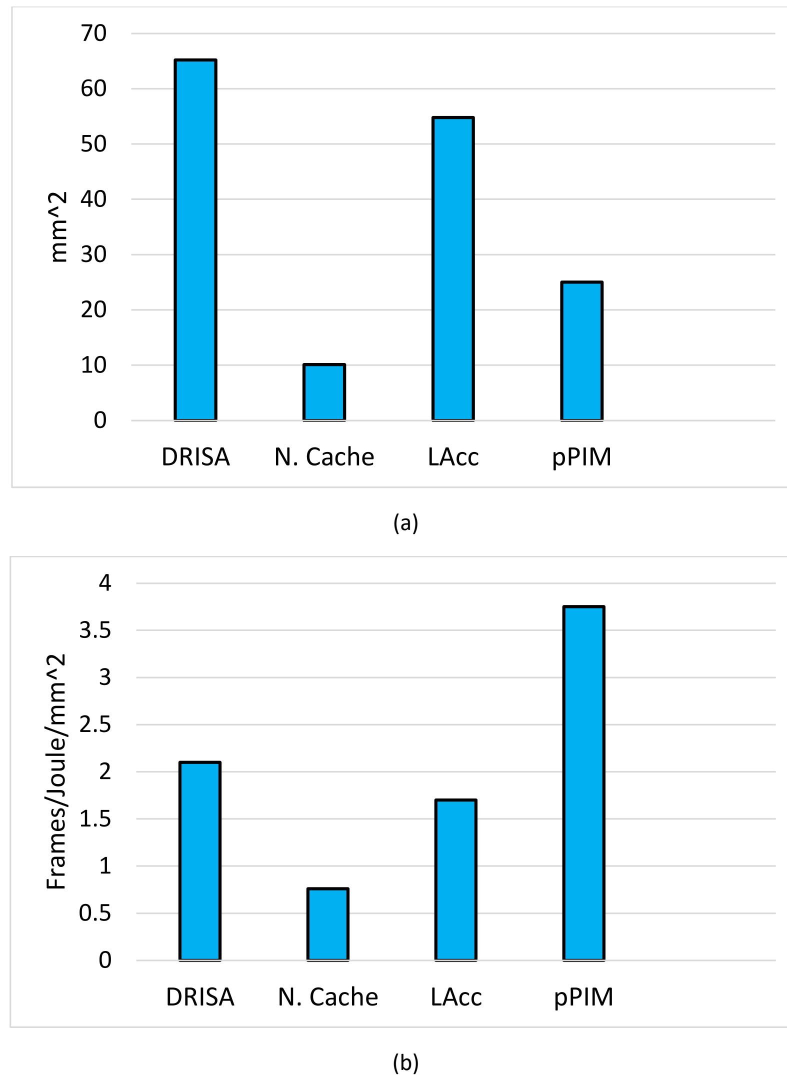

Figure 3.12: Comparison of (a) area and (b) efficiency/area of various architectures

As seen in Figures 3.11-3.12, the PIM architectures, especially pPIM, outperform the von Neumann architectures by a significant margin in throughput, power-consumption, and area overhead. This is because, the PIMs can largely avoid the overhead and latency associated with off-chip communications, unlike those of CPUs and GPUs. pPIM offers 12.8x and 45x higher throughput and 125x and 107.5x more power efficiency compared to P100 GPU and Knights Landing Processor, respectively. The Look-up Table (LUT) based architectures, LAcc [18], and pPIM offers high

throughput for the least amount of power consumption. DRISA [30], a DRAM-based bitwise processing accelerator, outperforms LAcc and pPIM, albeit at a higher power consumption. The pPIM architecture achieves better performance while consuming less power than LAcc, even with the implementation of the proposed RH-encoding scheme. This is due to the smaller scale of the LUTs utilized by the pPIM architecture as well as the efficient mapping of operations of our encoding and decoding schemes across the parallel processing sequences inside the pPIM architecture. Due to the superior performance and power efficiency of the pPIM compared to the other architectures, we choose the pPIM as a case study to evaluate the effect of RH and encoding.

3.8.3 Vulnerability Analysis of In-memory Processing Architectures

We perform a vulnerability analysis on the weights of a CNN trained under the MNIST dataset [33]. We investigate the vulnerability of the weights on a single-layer MLP. It consists of a single dense layer with a softmax activation function. The input size is a 28x28x1 image. The total number of parameters is 7,850, with 7,840 dedicated to the single dense layer and 10 dedicated to the activation function. The MLP was trained for 5 epochs with a batch size of 128. The reported accuracy of the network was 88.45%. The vulnerability analysis looked upon the weights of the dense layer as the majority of weights to be affected by the RH attack exist there. For each digit in the MNIST dataset, an bit-flip attack was simulated on the MSB of the exponent portion of the weights in the single dense layer to simulate a potential RH attack. The network performance is calculated for each individual bit-flip and not for all bit-flip attacks accumulated as some weights are found to be more sensitive than others. Figures 3.13 represents error heatmap on the MNIST dataset using the single layer MLP.

In Figure 3.13, the performance of the network is shown to degrade for each RH

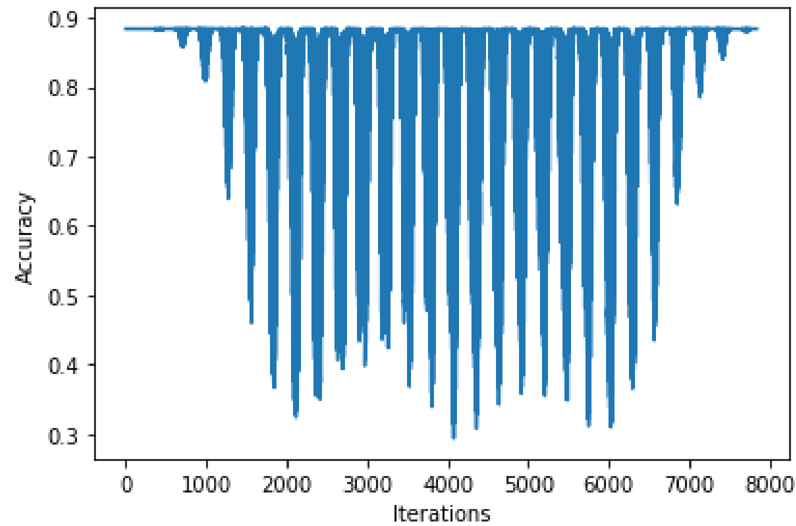

Figure 3.13: Weight Vulnerability Heatmap on the MNIST dataset

attack while classifying MNIST dataset. Due to the nature of MNIST frames, some weights are more vulnerable to the RH attacks than others due to their relative position within the frame. It can be observed that the largest drop in accuracy was down to 29.35%, showcasing the potential damage RH can do on simply a single layer CNN algorithm even for simple data sets like MNIST. The RH attack can occur on any PIM architecture with larger architectures having a more susceptible weights; henceforth, it is imperative to secure the weights to ensure the security of PIM architectures. The MNIST dataset contains multiple frames with either having the number or white-space in the frame. Some oscillations exist in the heatmap above due to some weights mapping to white-spaces as compared to other weights mapping to numbers. Weights that map to white-spaces that are flipped have less significance to the overall accuracy of the network. This happens at regular intervals because weights are scanned across the frame. When the weight is selected at edge of frame, it is not subject to a change in accuracy as there is white-space at edges of frames. When simulating the row hammer attack on these weights, there is very little impact on the overall network. However, weights mapped to the actual digit within the frame that are flipped have a significant impact on the overall accuracy of the network. A

flip in these bits would cause inference to be more difficult and cause larger accuracy drops as compared to white-space bit flips.

