Near-Memory acceleration of Convolutional Neural Networks by exploiting Parallelism, Sparsity, and Redundancy

Thesis submitted in partial fulfilment of the requirements for the award of the degree of

Doctor of Philosophy

in

Computer Science and Engineering

by

Palash Das

Under the supervision of

Prof. Hemangee K. Kapoor

Department of Computer Science and Engineering

Indian Institute of Technology Guwahati

Guwahati - 781039 Assam India

December, 2021

Dedicated to

My beloved Parents

and

Robi Mama (late)

It is from their love, care and support, I derive my strengths...

Acknowledgements

It is a great pleasure for me to thank all the people who have supported me during my Ph.D. at IIT Guwahati. First and foremost, I thank my supervisor, Prof. Hemangee K. Kapoor, for her guidance, encouragement, and extensive help over the last few years. She has given me the freedom to pursue research ideas and develop my research skills. I profusely thank her for correcting my mistakes and keeping me engaged in my Ph.D. work.

I am thankful to all my Doctoral Committee Members: Prof. Purandar Bhaduri, Dr. Aryabartta Sahu, and Dr. Moumita Patra, for their productive and constructive suggestions for my thesis work. I firmly believe that their opinions and comments helped me to shape my final thesis. Additionally, I would like to thank Prof. Jatindra Kumar Deka, the Head of the Department of Computer Science and Engineering, and other faculty members from the departments for their constant support and help. Furthermore, my sincere thanks to Prof. Kalpesh Kapoor for various academic/non-academic suggestions. I especially thank the EEE department and Dr. Nagarjuna Nallam for providing me synthesis setup, which I have extensively used in my thesis work.

During my Ph.D., I got the opportunity to work with Dr. Shirshendu Das, Dr. Shounak Chakraborty, Dr. Khushboo Rani, DR. Sukarn Agarwal, Sheel Sindhu Manohar, Arijit Nath, Aswathy N. S., Imlijungla Longchar, Neeraj Sharma, and Swati Upadhyay. Sharing knowledge and fruitful technical discussions with them countless times helped me carry out my research work.

I sincerely thank Mr. Raktajit Pathak, Mr. Nanu Alan Kachari, Mr. BhrigurajBorah, Mr. Monojit Bhattacharjee, Mrs. Gauri Deori, and all other department staff members for helping me in different ways and at different times during my stay at IIT Guwahati. I would also like to thank the student affairs section for providing an on-campus hostel facility. Last but not least, I am conveying my appreciation to security guards, janitors, hostel mess, and canteen staffs for making my life smooth on the IITG campus.

I want to thank Intel India for supporting my thesis by providing a fellowship. Additionally, technical discussion with Intel has also helped me to solve several problems in my thesis work.

I want to thank Prof. Ajay Joshi from Boston University for his suggestions/corrections during my thesis work. We arranged several online meetings to discuss ideas for my work.

I am extremely fortunate to have two of my closest friends on campus. I would like to thank Aparajita and Sayan for their continuous support during the whole journey. Discussion with Aparajita has also helped me in the technical contributions of my thesis. She has given me motivation, strength, and smiles during the several ups and downs in the entire journey. Sayan (the funniest guy I have ever seen) has brought countless smiles during my tough times. Thank you, Sayan, for bringing dinners several times when I slept early. I can keep writing lines about the contributions of these two persons in the entire journey.

I would like to especially thank Anasua, Abhijit, Arijit, Pawan, Ujjwal da, Chinmay sir, and Tamen for their support in this journey. You guys have provided beautiful moments which will remain green in the future.

Thank you, Dr. Pradip Das (da) and Milan, for helping me learn guitar, which will be there with me ever after.

Finally, I would like to thank my backbone (my family), without whom I would not have come to the position where I am. I am happy to make your (Bapi) dream come true to see "Dr." in front of my name. I am feeling extremely emotional to write about your contributions to the entire journey. Thank you for sending me WhatsApp messages during my failures, saying, "Don't worry, go ahead." Thanks to my mother for giving me all the updates about my home. The start of a day with your phone is magical. I would also like to thank my spouse for being so supportive during the entire journey. Thanks for all the sacrifices that you have made to shape my Ph.D. I also thank all my brothers and sisters for their best wishes during the journey. I express my apology if I am missing some important names to be acknowledged.

December 11, 2021 Palash Das

Declaration

I certify that

- The work contained in this thesis is original and has been done by myself and under the general supervision of my supervisor(s).
- The work reported herein has not been submitted to any other Institute for any degree or diploma.
- Whenever I have used materials (concepts, ideas, text, expressions, data, graphs, diagrams, theoretical analysis, results, etc.) from other sources, I have given due credit by citing them in the text of the thesis and giving their details in the references. Elaborate sentences used verbatim from published work have been clearly identified and quoted.
- I also affirm that no part of this thesis can be considered plagiarism to the best of my knowledge and understanding and take complete responsibility if any complaint arises.
- I am fully aware that my thesis supervisor(s) are not in a position to check for any possible instance of plagiarism within this submitted work.

December 11, 2021 Palash Das

Department of Computer Science and Engineering
Indian Institute of Technology Guwahati
Guwahati - 781039 Assam India

Dr. Hemange K. Kapoor
Professor
Email : hemangee@iitg.ac.in
Phone : +91-361-258-2363

Certificate

This is to certify that this thesis entitled "**Near-Memory acceleration of Convolutional Neural Networks by exploiting Parallelism, Sparsity, and Redundancy**" submitted by **Palash Das**, in partial fulfilment of the requirements for the award of the degree of Doctor of Philosophy, to the Indian Institute of Technology Guwahati, Assam, India, is a record of the bonafide research work carried out by him under my guidance and supervision at the Department of Computer Science and Engineering, Indian Institute of Technology Guwahati, Assam, India. To the best of my knowledge, no part of the work reported in this thesis has been presented for the award of any degree at any other institution.

Date: December 11, 2021
Place: Guwahati

Prof. Hemangee K. Kapoor
(Thesis Supervisor)

Abstract

The gap between the processing speed of the CPU and the access speed of the memory is becoming a bottleneck for many emerging applications. This gap can be reduced if the computation can be taken closer to the memory through near-memory processing (NMP). Among the logic options, application-specific integrated circuits (ASICs) are highly efficient in terms of power and area overhead for NMP logic integration. In this thesis, we aim to accelerate Convolutional Neural Networks (CNNs) by integrating custom hardware near the memory. As CNNs are widely used in several emerging applications, the designed hardware can be extensively used in all such cases. To design an NMP-based system with high performance and energy efficiency, we explore various techniques such as leveraging parallelism, exploiting data sparsity, and utilizing computation redundancy to reduce the number of operations. All such techniques result in hardware designs that implement the appropriate dataflow and data-parallel algorithm. The designs have positively impacted the system's performance and energy efficiency. To examine the deployability of the NMP approach, we perform experiments on various memory technologies like 3D memory, hybrid memory, and the commodity DRAM. Additionally, we also measure the efficacy of NMP for other applications like database operations. The proposed systems have performed substantially well while comparing them with various baselines and state-of-the-art works.

Contents

List of Figures

List of Tables

List of Tables

List of important Abbreviations

Terms	Abbreviations
NN	Neural Network
CNN	Convolutional Neural Network
DNN	Deep Neural Network
ReLU	Rectified Linear Unit
FC	Fully-Connected
NMP	Near-Memory Processing
PIM	Processing-In-Memory
ASIC	Application-Specific Integrated Circuit
FPGA	Field Programmable Gate Array
CGRA	Coarse-Grained Reconfigurable Array
CPU	Central Processing Unit
GPU	Graphics Processing Unit
PE	Processing Element
DRAM	Dynamic Random Access Memory

DIMM	Dual In-Line Memory Module
PCM	Phase-Change Memory
HMC	Hybrid Memory Cube
TSV	Through-Silicon Via
LoB	Logic Base
NVM	Non-Volatile Memory
MAC	Multiply-Accumulate
LAM	LookAside Memory

1 Introduction

Moore's law [10] coupled with Dennard scaling [11] has helped in improving the processor chip's performance in terms of increased clock frequency with every generation. However, with the end of Dennard scaling, the chip's performance (in terms of frequency) could not be improved as it became a challenging task to handle the dissipated heat by the processor because of the power-wall problem [12]. Alternatively, researchers have moved towards the multi-core processor to keep up with Moore's law. The multi-core processors have been proven to be a huge success in terms of delivering high performance in the last few years.

However, with the advent of emerging data-intensive applications, the memory access cost in terms of latency and energy has become a severe bottleneck for multi-core systems. Several of these applications that involve deep neural network (DNN) algorithms suffer from low temporal locality [13, 14]. Consequently, the cache hierarchy becomes less effective, and the main memory footprints increase, resulting in overall degradation in the system's performance and energy consumption due to the longer access latencies and increased energy consumption. The off-chip main memory access is costly as it requires additional cycles and energy per access. One solution, to avoid the longer data access latency and energy before the processing of data, is near-memory processing (NMP) [3, 15, 16] or processing-in-memory (PIM) [17, 18].

1.1 Near-memory Processing (NMP)

In near-memory processing (NMP), computations are moved close to the data by placing some processing units near the memory. Initial proposal of NMP was made in 1990s [19]. This approach could not be adopted at that time as incorporating costlier logic with memory increased the cost per bit, and memory industries are cost-sensitive. However, the recent inclination towards data-intensive applications, the energy constraints due to the end of Dennard scaling, lower cost per bit, and the emergence of 3D memory have again encouraged

Table 1.1: Data movement overhead across the memory hierarchy.

	Access latency [20, 21]	Dynamic read energy per cache line (nJ) @32 *nm* node[22]
L1cache	2-4 cycles	0.07
L2 cache	8-12 cycles	0.27
L3 cache	25-40 cycles	0.54
Main memory	100-200 cycles	14.2

the resurgence of the NMP. The concept of NMP promotes less data movement through the memory hierarchy, leading to substantial improvements in the system's performance, energy efficiency, and reliability. The data movement between any two consecutive levels of memory requires different latency and energy. The same can be verified in Table 1.1. Note that the values reported in Table 1.1 give a glimpse of the data movement overhead at different levels of the memory hierarchy. These values can also vary based on the different configurations, generations, and technology. Processing data near the memory can save the overhead of data movement through the long path of the memory hierarchy, resulting in improved performance and energy efficiency for NMP-based systems.

1.2 Near-memory Processing (NMP) vs. Processing-in-Memory (PIM)

Although both the processing-in-memory (PIM) and near-memory processing (NMP) advocate bringing the computations close to the memory, there are a few differences between the PIM and NMP based systems. The following sections explain their brief differences.

1.2.1 Processing-in-Memory (PIM)

Characteristics of the memory (subarrays/cells) are explored, and/or small circuits are integrated with the memory cells to perform computations in processing-in-memory (PIM) architectures. The PIM operations allow computations to be done inside the memory where they are actually stored, thus eliminating the need for most data movements during computations. PIM is proposed to exploit the high internal bandwidth of the memory chips to accelerate computations by modifying the internal architecture or operations of the memory chip. Both the DRAM and non-volatile memory (NVM) like resistive RAM (ReRAM), phase-change memory (PCM), and spintronic memory have been explored to implement PIM

designs. In the PIM approach, the implementation of logic operations is highly dependant on the specific memory technology used. However, the modifications are usually less to keep the area overhead minimal. Some PIM-based solutions minimally change the cell architecture while primarily relying on altering memory commands from the memory controller. Several operations like copying row of cells [23], logical operations such as AND, NOT, and OR [24–26], and arithmetic operations such as addition, multiplication [27–29] have been implemented using the PIM-based approach. In DRAM, the charge sharing technique has also been explored to enable bulk AND and OR operations [27].

The PIM approach has been primarily used inside the emerging NVM memories. In [26], multiple rows in a resistive memory sub-array are activated to enable bitwise logical operations. Besides conventional logic operations, NVM-based PIM has also proven helpful for accelerating neural network (NN) computations [30, 31]. The NNs often contain a high amount of dot-product operations of synaptic weights and input feature values as their primitive operations. The cell conductance in the rows of NVM-based memristive crossbar arrays (MCA) can be used to represent synaptic weights, while the input feature values can be represented by the wordline voltages [32]. When current flows through each bitline, a dot product of input and weight is achieved in a column. This process helps in accelerating NN operations using the PIM approach.

1.2.2 Near-memory Processing (NMP)

While PIM promotes modifying the memory itself to implement computations inside the memory, near-memory processing (NMP) typically suggests integrating additional processing elements (PEs) close to the memory. The NMP design utilizes traditional PEs for the processing of data. The researchers have put in efforts to integrate the PEs both in 2D memories [2] and 3D memories [3]. While conventional DRAM is exploited as 2D memory, 3D memories like hybrid memory cube (HMC) [3, 15], and high bandwidth memory (HBM) [33] provide a suitable environment for integrating PEs close to the memory to leverage benefits of NMP. Compared to HBM, HMC architecture provides highly parallel access to the memory [3], and HMC also includes bandwidth multipliers for the PEs in its logic layer, which is not present in the case of HBM [34]. However, as the computations are not performed directly inside memory arrays, the PEs in NMP do not leverage the same degree of internal memory bandwidth available in PIM designs. One advantage of NMP designs is that they can execute more coarse-grained and complex computations compared to PIM designs. Apart from that, the computations are done in the analog domain for the NVM-based PIM, unlike the NMP approach. Consequently, communication with digital circuitry necessitates

Table 1.2: Design issues in NMP logic options [8].

NMP logic options	Area Efficiency	Power Efficiency	Flexibility
Programmable Cores	No	No	Yes
FPGA (fine-grained)	No	Yes	Yes
CGRA (coarse-grained)	Yes	No	Yes
ASICs	Yes	Yes	No

the use of analog-to-digital (ADC)/digital-to-analog (DAC) converters. These conversions degrade the signal precision. Additionally, the area/power overhead of ADCs/DACs is also significantly high. The ADC/DAC can take 85% [35] to 98% [36] of the total area/power of a neuromorphic computing system.

1.3 Challenges in the Practical Implementation of NMP Approach

There are a few challenges in implementing the NMP approach. They are as follows:

- The hardware-software interface should be implemented for practical implementation.
- NMP systems are distributed, and hence maintaining coherency of data is also a challenge.
- If thread-based applications (multi-threaded applications) are executed in NMP, then communication and synchronization between the threads have to be handled [13].
- If the older conventional programming model has to be used, the proper runtime interface needs to be implemented so that the end-user can be unaware of NMP implementations.

1.4 NMP Logic Options

We can primarily use four types of processing elements as an NMP logic option. Each of them has its own pros and cons. The options are shown in Table 1.2.

- Programmable cores are flexible as they are general purpose. However, they require more area and power, which is not preferable in the strict area/power budget of near-memory integration.

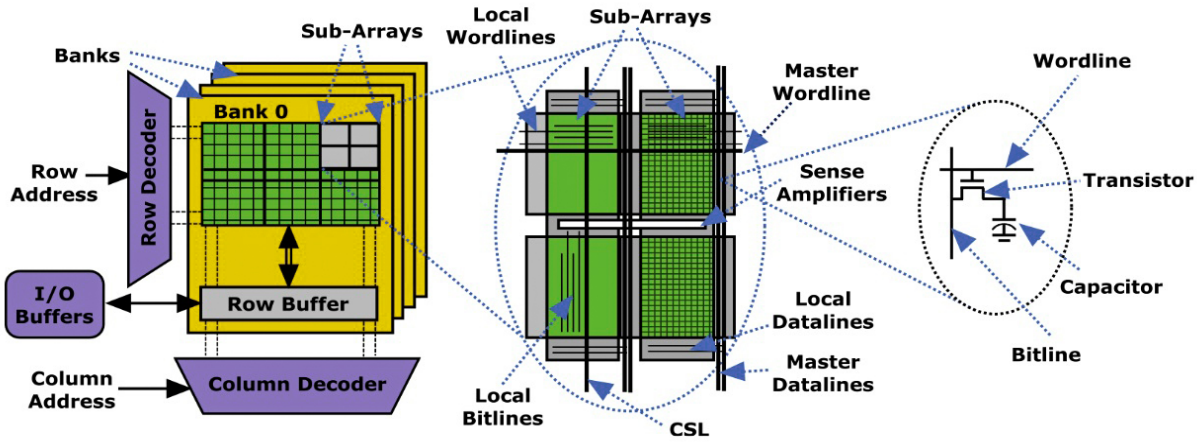


Figure 1.1: An abstract view of DRAM [1].

- Field Programmable Gate Arrays (FPGAs) are suitable in terms of both flexibility and power efficiency, but area efficiency is still an issue.
- Coarse-grain reconfigurable accelerators (CGRAs) provide decent performance and flexibility with reasonable area overhead in exchange for less power efficiency due to large interconnects.
- Application-specific integrated circuits (ASICs) are fast, power-efficient, and area-efficient. However, the flexibility is an issue as they are application-specific.

We choose ASICs and address the flexibility issue by targeting widely used algorithms that can be used in several applications. Below we discuss a few essential concepts used in the contributions of the thesis.

1.5 Memory Technologies

1.5.1 DRAM-based Main Memory

Dynamic random access memory (DRAM) is the most heavily used main memory system among available memory technologies. The architecture of a DRAM is explained in the following section.

1.5.1.1 An Abstract Architecture

Figure 1.1 shows an abstract view of a DRAM. Each DIMM (dual in-line memory module) of a DRAM is composed of multiple ranks, and each rank is made of multiple chips to increase

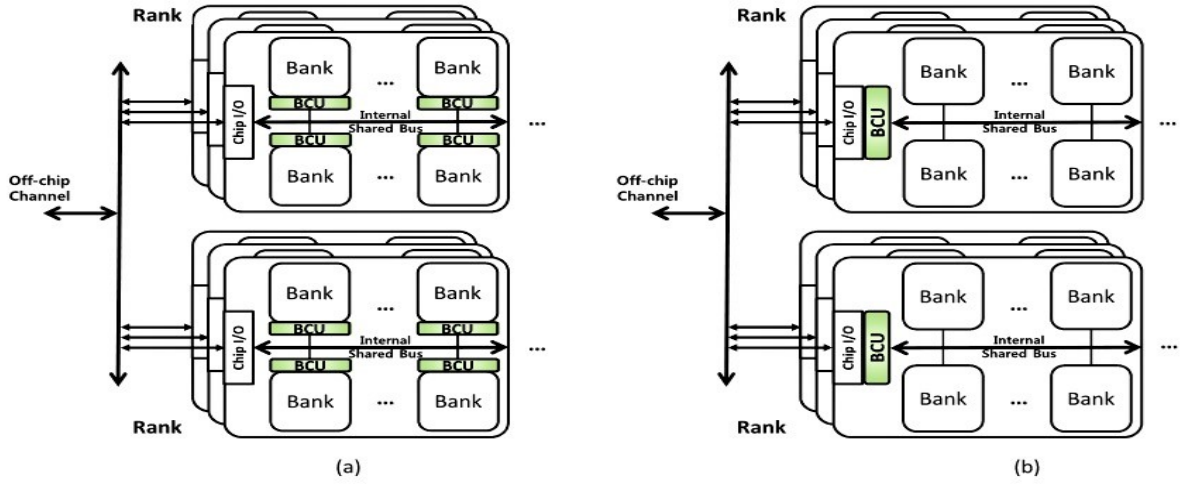


Figure 1.2: An abstract view of DRAM with integrated logic [2].

the width of the memory channel. Modern DRAM chips are typically multi-banked, each servicing requests independently from others. Each bank is composed of mats or the 2D-subarray of memory cells, as shown in Figure 1.1. An entire row of data is brought to the row buffer with the help of control wires like wordlines and bitlines. The row buffers primarily include a set of sense amplifiers that amplify the small change of voltage to a stable voltage level. A few bits are sent to the I/O pads by the column decoder from the entire row of a row buffer.

1.5.1.2 Opportunity of Near-Memory Logic Integration in DRAM

Apart from being used as the main memory, DRAMs have been used to implement the PIM concept [18, 37] by modifying the DRAM's architecture. Unlike PIM, researchers have also put in efforts to integrate additional logic in the DRAM's data path [2, 38] to leverage the benefits of near-memory processing. An example of bank-level and chip-level integration of additional logic is shown in Figure 1.2. The buffered compare units (BCU), proposed in the literature [2], are additional logic units integrated near the DRAM's bank and chip for data processing. The logic units can be integrated with each bank (bank-level integration) to exploit bank-level parallelism. Another place of near-DRAM logic integration can be the DRAM chip (chip-level integration). The chip-level integration helps in harnessing the benefits of chip-level parallelism. However, chip-level integration is comparatively lighter in terms of area/power overhead as fewer numbers of hardware (if 1 logic unit per chip/bank is used) are integrated compared to the bank-level integration. The additional logic close to the DRAM helps in faster access of data. However, there is also a strict area and power budget for near DRAM logic integration. Compared to DRAMs, 3D memories are more deserving

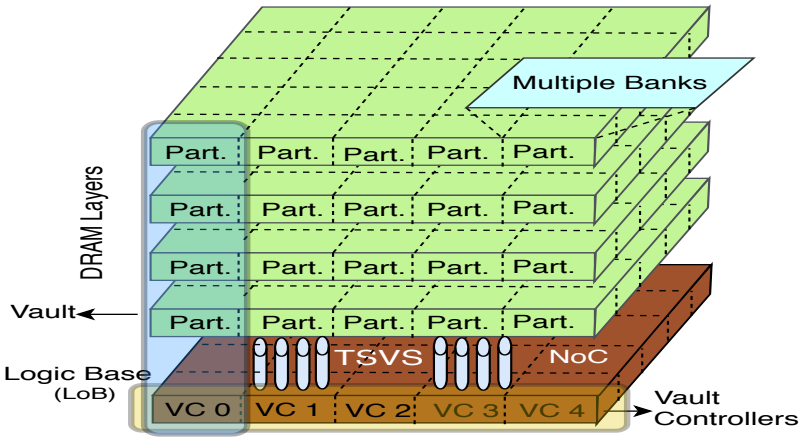


Figure 1.3: Hybrid Memory Cube (HMC).

candidates for near-memory integration of additional logic, as explained in the subsequent section.