Chapter 4

Row Hammer Reducing Encoding Scheme

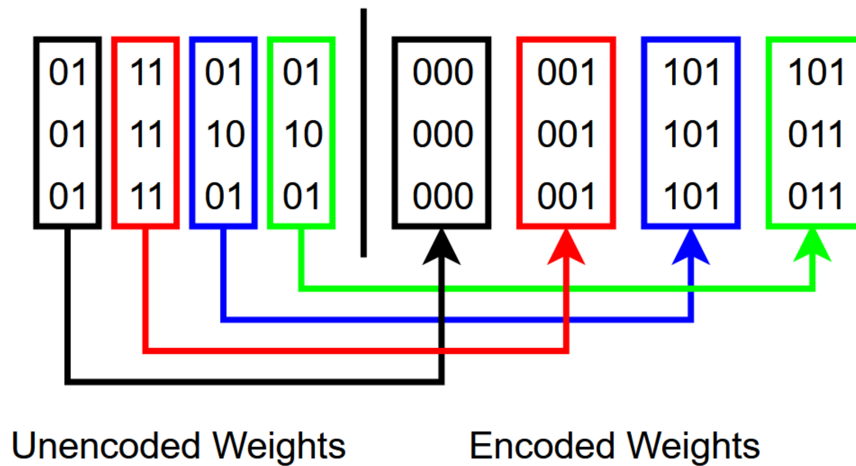

Figure 4.1: Proposed Encoding Scheme for handling RH attacks

The proposed RH-encoding scheme is designed to work within the pPIM architecture in securing data-intensive applications used in deep learning applications. Figure 4.1 illustrates the input and output of the RH-encoding scheme. When it comes to implementing the proposed RH-encoding, we found the pPIM [**?**] to be the most suitable platform for multiple reasons. First, a PIM architecture is capable of performing massively parallel encoding and decoded with very low latency and high energy efficiency which makes it a better alternative than encoding based on an external processor. Second, the LUTs in pPIM are programmable which makes it possible to implement the encoding scheme without modifying the existing architecture or adding new hardware to the pPIM.

4.1 In-memory Data Mapping

As shown in Figure 3.1 (a), the pPIM clusters are arranged in rows in-between memory subarrays. The weights and activations of CNNs are arranged inside the subarrays and are processed in the nearest cluster. The clusters can each read a large batch of such data operands (weights and activations) at a time. Since a CNN has a very large number of parameters, these parameters are arranged across multiple memory rows, prior to being read and processed inside the clusters. It is during this waiting period that these data become vulnerable to potential row-hammering attacks. Data bits across different memory rows can affect each other's capacitance when a particular row (attacker row) is activated repetitively. Therefore, the purpose of the encoding scheme is to ensure that the relative charge state of any two memory cells from two adjacent memory rows are such that the probability of capacitive interaction between them and therefore the probability of row-hammering corruption is minimal.

4.2 Encoding Connectivity

As shown in Figure 3.1, the vertically aligned pPIM clusters are interconnected in the pPIM architecture via interlinked bitlines [34]. These clusters can communicate data-operands among each other and effectively implement a Stream Processing architecture. We capitalize on these connectivity to include the Encoding and Decoding process in the stream of operations. A page of data is first encoded by a pPIM cluster programmed to execute the proposed encoding scheme and then stored in a memory row to protect it from potential row-hammering attacks. Once the data is required for the desired computational tasks, it is first send to a cluster programmed to perform decoding. The decoded data is then forwarded to the actual processing clusters via the interlinked bitlines. The encoding and the decoding clusters are just regular-cluster programmed to perform these specific tasks. In fact, since encoding

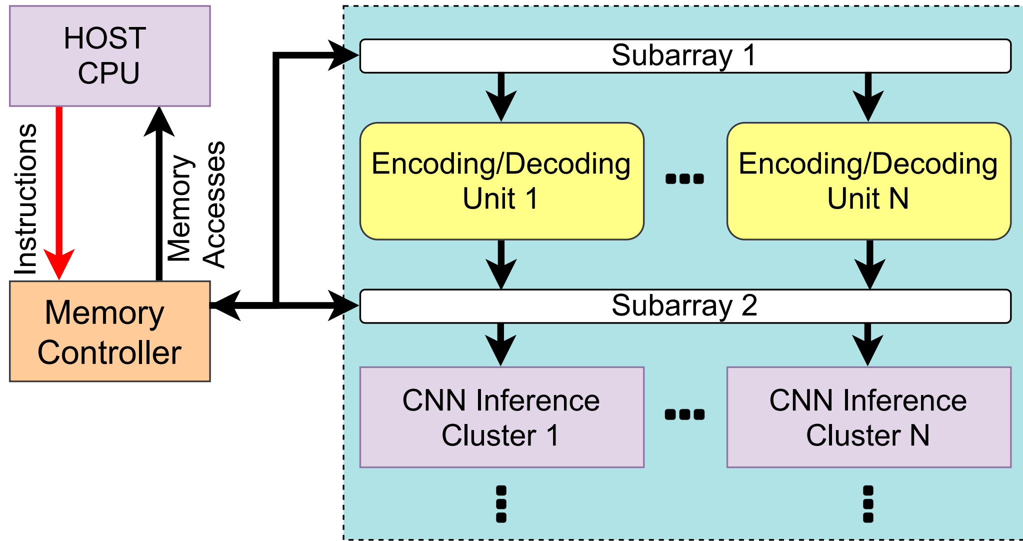

Figure 4.2: Model of the interface between the RH-encoding scheme with pPIM Clusters and memory elements

and decoding is not performed simultaneously, we envision reprogramming the same cluster dynamically for performing both tasks, sequentially, as shown in Figure 4.2.

4.3 Encoding Scheme

The proposed RH-encoding scheme is aimed at reducing the rate of data-corruption from row-hammering attacks by encoding the bit-combinations that are the most vulnerable to this attack. As discussed previously in Section 2, the row-stripe pattern of the stored bits in adjacent memory rows are the most vulnerable ones. For this purpose, we develop a coding scheme that investigates and encodes three rows of weights. The exponent portion of the weights is encoded since a bit-flip produces the highest amount of error. The coding scheme collects two triplet columns of weights, particularly two columns of bits at a time, and encodes them into three triplets of weights, or three columns of bits. The programming scheme for the cores in a cluster for performing encoding is presented in Table 4.1.

Table 4.1: Row Hammer Encoding Truth Table Samples

Input	Output
01	111
01	100
01	100
00	010
11	010
00	011
11	001
00	000
11	010
11	000
11	000
11	000

Input	Output
00	011
00	010
11	010
10	111
10	110
00	110
10	110
11	110
11	100
11	000
11	001
00	001

4.3.1 Alternative Encoding Schemes

Some alternative encoding schemes involved only taking the first two triplet columns of weights. These two columns would be transposed and encoded into a nine-bit output. Each triplet column would be encoded to another triplet value that does not exhibit the row-stripe pattern. The three MSB bits would be a pattern common to all encoded weights. Some advantages of this design involved fewer computation achieve this encoding scheme as some values were fixed. However, decoding proves to be difficult since only two triplet pairs are collected. Not all information is present to successfully decode weights for inference. In addition, some triplet values could not be converted to new triplet values as converting a three-bit space to another three-bit space involved the row-stripe pattern appearing. Figure 4.3 illustrates this encoding scheme below.

A dual 3-to-4 encoding scheme was also proposed which allow for 1-to-1 mapping for encoding/decoding purposes. The first and fifth triplet columns would be encoded but this time to two four-bit values making the total encoded value 8-bits long instead of 9. The first and fifth triplet pairs were taken to account for both MSB and LSB bits. Dual encoding allows prioritizing both MSB and LSB bits during encoding.

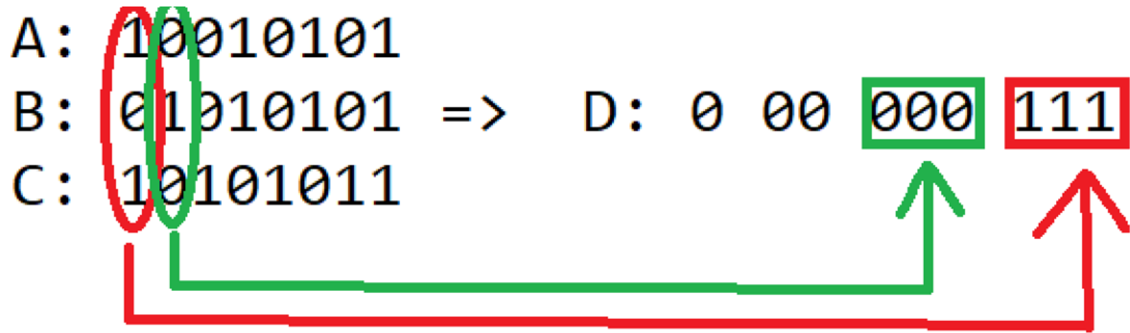

Figure 4.3: Original Encoding Scheme

More unique encoding outputs to try and test out for best decrease in chance of RH attacks. The 8-bit encoded output to substitute for the exponent portion of the weights. In addition, this scheme allows for better disguising to have 8-bit output as it can be directly substituted in for the 8-exponent bits in IEEE Floating Point Format. Some drawbacks associated with this encoding scheme involved difficulty in decoding as only two triplet columns out of the eight were collected. It would prove impossible to decode as not all information is gathered. Figure 4.4 shows illustrates encoding scheme below.

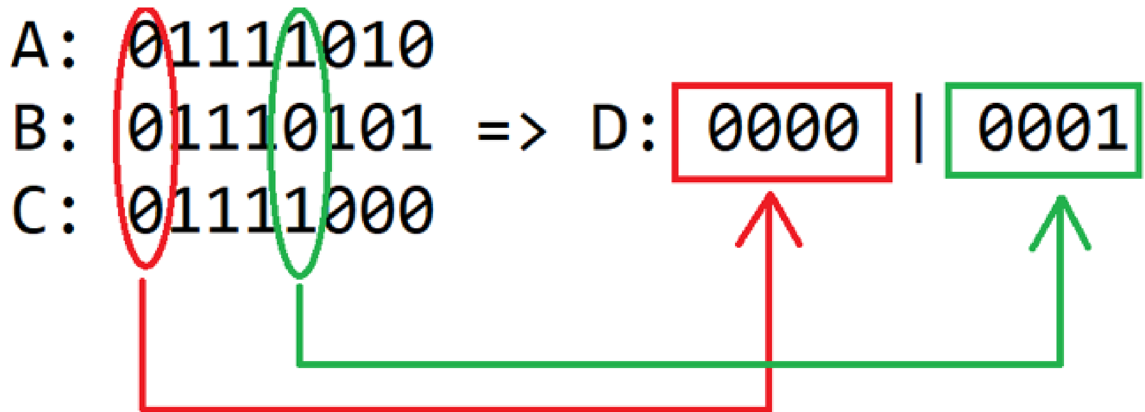

Figure 4.4: Dual 3-to-4 Encoding Scheme

The encoding scheme shown in Table 4.1 shows a six-bit to a nine-bit operation. A parity of three bits are added during encoding since three rows of weights are inputted and three rows of weights are outputted. A additional column of three bits allows for the row-stripe pattern to be checked across every three rows successfully as

the double-sided RH attack involves three rows of data. Double sided row hammer attacks are checked by looking at three rows at a time. The addition of three parity bits allow for proper investigation along with successful decoding.

The proposed encoding scheme will take up additional memory space. The proposed scheme adds four new triplet columns of bits, resulting in 12 new bits for every three weight values. As more bits of the weight are encoded, more memory space is taken up. This scheme demands larger memory space to operate successfully. Larger CNNs would require additional storage space to allow the encoding scheme to take place, making it more difficult to store encoded weights.

Some alternative encoding schemes could involve encoding one triplet column at a time to produce either one, two, or more columns of triplets. This could prioritize more bit positions of weights as compared to the proposed scheme. This would result at the minimum of eight new columns as compared to four. The addition of more columns results in more design constraints as new columns consume even more memory space compared to the proposed scheme. However, encoding larger groups of triplets may reduce the number of new triplet columns formed. This allows for less memory space to be consumed potentially at the cost of higher power consumption, a larger decrease in throughput, and less prioritization of all bit positions.

4.3.2 Comparisons with Error-Correction Codes

In 2014, three DRAM manufacturers (A, B, and C) were subject to the row hammer attack. All three DRAM manufacturers were equipped with the SECDED ECC mechanism to correct and detect the row hammer errors. Figure 4.3 below shows the results of the uncorrectable multi-bit errors (in bold) [2].

Most errors in the rows of $A_2 3$, $B_1 1$, and $C_1 9$ are subject to the row hammer attack. SECDED (Single Error Correction, Double Error-Detection) can only correct a single-bit error within a 64-bit word. However, if a row contains two victims SECDED

Figure 4.5: Uncorrectable multi-bit errors (in bold) [2]

Module	*Number of 64-bit words with X errors*			
	$X = 1$	$X = 2$	$X = 3$	$X = 4$
A_{23}	9,709,721	**181,856**	**2,248**	**18**
B_{11}	2,632,280	**13,638**	**47**	0
C_{19}	141,821	**42**	0	0

cannot correct the resulting double-bit error. It can only detect them. If there are three or more victims, SECDED can neither correct nor detect the multi-bit error, leading to silent data corruption within memory [2]

The advantages of the proposed encoding scheme include securing all weights from the row-stripe data pattern and can correct multiple bits in multiple segments of weights. However, the proposed encoding scheme only prevents against row hammer attacks. In contrast, the DRAM ECC looks into any particular bit flip in memory and corrects it, regardless of its data pattern type. This could be for any type of disturbance as well, regardless of row hammer. A significant disadvantage of DRAM ECC is that errors still linger after one error correction as shown in Figure 4.3. This means DRAM ECC does not secure data in memory against disturbance errors. A greater ECC is most likely required to combat row hammer attacks, which come at a cost of area, energy, performance, throughput, and DRAM capacity overheads.

4.3.3 Comparisons to logic/counter based approaches

A few logic/counter-based approaches to prevent row-hammering attacks have already been proposed, mostly dominated by the usage of counters and a probabilistic approach. These approaches include Counter-based Activation (CBA), Counter-based Tree (CBT), row remapping, probabilistic adjacent row activation (PARA), and Time

Window Counter (TWiCe). However, common to all these approaches is the involvement of complex and bulky intelligent logic circuitry to monitor and actively prevent row-hammering corruption. Several works have proposed that actively prevent row-hammering attacks. The common feature of these works is the presence of counters to monitor the memory rows to detect an aggressor row and prevent successful row-hammering attacks by taking interceptive measures.

One such counter-based activation (CBA) approach incorporates an intelligent logic unit that monitors the activation of DRAM rows. Once the number of repetitive activation reaches a certain threshold value; this intelligent unit refreshes the row to preserve the stored bits in the row from corruption. Another similar work utilizes a counter-tree (CBT) to actively prevent row hammer attacks. Based on a number of activations, different rows are marked as either 'warm', or 'cold' and are continuously monitored by the counter-tree. Once a row reaches the row-hammering threshold, all 'warm' rows, in general, are refreshed by the counter-tree. This results in overall fewer counters and a reduced footprint compared to the previous work. Another work leverages the 'Time-window Counters (TWiCe)' mechanism which features a table that stores entries of rows and prunes the number of active memory rows to track an aggressor row. For each table entry, the row is considered to be an aggressor candidate if the number of activations exceeds the pruning interval threshold. The number of counters required for TWiCe is limited as compared to other counter-based approaches. All of these approaches do not require modifying any data in memory and use simple hardware to detect row hammer attacks. The TWiCe approach uses less additional hardware as compared to the other counter-based approaches. Some downsides to these approaches involve the usage of additional hardware to monitor the rows of memory. This reduces power consumption and throughput loss significantly. In addition, more area is required to place the counters. This raises manufacturing costs. In comparison, the proposed encoding scheme does not need additional hard-

ware to encode the weights. It works within the existing architecture and can be scalable to any other architecture.