1.5.2 3D Memory

3D memory [39] is one of the promising technology for the main memory system. Multiple memory layers are vertically stacked on top of a logic layer within a single package by exploiting low-capacitance through-silicon vias (TSVs). The two popular state-of-the-art 3D memories are Micron's Hybrid Memory Cube (HMC) [40], and JEDEC's High Bandwidth Memory (HBM) [41]. For our work, we choose HMC for the near-memory integration of the logic. The architecture of the HMC device is explained in the following section.

1.5.2.1 Architecture

Figure 1.3 shows the abstract view of the HMC memory. The HMC [40] device contains 4 to 8 dies of DRAM memory layers, as shown in Figure 1.3. These memory layers are stacked on top of each other to form a 3D memory. Each DRAM layer comprises multiple DRAM banks. A dedicated logic layer (LoB) containing all the interconnects and controllers is placed at the bottom of the memory layers. The high-speed Through-Silicon-Vias (TSVs) are used to transfer data between the DRAM layers and the logic layer (LoB). This logic layer helps in fetching and storing the data in the memory die. Each of the memory layers is divided into 16 to 32 partitions (Part.). The partitions are primarily a collection of memory banks. An individual stack of these partitions constitutes a vault (vertical slice of the memory die, as indicated in Figure 1.3). Each vault is equipped with a separate memory controller, namely the vault controller (VC). All the memory references corresponding to the individual vaults

are managed independently by the VCs. HMC uses the host memory controller and SerDes links for off-stack communications. A crossbar network is used to connect all the vaults of the HMC. HMC provides higher bandwidth (160-250 GBps) with 3 to 5 times lower access energy than a traditional DDR3 memory [40].

1.5.2.2 Opportunity of Near-Memory Logic Integration in 3D Memory

With the emergence of 3D memories like hybrid memory cube (HMC) [40] or high bandwidth memory (HBM) [41], the CMOS logic integration near to the memory has become more convenient primarily because of a few reasons. The reasons are as follows.

- The 3D-stacked memories like HMC include a dedicated logic layer (LoB) that provides additional areas for easy integration of complex CMOS logics near the memory.
- The 3D-stacked memories like HMC provide native support for executing simple and atomic instructions.
- The additional processing elements can exploit the high bandwidth through-silicon vias (TSVs) to fetch/store data in the memory die.
- The partitioning of memory die into individual vaults enables superior memory-level parallelism that can be exploited for improved system performance.

Though 3D memories are more convenient for NMP, we perform experiments on both 2D and 3D memories to estimate NMP's efficacy for various scenarios.

1.6 Data and Compute Intensive Applications

The emergence of data- and compute-intensive algorithms like widely used convolutional neural networks (CNNs) has also stimulated the concept of near-memory processing (NMP). In our work, we primarily choose CNNs for near-memory acceleration through custom hardware. The CNNs are compute-intensive as a huge number of operations are required to be performed in both its training and the inference phases [42–44]. The CNNs are also data-intensive, and they take large datasets as input. Apart from that, the temporal locality of data [13] is also low during the execution of CNNs. Consequently, the cache hierarchy often fails to provide a decent hit rate of data. This increases the main memory footprints for the CNN algorithms. It has been observed that the majority of power consumption in a neural network circuit comes from the memory accesses [14]. A large amount of data movement involved in the CNN's execution results in increased off-chip memory accesses

[45] and thereby degrades the overall system's performance. The memory access bottleneck can be eased by avoiding unnecessary data movements while executing the CNN algorithm. One solution to prevent the data movements in the cache hierarchy is to execute the CNN algorithm close to the memory using the concept of NMP. The lack of temporal locality and the abundant parallelism in a target application suggests that the NMP processing should deliver substantial improvements over conventional processing that waste energy on power-hungry processor-to-memory links. In this thesis, we aim to accelerate the CNNs through near-memory integration of custom hardware modules. For a more comprehensive measurement of NMP's efficacy, we also perform experiments on data-intensive and parallel database operations.

1.7 Convolutional Neural Networks

Deep learning using convolutional neural networks (CNNs) has become ubiquitous in a wide range of applications and cloud services. These include AI applications like data analysis in back-end data centers [46], click-through prediction for placing ads [47], speech recognition (e.g., Siri, Cortana), image recognition [48], video analysis [49], natural language processing [50], robotics [51], pharmaceutical research [52], and so on. A typical deep learning algorithm includes two phases: training and inference [53]. While the training phase is all about learning new parameters by the model, the inference phase assesses the performance of the trained model on unseen data. In general, training is performed only once (before a model is

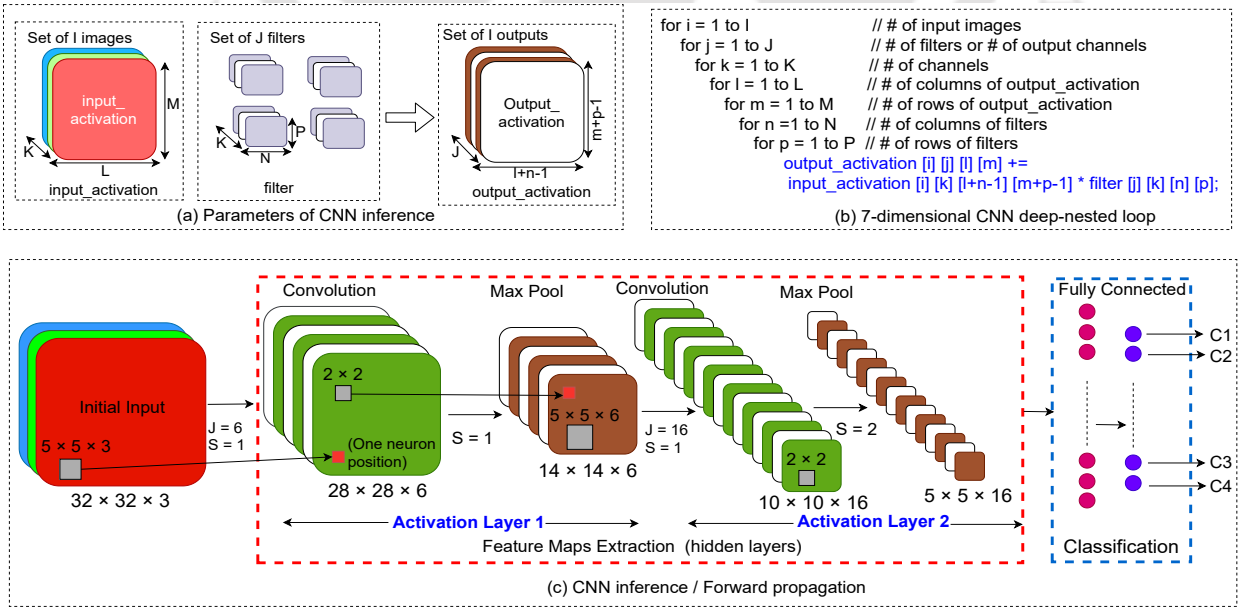


Figure 1.4: An abstract architecture of CNN inference.

used) with the support of GPUs or farms of GPUs, while real-time inferences are performed multiple times (as and when required) with the help of multi-core CPUs, GPUs, FPGAs, or dedicated hardware. In this thesis, we target to accelerate the inference phase near the memory to accelerate all the real-time inference runs. The detail of the inference is explained in the following section.

The CNN inference/forward propagation commonly consists of four layers: Convolution (CONV), Pooling, Non-linear, and Fully-Connected (FC). Figure 1.4 depicts the architecture of the feed-forward inference phase. The CNN algorithm starts by extracting the basic features and traverses through the hidden layers for more complicated features in the subsequent layers. The feature extraction layers of a CNN primarily consist of a series of convolutional layers that are made of dot products of two matrices (input_activations and weights) at each neuron position (shown in Figure 1.4). The pooling layers downsample the feature maps. Output feature maps of one layer are fed as the input feature maps to the next layer. A new set of weight matrices representing more sophisticated features is then used for the subsequent CONV layers. In the classification layer, the inference phase terminates by producing the prediction results for the set of input samples. The 7-dimensional deep-nested loops, shown in Figure 1.4 (b), represent operations of the activation layers. Interestingly, the sequence of loop executions is independent and can be reordered since multiply-add operations are associative in nature. Consequently, the computations can be distributed among the various processing units placed near the memory to achieve parallelism. The computations and data distribution patterns lead to different dataflows, one of the major differentiators of the CNN acceleration works. The different dataflows lead to distinct designs with substantial impacts on the system's performance and energy consumption.

1.8 Motivation

In the traditional Von Neumann computing architecture, the processor and the memory are strictly divided, and they are connected only through the bus. This architecture often creates hurdles for the data- and compute-intensive applications as latency and energy consumption for the data movements is considerably high. The extensive data movement also increases stress on the off-chip memory bandwidth. In Figure 1.5, an example of memory requirement in neural network processing can be visualized. Figure 1.5 presents the memory requirements for scene labeling with different input image sizes using convolutional neural network and MNIST with multilayer perceptron (MLP). It can be observed that it is hard to accommodate large input sizes and/or deep learning models even by the use of high-density eDRAM-based on-chip cache memory. This brings additional pressure on the off-chip memory bandwidth

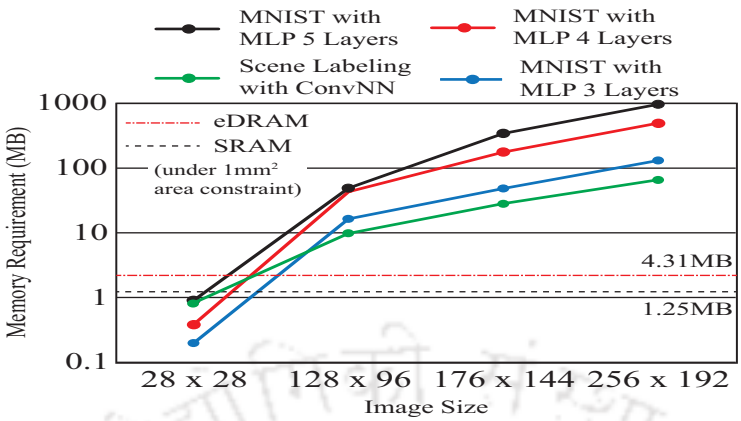


Figure 1.5: Memory requirement for scene labeling with different input image sizes using convolutional neural network and MNIST with MLP [3]. Memory capacity of SRAM and eDRAM is normalized by $1mm^2$ area constraint [4, 5].

Table 1.3: CNN Benchmarks [9].

Network	Application	Dataset	Number of Layers		MAC Operations (mils.)	
			CONV	FC	CONV	FC
AlexNet	Image Classification	ImageNet (224 × 224)	5	3	665.8	58.6
GoogleNet	Image Classification	ImageNet (224 × 224)	57	1	1233.0	1.0
VGG-M	Image Classification	ImageNet (224 × 224)	5	3	1141.8	85.9
VGG-S	Image Classification	ImageNet (224 × 224)	5	3	1901.5	96.4
VGG-19	Image Classification	ImageNet (224 × 224)	16	3	14999.8	123.6
MobileNet	Image Classification	ImageNet (224 × 224)	27	1	567.7	1.0
DenseNet-121	Image Classification	ImageNet (224 × 224)	120	1	3062.0	1.0
DPNet-92	Image Classification	ImageNet (224 × 224)	95	1	7384.5	2.7
ResNet-50	Image Classification	ImageNet (224 × 224)	53	1	3855.9	2.0
DnCNN	Image De-noising	FHD Images (1920 × 1080)	20	0	1.38×10^6	-
FFDNet	Image De-noising	FHD Images (1920 × 1080)	10	0	3.56×10^5	-
IRCNN	Image De-noising	FHD Images (1920 × 1080)	7	0	1.95×10^6	-
JointNet	Image De-mosaicking + De-noising	FHD Images (1920 × 1080)	19	0	3.25×10^4	-
VDSR	Super-resolution	FHD Images (1920 ×1080)	20	0	1.38×10^6	-

[3]. Processing the data near to the memory (NMP) can solve this problem by avoiding off-chip communications.

Apart from the memory requirements, CNNs are also computation demanding. Table 1.3 shows the number of multiply-accumulate (MAC) operations for the widely used CNN benchmarks. CPUs are not suitable for deep learning applications as there are usually fewer cores in a CPU chip. GPUs are often used to accomplish these millions of operations because of their several cores for parallel processing [54]. However, the GPUs are power-hungry platforms [15] and are not suitable for systems having power constraints. Additionally, high-end GPUs are also substantially expensive than the custom hardware-based accelerators [15]. The application-specific integrated circuits (ASICs) have become popular as energy-efficient and high-performing alternatives [55, 56]. Integrating these ASICs close to the memory can

further enhance the system's performance by amortizing the unnecessary data movement in the memory hierarchy. Hence, in this thesis, we arrive at the solution of integrating custom ASICs close to the memory while accelerating data- and compute-intensive applications like CNNs and database operations. The primary focus of the thesis is kept on CNN because of its widespread use in emerging applications like data analysis in back-end data centers, click-through prediction for placing ads, speech recognition (e.g., Siri, Cortana), image recognition, video analysis, natural language processing, robotics, pharmaceutical research, and so on.

In this thesis, we aim to accelerate convolutional neural networks (CNNs) through custom hardware integrated close to the memory. While the objective is to achieve high performance and high energy efficiency for the NMP architecture, the area overhead of the additional hardware is also kept minimum to accommodate them close to the memory. Apart from the hardware acceleration, other optimizations on the CNN algorithm have also been explored to optimize the hardware designs further. The following section summarizes the objective and outline of the works done for this thesis.

1.9 Objectives

The objective of this research is to investigate the efficacy of unconventional systems having near-memory processing ability. The proposed systems aim to deliver high performance and energy efficiency in exchange for minimal area overhead. The more specific goals of the research are mentioned in the following.

1. **Reducing data movements:** We aim to design near-memory processing-based systems that reduce the data movements in the memory hierarchy, leading to savings in latency and energy consumption.

2. **Designing accelerators:** We also aim to design custom accelerators for faster execution of the target application like CNNs and database operations. Towards achieving parallelism, multiple custom hardware modules are integrated into the proposed systems.

3. **Supporting widespread applications:** Providing support for a wide range of applications is another objective of this research. Custom accelerators are usually application-specific. Consequently, choosing a primitive algorithm for the accelerator is crucial for the usefulness of the system. We choose the widely used CNN algorithm and database operation to be accelerated near the memory. The selection of such algorithms can directly and indirectly benefit a number of recent applications.

4. **Achieving high throughput:** This research also aims to obtain high performance and energy efficiency while executing the target applications near the memory. Towards this end, we aim to explore optimizations like exploiting data sparsity and redundancy of computations to amortize the number of computations.

5. **Examining the deployability:** We integrate the proposed hardware units near the main memory. Towards Examining the deployability of the NMP approach, we aim to integrate the hardware units both with the 2D, 3D memories and measure its impact on the system's performance, energy consumption, and area overhead.

6. **Investigating the applicability:** To investigate applicability of the NMP approach, we also design systems that accelerate other applications like database operations near the memory.

1.10 Thesis Contributions

In this thesis, we primarily target to accelerate the inference phase of CNNs. Improving the performance for the inference benefits several real-time applications. We implement both the hardware- and software-based techniques to improve the system's performance and energy efficiency while executing the CNN inference phase. A brief overview of all the contributions is explained in the following section.

1.10.1 CLU: A Near-Memory Accelerator Exploiting the Parallelism in Convolutional Neural Networks (Contribution 1)

The gap between the processing speed and the memory-access latency in multi-core systems affects the performance and energy efficiency of the CNN/DNN tasks. This work aims to alleviate this gap by providing a simple and yet efficient near-memory accelerator-based system that expedites the CNN inference. Towards this goal, we design an efficient data-parallel algorithm to accelerate CNN/DNN tasks. The data is partitioned across multiple memory channels (vaults) to assist in the execution of the parallel algorithm, and the intermediate results are appropriately consolidated. We design a hardware unit, namely the convolutional logic unit (CLU), which implements the parallel algorithm. To harness the benefits of near-memory processing (NMP), we integrate homogeneous CLUs on the logic layer of the 3D memory. The proposed system achieves a substantial performance and energy reduction compared to multi-core CPU- and GPU-based systems with a minimal area overhead.

We choose HMC as our memory base for the NMP implementation. In the logic layer (LoB) of HMC, each HMC's vault is equipped with a separate vault controller. One CLU unit is integrated with one vault controller, leading to 16 CLUs working in parallel. The feature maps of each hidden layer are partitioned horizontally, and each slice of data (chopped feature maps) is then distributed across all the vaults for the concurrent processing in the CLUs. We compare the proposed system with quad-core, 64-core, and GPU-based systems. On average, we get around 101.83x and 74.69x improvements over the quad-core and 64-core CPU-based systems, respectively. We acquire a speedup up to 13.87x over a standard GPU. We also obtain around 218.40x, 160.33x savings in energy over 64-core CPU, quad-core CPU based systems, respectively, and maximum up to 33.76x energy savings over the GPU based system. The gain in performance and energy savings for the proposed system is achieved by the efficient CLU design, parallel processing of inference task, and from the NMP approach. The details of this work are described in chapter 3.

1.10.2 nZESPA: A Near-3D-Memory Zero Skipping Parallel Accelerator for CNNs (Contribution 2)

The inference phase of CNNs is primarily used in real-time applications. Consequently, the need for performance improvement is exceptionally high. State-of-the-art has either exploited the parallelism of CNNs, or eliminated computations through sparsity, or used near-memory processing (NMP) to accelerate the CNNs. We introduce NMP-fully sparse architecture, which acquires all three capabilities. The proposed architecture is parallel and hence processes the independent CNN tasks concurrently. To exploit the sparsity [57], the proposed system employs a dataflow, namely, Near-3D-Memory Zero Skipping Parallel dataflow or *nZESPA* dataflow. This dataflow maintains the compressed-sparse encoding of data that skips all ineffectual zero-valued computations of CNNs. We design a custom accelerator that employs the *nZESPA* dataflow. The grids of *nZESPA* modules are integrated into the logic layer of the hybrid memory cube. This integration saves a significant amount of off-chip communications while implementing the concept of NMP.

In the proposed system, a grid of *nZESPA* units is integrated with each vault by replacing the crossbar network with the 2D mesh network-on-chip (NoC) in the LoB. 16 *nZESPA* modules are connected through NoC and form a grid, leading to 256 *nZESPA* modules working in parallel to execute the inference tasks. The *nZESPA* hardware uses compressed data format with the help of a checker module and *nZESPA* controller. The compressed data format reduces the data to the number of non-zero elements both for weights and activations. We compare the proposed architecture with three other architectures which either

do not exploit sparsity (NMP-dense) or do not employ NMP (traditional-fully sparse), or do not include both (traditional-dense). The proposed system outperforms the baselines in terms of performance and energy consumption while executing CNN inference. Compared to the NMP based architectures, all traditional architectures suffer from longer data access latency and higher per bit energy cost due to the high off-chip communication cost. We have achieved the maximum speedup over traditional-dense architecture. The reasons are: 1) traditional-dense architecture works on dense data, and 2) it does not own the NMP capability. The gain over the traditional-fully-sparse architecture is comparatively less as it owns the similar property of exploiting sparsity for both the weights and activations. The gain over this architecture stems out only from the additional NMP capability of the proposed system. The proposed system performs better than NMP-dense because it is dense, and no computations are eliminated in NMP-dense architecture. On average, the proposed system obtains a maximum of up to 20.30x, 3.16x, and 6.09x improvement in the performance over the traditional-dense, traditional-fully-sparse, and NMP-dense architecture, respectively. Additionally, the proposed system achieves a maximum of up to 19.9x, 5.69x, and 3.44x energy efficiency on an average, compared to the traditional-dense, traditional-fully-sparse, and NMP-dense architecture, respectively. Chapter 4 covers this work in more detail.

1.10.3 ALAMNI: Adaptive LookAside Memory based Near-memory Inference engine for eliminating multiplications in real-time (Contribution 3)

The primary aim of this thesis is to improve the performance and energy efficiency of our proposed system while accelerating CNNs. Towards this goal, we incorporated another optimization of skipping the redundant computations in our designed hardware and integrated them close to the memory. Our proposed hardware, namely Adaptive LookAside Memory based Near-memory Inference engine (ALAMNI), reduces costly multiplications of CNNs with the help of lookaside memory (LAM). The ALAMNI controller keeps the most frequent triplets of weight (W), activation (A), and multiplication result (M), $\langle W, A, M \rangle$, in the LAM, which is used to eliminate matching computations. The LAMs are updated at runtime. It is effective on unseen data as it does not require any data pre-profiling overheads. As an additional optimization, we introduce a bitmasking concept to increase the hit rate of LAMs and further amortize computations. This bitmasking can be reconfigured to achieve the desired classification accuracy.