PARA is a probabilistic logic based approach to counter row hammer attacks. Each time a row is accessed, one of its adjacent rows is also statistically accessed with some low assumed probability 'p' [3]. If one row happens to be accessed repeatedly, then it is statistically certain that the adjacent row will be accessed as well, assuming 'p' is chosen carefully. Some advantages of PARA is that it does not require additional hardware to work successfully, low power-consumption and high performance. In comparison, the proposed encoding scheme also does not need additional hardware to work successfully. It also has a low-power consumption and high throughput as well. However, the proposed encoding scheme does not rely on an assumed probability to encode data.

The proposed encoding mechanism is adopted such that the overhead translated to the memory space in the PIM architectures. Memory is relatively abundant in a majority of architectures whereas logic is expensive. Therefore, overhead in regards to throughput, energy consumption, and area are minimized with the proposed encoding scheme.

4.4 Operation Mapping

Our proposed encoding scheme is implemented on multiple stages inside pPIM clusters. This requires the LUT cores in a cluster to be programmed in a specific way to perform a desired set of operations. For the 8-bit encoding operation, four cores are programmed as 4-bit read cores, two cores are programmed as 4-bit adders, one core is programmed as a 4-bit encoding core, and one core is programmed as a 4-bit write-back core. Each cluster can encode all triplet pairs in a cluster at once.

The three 8-bit inputs of the encoding operation are each split into pairs of 4-bit segments, each denoting the row and upper or lower half of data. Three rows of data

are accessed from memory and are split into low and high bits, corresponding to the upper four or lower four bits of data, respectively. Data is sent through four read cores of Type 1 (shown in Table 4.2) in which each one receives different bit-segments of each of the four triplet pairs. The third row of data is then forwarded to the add cores to concatenate with the different bit-segments for appropriate encoding format. The triplet pair bits are then collected and sent to the encoding core. The encoded data is forwarded to the write-back core to generate the ninth bit, identical to the eighth bit. The encoding is done sequentially for each of the triplet pairs, resulting in a total of eleven clock cycles to complete. The eleven-stage encoding and nine-stage decoding routing is shown in Figure 4.6.

Data is decoded prior to performing CNN inference. The cluster previously used for encoding is reprogrammed to perform decoding now. Two read cores of Type 2 (as shown in Table 4.2) are required for appropriate decoding format. Figure 4.6 (b) displays the sequential model required for decoding. Three columns of bits represent one triplet set to be decoded. The decoding core takes the format and recovers original weight values sequentially.

Table 4.2: Read Core Functionalities

Read Core Type	Output	Input A	Input B
1	0xx0 0xx0	Any 4-bit Value	Any 4-bit Value
2	0xxx 0xxx	Any 4-bit Value	Any 4-bit Value

4.5 Results

4.5.1 Encoding and pPIM architecture characteristics

The encoding/decoding scheme and pPIM are characterized using post-synthesis models using the Synopsys Design Compiler at the 28 nm technology node. The hardware synthesis and pPIM results are shown below in Table 4.3. It can be observed that there is minimal overhead from the encoding scheme thanks to the micro-code-based

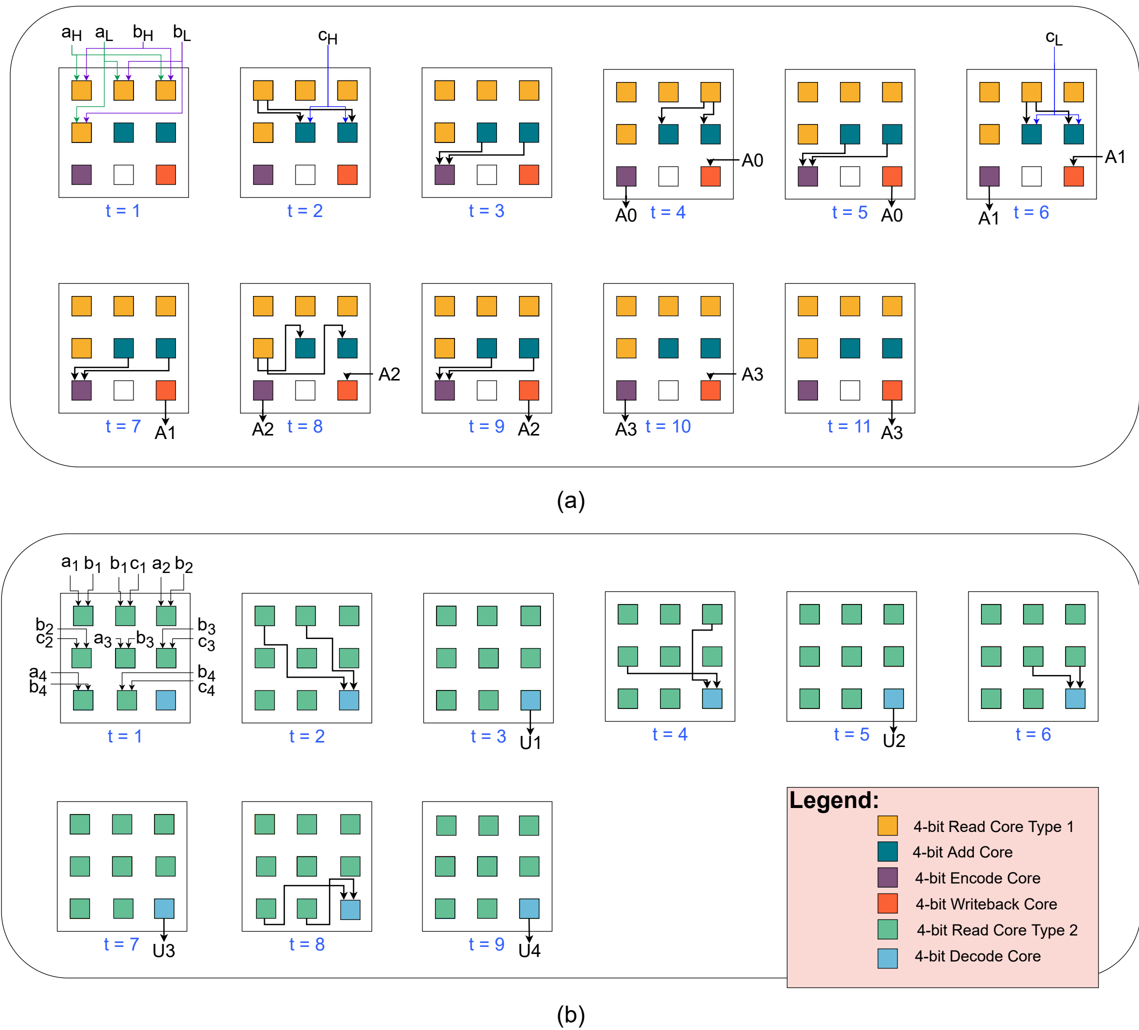

Figure 4.6: Sequential model of both encoding and decoding operations on the pPIM architecture including (a) encoding sequential model and (b) decoding sequential model. The inputs **'a'**, **'b'**, and **'c'** indicate input to the cluster from memory with the high and low segments of memory dictated by the subscripts **'H'** and **'L'**, respectively. The input and the output of the encoding and decoding schemes are dictated by **'A'** for encoding and **'U'** for decoding.

implementation of the Control Unit. The encoding component utilizes more power due to its higher latency values; however, the decoding component consumes fewer wattage as it takes fewer clock cycles as compared to the encoding process.

In addition, Table 4.3 compares a MAC operation to the proposed encoding and decoding operations. The MAC operation characteristics are displayed without the usage of the encoding and decoding schemes. The encoding and decoding schemes

Table 4.3: Synthesis Results

Component	Delay (ns)	Power (mW)	Active Area (μm^2)
PIM Core	0.8	2.7	4616.85
PIM Encoding	8.8	11.18	6250
PIM Decoding	7.2	10.02	6250
PIM Cluster (MAC Operation)	7.2	5.2	41551.66

both consume more power and have larger amounts of delay in comparison to the MAC operation. More clock cycles are used to successfully encode the data. This contributes to larger amounts of latency. In addition, more cores are used per clock cycle in transporting data and performing operations. This demands a higher power consumption. However, the active area consumed is much lower in comparison to the MAC operation. This allows for minimal overhead to the pPIM architecture.

4.5.2 Encoding Functionality

We evaluate the functionality of the encoding design via number of vulnerable weights before and after encoding. We evaluate the functionality of the RH-encoding on two CNNs: AlexNet [35] and VGG-16 [36]. Figures 4.7 (a) and 4.7 (b) present the comparison of vulnerable weights found within the two networks with and without encoding for each triplet set (denoted as input/output bit positions) of weights within AlexNet and VGG-16. It can be observed that the RH-encoding scheme successfully decrease the number of vulnerable, or row-stripe, weights for a majority of triplet pairs. However, there is an increase in the number of vulnerable weights for the final triplet pair as the RH-encoding scheme favors bits towards the MSB as a bit flip there would be less significant than a bit flip towards the most significant bits. Bit patterns have a wider variety of data patterns towards the final triplet pair. The proposed RH-encoding scheme may sometimes create a larger number of row-stripe

patterns for the final triplet pair, but compensates and secures for the bits towards the most significant bits, those that can be impacted the most via a bit-flip attack. The proposed RH-encoding scheme successfully secures bits towards the MSB portion, preserving the most sensitive bits from the RH attacks.

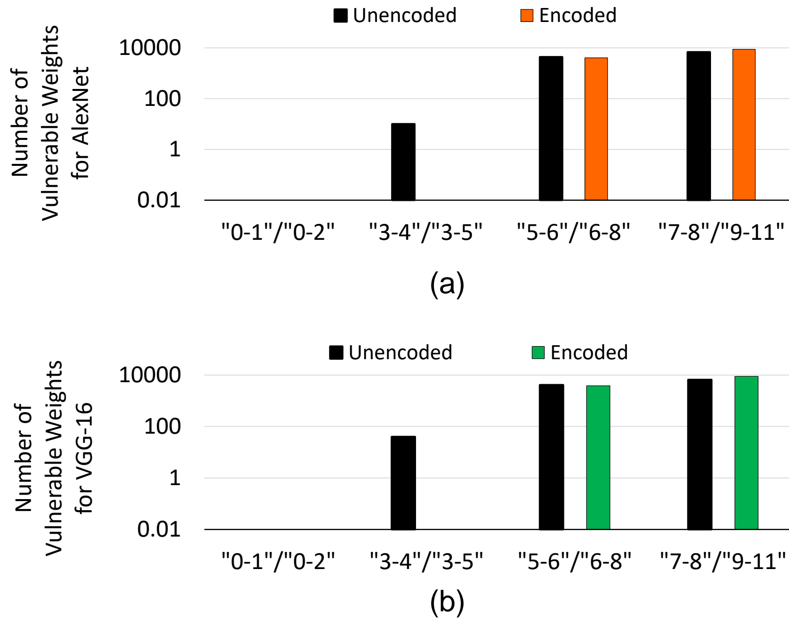

Figure 4.7: Encoding Functionality Comparison of (a) AlexNet, and (b) VGG-16, with and without encoding

4.5.3 Encoding/Decoding Performance Evaluation

We evaluate the performance of the pPIM architecture equipped with our proposed encoding/decoding schemes alongside CNN inference. We evaluate the performance of the pPIM configurations for four other CNNs: ResNet 18, ResNet 34, ResNet 50, and VGG 16. Figure 4.8 present the comparison of energy consumption and performance throughput of CNN inferences with and without encoding/decoding. It can be observed that the proposed encoding/decoding scheme offers both a low-energy usage and a low drop in throughput due to a low computational workload from our proposed encoding/decoding design. This highlights the merit of the proposed encoding/decoding scheme on PIM-based architectures, reducing the total number of

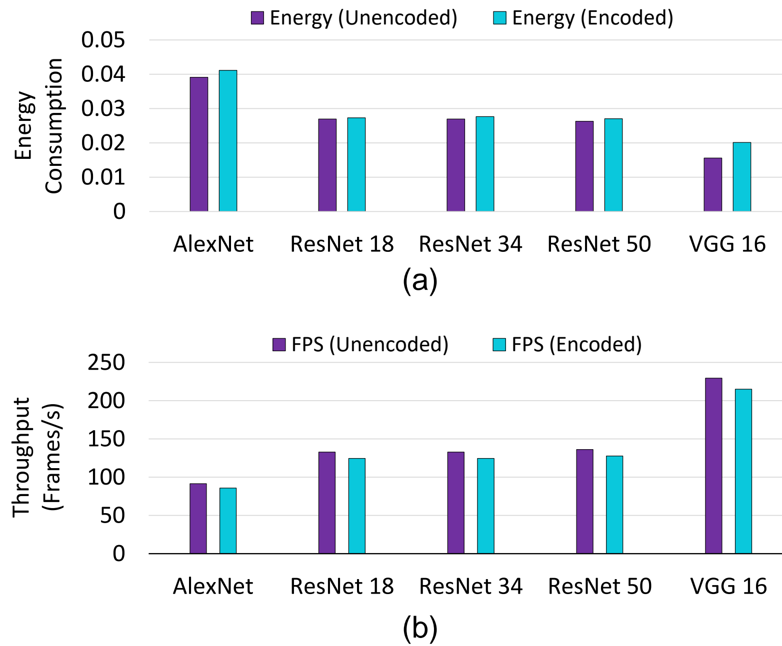

Figure 4.8: Comparison of (a) throughput and (b) power consumption of various CNN algorithms with and without RH-encoding scheme

vulnerable weights within CNN algorithms with minimal energy and throughput.

4.5.4 Pipelined RH-encoding scheme

Our proposed encoding scheme can be implemented with fewer states inside the pPIM cluster in a pipe-lined manner. Certain operations can take up portions of the clock cycle during a stage. This allows for both encoding and decoding to be done in fewer cycles and with multi-step operations within a clock cycle. The encoding operation is reduced from eleven stages to eight stages. The deocoding operation is reduced from nine stages to six stages. Figure 4.9 shows the pipelined RH-encoding operation mapping scheme.

The pipelined RH-encoding scheme is characterized using post-synthesis models using the Synopsys Design Compiler at the 28 nm technology node. The hardware synthesis and pPIM results are shown below in Table 4.4.