The designed ALAMNI unit is integrated with the vault controller of the HMC device,

similar to the previous contributions. In ALAMNI, we modify the MAC units into LAM-MAC$_i$ units by integrating two 64-entry content addressable memories (CAM) and a result buffer (64-entry). The first CAM caches the activations (A), and the second CAM caches the weights (W) from the most frequent $\langle W, A \rangle$ pairs. The associated result buffer stores the multiplication result (M) for the given $\langle W, A \rangle$ pair that hits in the CAMs. To skip the multiplications, the ALAMNI controller uses this LAM-MAC$_i$ units by taking precomputed results directly from the result buffer for the matching $\langle W, A \rangle$ pairs in LAM. To measure the efficacy of the proposed system, we evaluate three NMP-based architectures.

1. **NLN:** The system with near-memory hardware acceleration without using LAM search is termed as No_LAM_NMP or NLN. This architecture does not skip any multiplication operations.

2. **ALAMNI:** Here, the system with ALAMNI units skips the multiplications using LAM hits without using any bit-masking approximation.

3. **ALAMNI-Opt:** Here, the system with ALAMNI units skips multiplications using LAM hits with the approximation of using 5-bit masking.

The NLN architecture performs the worst among all three architectures. The reason is: NLN has neither got the benefits of LAM search nor the bit-masking. ALAMNI performs better than the traditional NLN, but not the best as it only leveraged the advantage of the temporal locality from the unmasked data through LAM search. Consequently, a lesser amount of multiplications are skipped here compared to the ALAMNI-Opt. The ALAMNI-Opt stands out to be the best in terms of performance because of the increased hits in LAMs due to bit-masking. Hence, more multiplications are skipped in ALAMNI-Opt, compared to the ALAMNI. ALAMNI achieves around 43.48% performance gain and 42.65% energy savings compared to NLN. On a 5 bit-masking, the performance gain and energy savings can rise up to 54.02% and 53.35%, respectively, over the NLN. Chapter 5 presents the full description of this work.

1.10.4 Contribution 4: Exploring Other Avenues for NMP Processing

The fourth contribution focuses on two objectives, thereby manifesting the NMP approach's efficacy in a broader domain. They are as follows.

1. Till this point, we explored near-memory acceleration of CNNs only in the 3D-memory. However, to establish the efficacy of the NMP approach, it is also essential to test the

concept for other popular memory technologies. Towards this end, we implement NMP-based systems for popular memory technologies like hybrid main memory and DRAMs. In this context, we accelerate the CNN inference close to the DRAM-PCM-based hybrid memory and close to the DRAM.

2. Apart from the CNNs, we also explore near-memory acceleration for another parallel and data-intensive application such as database operations.

This broader domain of experiments in terms of memory technology and application provides deeper insights about the system's performance and energy efficiency of NMP-based systems.

1.10.4.1 Hydra: A Near Hybrid Memory Accelerator for CNN Inference

It is essential to integrate additional processing elements close to other emerging main memory subsystems to examine the efficacy of the near-memory approach. Non-volatile memory, such as phase-change memory (PCM), has emerged as a promising DRAM alternative. It is also used in combination with DRAM, forming a hybrid memory. Though near-memory processing (NMP) has been used to accelerate the CNN inference, the feasibility/efficacy of NMP remained unexplored for a hybrid main memory system. Additionally, PCMs are also known to have low write endurance, and therefore, the tremendous amount of writes generated by the accelerators can drastically hamper the longevity of the PCM memory. In this work, we propose Hydra, a near hybrid memory accelerator integrated close to the DRAM to execute inference. The PCM banks store the models that are only read by the memory controller during the inference. For entire forward propagation (inference), the intermediate writes from Hydra are entirely performed to the DRAM, eliminating PCM-writes to enhance PCM lifetime. Unlike the other in-DRAM processing-based works, Hydra does not eliminate any multiplication operations by using binary or ternary neural networks, making it more suitable for the requirement of high accuracy. We also exploit inter- and intra-chip (DRAM chip) parallelism to improve the system's performance. On average, Hydra achieves around 20x performance improvements over the in-DRAM processing-based state-of-the-art works while accelerating the CNN inference. The details of this work are explained in chapter 6.

1.10.4.2 Exploring the Design Space for Near-DRAM MAC-based Inference Engine

In recent works, near-memory processing (NMP) has emerged as a reliable solution in accelerating CNNs. However, prior works either have used 3D memories to integrate the CNN accelerators or modified the cell arrays in 2D memories to introduce some computing capability to accelerate the CNNs. In this work, we propose to integrate custom hardware

with 2D memory, specifically the widely used DRAMs, for accelerating the CNN inference using the NMP concept. Integrating logic near the DRAM is challenging because of the strict area and power constraints. Consequently, exploration of the design space based on the performance, power consumption, and area overhead can be beneficial for the practical implementation of the near-memory MAC-based inference accelerators. Towards this aim, we design near DRAM inference accelerator (DiA) and integrate multiple such units inside the DRAM's chip. We explore both the chip-level and bank-level integration of DiA units to see the effects on the system's performance, power consumption, and area overhead. We also design two optimized versions of DiA, namely DiA-light1 and DiA-light2, and study their effect on the design space for near-DRAM logic integration. We exploit the intra-chip/bank and inter-chip/bank parallelism through our data partitioning scheme to execute the inference tasks concurrently. Among the proposed architectures, the most aggressive DiA at bank-level integration delivers 41x - 43x better performance, while the lightest DiA-light2 at chip-level integration achieves around 17x speedup compared to the state-of-the-art works. The proposed architectures incur area overheads of around 1.24 - 5.56% based on the hardware and its place of integration. The details of this work are included in chapter 6.

1.10.4.3 Towards Near-Memory Processing of Compare Operations in 3D-Stacked Memory

We explored CNN acceleration using the NMP approach. Apart from CNNs, we also examine the system's throughput for other data-intensive and parallel applications like database operations. Compare or scanning is the core operations of many applications, typically in a database. Such operations can leverage the benefits of NMP as they involve a significant amount of off-chip communications. We propose *near-data compare unit (NDCU)*, a less-invasive hardware that can be integrated with the existing ecosystem of the hybrid memory cube (HMC). While integrating NDCU, we have designed two full-system architectures, one is lighter NMP with no parallelism (NNP), and the second is NMP with vault level parallelism (NVLP). While the first architecture is more power and area efficient, the second one is fast with negligible overheads. With the motive of carrying out scan operation near the memory, we have specifically implemented NDCU to perform *'compare-n-hit'*, *'compare-n-count'* and *'compare-n-max'* operations on both row-store and column-store databases. We observe significant improvements over a conventional CPU-based system. We get around 2.3x and 37x performance improvement in NNP and NVLP architectures, respectively. In both designs, we reduce the energy consumption by around 8x on average. Chapter 6 presents the full information of this work.

1.11 Thesis Organization

The thesis comprises seven chapters which are as follows.

- Chapter 1 provides an introduction, motivation, objectives, and brief ideas of contributions of the research work.
- Chapter 2 summarizes the background and prior works related to the contribution of the thesis.
- Chapter 3 presents the first contribution. Here, we accelerate the CNN inference in the logic layer of the HMC device (NMP approach) by exploiting the inherent parallelism through our proposed dataflow **(Objective 1, 2, and 3)**.
- Chapter 4 illustrates the second contribution where we exploit the sparsity property as an optimization to the CNN inference. We design the hardware that can take the benefit of the parallelism, sparsity, and the NMP approach **(Objective 1, 2, 3, and 4)**.
- Chapter 5 contains the details of the third contribution. While exploiting parallelism, we aim to eliminate the costly multiplication operations as another optimization to the CNN accelerator close to the memory **(Objective 1, 2, 3, and 4)**.
- In Chapter 6, we include the details of our fourth contribution, where we perform experiments on various memory technologies and applications for the extensive investigation of the NMP's efficacy **(Objective 1, 2, 3, 5, and 6)**.
- Chapter 7 finally concludes the thesis.

2
Background

Artificial intelligence (AI) [58] is the science and engineering of creating intelligent machines that aim the goals as humans do. The relation between our target algorithm, Convolutional Neural Networks (ConvNets / CNNs), and the whole of AI is shown in Figure 2.1. Within the domain of AI, machine learning [59] is a large subfield that empowers computers with the ability to learn without being explicitly programmed. In other words, a single program can be created once, and it will be able to learn new intelligent activities outside the notion of programming. Inside machine learning, there is a sub-area that is often termed brain-inspired computing [60, 61]. In this field, the algorithms try to mimic the way how the human brain works. Within brain-inspired computing, spiking [62] is another sub-area where it incorporates time into the working model, in addition to neuronal and synaptic status. In neural networks (NNs), a neuron's computation involves the weighted sum of the input values. The NNs are the series of algorithms that are primarily used to recognize the relationship between the vast amount of data. Within neural networks, there is an area called deep learning [63] where the NNs include more than three layers, i.e., more than one hidden layer. Deep neural networks (DNNs) are the neural networks primarily used in deep

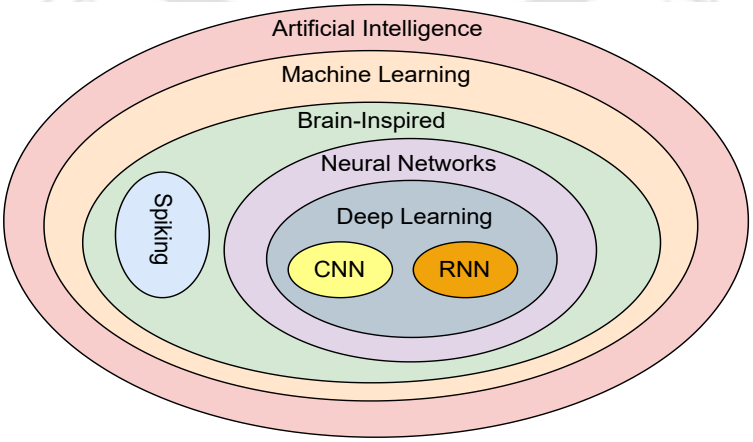


Figure 2.1: Taxonomy of Artificial Intelligence [6, 7].

learning, and these DNNs can have five to more than a thousand network layers. A recurrent neural network (RNN) is a class of artificial neural networks (ANNs) where connections between the nodes produce a directed or undirected graph along a temporal sequence. And the convolutional neural networks (CNNs) are a common form of DNNs, which includes multiple convolution (CONV) layers. The CNNs are widely used in classification and prediction works. In the following section, we discuss CNNs in more detail.

2.1 Convolutional Neural Networks

Convolutional Neural Networks (ConvNets / CNNs) are kind of similar to ordinary neural networks with minor differences. The ConvNets are primarily used in visual imagery or computer vision. They are also known as shift invariant or space invariant artificial neural networks. The CNNs include two phases: (1) training and (2) inference. In training, a model/network is built for future use, and the models are used in real-time for classification or prediction in the inference phase. The ConvNets include multiple layers of artificial neurons. The artificial neurons are an approximate imitation of their biological counterparts, and they are mathematical functions that compute the weighted sum of multiple inputs to produce output activations. Every neuron obtains a portion of inputs, performs a dot product with the weights, and is optionally followed by non-linearity. The entire network expresses a single differentiable score function where the raw image pixels remain on one end, and the class scores remain at the other. A loss function in the last (fully-connected) layer provides the prediction error of a network. In general, the ConvNet architecture makes an assumption of the input as image samples. The forward function (inference) becomes more efficient as the number of parameters reduces in the network.

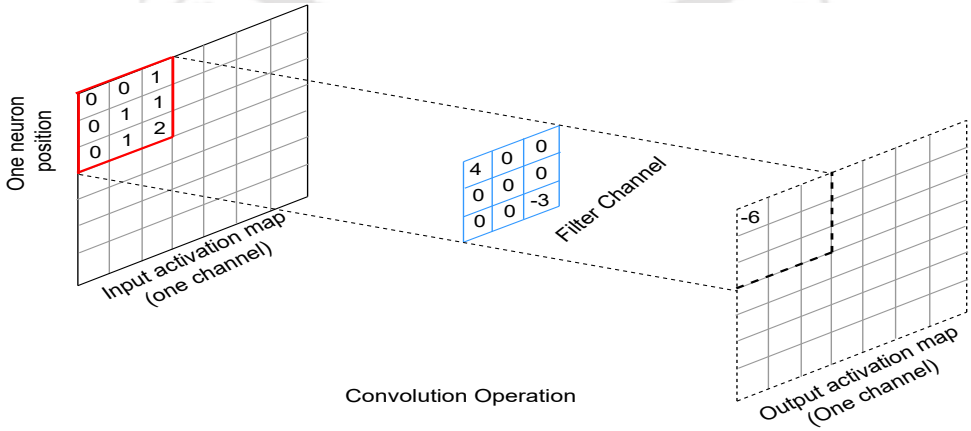


Figure 2.2: Convolution Operation for One Neuron Position.

2.1.1 Architectural Overview

Unlike a regular neural network, the layers of a ConvNet consist of neurons arranged in three dimensions: width, height, and depth. For example, the input images in the CIFAR-10 dataset have the dimensions of 32×32×3 (width, height, and depth, respectively). The depth is often referred to as a channel. A simple ConvNet is a sequence of layers where every layer transforms one volume of activations to another through a differentiable function. The primary layers/operations of the forward phase (shown in Figure 1.4) are as follows.

1. **Convolution layer (CONV):** Figure 2.2 shows the example of convolution operation of one neuron position. The CONV layers compute the output neurons that are connected to the local regions in the sample image (input activation map). The output neurons are obtained by performing dot product operations between the weights (filter channel) and a small region of the input sample. In addition, the bias values can also be added to each neuron position during the convolution operations.

   Formally, each kernel/filter can be represented by a 3D array of size $N_z \times F_x \times F_y$. The input feature map is a set of N_z channels, each having a dimension of $D_x \times D_y$. F_n denotes the number of kernels used in one activation layer, which is the number of output channels for the next hidden layer. The (x, y) element of the l^{th} output channel is defined as:

$$D_l^{out}(x,y) = \delta(b + \sum_{m=0}^{N_z-1} \sum_{n=0}^{F_x-1} \sum_{p=0}^{F_y-1} D_m^{in}(x+n, y+p) * F_{(l,x,y)}(m,n,p))$$

   Here, $F_{(l,x,y)}(m,n,p))$ is the weight at position (m,n,p) of the l^{th} kernel and $D_m^{in}(x,y)$

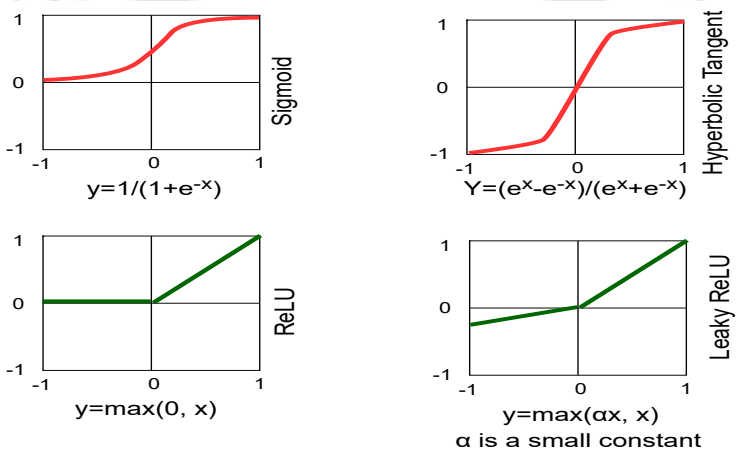


Figure 2.3: Various Nonlinear Function.

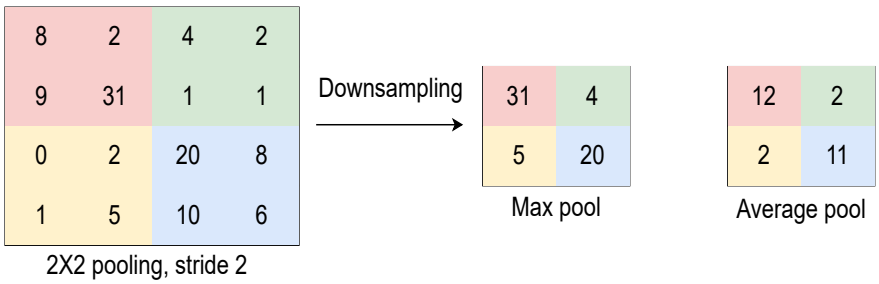


Figure 2.4: Various pooling operations.

represents the neuron at position (x, y) of input channel 'm'. The next position of $D_l^{out}(x, y)$ depends on the stride (S). In all activation layers, the bias (b) is added once, and also, the non-linear function $\delta()$ is evaluated. This background of CNN gives us the insight that the dot-products of matrices can be computed in parallel, closer to the memory.

2. **Nonlinearity:** Each CONV layer can have a nonlinear function. As shown in Figure 2.3, there are various nonlinear functions that can be used to introduce nonlinearity into the DNN. These include nonlinear functions like the sigmoid, hyperbolic tangent, rectified linear unit (ReLU) [64], which has become popular in recent years due to its simplicity and ability to enable fast training. Leaky Rectified Linear Unit (Leaky ReLU) [65] is a special type of activation function based on ReLU. However, it has a small slope for negative values instead of a flat slope like ReLU. The slope coefficient is determined before the training phase as it is not learned during training.

3. **Pooling layer:** Pooling layers, also known as downsampling, are responsible for reducing the spatial size of the convolved feature. Similar to CONV layers, the pooling operation moves a filter across the entire input channel, but the difference is that the filters do not involve any weights in it. The kernel/filter applies an aggregation function to the values within the receptive field to populate the output array, leading to the extraction of dominant features for efficient training. As shown in Figure 2.4, there are two types of pooling: (1) Max Pool and (2) Average Pool. In Max Pool, as the filter moves across the input channel, the pixel with maximum value is selected and sent to the output array. In Average Pool, when the filter moves across the channel, it computes the average value within the receptive field and sends the result to the output array. The same can be verified from Figure 2.4.

4. **Fully-connected (FC) layer:** After the feature extraction in the CONV layers, we classify the data into various classes. This can be done using a fully connected (FC)

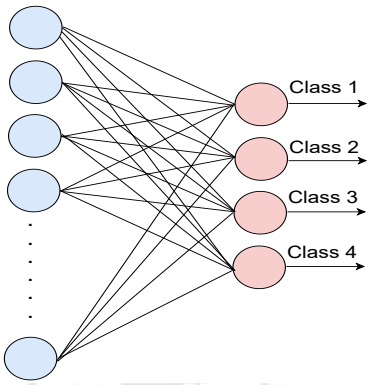


Figure 2.5: Fully Connected Layer (FC).

layer. The input to the fully connected layer is the output from the final pooling or CONV layer, which is flattened and then fed into the FC layer. As shown in Figure 2.5, the neurons in a fully connected layer have full connections (weights) to all activations in the previous/subsequent layer, as seen in regular Neural Networks. Their activations can hence be computed with a matrix multiplication followed by a bias offset. After the input is passed through the FC layer, the final layer applies the softmax activation function that produces the probabilities of input being in a particular class (classification). Hence, we finally obtain the probabilities of the objects in the image belonging to the different classes. In this way, CNNs classify an input image with its particular label.

All the layers explained above are part of the forward or inference phase where we use a trained model. However, training includes an additional phase that is called the backward phase. In the backward phase, the gradients are backpropagated, and weights are updated over multiple epochs till the model is trained. The details of this phase are not explained here, as training is out of the scope of this thesis.

2.2 Memory Technologies used for Near-Memory Processing

While approaching towards the unconventional NMP-based system instead of the traditional systems, it primarily leaves two design options:

- NMP with emerging 3D-stacked memories, like hybrid memory cube (HMC) [66], high bandwidth memory (HBM) [41].

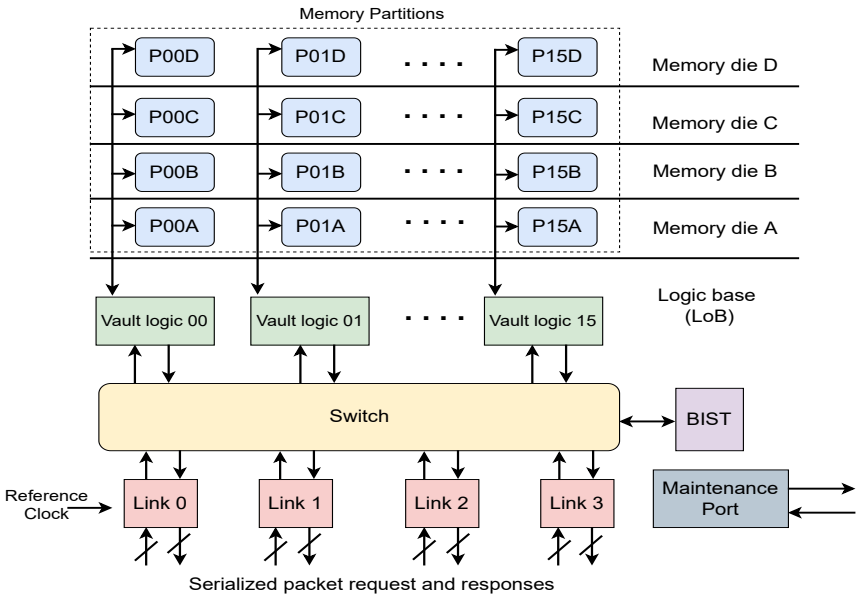


Figure 2.6: The architecture of hybrid memory cube (HMC).