It can be observed that there is a reduction in the latency of the encoding and decoding operations. As compared to the latency values in Table 4.3, the latency

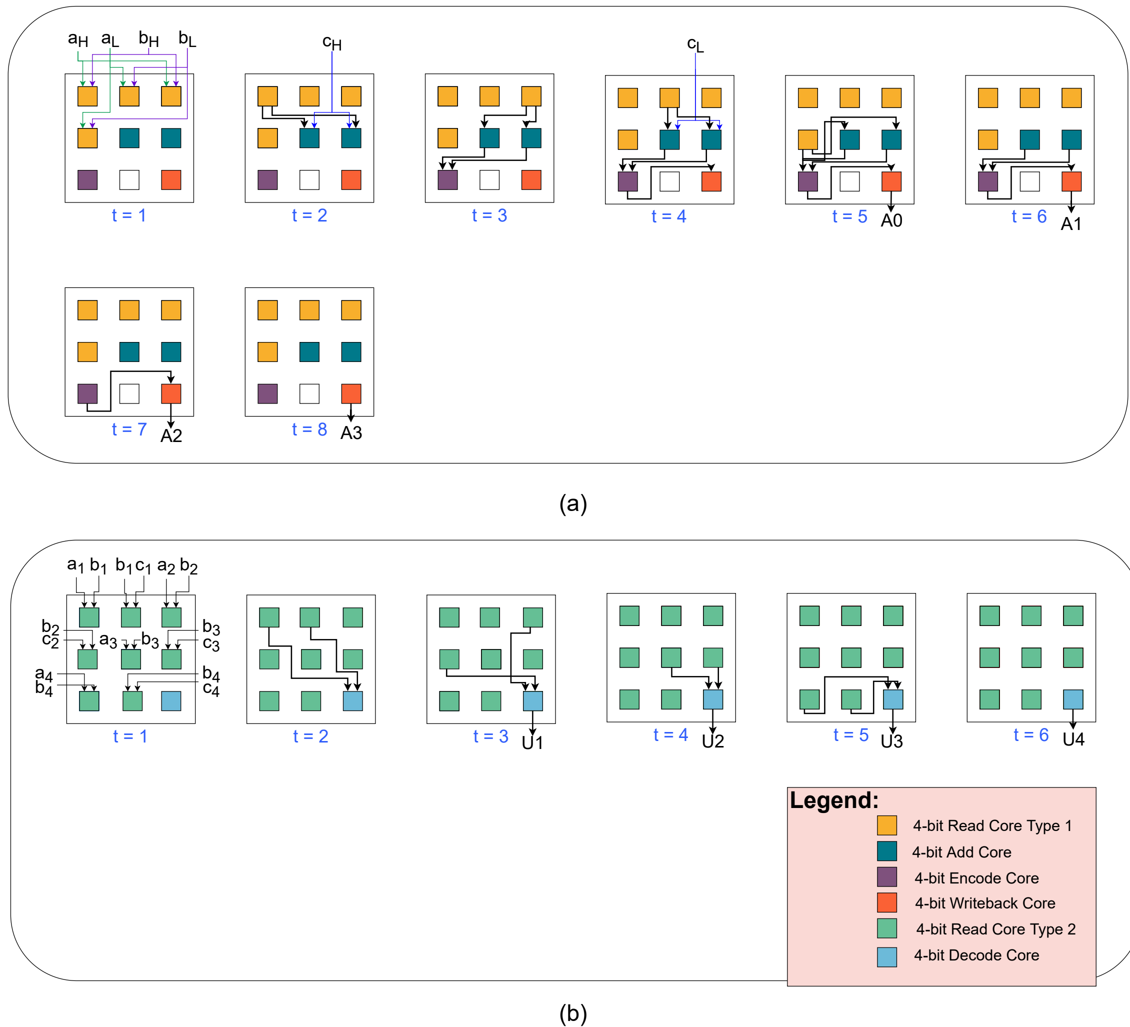

Figure 4.9: Pipelined Sequential model of both encoding and decoding operations on the pPIM architecture including (a) encoding sequential model and (b) decoding sequential model. The inputs **'a'**, **'b'**, and **'c'** indicate input to the cluster from memory with the high and low segments of memory dictated by the subscripts **'H'** and **'L'**, respectively. The input and the output of the encoding and decoding schemes are dictated by **'A'** for encoding and **'U'** for decoding.

has decreased due to pipelining. This allows for the RH-encoding scheme to be accomplished in a shorter amount of time with equal functionality on the pPIM architecture.

We also evaluate the performance of the pPIM architecture equipped with the pipelined encoding/decoding schemes alongside CNN inference. We evaluate the performance of the pPIM configurations for four other CNNs: ResNet 18, ResNet

Table 4.4: Pipelined Synthesis Results

Component	Delay (ns)	Power (mW)	Active Area (μm^2)
PIM Core	0.8	2.7	4616.85
PIM Encoding	6.4	11.18	6250
PIM Decoding	4.8	10.02	6250
PIM Cluster (MAC Operation)	7.2	5.2	41551.66

34, ResNet 50, and VGG 16. Figure 4.10 presents the comparison of performance throughput and energy consumption of CNN inferences with and without encoding/decoding as energy consumption remains the same. It can be observed that the proposed encoding/decoding scheme allows for higher throughput and fewer energy consumption due to the fewer amount of stages as compared to Figure 4.8. This highlights the merit of the pipelined encoding/decoding scheme on PIM-based architectures, reducing the total number of vulnerable weights within CNN algorithms with minimal energy consumption and throughput.

4.6 DRAM Design Analysis

As the technology node improves, memory cells gradually become closer, with both rows and columns within memory becoming denser; therefore, as the area of memory cells reduce, disturbance issues can appear within DRAM, hindering novel ideas such as Deep Learning (DL) and Artificial Intelligence (AI), which utilize large data sets to understand applications such as automotive, mobile edge devices, medical imaging applications, and enterprise storage. Some of these disturbance errors can be associated with DRAM parasitic capacitance as well as Row Hammer attacks. Parasitic capacitance can occur due to effects of bit line coupling as the signal between adjacent interconnect values vary. If two adjacent interconnects switch in the same direction, no coupling capacitance would occur, however, if one or both neighboring

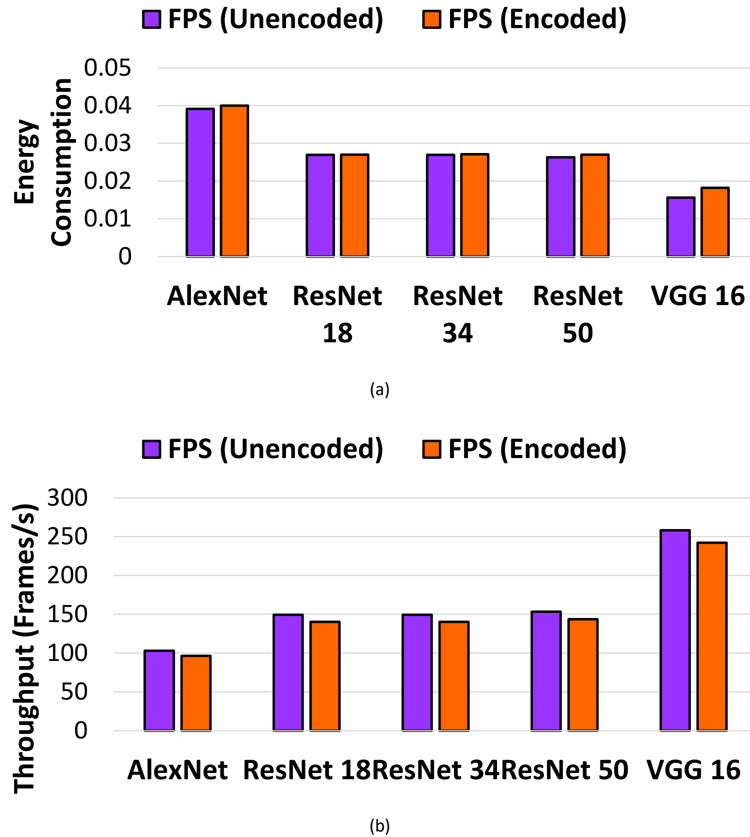

Figure 4.10: Comparison of throughput of various CNN algorithms with and without pipelined RH-encoding scheme

interconnects vary in opposite directions, coupling capacitance is introduced, creating additional noise and disturbance errors [37]. Equation 1 below shows the RH induced noise voltage swing, V_{RH}, generated from N number of row hammers both above and below the victim cell, induced due to the coupling capacitance, C_c between the memory cells. The DRAM cell capacitance and power supply voltage are denoted by C_{DRAM} and V_{DD}, respectively.