- NMP with 2D memories, like conventional DRAM, non-volatile memory, and hybrid memory.

We discuss some of the memory technologies that are used in our experiments to implement the NMP-based system in the following section.

2.2.1 Hybrid Memory Cube (HMC)

The HMC includes a single package containing multiple memory die and one logic die, stacked on top of each other using through-silicon via (TSV) technology. As mentioned in Section 1.5.2.1, the memory is organized in the form of vaults within HMC. Each of these vaults is operationally and functionally independent of each other. As shown in Figure 2.6, in the logic layer (LoB), each vault is controlled by a separate controller, namely the vault controller. Refresh operations are also controlled by these vault controllers, leading to a reduction of this overhead from the host memory controller. The vault controller also determines its own timing requirements. Each vault controller is also equipped with a queue that is used to buffer references for the corresponding vault's memory. The references from the queue can be served based on the need rather than the order of arrival. Consequently, the responses from vault operations back to the external serial I/O links (shown in Figure 2.6) can be out of order. However, requests from a single external serial link to the same vault/bank address are executed in order. Requests from different external serial links to the same vault/bank address are not guaranteed to be executed in a specific order and must be managed by

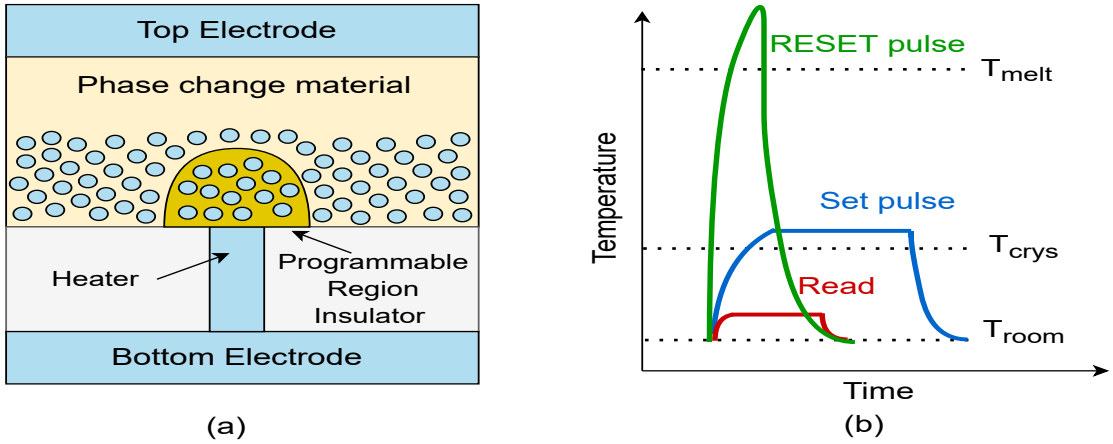


Figure 2.7: (a) The cross-section schematic of the PCM cell. (b) The PCM cells are programmed and read through electrical pulses, leading to change temperature.

the host controller. The HMC I/O is implemented as multiple serialized, full-duplex links. The I/O links access the collective and internally available bandwidth from all the vaults. This can be achieved by implementing a crossbar switch. The external I/O links include multiple serial links with the duplex operation. We use HMC's 16 vault configuration for our experiments though 32 vault configuration is also available. Our experiments can also be performed on HMC's 32 vault configuration with additional area/power overhead (if more hardware units are integrated with the other vaults).

2.2.2 Phase Change Memory (PCM)

A cross-section of PCM cell and its operations are shown in Figure 2.7. The PCM memory uses a large resistivity contrast between the phase change material's amorphous (high resistivity) and crystalline (low resistivity) phases. In PCM, the reset and set state refer to high and low resistance states, respectively. As the processing temperature of the metal interconnect layer is sufficient to crystallize the phase change material, it remains in the crystalline. The programming region is first melted and then quenched rapidly by applying a large electrical current pulse for a short period of time to reset the PCM cell into the amorphous phase. Consequently, it leaves a region of amorphous or highly resistive material in PCM cells. This amorphous region is in series with any crystalline region of the PCM. It effectively determines the resistance of the PCM cell between the bottom electrode contact (BEC) and the top electrode contact (TEC). A medium electrical pulse is also put in to anneal the programming region to set the cell into crystalline phase with a certain temperature between melting and crystallization temperature for the duration to crystallize. The cell resistance is estimated by passing a minimum electrical current that does not affect the

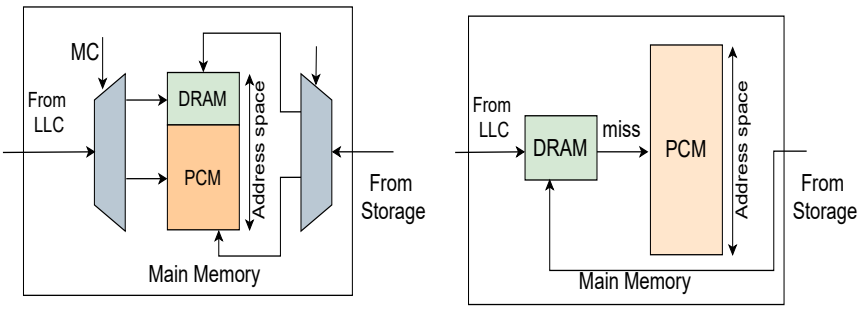


Figure 2.8: (a) Parallel organization for hybrid memory. (b) Hierarchical organization for hybrid memory.

current state. We have shown the diagram of the pulse shapes in Figure 2.7(b).

2.2.3 Hybrid Main Memory

High memory capacity has become one of the major requirements for emerging applications. The increasing memory requirement can not be easily fulfilled due to the limitations in DRAM scaling [67]. Additionally, DRAM faces several challenges like frequent refresh operations for high-leakage cells, longer write recovery time, and difficulty in building high-density arrays. The overview of the DRAM modules is explained in Section 1.5.1.1. As the main memory alternatives, non-volatile memories (NVM) like phase-change memory (PCM) and resistive RAM (ReRAM) have the potential to satisfy the high memory requirements of emerging applications. The NVMs provide several capabilities such as high memory density, low cost per bit, and near-zero standby power consumption in exchange for low performance and limited endurance [68, 69]. Despite various advantages of NVMs, DRAMs can not be entirely replaced, and it is more beneficial to use the NVMs in conjunction with DRAM. This leads to the use of hybrid memory. We choose PCM from the NVM options for our experiments as it is the most mature technology for the main memory [70].

From the design point of view, the DRAM-PCM-based hybrid memory can be divided into two categories, as shown in Figure 2.8. The first category is DRAM-PCM parallel organization, where DRAM and PCM can have a size of a similar order. In the second category, DRAM and PCM memory are hierarchically organized, and the DRAM is used as a small cache in this architecture. Both the architectures work on the principle of placing the write-intensive pages in DRAM and read-intensive pages in PCM. The hybrid memory controller often takes the responsibility of page migration between the two memories. The address space is split into two non-overlapping regions in the case of parallel organization of DRAM-PCM hybrid memory, as shown in Figure 2.8(a). In the hierarchical organization (shown in Figure 2.8(b)), DRAM is utilized as a cache to the PCM memory. All memory

references are sent to DRAM, and PCM is accessed only when there is a miss in the DRAM. Usually, the size of DRAM is substantially small and can be in a similar order as the size of the last level cache (LLC). However, DRAM can still encounter hits if LLC has the misses since LLC's contents are not forced to be inclusive in DRAM. In this thesis, we have explored the NMP's efficacy with the parallel organization of DRAM-PCM hybrid memory.

2.3 Hardware Accelerators

Hardware accelerators are used to perform specific tasks more efficiently compared to an application running on a general-purpose processing unit. In computing systems ranging from general-purpose processing units to fully customized hardware, there is a trade-off between flexibility (ease of executing various applications) and efficiency (system's performance). While the general-purpose processors execute instructions, the accelerators, connected on the bus, are controlled by the registers. The customized hardware modules can deliver orders of magnitude higher efficiency than the applications running on general-purpose processing units. Accelerators provide a better cost per performance as the CPU cost is generally a non-linear function of performance. Accelerators also deliver better real-time performance by putting time-critical functions on less-loaded processing elements. Additionally, accelerators are suitable for several other tasks like I/O processing in real-time, data streaming (audio, video, and network traffic), and solving complex operations (fast Fourier transform, exponential, and Logarithmic function). There are primarily three types of hardware accelerators: (1) specialized processor, (2) field-programmable gate array (FPGA), and (3) application-specific integrated circuits (ASICS). A popular example of a specialized processor is the GPU which contains several processing cores. The FPGA is a processor that can be configured after it is manufactured, leading to the feature of reconfigurability to implement various logic functions. The ASICS are dedicatedly built for a specific task, and they are extremely efficient in terms of the system's performance and power consumption. There is a bundle of works that have used the accelerators to harness substantially high system throughput. They are explained in the following section.

2.3.1 GPU-based Acceleration

To obtain high performance, the GPUs are used in several domains of work [31, 71, 72] like graph processing, database systems, deep learning, and so on.

In the era of big data, real-world data are often represented as graphs. Consequently, several application domains can be modeled as graph processing. Data processing has turned

into processing the number of vertices/edges in the order of billions or even hundreds of billions. The highly parallel architecture and high memory access bandwidth of GPUs show enormous potential to address the graph processing problems. In the previous work [73], G2, an extension of GraphIT [74] compiler framework, has been proposed. The GraphIT programming language is used to write graph algorithms. However, it has no support for generating high-performance code for GPUs. The G2 expands the optimization space of GPU graph processing frameworks with a novel GPU scheduling language and compiler. It includes capabilities like combining load balancing, edge traversal direction, active vertex set creation, the active vertex set processing ordering, and kernel fusion optimizations. NN-Descent [75], a k-NN graph construction approach, has been made faster by redesigning it to be adopted for the execution on the GPU. This work also proposes a graph update strategy, namely selective update. While executing NN-Descent, the experimental results show a substantial speedup compared to the existing GPU-based approaches.

Database operations are also accelerated using GPUs. The authors [76] proposed a generic technique for nested query processing on GPU without unnesting queries. The execution of nested queries is complex and time-consuming. For faster execution of nested queries, a new code generation framework that best fits GPU is also proposed in this work. The computational complexity of the nested method is reduced by optimized parallel processing on GPU. Database select queries have been accelerated using the GPU in the literature [77]. Here, the authors have implemented a subset of the SQLite command processor directly on the GPU.

Deep learning has become a widely used tool because of the emergence of GPU-based systems [78]. In a popular work [79], the authors have proposed to execute AlexNet on two different GPUs: a server GPU (Tesla K40 with Kepler architecture) and a desktop GPU (GTX1080 with Pascal architecture). They perform experiments with varying batch sizes of 16, 64, and 128. They find that GTX1080 provides higher performance compared to K40 GPU because of its higher clock frequency and the larger number of processing cores. In LookNN [80], CNNs have been accelerated in the GPU-based platforms. The multiplication operations are replaced by look-up table-based search operations. Each processing element of our GPU can access a small associative memory, enabling it to bypass redundant computations. The LookNN architecture can achieve up to 3x energy improvement and 2.6x speedup compared to a traditional GPU. In ALook [81], neural network acceleration has been implemented in the GPU platform. They propose an adaptive look-up-based approach that implements a dynamic update policy to maintain a set of recently used operations in associative memory. This approach decreases the energy consumption of GPGPU applications. An analytical model related to performance and memory traffic in GPU has been

proposed in DeLTA [82] while accelerating the CNNs. They have also manifested that their model can be utilized to balance the scaling of various GPU resources to improve CNN's performance.

2.3.2 FPGA-based Acceleration

FPGA-based acceleration is highly popular for its efficiency in performance and energy consumption. In various domains [83–86], there are several works that have used FPGA-based acceleration. A few of them are described in the following section.

In the previous work [87], an FPGA-based storage engine is designed for the database with a focus on data filtering operation. The FPGA's parallelism is exploited to implement a hardware-based data filter that substantially improves the performance of filtering operations. The design can also support various queries without partial reconfiguration. The obtained results have achieved 2.8x and 1.95x improvement compared to the software baseline and conventional storage engine, respectively. GraVF [88] proposes a design framework for distributed graph processing on FPGAs. The designed system is evaluated with four widely used graph algorithms like breadth-first search, single-source shortest path, PageRank, and connected components. This FPGA-based design can deliver performance similar to custom designs with fewer requirements of user inputs. In BlueDBM [89], a system architecture for Big Data analytics has been proposed with the aim of delivering better cost-performance trade-off. The designed system includes a homogeneous cluster of host servers integrated with one BlueDBM storage. Each BlueDBM storage is connected with the host server through the PCIe link. The BlueDBM contains flash storage, an in-store processing element, network interfaces, and a DRAM. An FPGA is used to implement the in-store processor for each BlueDBM node. The BlueDBM outperforms a flash-based system by a factor of 10. In ForeGraph [90], a multi-FPGA-based large-scale graph processing framework is proposed. Instead of the entire graph, a portion of the graph is kept in the off-chip memory of one FPGA board. Data has been partitioned for parallel processing, and the communication over the partition is also reduced. On average, the throughput is around 2.03x higher than state-of-the-art.

FPGA-based acceleration has also been largely used to accelerate neural networks (NNs) [83] because of its high performance, reconfigurability, and fast development round. In SparkNoC [91], the authors have developed a lightweight neural network, SparkNet, that includes fewer numbers of parameters, hence leading to a reduced number of computations. The designed SparkNet is also suitable for deployment on FPGAs. They have deployed the SparkNet on Intel Arria 10 FPGA platform and achieved a throughput of 337.2 GOP/s

and an energy efficiency of 44.48 GOP/s/w. An FPGA-based architecture that accelerates the CNNs is proposed in the previous work [92] for the extraction of facial features. They exploit the parallelism and pipeline structure to obtain optimized resource utilization. For a fully connected layer of the forward pass, they have used a batch-based technique to reduce the number of data accesses. In another work [93], efficient utilization of logical resource and memory bandwidth has been taken care of while accelerating CNNs on the FPGA platform. Here, the proposed work first estimates the throughput and required bandwidth using techniques like loop tiling and transformation. The authors then use the roofline model to identify the solution with the best possible performance and lowest utilization of FPGA resources. This work has obtained a peak performance of around 61.62 GFLOPs which outperforms their previous approaches. The FPGA-based accelerator is also proposed for large-scale CNNs [94] . In this work, a technique is built to find an optimized parallelism strategy for each CNN layer, leading to high throughput and resource utilization. The size of on-chip buffers is also reduced substantially with the help of two different computing patterns on the fully-connected layer. This work has achieved a peak throughput of around 565 GOP/s with a clock frequency of 156 MHz. In the literature [95], another high-performance FPGA-based CNN accelerator is proposed. This work has adopted block-floating-point (BFP) arithmetic to increase the efficiency of the accelerator during inference. Instead of single-precision, they have used the mixed-precision: 16bit format for activation and 8bit format for model parameters, leading to savings in memory and bandwidth. They have also reduced the retraining overhead in exchange for a loss of accuracy of up to 0.12%. This work has achieved a performance of 760.83 GOP/s and an energy efficiency of 82.88 GOP/s/W.

2.3.3 ASIC-based Acceleration

While executing various algorithms, the application-specific integrated circuits (ASICs) are highly efficient in providing performance and energy efficiency in exchange for their limited flexibility. There are several domains [96–98] where the ASICs have played a crucial role in solving problems. A few of them are as follows.

In Beyond the wall [38], an accelerator, namely JAFAR, has been integrated close to the DRAM. JAFAR can perform the select operation on the column store database and send only qualifying data through the off-chip link, leading to a reduction of data movement in the memory hierarchy. Their experiments show that JAFAR can deliver up to 9x speedup compared to moving the data directly towards the CPU. In another work [2], buffered compare units (BCUs) are proposed and integrated inside the DRAM's chip to explore the high unexplored internal bandwidth. The BCUs are integrated with the individual banks to ac-

celerate 'compare-n-op' operations that are primitive to many big data applications. The BCUs include a small buffer, simple ALU, and command generator that generates commands to fill the buffer and feed data to the ALU. After processing the data close to the memory, it returns the result back to the host memory controller, leading to savings in costly off-chip communications. The experimental results show a substantial improvement in the performance and efficiency of the system for various workloads. In the previous work [99], logic-in-memory accelerators have been employed with a 3D-stacked DRAM to perform near-memory operations on SpGEMM, 2DFFT. Their system includes a fine-grained rank-level 3D die-stacked DRAM and an additional layer implementing logic-enhanced SRAM blocks that are dedicated to a particular application. They also perform design space exploration and exploit efficient architectures to accelerate computing while keeping a balance between performance and power. Their experiments manifest orders of magnitude of performance and power efficiency improvements compared to traditional multi-threaded software implementation on the CPU. NDA [100] proposes a near-DRAM accelerated (NDA) architecture that processes data using accelerators 3D-stacked on DRAM. NDA requires no changes in the host processor design and a minimal change in the commodity DRAM while keeping the compatibility with standard DRAM interface and DIMM architecture. They have explored three different microarchitectures that have different impacts on DRAM area, timing, and energy. NDA targets to accelerate various big data applications with high parallelism and localized memory accesses on Map-Reduce frameworks [101]. The experiments show that NDA incurs substantially lower energy and delivers higher performance compared to a system that integrates a similar accelerator on-chip. In GraphPulse [102], the authors propose a hardware accelerator that is used for asynchronous graph processing with event-driven scheduling. They optimize the model by coalescing events to control event population. They have also achieved efficient memory access patterns that enable implementation in reconfigurable hardware or ASIC. They have enhanced their model with prefetcher and steaming scheduler to obtain high throughput. Their design outperforms software frameworks because of the efficient memory usage and bandwidth utilization.

The ASICs play a highly crucial role in accelerating neural networks (NNs). Below we explain a few of the works related to NN acceleration in a separate section.

2.4 Neural Network (NN) Accelerators

There have been several efforts put into building accelerated architectures for neural networks. Some of them are on-chip accelerators, while there are many memory-based accelerators as well. We explain some of them in the following section.

2.4.1 On-chip NN Accelerators

In general, the on-chip accelerators are integrated close to the processor's chip. They have comparatively higher memory access costs in terms of access latency and access energy compared to the memory-based accelerators. In DaDianNao [45], the CNN/DNN algorithms are accelerated through a custom multi-chip machine-learning architecture. To reduce the off-chip communications from the custom hardware, they have used nearby SRAM buffers and eDRAM banks to fetch/store data. Experimental results show substantial speedup and energy savings over a GPU-based system. In Eyeriss [103], the CNNs are accelerated through a spatial architecture with 168 processing elements. They have also proposed a dataflow, namely row stationary (RS), which reduces the energy efficiency by maximally reusing data to reduce the data movement in terms of DRAM accesses. Cnvlutin [104] design an accelerated architecture that removes a substantial amount of ineffectual zero-valued computations from deep neural networks. They have explored the dynamic sparsity property. In other words, they have amortized the ineffectual computations related to the activations. On average, they obtain a performance improvement of around 1.52x without any loss in accuracy with a broader ineffectual identification policy. In Cambricon-X [105], neural networks have been accelerated through an accelerator that can exploit the sparsity and irregularity of the models for increased efficiency. SCNN [57] proposes accelerated architecture for CNNs. They have exploited both weight and activation sparsity to eliminate the zero-valued computations. They have achieved performance gain and energy savings by a factor of 2.7x and 2.3x, respectively, over a comparably provisioned dense CNN accelerator. In EIE [106], an energy-efficient inference engine has been proposed. The designed hardware works on the compressed network model and accelerates the resulting sparse matrix-vector multiplication with weight sharing. EIE also leverages the benefits of both weight and activation sparsity, and it primarily targets to accelerate the fully-connected layer.

In our second contribution, we have also exploited the sparsity property. While Cnvlutin and Cambricon-X either exploit static (weight) or dynamic (activation) sparsity, we have explored both sparsity in our proposed design, similar to SCNN and EIE. However, EIE targets only the fully-connected layer of the CNN algorithm. Further, SCNN uses a Cartesian product-based solution with an assumption of unit strides for convolution, when unit stride may not always be applicable for all the hidden layers of various networks. This Cartesian-product strategy restricts SCNN's applicability only to networks with unit-stride convolutions. In our work, we have implemented a dot-product-based solution that retains its applicability to any CNNs of diverse shapes and sizes.

2.4.2 Memory-based NN Accelerators

The memory-based accelerators are integrated closed to the memory, or the memory itself owns some amount of computing capability. The former approach is called near-memory processing (NMP), while the latter is known as processing-in-memory (PIM). There have been several works in each of these categories, which are explained in the following section.