$$V_{RH} = N V_{DD} \frac{2C_c}{C_c + C_{DRAM}} \tag{4.1}$$

Row hammer attacks can occur due to continuous pre-charge and activation of rows of memory, effectively moving charge between rows of memory via coupling capacitance between cells from adjacent rows. This causes bit flips to occur in memory

without directly accessing the victim rows. The ratio of DRAM cell-to-cell coupling capacitance, defined here as β, versus the potential number of row hammerings, N, occurring in a DRAM cell is derived via (1) as shown in (2). We considered the noise margin of the DRAM cells to be 10% of V_{DD} in Equation 2.

$$N = (0.1)\frac{\beta + 1}{2\beta} \tag{4.2}$$

Using (2), the variation of N as a function of β is shown in Figure 4.11.

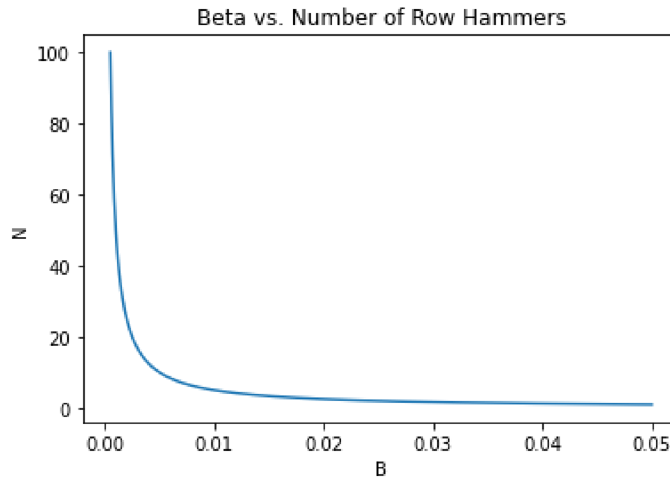

Figure 4.11: DRAM Coupling Capacitance vs. Number of Row Hammer Attacks

Figure 4.11 shows that a very low β value, typically less than 0.01, the number of row hammerings vary significantly. however, after the parameter surpasses 0.01, the number of row hammerings becomes constant. As there is more coupling capacitance introduced signified by increasing in *beta*, memory cells are subjected more to the parasitic effects of coupling capacitance such as adjacent aggressor rows switching in opposite directions or remaining quiet. On the contrary, with very little coupling capacitance as signified by a decreasing *beta*, a higher number of row hammering attacks are necessary to inject the same 10% noise.

Chapter 5

Conclusion and Future Work

As the technology node advances, the gap between computational and memory performance widens, causing a major performance bottleneck in contemporary computing devices. As computational power improves at a consistent rate, memory lags behind, creating a 'memory-wall' bottleneck. In traditional von Neumann Architectures, both the computing and storage unit are separated via interconnects over which data travel for either computation or storage. Both power-consumption and data-transfer bandwidth are impeded due to the physical separation of the processor and memory component and constant data-transfer over interconnects, preventing a processing device from reaching its peak performance.

Processing-in-Memory is a non-von Neumann computing model that aims to alleviate the performance bottleneck faced by traditional computing devices for data-centric applications. The traditional computing devices are based on the von-Neumann computing model that draws a distinct separation between the processor and the memory and imposes a sequential or pipe-lined execution model. However, the off-chip data communications between the memory and the processor are associated with high latency, high power consumption, and a limited bandwidth. PIM offers a solution to this phenomenon by locating the processing elements inside the memory chip itself. The benefit of this is that the data-communication latency and energy consumption are virtually eliminated as the data is processed within the same chip.

Moreover, implementation of massively parallel processing within the memory chip is possible by leveraging its ultra-high internal data-movement bandwidth.

The programmable PIM architecture delivers provides high-performance along with the use of programmable LUT-based multiplexers in accomplishing a plethora of applications. In comparison to contemporary PIM architectures, the pPIM architecture achieves better performance while consuming less power. Due to the superior performance and power efficiency of the pPIM compared to the other architectures, we choose the pPIM as a case study to evaluate the effect of RH-encoding.

The past few decades were marked by the rapid downscaling of semiconductor technologies. This has resulted in a steady growth of the number of transistors and therefore the processing capabilities of the computing devices. The memory devices such as the DRAM have also undergone a similar downscaling trend, packing more storage capacity with lower power consumption, at lower manufacturing costs . However, the downscaling of the technology has also lead to unique reliability issues in the DRAM. Capacitive interaction between neighboring cells in the DRAM memory chips on 35nm or lower technology nodes can lead to data corruption. Termed as '*Row-Hammering*', it was shown that severe data corruption and manipulation can be pulled off by cleverly manipulating this phenomenon in the off-the-shelf commodity DRAM chips. The security concern arising from the possibility of row-hammering attacks becomes even more critical for the emerging Processing in Memory (PIM) computing paradigm.

The security and reliability concerns arising from the discovery of Row-Hammering on the DRAM technology affect all PIM architectures implemented on this memory platform. A few approaches to prevent row-hammering attacks have already been proposed, mostly dominated by the usage of counters and a probabilistic approach. These approaches include Counter-based Activation (CBA), Counter-based Tree (CBT), row remapping, probabilistic adjacent row activation (PARA), and Time Window Counter

(TWiCe). However, common to all these approaches is the involvement of complex and bulky intelligent logic circuitry to monitor and actively prevent row-hammering corruption.

A method is proposed for preventing the row-hammering attacks on the DRAM memory by performing RH-encoding of the data. We leverage a recent programmable PIM architecture called pPIM for implementing the RH-encoding/decoding scheme with low latency and ultra-high efficiency. The pPIM architecture features clusters of programmable LUTs that work together to perform virtually any logic or arithmetic operations. We also evaluate the latency, energy efficiency, and area overhead to the baseline pPIM architecture caused by the implementation of our proposed row-hammering preventing coding scheme and benchmark its performance with other latest PIM architectures. We achieve a reduction in the number of vulnerable bits for a majority of the bits at the cost of throughput and power consumption. We achieve a reduction in the number of vulnerable bits for a majority of the bits at the cost of throughput and power consumption.

5.1 Future Work

The presented work delivers synthesized models for a PIM architecture and the RH-encoding scheme. In the future, improvements on the DRAM architecture should be made to minimize the effects of the novel Row Hammer security bug as the technology node improves alongside finding alternative solutions to counter RH attacks. As the technology node improves, DRAM architecture will become denser and smaller. Rows and columns inside the memory architecture may cause higher amounts of induction and capacitance. In addition, new issues other than row hammer may appear as the technology node improves. Therefore, an investigation into DRAM design is necessary to combat and secure against vulnerabilities.

Further steps are needed to minimize the vulnerability of weights in all bit po-

sitions of weights from the row hammer attack. An investigation into the encoding scheme is necessary to combat the row hammer attack further. These steps involve the development of multiple encoding schemes operating on different sections or portions of the weights in neural networks. The usage of multiple encoding schemes on different portions of the weights would reduce the number of vulnerable weights in a CNN. However, multiple encoding schemes may require more multiple operation mapping configurations of the pPIM architecture. This may incur higher power consumption and latency issues but aid in securing all of the weights in a network.

Bibliography

[1] S. Gogna and A. Ankolekar, "Processing in memory," pp. 1–6, 2020.