2.4.2.1 NMP based NN Accelerators

Neurostream [15] proposes a NMP-based solution to accelerate CNNs. They have used the hybrid memory cube (HMC) to integrate logic, namely NeuroCluster. NeuroClusters include modular design based on NeuroStream coprocessors and general-purpose RISC-V cores. While executing the CNN workloads, they have achieved a throughput of 240 GFLOPS which can be scalable up to 955 GFLOP with a network of four memory cubes. In Neurocube [3], a neuromorphic architecture has been proposed to accelerate both CNN's training and inference. Towards this end, clusters of processing engines connected by a 2D mesh network are integrated into the logic layer of HMC. These clusters can also access the vaults of HMC in parallel to others. Another 3D-memory-based NN accelerator has been proposed in TETRIS [16]. Apart from the hardware architecture, the authors have also proposed scheduling and partitioning techniques for the efficient execution of the NN algorithm. They have proposed a hybrid partitioning scheme to parallelize the NN computations over multiple accelerators. TETRIS has achieved a performance gain of around 4.1x while saving energy by around 1.5x compared to NN accelerators with a DRAM-based main memory system. DeepTrain [107] is another deep neural network (DNN) accelerator that is primarily designed for high-performance and energy-efficient training. They propose spatially homogenous computing hardware with temporally heterogeneous dataflow for optimizing memory mapping and data reuse. The designed hardware modules are also integrated into the logic layer of HMC. On 15 nm FinFET technology, their architecture has achieved an efficiency of 500 GFLOPS/W for a wide range of DNNs like convolutional, recurrent, and mixed (CNN+RNN) networks. In [13], both the hardware and software for the NMP architecture are made for in-memory analytics frameworks, including deep neural networks. The proposed hardware also supports coherence, communication, and synchronization to support the execution of the analytic frameworks with complex data patterns.

2.4.2.2 PIM-based NN Acceleration

One of the state-of-the-art, PRIME [108], has used the computations capability of metal-oxide resistive random access memory (ReRAM) to build an NN accelerator. The crossbar

array of ReRAM can perform matrix-vector multiplication that is the elementary operation of the neural networks. PRIME configures a portion of the crossbar array for NN computation. They have also designed software/hardware interfaces for various NN workloads. Experimental results show a performance gain of around 2360x and energy savings of around 895x compared to their previous work. To implement an accelerated architecture, ISAAC [17] proposes to use memristor crossbar arrays both for computing the NN's dot-product operations and storing input weights. In their pipeline-based architecture, the computation works on the analog domain, which involves analog-to-digital (ADC) and digital-to-analog (DAC) conversion overhead. They have proposed data encoding techniques that reduce this overhead. They implement several supporting digital components for CNN accelerator and identify the best balance of memristor storage/compute, ADCs, and eDRAM storage on a chip. ISAAC achieved substantial improvement in throughput and energy consumption compared to DaDianNao [45]. In PipeLayer [109], they include the additional capability of training the networks into the ReRAM-based PIM accelerator, unlike ISAAC and PRIME. They exploit both intra- and inter-layer parallelism to enable highly pipelined execution of both training and testing. Their experiments show a speedup of 42.45x and energy savings of 7.17x compared to a GPU implementation.

Neural networks have also been accelerated closer to the DRAM using the PIM concept. In DrAcc [37], a DRAM-based CNN accelerator is proposed, and it leverages the benefits of the PIM approach. DrAcc obtains a high inference accuracy with a ternary weight network. They enhance the in-DRAM bit operation to execute the NN operations. DrAcc achieves a throughput of 84.8 FPS (frame per second) with an additional expense of 2W. DRISA [18] also proposes a DRAM-based accelerator that provides both powerful computing capability and large memory capacity/bandwidth. In DRISA, every memory bitline of DRAM is used to implement bitwise boolean logic operations. DRISA is reconfigurable and can compute various functions with the combination of boolean logic operations and hierarchical internal data movement. Experimental results show a substantial performance improvement and energy efficiency compared to other ASICs and GPUs.

2.5 Summary

This chapter summarizes the background of CNN inference, memory technologies, and hardware accelerators as this knowledge are the prerequisite for NMP integration of the custom hardware. We briefly discuss the memory technologies like HMC, PCM, and hybrid memory, which we use for our experiments. We also list the accelerator design space ranging from GPUs-FPGAs-ASICs to memory-based accelerators. We demonstrate various state-of-the-

art works to present a glimpse of each category of work. Among the memory-based accelerators, PIM involves the benefits of high memory bandwidth with additional challenges of fabrication and ADC/DAC conversion overhead. The NMP-based processing provides good system performance with the challenge of integrating additional logic that fits in the area/power budget of the corresponding memory technology. This thesis aims to provide such NMP-based optimized designs for emerging applications.

3

CLU: A near-memory accelerator exploiting the parallelism in Convolutional Neural Networks

In this chapter, we aim to accelerate the CNNs by exploiting the inherent parallelism of its inference phase. We design a three-phase algorithm and data partitioning scheme to exploit the parallelism efficiently. We design custom hardware, namely convolutional logic unit (CLU), and integrate multiple such instances in the logic layer of the HMC memory (NMP). This integration helps in leveraging intra- as well as inter-vault parallelism. This two-level of parallelism, along with NMP, substantially increases the system's throughput.

3.1 Introduction

As discussed earlier, CNN is a widely used machine learning tool for emerging applications. Examining the CNNs, it can be observed that it requires ten to a hundred megabytes of parameters on billions of operations in a single inference pass. Consequently, the computations and energy requirement of the inference phase can be enormous, especially with the deeper networks and larger data sets such as high-definition videos. The operations involved in inference often surge the data movement in on-chip as well as off-chip links to support the high degrees of computations. In 2009 [78], the inclusion of GPUs in deep learning was a pioneering contribution. Processing neural networks (NNs) on massively parallel GPUs greatly surpassed the traditional methods of executing NNs on multi-core CPUs. However, GPUs can only provide the required speed in computations. Memory bandwidth could not be increased in sync with the computational capability due to the limitations on pins, leading

to the *memory-wall* problem [110]. Consequently, the cost of data movement: comprising of memory access latency and energy consumption, especially on off-chip links, significantly affects the CNN inference for massive datasets despite the use of GPUs.

This constraint endorses the movement towards the resurgence of the old unavailing idea of near-memory processing (NMP). In NMP, some computations are offloaded to the processing units, placed near the memory while leaving complex or unbounded controls to the host processor. Integrating general-purpose cores near to the memory [13] has challenges in terms of power budget, thermal issues [111], management of cache coherence, and virtual memory.

Here, we choose dedicated hardware over the general-purpose cores as an NMP logic option. Many researchers have built PIM prototypes since 1990s [112–114], and have shown promising results. However, these could not be adopted at that time, as incorporating costlier logic and memory on the same die was difficult. Moreover, programmers needed to grapple with newer programming models. However, with the advent of 3D-stacked memory like Hybrid Memory Cube (HMC) [66], High Bandwidth Memory (HBM) [41], etc., there is scope for NMP on account of the tight integration of logic and memory with the help of Through-Silicon-Vias (TSVs). 3D-stacked memories have a logic layer (LoB) and high internal bandwidth that can be utilized by NMP approaches. One can utilize the LoB to place logic components near the memory. In the LoB, we placed specialized processing elements dedicatedly built for the convolution ($CONV$) operations. The CONV operations are the centroid of any CNN inference. Certainly, dedicated hardware has limited flexibility. However, here it turns out to be an eccentric opportunity to design highly specialized and thus extremely efficient hardware, which can inherently benefit many of the emerging applications. We choose HMC for implementing near-memory processing of $CONV$ operations. CNNs (ConvNets), having highly parallel operations, can easily leverage the benefits of the parallel architecture of the HMC device with its high internal bandwidth.

In this chapter, we present a simple and efficient system that harnesses the benefits of the NMP approach for CNNs. We design lightweight processing engines, namely convolutional logic units (CLUs), to do $CONV$ operations near the memory. Accelerating $CONV$ operations provides speedup in the inference phase. These CLUs are integrated with the individual vaults (memory channels) of the HMC. We partition the data and place them in the separate vaults of the HMC to perform convolutions simultaneously using the CLUs on each vault. In [115], we presented the basic design of the CLU, which had limited functionality in terms of executing bigger ConvNets. Here we extend our previous work to make the system more generalized. The salient contributions are as follows.

1. We design simple, dedicated hardware (CLUs) and integrate these units with the in-

dividual vaults in the LoB of the HMC device. This integration helps in leveraging the benefits of NMP. The CLUs are area efficient and can be easily integrated into the existing ecosystem of the HMC with minimal changes. The CLU can also handle any number of strides.

2. Data parallelism is exploited in the form of intra-vault and inter-vault parallelism. To achieve this parallelism, we propose an algorithm having three different phases. The designed hardware (CLU) is implemented accordingly to execute these three phases of the algorithm.

3. The proposed design of CLU delivers speedup in performance with substantial savings in energy consumption. Our proposed system outperforms the multi-core CPU and GPU based systems. Additionally, we also compare our approach with the recent state-of-the-art works. Compared to the previous version of CLU [115], we add new capabilities to the design like handling any number of strides, a way of accommodating larger features in the small SRAM buffer of the CLU unit, etc. The finite state machine (FSM) of the CLU controller has been redesigned to add all the new capabilities. These make the design scalable for all types of ConvNets. The throughput of the proposed system is benchmarked with five widely used state-of-the-art ConvNets: AlexNet [116], ZFNet [117], VGG-16 [118], VGG-19 [118] and ResNet-34 [48], pre-trained on ImageNet dataset [119].

3.2 System Architecture

3.2.1 Overview

CNNs primarily involve dot-products on matrices (describes in Chapter 2). These dot-products can be computed in parallel, closer to the memory. Towards achieving this, we propose a system as follows:

- To parallelize the operations, we divide the data matrix into 16 parts (as there are 16 vaults in our selected HMC). Each vault would then process its corresponding data using the multipliers and adders. However, in the case of a different number of vaults, the data can be partitioned accordingly.

- Another thing to be observed is that convolution is highly compute and data intensive. If all of these *CONV* layers are evaluated near the DRAM, this will reduce the traffic between the processor and the DRAM.

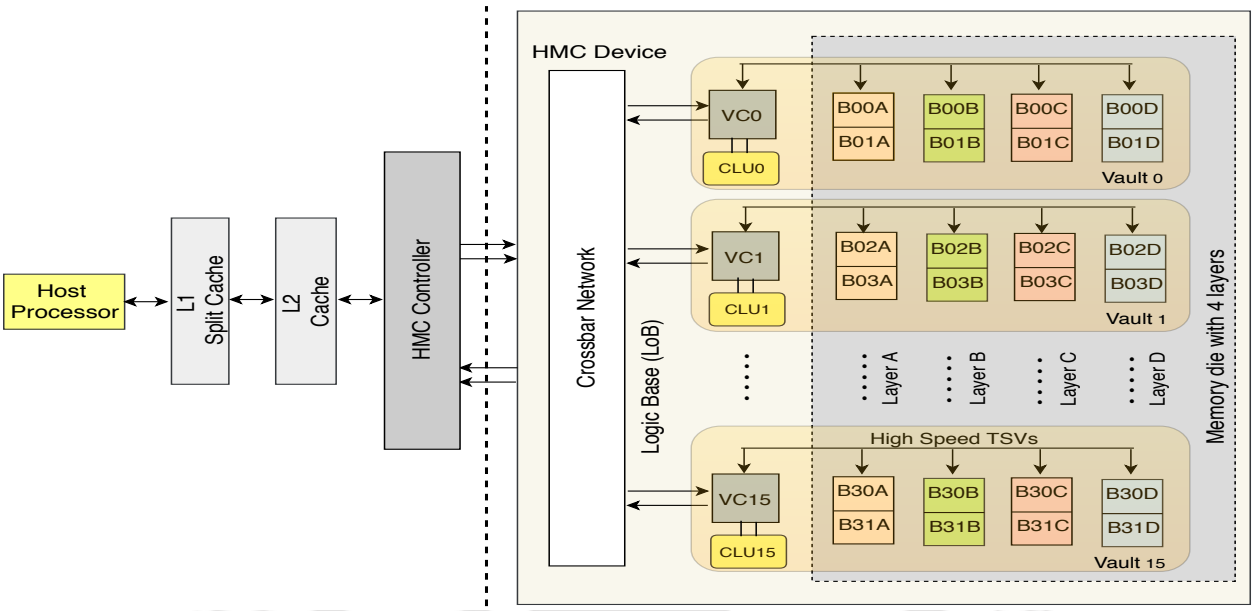


Figure 3.1: A conceptual view of proposed full system architecture for convolution operation.

Thus we arrive at the idea of designing a data-parallel algorithm and using HMC for near-memory processing of convolution. The architecture, shown in Figure 3.1, is the full system representation of the proposed model. Here, we choose HMC as our memory base for the NMP integration. The host processor communicates with the HMC through a packet-based abstract protocol. The system includes a two-level memory hierarchy with one L1 split cache, and one shared L2 cache. One on-chip HMC controller takes the role of handling all the global memory references. The specific HMC device consists of 4-Gbit layers (4 layers: layer A, layer B, layer C, and layer D, shown in Figure 3.1), which sum up to 2 GB of memory in total. One slice of all the four memory layers forms a vault, as shown in Figure 3.1. We use 16 vaults-configuration of the HMC. Here, each vault consists of 2 banks per memory layer. For example, the banks of vault0 are (B00A, B01A), ..., (B00D, B01D), as shown in Figure 3.1. In LoB, each vault is equipped with a separate vault controller (named as VC0, VC1,.., VC15), which deals with the memory references local to the individual vault. These vault controllers also manage the inter-vault communications (fetch/store operations) with another vault with the help of the crossbar network. The CLUs are integrated on top of these vault controllers, which are capable of working independently from the others. The design of the CLU is independent of the existing memory technology. As a result, it can be integrated with other memory technologies with minimal changes in the respective systems.

The architecture explained above leverages twofold benefits: (1) Minimum memory access latency due to NMP compared to the conventional CPU/GPU based systems having a relatively slower memory hierarchy. (2) The inherent parallelism of the CNN operations,

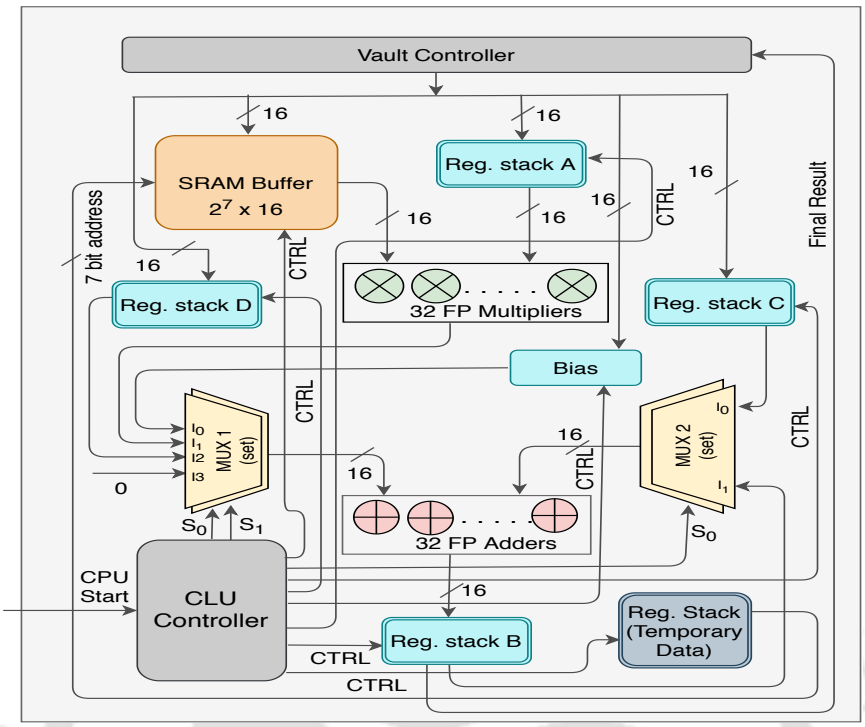


Figure 3.2: An abstract design of the CLU.

which is exploited by placing 16 hardware units (CLUs), one for each vault. To realize these benefits, we propose a parallel algorithm, design hardware (CLU) to implement it, and synthesize the hardware using the Cadence tool-set (explained in Section 3.4.4). Similar to [120], [121], [122], and [57], our proposed hardware (CLU) is primarily designed for the convolution layers, and the remaining layers like FC are executed by the host system. The convolution layer being the major consumer (around 90-95%) of the CNN computations [123], most of the operations are performed near to the memory, consequently delivering decent speedup (actual throughput of 1.4 TFLOPS), substantial savings in power (only 0.92 W), and high efficiency (1521 GFLOPS/W) with a minimum area overhead. The idea of accelerating the CONV layers makes our design simple yet efficient for all the CNNs including fully convolutional neural networks [124], [125] and CNNs with other classifiers [126], [127].

3.2.2 Convolutional Logic Unit (CLU)

The absolute data path with all the components and connections of CLU being complex, we provide an abstract design of it in Figure 3.2. The multiple connections between any two modules are also represented by a single connection for simplicity. Each CLU is primarily composed of SRAM buffer, operand register stacks (A, B, C, and D), bias, register-stack (for temporary data), one CLU controller, and combinational logic like fixed-point adders, fixed-

point multipliers, and multiplexers. The host processor offloads its computational loads by sending a start signal to the CLU controller. Additional information like dimensions/depths of the kernels and data matrices, initial addresses of the metadata (like data, kernels, strides, and bias) are also offloaded to the register-stack by the host processor. The CLU controller drives the entire modules and makes the results ready for high-level processing by the host processor. The CNN algorithm essentially includes a high degree of dot product operations. To execute these operations, we include 16-bit fixed-point multipliers and adders. The CLU controller brings all the data needed by the algorithm through the local vault controller. Initially, a single layer (or matrix) out of all the matrices along the depth of a kernel is loaded in the SRAM buffer. After the entire convolution operations using one layer (or matrix) of a kernel on the corresponding data layer (or matrix), the next layer (or matrix) along the depth of the kernel is loaded in the SRAM. As it only loads kernel depth-wise (one layer or matrix at a time), it is sufficient to have a buffer size of only 256 B (2^7 rows, 16-bit each) for most of the well-known ConvNets. Although this small buffer increases the DRAM-layer accesses to some extent, it limits the energy consumption of each CLU. The larger buffers also result in increased area overhead. For each layer (or matrix) of a kernel, data items from the corresponding layer are loaded in the Reg. stack A. After the *CONV* operations at each neuron position by the individual multiplier-adder unit, the final results are accumulated in Reg. stack B, and the output matrix is updated in the DRAM die by the CLU controller. The bias register holds the bias value, which is added only once at each neuron position. The other two register stacks (D and C) are used to hold the partial results (explained in the subsequent section), which are needed to complete the *CONV* operations.

3.2.3 Data Distribution and Parallel Processing

Two approaches [128], data parallelism [116] and model parallelism [129], are increasingly popular and important in the distributed processing of deep learning. The data parallelism partitions the batch of images while replicating the model. And the model parallelism deals with the aspects of concurrent processing of neural network models (weights). To exploit the parallelism efficiently, we implement a variant of data parallelism with a horizontal partition scheme for our proposed system. Initially, the preprocessing of data is done by replicating the model parameters and partitioning the feature maps horizontally in the individual stack (vault) of the HMC module. The finite state machine (FSM) of the CLU controller is designed for layer-wise streaming of those model parameters into the local SRAM cache of the individual CLU. The host system offloads the initial addresses for the data. All the multiply-accumulate (MAC) units of a CLU work simultaneously on the horizontal strip

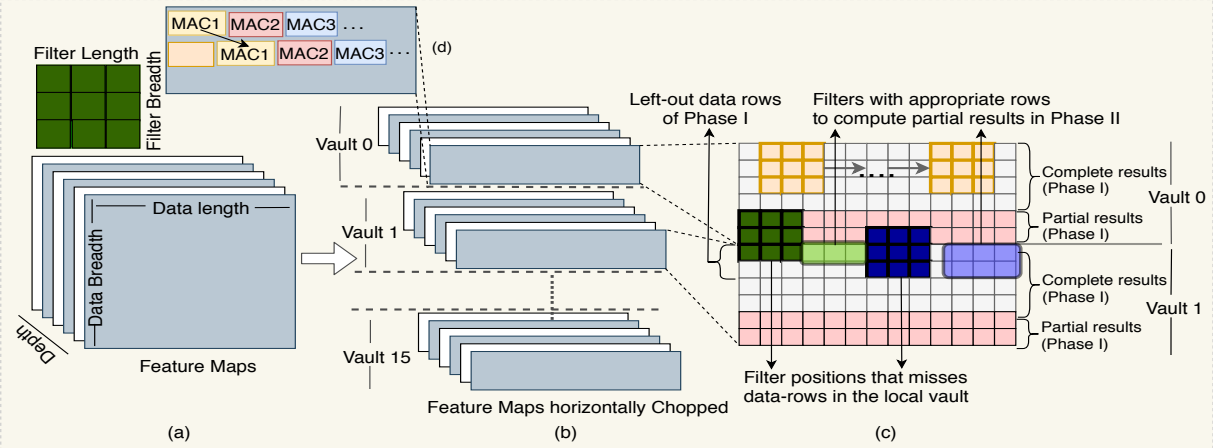


Figure 3.3: Data distribution and processing across vaults.

of the data to leverage the intra-vault parallelism. Figure 3.3 (d) shows the intra-vault parallelism, which depicts how the MAC units compute different neuron positions in parallel to others. All the MACs work in parallel on different portions of the image using the same filter from SRAM. As the entire strip of data belongs to one vault, the data dependencies on the adjacent vaults are avoided in most cases except for the tile edges. Figure 3.3 also shows the inter-vault data parallelism by distributing the input feature maps across multiple vaults for parallel processing by the CLUs (1 CLU/vault). The input feature maps are chopped horizontally on all the matrices (or layers). Each slice of data is then distributed over all the vaults by the host processor during the load time. The concurrent processing of the CLUs is one of the reasons for achieving this high speedup over the baselines.

One of the challenges of this distributed system during the processing of the *CONV* operations is non-local data. From Figure 3.3, it can be observed that the kernels at certain neuron positions (indicated by dark green and dark blue) will produce partial results as some of its data (non-local) will reside in the adjacent vault due to data partitioning. Note that we do not keep the replica of the data (feature maps) for space efficiency. This necessarily produces some partial or incomplete results (indicated by pink) in phase 1 of the processing. Upon completion of phase 1, the CLU controller enters into phase 2. In phase 2, the controller again starts computing new sets of partial results. These new sets of partial results of phase II are obtained by the convolution of the missing rows of input data (left-out rows located in the adjacent vault) and appropriate rows (indicated by light blue and light green) of the kernels as shown in Figure 3.3. These partial results of phase II are needed to make the partial results of phase 1 (indicated by the pink rows) complete. In the final phase 3, the controller only adds the partial results of phase 1 and phase 2 to obtain the final results, and it terminates the process. The detailed algorithm to implement the *CONV* operations

through these three phases (1, 2, and 3) is explained in the subsequent section.

Algorithm 1 Proposed Parallel Processing Algorithm.

Input: Data matrices (D) and filter matrices/kernels (F)
Output: Final result which is one output-matrix of the next activation layer

1: **for** $d_i = 0$ to depth **do**
2: *Load one layer of filter in SRAM buffer*
3: $write\_addr_{mID} \leftarrow initial\_result\_addr_{mID}$
4: phase_1 ()
5: phase_2 ()
6: d_i++
7: **end for**
8: phase_3 ()

3.3 Operational Steps

To leverage substantial speedup over the baselines (especially a standard GPU platform), we design a parallel algorithm (shown in Algorithm 1) with three different phases (1, 2, and 3). The proposed hardware (CLU) is implemented in such a way that it can execute this parallel algorithm and its three phases to parallelize the CNN operations. Consequently, the components of the CLU, shown in Figure 3.2, are used here to implement this proposed parallel algorithm. At the beginning of this algorithm, a single layer of the kernel (filter) is loaded from the DRAM into the SRAM buffer (line 2) instead of loading all the layers. Note that the Kernel elements are frequently used during the *CONV* operations. Thus, using SRAM as a small cache memory for storing kernel saves costly DRAM die accesses in this context. Each CLU has an array of 32 multiply-accumulate (MAC) units. Each MAC unit has a set of registers that allocate a portion of memory to the unit for its computations. All the MAC units work on their respective data partition to perform convolution using the three phases 1, 2, and 3. The $write\_addr_{mID}$ of the line 3 contains the initial addresses of the main memory location where the results of phase 1 and phase 2 are to be kept by the corresponding MAC units. Here 'mID' denotes unique MAC-ID, and it is subscripted with the registers and other modules to represent a one-to-one association with the MAC units. Initially, the entire location of the main memory, where the results are to be kept, is assigned with the zero (0) values. For each layer of the data and filter matrix along the depth (loop of the line 1), phase 1 and phase 2 are invoked (line 4-5) repeatedly. After all the layers along the depth have completed phase 1 and phase 2, phase 3 is called to terminate the process by adding the partial results of phase 1 and phase 2. At the end of phase 3, we

eventually obtain one matrix (or layer) of the output activation layer, generated by applying one kernel. Similar steps are repeated for individual kernels to form the next matrices of the output activation layer, and the depth increases by one for each kernel. Note that, unlike the previous version of CLU [115], which loads the whole kernel in the SRAM cache, the proposed CLU controller loads the kernel layer-wise (one layer at a time) to accomplish phase 1 and phase 2 consecutively (shown in algorithm 1). This approach makes this design capable of executing the deep CNNs with a small SRAM cache.

3.3.1 Phase 1

The CLU controller can handle any number of strides. In addition to this, the *depth-wise* filter loading technique has made the design capable of executing bigger ConvNets of different sizes and shapes. The CLU controller starts its process of phase 1 with the following steps.

- After the initialization of the kernel elements into the SRAM buffer, the CLU controller brings the input data items into the Reg. stack A (Reg_{mID} A) to be used by all MACs.
- The multipliers multiply the two values (one filter element and one data item), and the results are added to the cumulative value in Reg. stack B (Reg_{mID} B) of the respective MACs.
- When the *depth* (d_i) is zero for the first layer, a bias value is added only once at each neuron position by the CLU controller with the results available in Reg. stack B.
- The Reg. stack B contains the convolution results, computed by the multipliers and adders of each CLU, at each neuron position of the current layer. These results are added to the corresponding values of other layers to finally obtain one 2D output matrix for the next activation layer.

This process runs concurrently in all vaults and results in a snapshot of data, as shown in Figure 3.3. This gives some complete results (indicated by white) and some partial results (indicated by pink). Algorithm 2 is the step-wise implementation of the above-explained procedure. The $initial\_data\_addr_{mID}$ registers of the line 1 hold the initial addresses of the portion of input image data (D). The MEM_addr_{mID} registers of line 2 are used to fetch data from the DRAM die. The $current\_elem\_addr_{mID}$ registers are used to update the MEM_addr_{mID} registers after the CONV operations at each neuron position. The strides of the filters along the *data_breadth* and *data_length* are controlled by the two while loops of lines 4 and 6, respectively. For the first layer of the data matrix ($d_i = 0$ at line 1 of Algorithm 1), the bias value is added at each neuron position by setting the proper select

Algorithm 2 phase_1 ()

Input: Data matrices (D) and filter matrices/kernels (F)
Output: One matrix of the activation layer with few complete and partial results

$current\_elem\_addr_{mID} \leftarrow initial\_data\_addr_{mID}$
$fi_{mID} \leftarrow (mID * stride)/data\_length$, $MEM\_addr_{mID} \leftarrow current\_elem\_addr_{mID}$
$write\_addr_{mID}$: Initial addresses of the results in main memory
while $fi_{mID} < data\_breadth$ **do**
  $fj_{mID} \leftarrow (mID * stride)\%data\_length$
  while $fj_{mID} < (data\_length - filter\_length) + 1$ **do**
    Reg_{mID} B $\leftarrow 0$
    if $d_i == 0$ **then**
      MUX_{mID} 1 with $S_0 = 0, S_1 = 0$
      MUX_{mID} 2 with $S_0 = 1$
      Reg_{mID} B $\leftarrow \text{Reg}_{mID}$ B $+ Bias$
    end if
    if $(fi_{mID} + filter\_breadth) \leq data\_breadth$ **then**
      $i\_range_{mID} \leftarrow fi_{mID} + filter\_breadth$
    else
      $i\_range_{mID} \leftarrow data\_breadth$
    end if
    $K \leftarrow 0$
    for $i = fi_{mID}$ to i_range_{mID} **do**
      for $j = fj_{mID}$ to $fj_{mID} + filter\_length$ **do**
        Reg_{mID} A $\leftarrow$ D $[MEM\_addr_{mID}]$
        $Multiplier\_out_{mID} = SRAM[k] *$ Reg_{mID} A
        MUX_{mID} 1 with $S_0 = 0, S_1 = 1$
        MUX_{mID} 2 with $S_0 = 1$
        Reg_{mID} B $\leftarrow \text{Reg}_{mID}$ B $+ Multiplier\_out_{mID}$
        $K++$
        $MEM\_addr_{mID}++$
      end for
      $MEM\_addr_{mID} \leftarrow MEM\_addr_{mID} + data\_length - filter\_length$
    end for
    Reg_{mID} D $\leftarrow$ Mem $[write\_addr_{mID}]$
    MUX_{mID} 1 with $S_0 = 1, S_1 = 0$
    MUX_{mID} 2 with $S_0 = 1$
    Reg_{mID} B $\leftarrow \text{Reg}_{mID}$ B + Reg_{mID} D
    Store (Reg_{mID} B, $write\_addr_{mID}$)
    $write\_addr_{mID}++$
    $fj_{mID} \leftarrow [fj_{mID} + (stride * num_{mac})]\%data\_length$
    $current\_elem\_addr_{mID} \leftarrow current\_elem\_addr_{mID} + (stride * num_{mac})$
    $MEM\_addr_{mID} \leftarrow current\_elem\_addr_{mID}$
  end while
  $fi_{mID} \leftarrow [fi_{mID} + (stride * num_{mac})]\%data\_breadth$
end while

signals for MUX_{mID} 1 and MUX_{mID} 2 (line 8-11) of each MAC unit. The lines of 13-17 set the proper boundary conditions for the succeeding loops (lines 19-30), which calculate the dot products of filter and part of the data matrix for each layer at each neuron position by the MAC units (shown in Figure 3.3 (d)). Line 29 updates the memory address to fetch the data of the next row in the same neuron position. At each neuron position, the results of the convolution from the previous layer (d_i-1) are fetched into Reg. stack D (line 31). The values are added cumulatively with the results of the present layer (d_i), and the final result is updated (line 32-35). Finally, lines 37, 38, 39, and 41 shift the filters based on the number of MACs (num_{mac}) and the stride value available in register-stack. All the additional stack of registers (except A, B, C, and D) are the part of the register stack shown in Figure 3.2.

3.3.2 Phase 2

Phase 2 deals with the problem raised in phase 1 due to the data distribution. During the convolution process completed till phase 1, filters or kernels will lag a few rows of data elements at certain neuron positions (pink rows in Figure 3.3) due to the data distribution. In other words, the filters (dark green and dark blue in Figure 3.3) need to be applied on a set of rows located on two different vaults. In phase 2, the CLU controller computes the needed partial results separately in the local vaults. The controller, then, sends those results to the adjacent vault using the inter vault crossbar network in phase 3.

To accomplish phase 2, the step-wise pseudocode is presented in Algorithm 3. The 'trace_par'of Algorithm 3 tracks the number of sets of the left-out data rows due to the missing data elements in the local vault caused by the data distribution. For example, the value of the trace_par (stored in the register-stack) is 2, according to Figure 3.3. The reason is, for each pink row, there will be a set of left-out data rows in the adjacent vault. In this case, one set will contain two left-out rows, and the other will contain one left-out row. This means the while loop in line 2 will iterate for each of such pink rows (shown in Figure 3.3) in the respective MAC units. The $initial\_data\_addr_{mID}$ and MEM_addr_{mID} registers of lines 3-4 are used for the same purpose as in Algorithm 2. The loops from lines 6-9 compute the dot products (line 10-16) using the appropriate rows (lesser rows than the original $filter\_breadth$) of the filter over the data rows, left-out by phase 1. The partial results are computed simultaneously by all the MAC units. Here, lines 20-25 have the same semantics as line 31-36 of Algorithm 2. The required data for the next iteration are brought based on the num_{mac} and the value of the *stride* register (lines 26, 27, and 29). Here, all these partial results are updated in the DRAM die by the CLU controller. Hence, at the end of phase 2, all the partial results, needed to complete the partial results of phase 1, are

Algorithm 3 phase_2 ()

Input: Data matrices (D) and filter matrices/kernels (F)

Output: Convoluted partial data for sending to the previous vault (using NoC) to make the partial results of phase 1, complete

1: $trace\_par$: Number of left-out rows where the data elements were missing in the phase 1
2: **while** $trace\_par > 0$ **do**
3: $current\_elem\_addr_{mID} \leftarrow initial\_data\_addr_{mID}$
4: $fj_{mID} \leftarrow (mID * stride)\%data\_length$, $MEM\_addr_{mID} \leftarrow current\_elem\_addr_{mID}$
5: Reg_{mID} B $\leftarrow 0$
6: **while** $fj_{mID} < (data\_length - filter\_length) + 1$ **do**
7: $K \leftarrow no\_of\_filter\_elem - (trace\_par * filter\_length)$
8: **for** $i = 0$ to $trace\_par$ **do**
9: **for** $j = fj_{mID}$ to $fj_{mID} + filter\_length$ **do**
10: Reg_{mID} A $\leftarrow$ D $[MEM\_addr_{mID}]$
11: $Multiplier\_out_{mID} = SRAM[k] * \text{Reg}_{mID}$ A
12: MUX_{mID} 1 with $S_0 = 0, S_1 = 1$
13: MUX_{mID} 2 with $S_0 = 1$
14: Reg_{mID} B $\leftarrow \text{Reg}_{mID}$ B $+ Multiplier\_out_{mID}$
15: $K++$
16: $MEM\_addr_{mID}++$
17: **end for**
18: $MEM\_addr_{mID} \leftarrow MEM\_addr_{mID} + data\_length - filter\_length$
19: **end for**
20: Reg_{mID} D $\leftarrow$ Mem $[write\_addr_{mID}]$
21: MUX_{mID} 1 with $S_0 = 1, S_1 = 0$
22: MUX_{mID} 2 with $S_0 = 1$
23: Reg_{mID} B $\leftarrow \text{Reg}_{mID}$ B + Reg_{mID} D
24: Store (Reg_{mID} B, $write\_addr_{mID}$)
25: $write\_addr_{mID}++$
26: $fj_{mID} \leftarrow [fj_{mID} + (stride * num_{mac})]\%data\_length$
27: $current\_elem\_addr_{mID} \leftarrow current\_elem\_addr_{mID} + (stride * num_{mac})$
28: $MEM\_addr_{mID} \leftarrow current\_elem\_addr_{mID}$
29: **end while**
30: $trace\_par \leftarrow trace\_par - stride$
31: **end while**

obtained.

Algorithm 4 phase_3 ()

Input: Partial results of phase 1 and partial data computed in phase 2
Output: Complete results for one layer (or matrix)

1: $ph1\_par\_addr_{mID}$: Initial addresses of the partial results of phase 1
2: $ph2\_par\_addr_{mID}$: Initial addresses of the partial results of phase 2
3: $no\_of\_partial\_results$: Number of such partial results that need to be converted into complete results. This number is obtained during the phase 1 and phase 2 computations.

4: **for** $i = 0$ to $no\_of\_partial\_results/num_{mac}$ **do**
5: Reg_{mID} D $\leftarrow$ D $[ph1\_par\_addr_{mID}]$
6: Reg_{mID} C $\leftarrow$ D $[ph2\_par\_addr_{mID}]$
7: MUX_{mID} 1 with $S_0 = 1, S_1 = 1$
8: MUX_{mID} 2 with $S_0 = 0$
9: Reg_{mID} B $\leftarrow$ Reg_{mID} D + Reg_{mID} C
10: Store (Reg B, $ph1\_par\_addr_{mID}$)
11: $ph1\_par\_addr_{mID}++$
12: $ph2\_par\_addr_{mID}++$
13: $i++$
14: **end for**

3.3.3 Phase 3

We use two registers ($ph1\_par\_addr_{mID}$ and $ph2\_par\_addr_{mID}$ of the register-stack), which keep track of the partial results generated in phase 1 and phase 2 (shown in Algorithm 4). The register, $ph1\_par\_addr_{mID}$, holds the initial address of the partial results of the local vault (generated in phase 1), whereas the $ph2\_par\_addr_{mID}$ holds the beginning address of the partial results (generated in phase 2) of the adjacent vault. The CLU controller (shown in Figure 3.2) loads the local partial values in the Reg. stack D (line 5). Then, it loads the remote partial values from the adjacent vault into the Reg. stack C (line 6), using the NOC of the LoB. Note that we consider 4×4 2D-mesh NOC and assume 3 cycles for router and 1 cycle for wire as the zero-load delay [13]. By activating the proper select lines of the MUX_{mID} 1 and MUX_{mID} 2 (line 7-8) of each MAC, the CLU controller feeds the values of Reg. stack D and Reg. stack C to the adder and the final result is stored in Reg. stack B after the addition operation (line 9). The partial values of phase 1 are replaced in the memory when the results from register stack B are written back to the corresponding memory locations (line 10) to make them complete.

Table 3.1: Workload Details.

CNNs	AlexNet	ZFNet	VGG-19	ResNet-34
Year	2012	2013	2014	2015
Place	1^{st}	1^{st}	2^{nd}	1^{st}
Top-5 error	15.3%	14.8%	7.3%	5.7%
# of *CONV* layer	5	5	16	33
ConvNet Configuration	{[227×227×3], [11×11×3, 96], 4, 0} {[27×27×96], [5×5×96, 256], 1, 2} {[13×13×256], [3×3×256, 384], 1, 1} {[13×13×384], [3×3×384, 384], 1, 1} {[13×13×384], [3×3×384, 256], 1, 1}	{[224×224×3], [7×7×3, 96], 2, 0} {[55×55×96], [5×5×96, 256], 2, 0} {[13×13×256], [3×3×256, 512], 1, 1} {[13×13×512], [3×3×512, 1024], 1, 1} {[13×13×1024], [3×3×1024, 512], 1, 1}	{[224×224×3], [3×3×3, 64], 1, 1} {[224×224×64], [3×3×64, 64], 1, 1} {[112×112×64], [3×3×64, 128], 1, 1} {[112×112×128], [3×3×128, 128], 1, 1} {[56×56×128], [3×3×128, 256], 1, 1} {[56×56×256], [3×3×256, 256], 1, 1}×3 {[28×28×256], [3×3×256, 512], 1, 1} {[28×28×512], [3×3×512, 512], 1, 1}×3 {[14×14×512], [3×3×512, 512], 1, 1}×4	{[224×224×3], [7×7×3, 64], 2, 0} {[56×56×64], [3×3×64, 64], 1, 1}×6 {[56×56×64], [3×3×64, 128], 2, 0} {[28×28×128], [3×3×128, 128], 1, 1}×7 {[28×28×128], [3×3×128, 256], 2, 0} {[14×14×256], [3×3×256, 256], 1, 1}×11 {[14×14×256], [3×3×256, 512], 2, 0} {[7×7×512], [3×3×512, 512], 1, 1}×5
Parameter Size (Conv Layers)	7.15 MB	21.45 MB	38.19 MB	40.25 MB
Total size (Conv Layers)	9.30 MB	26.27 MB	87.87 MB	53.83 MB

3.4 Experimental Evaluation

We perform experiments on widely used state-of-the-art benchmarks from industries and academia. The selected benchmarks stand at the top in terms of reduced *'top*-5 *error rate'* in the ImageNet Large Scale Visual Recognition Challenge (ILSVRC) [130], which runs annually since 2010. In this challenge, the object detection and image classification algorithms are evaluated on over 15 million labeled high-resolution images from roughly 22 thousand categories. The details of the workload are described in the subsequent section. To quantify the efficacy of the proposed NMP based system, we compare it with the multi-core CPU and GPU based systems.

3.4.1 Workloads

Table 3.1 presents the particulars of the workloads. Note that we did not show the details of VGG-16 as VGG-19 has a similar configuration with additional layers. The *top*-5 *error* of these selected workloads reduced significantly over time. We represent the configuration of the ConvNets by the quadruplet, $\{[D_x \times D_y \times N_z], [F_x \times F_y \times N_z]$, S, P$\}$ as discussed in Section 2.1.1. The last term, 'P', is the padding used in each *CONV* layer of the CNNs. In some of the *CONV* layers, we multiply the quadruplet by the number of layers with the same configuration (shown in Table 3.1). The total size in Table 3.1 is the summation of the size of input feature maps, the filter elements, the output feature maps, and bias from all the CONV layers in a single forward pass. For example, the total size of the AlexNet is shown as 9.30 MB. AlexNet consists of five *CONV* layers (CONV1, CONV2, CONV3, CONV4, and CONV5). The following operations are performed in the respective layers.

- CONV1: $[227 \times 227 \times 1]$ *conv* $[11 \times 11 \times 3, 96] \rightarrow [55 \times 55 \times 96]$
- CONV2: $[27 \times 27 \times 96]$ *conv* $[5 \times 5 \times 96, 256] \rightarrow [27 \times 27 \times 256]$
- CONV3: $[13 \times 13 \times 256]$ *conv* $[3 \times 3 \times 256, 384] \rightarrow [13 \times 13 \times 384]$
- CONV4: $[13 \times 13 \times 384]$ *conv* $[3 \times 3 \times 384, 384] \rightarrow [13 \times 13 \times 384]$
- CONV5: $[13 \times 13 \times 384]$ *conv* $[3 \times 3 \times 384, 256] \rightarrow [13 \times 13 \times 256]$

The total number of elements of CONV1 is 479931 which includes input features, kernels, output features, and bias of $227 \times 227 \times 3 = 154587$, $11 \times 11 \times 3 \times 96 = 34848$, $55 \times 55 \times 96 = 290400$ and 96 elements, respectively. Similarly, CONV2, CONV3, CONV4, and CONV5 are having 893536, 1007616, 1478784, and 1014656 elements, which lead to the total memory requirement of $1014656 \times 16bits \approx 9.30$ MB. We kept the size of the HMC memory as 2 GB, as explained in Section 3.2.1. Although the program will consume a little more memory than the reported total memory of Table 3.1 due to the temporary variables, our proposed system suffices the need of all the ConvNets. However, bigger (in size) HMC devices or multiple connected HMC devices (as shown in [131]) can also be used for further memory requirements. Note that our design of CLU is generic and independent of the memory size and number of HMC chips, thus making it scalable for the ConvNets of any shape and size. The dimensions of the output feature maps, being straightforward, are not shown in Table 3.1. The total size of the weights (parameters) of the individual ConvNets is also shown separately in Table 3.1. For this work, we assume a batch size of 1, which is common for inferencing tasks [57].