[2] Y. Kim, R. Daly, J. Kim, C. Fallin, J. H. Lee, D. Lee, C. Wilkerson, K. Lai, and O. Mutlu, "Flipping bits in memory without accessing them: An experimental study of dram disturbance errors," in *2014 ACM/IEEE 41st International Symposium on Computer Architecture (ISCA)*, 2014, pp. 361–372.

[3] M. Connolly, "A programmable processing-in-memory architecture for memory intensive applications," *Rochester Institute of Technology*, vol. 1, no. 1, pp. 1–53, 2021.

[4] P. R. Sutradhar, M. Connolly, S. Bavikadi, S. M. Pudukotai Dinakarrao, M. A. Indovina, and A. Ganguly, "ppim: A programmable processor-in-memory architecture with precision-scaling for deep learning," *IEEE Computer Architecture Letters*, vol. 19, no. 2, pp. 118–121, 2020.

[5] L. Song, Y. Zhuo, X. Qian, H. Li, and Y. Chen, "Graphr: Accelerating graph processing using reram," in *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2018, pp. 531–543.

[6] A. Ganguly, "Interconnect architectures and signal integrity," in *SoC Physical Design Issues*, 2020, pp. 1–38.

[7] J. Jiang, K. Parto, W. Cao, and K. Banerjee, "Monolithic-3d integration with 2d materials: Toward ultimate vertically-scaled 3d-ics," in *2018 IEEE SOI-3D-Subthreshold Microelectronics Technology Unified Conference (S3S)*, 2018, pp. 1–3.

[8] A. Shacham, B. G. Lee, A. Biberman, K. Bergman, and L. P. Carloni, "Photonic noc for dma communications in chip multiprocessors," in *15th Annual IEEE Symposium on High-Performance Interconnects (HOTI 2007)*, 2007, pp. 29–38.

[9] S. Gogna, "An investigation of the effects of the row hammering bug on cnns/dnns," pp. 1–5, 2020.

[10] J. S. Kim, M. Patel, A. G. Yağlıkçı, H. Hassan, R. Azizi, L. Orosa, and O. Mutlu, "Revisiting rowhammer: An experimental analysis of modern dram devices and mitigation techniques," in *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, 2020, pp. 638–651.

[11] S. Gogna, "Evaluation of crosstalk avoidance coding schemes," pp. 1–5, 2020.

[12] K. S. Yim, "The rowhammer attack injection methodology," in *2016 IEEE 35th Symposium on Reliable Distributed Systems (SRDS)*, 2016, pp. 1–10.

[13] O. Mutlu and J. S. Kim, "Rowhammer: A retrospective," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 39, no. 8, pp. 1555–1571, 2020.

[14] V. Van der Veen, "Drammer: Deterministic rowhammer attacks on mobile platforms," 2016.

[15] P. Frigo, "Grand pwning unit: Accelerating microarchitectural attacks with the gpu," 2018.

[16] I. Kang, E. Lee, and J. H. Ahn, "Cat-two: Counter-based adaptive tree, time window optimized for dram row-hammer prevention," *IEEE Access*, vol. 8, pp. 17 366–17 377, 2020.

[17] E. Lee, I. Kang, S. Lee, G. E. Suh, and J. H. Ahn, "Twice: Preventing row-hammering by exploiting time window counters," in *2019 ACM/IEEE 46th Annual International Symposium on Computer Architecture (ISCA)*, 2019, pp. 385–396.

[18] Q. Deng, Y. Zhang, M. Zhang, and J. Yang, "Lacc: Exploiting lookup table-based fast and accurate vector multiplication in dram-based cnn accelerator," in *2019 56th ACM/IEEE Design Automation Conference (DAC)*, 2019, pp. 1–6.

[19] C. Wu, W. Shan, and J. Xu, "ynamic adaptation of approximate bit-width for cnns based on quantitative error resilience," in *2019 IEEE/ACM International Symposium on Nanoscale Architectures (NANOARCH)*, 2019, pp. 1–6.

[20] J. T. Pawlowski, "Hybrid memory cube (hmc)," in *2011 IEEE Hot Chips 23 Symposium (HCS)*, 2011, pp. 1–24.

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

[22] S. Xu, X. Chen, Y. Wang, Y. Han, X. Qian, and X. Li, "Pimsim: A flexible and detailed processing-in-memory simulator," *IEEE Computer Architecture Letters*, vol. 18, no. 1, pp. 6–9, 2019.

[23] M. Hailesellasie, J. Nelson, F. Khalid, and S. R. Hasan, "Vaws: Vulnerability analysis of neural networks using weight sensitivity," in *2019 IEEE 62nd International Midwest Symposium on Circuits and Systems (MWSCAS)*, 2019, pp. 650–653.

[24] S. M. Seyedzadeh, A. K. Jones, and R. Melhem, "Mitigating wordline crosstalk using adaptive trees of counters," in *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, 2018, pp. 612–623.

[25] Y. Wang, Y. Liu, P. Wu, and Z. Zhang, "Detect dram disturbance error by using disturbance bin counters," *IEEE Computer Architecture Letters*, vol. 18, no. 1, pp. 35–38, 2019.

[26] M. Kim, J. Choi, H. Kim, and H. Lee, "An effective dram address remapping for mitigating rowhammer errors," *IEEE Transactions on Computers*, vol. 68, no. 10, pp. 1428–1441, 2019.

[27] Q. Liu, W. Wen, and Y. Wang, "Concurrent weight encoding-based detection for bit-flip attack on neural network accelerators," in *2020 IEEE/ACM International Conference On Computer Aided Design (ICCAD)*, 2020, pp. 1–8.

[28] A. Sodani, "Knights landing (knl): 2nd generation intel® xeon phi processor," in *2015 IEEE Hot Chips 27 Symposium (HCS)*, 2015, pp. 1–24.

[29] A. A. Awan, H. Subramoni, and D. K. Panda, "An in-depth performance characterization of cpu- and gpu-based dnn training on modern architectures," in *Proceedings of the Machine Learning on HPC Environments*, ser. MLHPC'17. New York, NY, USA: Association for Computing Machinery, 2017. [Online]. Available: https://doi.org/10.1145/3146347.3146356

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

[31] Q. Deng, L. Jiang, Y. Zhang, M. Zhang, and J. Yang, "Dracc: a dram based accelerator for accurate cnn inference," in *2018 55th ACM/ESDA/IEEE Design Automation Conference (DAC)*, 2018, pp. 1–6.

[32] C. Eckert, X. Wang, J. Wang, A. Subramaniyan, R. Iyer, D. Sylvester, D. Blaaauw, and R. Das, "Neural cache: Bit-serial in-cache acceleration of deep neural networks," in *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, 2018, pp. 383–396.

[33] L. Deng, "The mnist database of handwritten digit images for machine learning research," *IEEE Signal Processing Magazine*, vol. 29, no. 6, pp. 141–142, 2012.

[34] K. K. Chang, P. J. Nair, D. Lee, S. Ghose, M. K. Qureshi, and O. Mutlu, "Low-cost inter-linked subarrays (lisa): Enabling fast inter-subarray data movement in dram," in *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, March 2016, pp. 568–580.

[35] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *Proceedings of the 25th International Conference on Neural Information Processing Systems - Volume 1*, ser. NIPS'12. Red Hook, NY, USA: Curran Associates Inc., 2012, p. 1097–1105.

[36] S. Liu and W. Deng, "Very deep convolutional neural network based image classification using small training sample size," in *2015 3rd IAPR Asian Conference on Pattern Recognition (ACPR)*, 2015, pp. 730–734.

[37] T. Sakurai, "Closed-form expressions for interconnection delay, coupling, and crosstalk in vlsis," *IEEE Transactions on Electron Devices*, vol. 40, no. 1, pp. 118–124, 1993.