3.4.2 CPU based baseline Systems

We use PyTorch 1.4.0 [133] along with the *MKL* library to execute all the ConvNets in the CPU-based baselines. The inference operations are performed on the pre-trained models available with the PyTorch framework. The execution of the models is performed on the quad-core and the 64-core CPU-based systems, having x86 instruction set architecture (ISA). For the 64-core CPU-based system, we use a cluster of 4 nodes, each having 16 cores. The systems include a three-level memory hierarchy (private L1, shared L2, and shared L3) with a traditional DDR4 main memory. We summarize the specifications of the simulation setups in Table 3.2. The layer-wise execution times of the *CONV* layers are obtained from the python code of the framework. We model similar cores in McPAT [134] to obtain the power values and measure the energy consumption of CPU-based baselines.

Table 3.2: Specification of the simulation setup.

Host Processor / CPU-based baselines	
Core ISA	x86-64
Host processor's core count	4
Baseline CPU's core Count	4 (Baseline 1), 64 (Baseline 2)
Thread(s) per core	1
Core Frequency	2.5 GHz
Cache Memory	
L1 i-cache	32 KB, private, 8 way associative, 64B blocks
L1 d-cache	32 KB, private, 8 way associative, 64B blocks
L2 cache	256 KB, shared, 8 way associative, 64B blocks
L3 cache	30 MB, shared, 20 way associative, 64B blocks
Main Memory	
Size	128 GB
Type	DDR4_1600_x64
GPU	
GPU Name	NVIDIA Tesla P100 (PCIe-Based)
CUDA Cores frequency	1.33 GHz
GPU Memory	16 GB HBM2
Maximum Power Consumption	250 W
Proposed logic unit for NDP (CLU)	
CLU	16 in number (one for each vault)
Multipliers and adders	32, 16 bit fixed-point / CLU
Cache	1 SRAM cache per CLU, each is 256 B in capacity
Frequency	550 MHz - 1 GHz on UMC 90 nm technology, 3.3 GHz - 5.9 GHz on 15 nm technology
Peak performance	3.4 TFLOPs (15 nm)
Host cache	2-level (L1 and L2)
3D Memory Stack (HMC)	
Timings	t_{RP} = 7.7ns, tt_{CCD} = 3.3ns, t_{RCD} = 10.2ns, t_{CL} = 9.9ns, t_{WR} = 15ns, t_{RAS} = 21.6ns, t_{CK} = 0.8ns
Energy [40]	3.7 pJ/bit for DRAM read, 6.78 pJ/bit for SerDes hop
Power[132]	11.08W
Size	2 GB
Logic Die	90 nm, dimension $27 \times 27mm^2$

3.4.3 GPU based baseline System

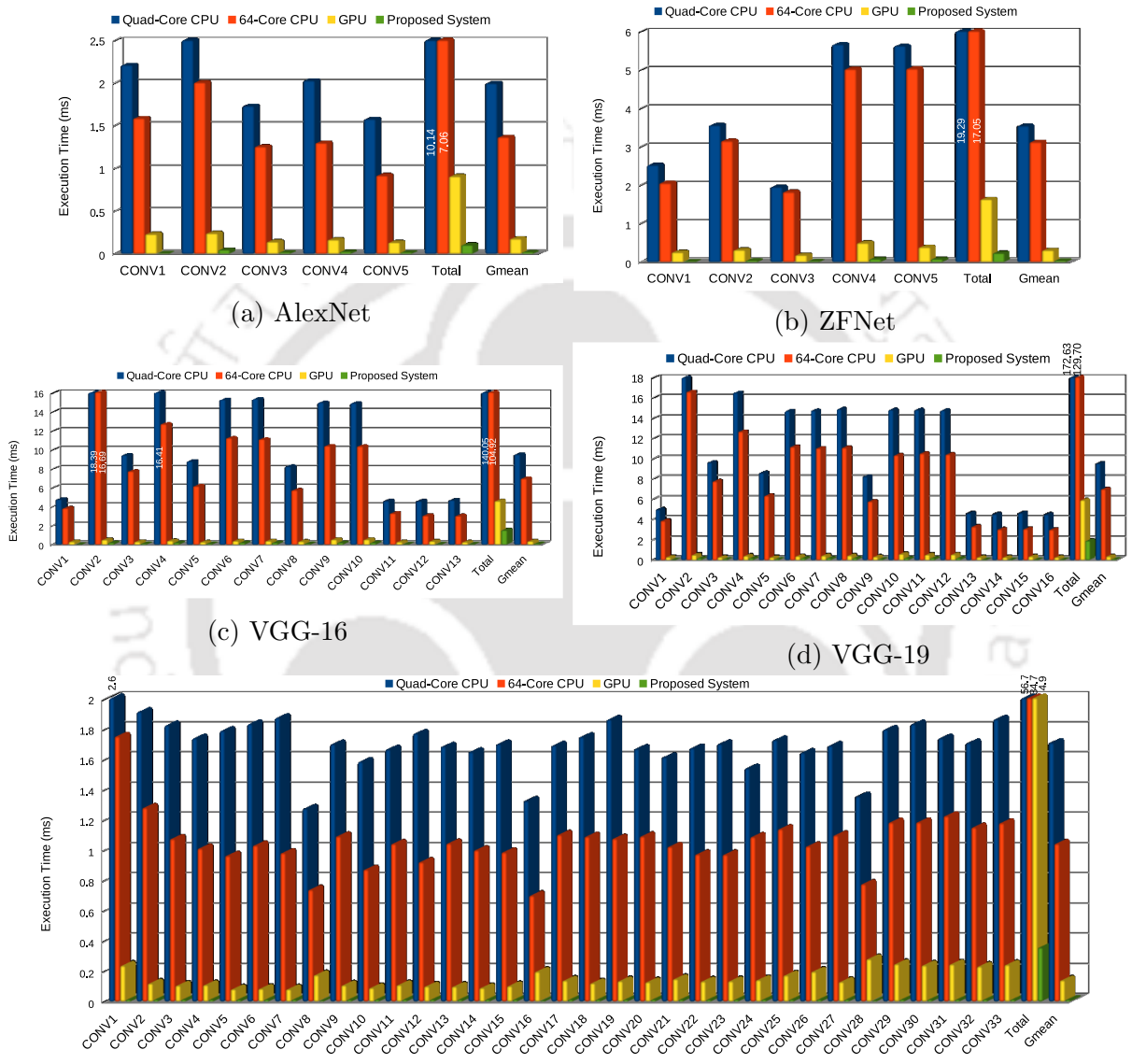
We compare the throughput of the proposed system with a high-end GPU-based system with NVIDIA Tesla P100 (Pascal architecture) GPU. The performance and energy efficiency of the Tesla P100 is shown in [135]. The size of the global memory (HBM2 Stacked) of the GPU is 16 GB. The details of the GPU are shown in Table 3.2. Note that Tesla P100 being a high-end GPU, costs more than $9 K. On the contrary, our NMP solution is cost-effective and can easily be integrated with the existing ecosystems of the HMC devices in the same package structure and within the same power budget. Moreover, an HMC module costs less than $1.4 K, which is expected to reduce with its growing market size [15]. We use PyTorch 1.4.0 along with the *cuDNN* library to execute all the ConvNets (shown in Table 3.1) in the GPU platform. The GPU's throughput is measured from the Python code of the framework. We have used a similar instruction set architecture (ISA), x86, for all the CPU cores across all the systems, including the proposed system, as shown in Table 3.2. The host offload overhead is excluded for the GPU-based system and the proposed architecture.

3.4.4 Designing NMP Hardware using CAD tools

We implement the CLUs in Verilog and perform the synthesis using Genus Synthesis Solution (version 15.21) from Cadence. We obtain the operating frequency of the CLU around 550 MHz - 1 GHz after the synthesis at 90 *nm* technology. We develop an in-house cycle-accurate simulator that resembles the state machine of the design hardware (CLU). The execution times of the CONV layers are measured from the simulator by using the frequency obtained from the synthesis. Note that the layer's properties are extracted from the PyTorch simulation to maintain consistency while comparing the results. The obtained execution times include the data processing time in the CLU and the data transfer time to/from the DRAM die. The area analysis is performed using UMC 90 *nm* technology. The CLU net-list, generated by the Genus, is fed to Innovus (from Cadence tool-set) for layout and placement of the standard cells to perform a detailed area analysis, which is discussed in the subsequent section. The power consumption and area overhead of the proposed system is measured from the Cadence tool-set (Genus and Innovus) after the synthesis and layout.

3.4.5 Results

While investigating the concept of near-memory processing to accelerate CNN algorithms, we obtain motivating results with substantial performance gain with reduced energy consumption compared to the baselines. The area, being extremely critical for integrating logic



(a) AlexNet

(b) ZFNet

(c) VGG-16

(d) VGG-19

(e) ResNet-34

Figure 3.4: Performance analysis of ConvNets while accelerating CONV layers.

Table 3.3: Speedup of proposed design over respective baselines (baseline/proposed) while accelerating CONV layers.

ConvNets	Quad-Core	64-Core	GPU
AlexNet	98.87	68.84	8.84
ZFNet	82.76	73.14	7
VGG-16	93.18	69.81	3.04
VGG-19	90.36	67.89	3.09
Rsnet-34	158.90	97.42	13.87
Gmean	**101.83**	**74.69**	**6.04**

on the same memory chip, the practical implementation of the CLU hardware massively depends on the area overhead. Thus, we perform a detailed area analysis of the designed hardware and find overhead significantly low for near-memory integration.

3.4.5.1 Performance Analysis

Figure 3.4 shows the performance of the different systems while executing the ConvNets. In Figure 3.4, layer-wise execution times (ms) are plotted along the Y-axis, and all the *CONV* layers of the respective ConvNets are shown along the X-axis. We execute all the five ConvNets on four individual systems (2 on the CPU based + 1 on the GPU platform + 1 on the proposed system). From the results, it can be visualized that we achieve high speedups over the conventional multi-core CPU systems. On average, we procure around 101.83x and 74.69x improvements over the quad-core and 64-core CPU-based systems, respectively, as shown in Table 3.3. We acquire a speedup up to 13.87x (shown in Table 3.3) over a standard GPU, which demonstrates the efficacy of the proposed NMP-based system. The substantial gain in the performance is achieved primarily because of three reasons: (1) Minimum memory access latency as data travels much shorter distance due to the NMP approach compared to the baselines, (2) The proposed algorithm performs the CNN tasks in parallel, and (3) The efficient design of CLUs which are dedicatedly built for accelerating the *CONV* operations.

Table 3.3 shows the speedup over each baseline of the individual ConvNets. Indeed,

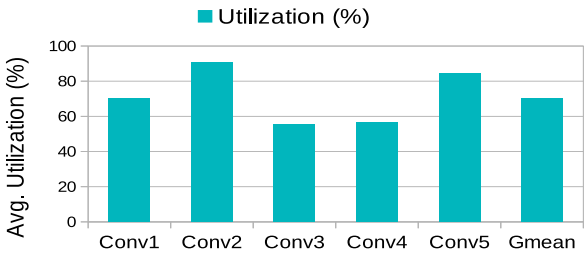


Figure 3.5: Average utilization for the arithmetic units (AlexNet).

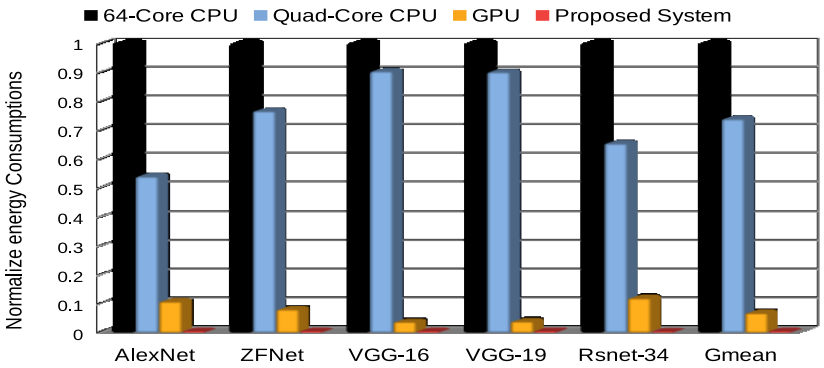


Figure 3.6: Energy consumption of different ConvNets/ImageNets while accelerating CONV layers; normalized w.r.t 64-core CPU-baseline.

in the case of a relatively slow quad-core CPU based system, we get maximum speedups, which gradually diminish with the more powerful systems like 64-core CPU and GPU based systems. Here, the execution times of the proposed NMP-based system include both the computation time as well as the data fetch/store time in the DRAM die from the logic layer. In the case of CPU/GPU based baselines, the execution times also comprise the computation times of the CPU/GPU and the data fetch/store times through their respective conventional memory hierarchy.

We also study the average utilization of the arithmetic units of the designed CLU modules. The Y-axis of Figure 3.5 shows the percentage of utilization, and the X-axis represents the CONV layers for the AlexNet. On average, we found around 70 % utilization for the arithmetic units of our designed hardware.

3.4.5.2 Energy Savings

We design an energy-efficient system for the applications that implement CNN algorithms. The objective is to develop a low power system with limited resources yet comparable with a GPU integrated system. Figure 3.6 shows a comparative analysis of the energy consumption of the different systems while executing the CNN algorithm. We achieve a substantial

Table 3.4: Energy savings of proposed design over respective baselines (baseline/proposed).

ConvNets	64-Core	Quad-Core	GPU
AlexNet	201.29	108.41	21.55
ZFNet	213.85	163.35	17.06
VGG-16	204.12	183.91	7.40
VGG-19	198.51	178.35	7.51
Rsnet-34	284.86	185.85	33.76
Gmean	**218.40**	**160.33**	**14.72**

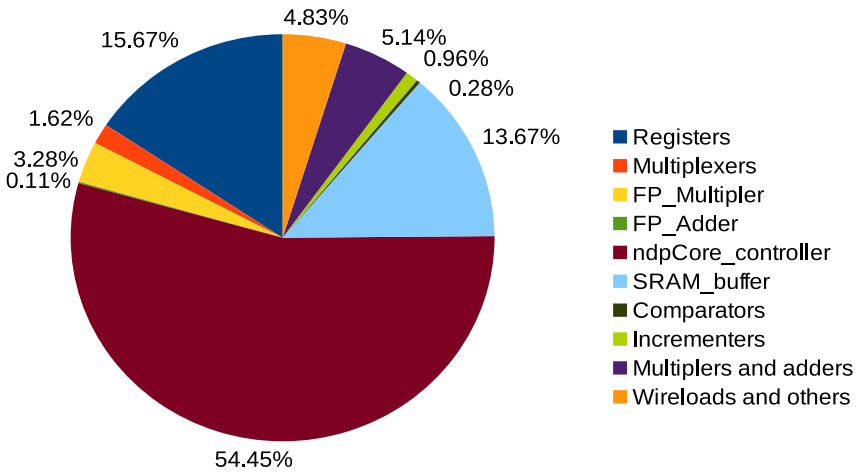


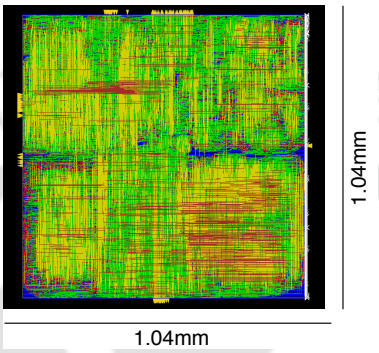
Figure 3.7: Power (Leakage + Dynamic) breakdown of CLU.

reduction in energy consumption in the proposed system compared to the baselines for all the CNN benchmarks. Tables 3.4 exhibits the energy savings of the proposed system over all the baselines while executing the respective ConvNets. On average, we get around 218.40x, 160.33x savings in energy over 64-core CPU, quad-core CPU based systems, respectively, and maximum up to 33.76x energy savings over the GPU based system. The reasons for the saving are the energy-efficient design of the CLU and the NMP approach, where we place these logic units on the logic die close to the memory. Consequently, the incurred energy of transferring bits becomes less in the proposed system as compared to the conventional multi-core CPU/GPU based systems.

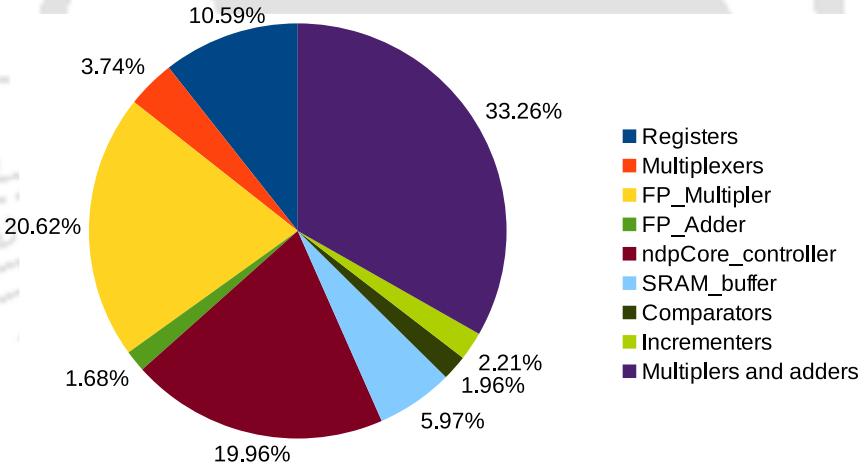
Figure 3.7 shows the power consumption of the discrete modules within the CLU. From Figure 3.7, it can be observed that the SRAM buffer consumes the power of around 13.67% as we keep a small buffer of only 256 B. This motivates our design approach of integrating a small buffer and loading the kernel depth-wise (one layer or matrix at a time) without loading the entire kernel into a bigger-sized cache. Note that we consider the total energy consumption, which includes both the computation and the data fetch/store energy for all the baselines as well as the proposed system. The parameters like timings, power, and transfer-energy/bit for the proposed 3D-memory based system are kept similar to the state-of-the-art, as mentioned in Table 3.2.

3.4.5.3 Area Analysis

Since memory industries are extremely area sensitive, area overhead plays a central role in the near-memory integration of the logic. We use Innovus (from Cadence tool-set) for area analysis, and our CLU net-list got placed on $1.04mm \times 1.04mm$ (shown in Figure 3.8 (a))



(a) Chip Layout from Innovus.



(b) Area breakdown of the designed CLU hardware.

Figure 3.8: Area analysis.

square box at 90 *nm* technology. This leads to a negligible area overhead of 2.37% in total for 16 CLUs on the logic die. Figure 3.8 (a) shows the layout of the designed hardware where the standard cells are placed. The area breakdown of the different elements of the CLU is presented in Figure 3.8 (b). The sequential elements have imposed an area overhead of around 16.56% while the logic components consume the rest of the area.

3.4.6 Comparison with Previous Accelerators

We perform a detailed comparison with existing DNN accelerators, NeuroStream [15], and NeuroCube [3], that incorporate the idea of NMP. For parallelizing the DNN operations, a data partitioning approach is used in NeuroCube [3]. This approach needs a specialized state machine integrated with each vault controller to drive the data into the processing elements. Unlike their approach, our proposed design needs no change in the controller circuit. The proposed hardware (CLU) has the inbuilt capability of bringing the data for processing. Moreover, their experiment is limited to the small ConvNets having only 6 layers. In NeuroStream [15], deep learning operations have been performed in the LoB of the 3D memory. They adopted RISC-V cores and streaming coprocessors as an NMP logic option to accelerate the CNNs. We choose dedicated hardware (CLU) instead of the RISC-V cores and streaming coprocessors in our proposed design. The reasons are: 1) for efficient utilization of these RISC-V cores, they require more programming effort, 2) their register-file bottleneck affects the scalability [136]. The amount of efficiency loss due to the use of RISC-V cores in [15] is reported to be 17%. The design in [15] increases 8% of the LoB die area on each HMC, which is significantly higher than the area overhead of our proposed system. Moreover, their data partitioning scheme (4D tiling) puts a significant workload on the RISC-V cores as data needs to be re-tiled after every layer before storing into DRAM. Our proposed design survives from such re-tiling overhead as we partition the data among all the vaults at the beginning of the CNN algorithm and keep the input and output activation in the corresponding vaults. Only a negligible amount of data movement is needed between the adjacent vaults in phase 3 (explained in Section 3.3.3) of our proposed algorithm.

Table 3.5: Comparison with previous near-memory DNN accelerators.

Node (nm)	SOA	Peak (TFLOPS)	Power (W)	Efficiency (GFLOPS/W)	*Area (mm^2)*
28 nm	NS [15]	0.24	11	22.5	8.3
	CLU	0.86*	2.36	364.41	1.73
15 nm	NC [3]	0.13	3.4	38.8	0.98
	CLU	1.4*	0.92	1521	1.07

NS: NeuroStream, **NC:** NeuroCube,
On average, actual throughput achieved during execution (Peak is even higher)

Table 3.5 summarizes the obtained key-design specifications of the NMP-based accelerators. On a similar technology node, the proposed work (CLU) outperforms NeuroStream [15], and NeuroCube [3] in terms of performance, power, efficiency, and area overhead. The proposed CLU achieves a peak throughput of 0.86 TFLOPS and 1.4 TFLOPS over 0.24 TFLOPS of NeuroStream [15] and 0.13 TFLOPS of NeuroCube [3] respectively, as shown in Table 3.5. While the efficiency of 364.41 GFLOPS/W and 1521 GFLOPS/W is substantially higher than both the corresponding state-of-the-art architectures, the proposed CLU incurs a comparatively lesser area overhead as well (shown in Table 3.5). Another work, TETRIS [16], provides an HMC-based near-memory architecture to accelerate the neural network (NN). This work adopts "row-stationary" computation paradigm proposed in [120]. Similar to our CLU, TETRIS uses a feature map (Fmap) based data partitioning scheme for the parallel execution of the NN tasks in the CONV layers. However, they use a different partitioning technique in the Fmaps. Unlike our horizontal Fmap partitioning scheme, they chop the data both horizontally and vertically, leading to more partitions and data tiles. Although more partitions can lead to more parallelism, this scheme also increases the inter-tile data dependency from different vaults at the tile edges due to the striding window nature of the convolution operations. Thus, it increases the inter-vault communications overhead. A throughput of 159 GOPS can be estimated from the relative results reported in TETRIS [16] with an average power consumption of 6.9 W. At 28 nm technology, TETRIS's area overhead can be scaled to 21 mm^2. A similar estimation of performance and area overhead of TETRIS is also observed in NeuroStream [15]. Our CLU provides higher performance and energy efficiency compared to TETRIS while being more area efficient.

Apart from the above similar near-memory accelerators, efforts have also been spent to build on-chip custom accelerators like Tensor Processing Unit (TPU) [137]. Although this accelerator delivers substantially high throughput (92 TOPS), it is difficult to integrate a similar bulky unit (331 mm^2) with a larger MAC array (64 K) in the logic layer of 3D memory. The logic die of the HMC device that we consider permits approximately 50-60 mm^2 in total or 3.5 mm^2 per vault of additional space for near-memory logic apart from its existing circuitry [40].

3.5 Summary

In this chapter[1], we propose an efficient parallel algorithm (having three phases) to maximize the speedup of the proposed system while accelerating the CNN inference. We design the hardware (CLU), which implements this parallel algorithm. To exploit the benefits of the near-memory approach and harness more throughput from the proposed system, we further integrate these CLUs in the logic layer of the HMC device. The cumulative outcome of these contributions results in our proposed architecture, which delivers substantially better throughput (in terms of performance and energy savings) than the multi-core CPU and GPU based systems. On average, we achieve around 101.83x and 74.69x performance gain over the quad-core and 64-core CPU based system, respectively. The proposed system also procures around 218.40x and 160.33x energy savings over the 64-core and quad-core CPU based system. In the case of GPU, our proposed system has accomplished a speedup of up to 13.87x while reducing the energy consumption up to 33.76x. Moreover, at 90 *nm* technology, the additional hardware units incur 2.37% area overhead, which is also negligible. Our proposed CLU is also comparable with other existing NMP-based architectures and outperforms them in terms of performance, power consumption, and efficiency. Certainly, the obtained results make this architecture a lucrative and more deserving candidate for practical implementation. This chapter covers the objectives like reducing data movement (objective1), obtaining high performance and energy efficiency through accelerated architecture (objective2), and providing support for various applications (objective3).

[1]The publication details related to this chapter are as follows.
(1) Das, Palash, et al. "Towards near data processing of convolutional neural networks." 2018 31st International Conference on VLSI Design and 2018 17th International Conference on Embedded Systems (VLSID). IEEE, 2018.
(2) Das, Palash, and Hemangee K. Kapoor. "CLU: A Near-Memory Accelerator Exploiting the Parallelism in Convolutional Neural Networks." ACM Journal on Emerging Technologies in Computing Systems (JETC) 17.2 (2021): 1-25.

4

nZESPA: A Near-3D-Memory Zero Skipping Parallel Accelerator for CNNs

Apart from exploiting parallelism like the previous chapter, we explore additional optimizations of skipping ineffectual zero-valued computations by leveraging the abundant data sparsity in both weights and activations. We use a compression/decompression technique of data to take the benefits of sparsity. We design dataflow that helps implement intra/inter-vault parallelism and utilize data sparsity. We implement custom hardware that works with the dataflow to skip all ineffectual computations. A grid of 16 hardware units is integrated with each vault of the hybrid memory cube (HMC) to exploit the benefits of NMP. The proposed architecture achieves high speedup and energy efficiency compared to all baselines and state-of-the-art.

4.1 Introduction

As we mentioned in our previous chapter, the CNN models have grown deeper (with several hidden layers) and exceptionally large in size. The number of both neurons and synaptic weights of the CNNs has increased dramatically (neurons: kilo-million neurons and synaptic weights: up to 10 billion) [116], [129]. Consequently, an extensive amount of computations and memory accesses are involved in CNN inference operations. These compute and data-intensive CNN operations paralyze the memory hierarchy and increase the costly off-chip DRAM accesses, which require higher cycles and energy to transfer data between the last-level cache and main memory. Eventually, the system's throughput in terms of performance and energy consumption degrades. Any optimization of the CNN algorithm, regarding its performance and energy consumption, benefits varieties of applications that include CNNs

as the core computation module. Towards optimizing the CNN algorithm, researchers have exploited the inherent parallelism of the CNN operations [45], [103], [137]. A few groups have eliminated unnecessary zero-valued computations by utilizing the abundant sparsity present in the neurons (Dynamic) [120], [104] and synaptic weights (static) [105], [138]. SCNN [57] and EIE [106] have exploited both static and dynamic sparsity to reduce the number of computations of the CNNs. Researchers have also put in efforts to reduce the costly off-chip main memory accesses by near-memory processing (NMP) to achieve a high system's throughput while accelerating the CNNs [15], [139], [115].

In the first category, the parallelism of the CNN operations has been exploited by the use of multiple processing units, resulting in an improved systems throughput in terms of performance and energy consumption. In the second category, people have utilized the sparsity property to reduce the number of computations in CNNs, thereby improving the systems throughput. Sparsity can be defined as the number of zeros present in the weight (synapse) and activation (neuron) matrices [140]. This sparsity is usually static in nature for the weights as it stems out by pruning connections in the network during the training. In contrast, the sparsity in activations, raised by the thresholding of ReLU (Rectified Linear Unit), is data-dependent and hence dynamic in nature. Multiplications and additions of these weights and activations are the primitive operations for the inference. Since multiplication or addition with zero produces nothing, these operations can be eliminated by over an order of magnitude to skip the costly computations. Around 18-85% static and 25-60% dynamic sparsity is observed for the different layers of CNN benchmarks in the previous work [140]. A similar observation is made (static: 20-80%, dynamic: 50-70%) in another work [57]. This has motivated many researchers to exploit the sparsity property. In some research, only the one-sided sparsity, like the static or the dynamic sparsity, has been investigated. Others have explored the benefits of both static and dynamic sparsity and thus fully-sparse. Nonetheless, the above two approaches still fail to address the issue of costly main memory accesses while fetching/storing data in the main memory. One solution to this problem is near-memory processing (NMP), where some computations are offloaded to the processing units, placed near to the memory. The NMP approach helps in amortizing the off-chip accesses. Several recent projects on near-memory processing of neural networks have manifested that the systems having NMP capability expedite the performance while being significantly energy efficient. As explained above, the existing works have been able to leverage isolated benefits of parallelism or sparsity or NMP by their proposed architecture. Addressing all three issues by one novel architecture is still a problem and has not been implemented.

In this chapter, our proposed system exploits the parallelism by distributing the CNN tasks among the proposed processing units placed near to the memory. In this context,

we choose 3D memory, specifically Hybrid Memory Cube (HMC) [40] for the near-memory integration of the designed processing units. We utilize both static and dynamic sparsity to eliminate the costly multiply-accumulate (MAC) operations of the CNN inference without affecting the accuracy. Apart from being near-memory, we use data compression on both weights and activations. Encoding of sparse weights and activations provides twofold benefits: 1) It reduces the number of DRAM access from the HMC's logic layer (LoB) where we place the proposed hardware, 2) It helps to accommodate the required weights and activations in a comparatively smaller sized buffer in the proposed hardware. Lastly, we have successfully curtailed the off-chip main memory accesses using the NMP approach while executing the inference. The reason is, the majority of the computations on the data take place inside the LoB of the HMC device without bringing them to the conventional cache hierarchy. Towards achieving these goals, we design our proposed hardware, namely Near-3D-memory Zero Skipping Parallel Accelerator (*nZESPA*), and integrate multiple such hardware units in the LoB of the HMC device. The proposed system outperforms the baselines in terms of performance and energy consumption, as shown in Section 4.6.3. The salient contributions are as follows.

1. To harness parallelism, we design an efficient sparsity-aware near-3D-memory dataflow for layer-wise processing of data, namely *Near-3D-Memory Zero Skipping Parallel dataflow* (*nZESPA*).

2. We design specialized hardware (*nZESPA*) and integrate a grid of those units with the individual vaults (explained in Section 4.4) in the LoB of the HMC device. The *nZESPA* is area efficient and can be easily integrated into the existing ecosystem of the HMC.

3. We exploit static and dynamic sparsity, which reduces the computation cost by eliminating the zero-valued computations. The sparse data is also represented and utilized in a compressed form reducing the buffer requirement in the implemented hardware.

4. To quantify the ability of the proposed system having fully-sparse CNN accelerator with NMP (NMP-fully-sparse), we compared it with three other variants of architectures: ⓐ System having dense CNN accelerator without NMP (traditional-dense), ⓑ System having fully-sparsed CNN accelerator without NMP (traditional-fully-sparse), and ⓒ System having dense CNN accelerator with NMP (NMP-dense). The NMP-fully-sparse outperforms the other three architectures in terms of performance and energy consumption.

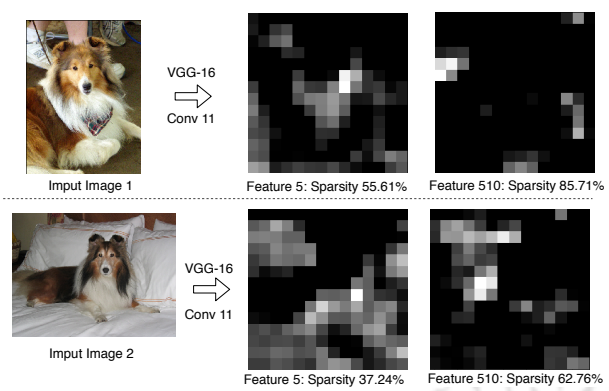


Figure 4.1: Dynamic nature of the activation-sparsity.

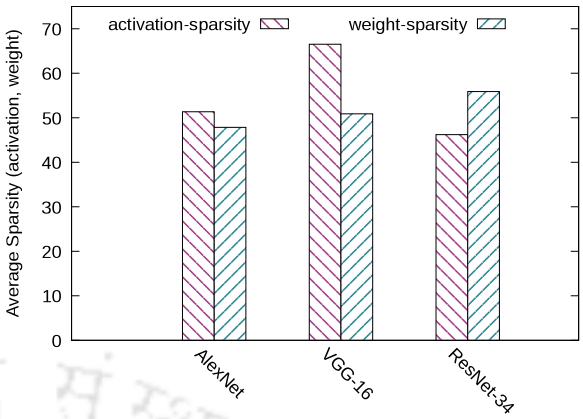


Figure 4.2: Exploitable Sparsity for the ConvNets.

5. We also compare the proposed NMP-fully-sparse architecture with state-of-the-art approaches exploring either static or only dynamic sparsity.

6. The throughput of the proposed system is evaluated with three widely used state-of-the-art ConvNets/networks with different shapes and sizes: AlexNet [116], VGG-16 [118], and ResNet-34 [48].

4.2 Motivation

The term sparsity can be defined as the fraction of zero values found in the layer's activation and weight matrices. The abundant sparsity, present in the CNNs, brings enormous opportunities for the system architects to design highly efficient systems with optimized energy consumption. In practice, all the widely used ConvNets/networks include sparsity in both the activation and weight matrices. The effective approach to create weight sparsity is to prune the network during the training phase. One of the effective techniques to prune the weights is shown in [141]. Here, the weight values which are closed to zero (e.g., within a certain threshold) are set to zeros. This pruning process affects the accuracy. To regain accuracy, they retrain the network. These two pruning and retraining steps result in a smaller network with accuracy, very close to the original network. Once the training is done, the weight sparsity remains unchanged and hence static in nature.

Activation sparsity is created primarily by the non-linear operator like ReLU as it forces all negative activation values to be clamped to zeros. The sparsity of activations is highly data-dependent and hence dynamic in nature. Figure 4.1 shows the dynamic nature of the activation sparsity. The CONV11 layer of VGG-16 is shown for the two different input images

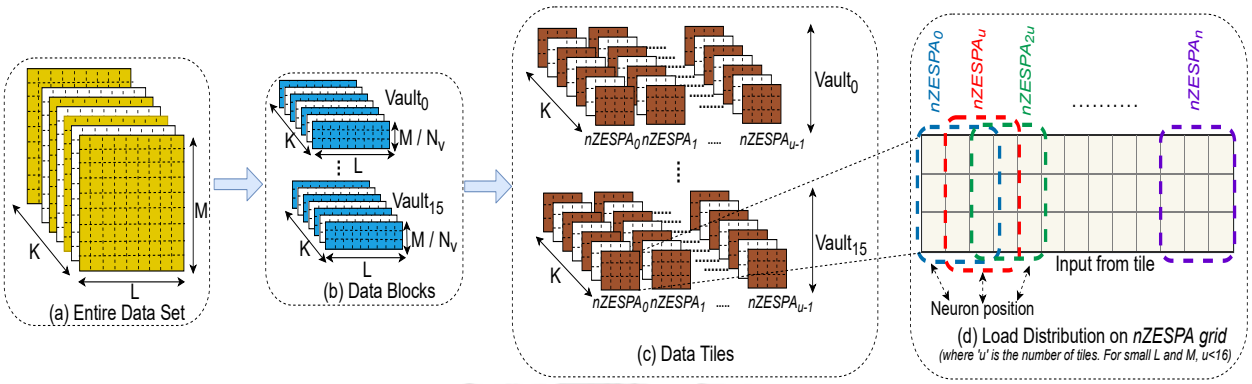


Figure 4.3: Data distribution for *nZESPA* dataflow.

from the ImageNet dataset. A variance in sparsity is observed across the input images. Note that, here, the black pixels represent the zero values that are responsible for the sparsity. To inspect the activation and weight sparsity across the hidden layers of all three networks, we use the PyTorch framework [133] along with the pruning algorithm of [141]. Figure 4.2 illustrates the average sparsity (in percentage) in the activation and weight matrices of all three networks that have been used for evaluation. We observed a range of ∼20-70% weight and ∼28-80% activation sparsity across the different layers of the networks. For this work, we assume a batch size of 1, which is common for inferencing tasks.

4.3 nZESPA Dataflow

The *Near-3D-Memory Zero Skipping Parallel* dataflow or *nZESPA* dataflow is specifically designed to maintain the streaming of data through TSVs between the DRAM layers of the memory die and the *nZESPA* modules placed on the LoB of the HMC device. The objectives of the dataflow are multidimensional while executing the inference. The *Near-3D-Memory* term represents its active workspace, which is the logic-layer (LoB), close to the memory die. The *Zero Skipping* term is used as it is specially designed to perform its computations on the compressed form of data (activations + weights) obtained after removing zeroes. The compressed data helps in removing all multiply-accumulate operations associated with zero-valued weights and activations, leading to the exploitation of sparsity property. The *nZESPA* dataflow keeps both the activations and weight model in the vaults until the end of the entire algorithm. This property increases the data reuse to some extent and reduces the overhead of data-partitioning after each activation layer. To harness the parallelism, the weight matrices are replicated in the vaults while the data is partitioned (shown in Figure 4.3) across the vaults for the parallel processing by the *nZESPA* grids. The dataflow provides

parallelism in the form of intra and inter vault parallelism, as explained in the subsequent section. Since we integrate the *nZESPA* grids close to memory (NMP), the data can be transferred between the DRAM layers and *nZESPA* modules with negligible transmission overhead [15]. Consequently, we can exchange the activations and the filters between the DRAM layers and the local buffers of *nZESPA* modules with minimal effect on the system's throughput. One inefficiency that stems out during the parallel execution of the inference tasks is load imbalance. Load imbalance often occurs as an effect of the data reuse scheme. Like in [57], the filter elements can be broadcast among the processing elements (PEs) for performing the MAC operations while keeping the input_activations stationary in the local buffers of PEs. The input maps having different sparsity, sparser maps finish early than denser maps, leading to the under-utilization of resources due to the load imbalance. Alternatively, holding the filter elements and broadcasting the input_activations incurs the same problem. The load imbalance can be alleviated if the PEs independently fetch the weights and activations to the local buffers by sacrificing the data reuse. Thus there is a fundamental reuse-imbalance trade-off while designing the dataflow. We fetch the activations and filters in the local buffers instead of broadcasting them among the *nZESPA* modules as a solution to the load imbalance problem.

4.3.1 Data Partitioning for *nZESPA*

Figure 4.3 explains the data distribution and its granularity in more detail. The entire $L \times M \times K$ amount of activation inputs (Figure 4.3a) are chopped horizontally, leading to data blocks, as shown in Figure 4.3b. Here, L and M are the counts of columns and rows, respectively, for each input channel; and the total number of channels is represented by K (same as shown in Figure 1.4). Each block of $L \times (M/N_v) \times K$ elements is distributed across the N_v number of vaults of the HMC. We choose the 16-vault configuration of the HMC 2.0 for our experiment. However, one can choose the other available HMC configurations as in HMC 2.1. It (2.1 version) has 32 vaults, which can be used for the integration of 32 *nZESPA* grids. Note that our design of *nZESPA* modules are independent of memory architecture and can be adopted in any 3D-stacked memory with minimal changes in the existing ecosystem. Each block in the respective vaults is logically divided into smaller pieces of tiles (Figure 4.3c), each having $L/N_{spc} \times (M/N_v) \times K$ elements. Here, N_{spc} represents the number of *nZESPA* modules (16 in proposed NMP-fully-sparse), integrated into each vault. Figure 4.3d shows the load distribution in a finer granularity when L, M is small, and the number of tiles becomes less than N_{spc}. To maximize hardware utilization for smaller layers (small L

and M), the *nZESPA* modules within a grid perform the convolutions simultaneously in the sequential neuron positions of a tile (based on the stride value available in register-stack), as shown in Figure 4.3d. However, for larger layers (large L, M) where the number of tiles is at least equals to N_{spc}, the *nZESPA* modules of a grid perform convolution simultaneously on multiple tiles, as shown in Figure 4.3c. This approach reduces the number of idle cycles of *nZESPA* modules.

4.3.2 Intra and Inter Vault Parallelism of *nZESPA*

The *nZESPA* dataflow provides twofold parallelism. The intra-vault parallelism is achieved within a single vault. In NMP-fully-sparse architecture, we placed a grid of 16 *nZESPA* modules in each vault of the HMC device. Each *nZESPA* module within a vault can perform the MAC operations on the corresponding tile (Figure 4.3c) independently in parallel with others. This concurrent processing of the tiles leads to intra-vault parallelism. The intra-vault parallelism is exploited on a single block of data (Figure 4.3b). Further, as shown in Figure 4.3d, the *nZESPA* modules can compute on different sections of data of a tile in parallel, leading to a reduced number of idle cycles for the smaller layers. HMC, having multiple vaults, provides a suitable environment to implement a highly parallel system. We integrate the *nZESPA* grids with all the vaults. These *nZESPA* grids parallelly process all the data blocks (Figure 4.3b), resulting in inter-vault parallelism. Note that our architectural setup includes 16-vault HMC configurations with 16 *nZESPA* modules (1 grid) in each vault to leverage intra and inter vault parallelism. Increasing the number of grids by increasing the number of vaults as in HMC 2.1 can increase the performance by additional inter-vault parallelism in exchange for additional power consumption by the system. Similarly, increasing the number of *nZESPA* modules in a single grid can increase the intra-vault parallelism while also incurring additional overhead on area and power, leading to design trade-offs for the system designers.

Due to the sliding window nature of the convolutions along with the horizontal and vertical partitioning scheme of *nZESPA* dataflow, there will be some missing input activations at the edges of each block and tile (data blocks and tiles are shown in Figure 4.3), resulting in partial results at some neuron positions. To address the cross tile dependencies, we replicate additional input activations in the activation buffers of each *nZESPA* module. We also replicate additional activations in each vault for the edges of the data blocks. While the former solves the cross tile dependency problem for intra-vault parallelism, the latter solves the issue for inter-vault parallelism.

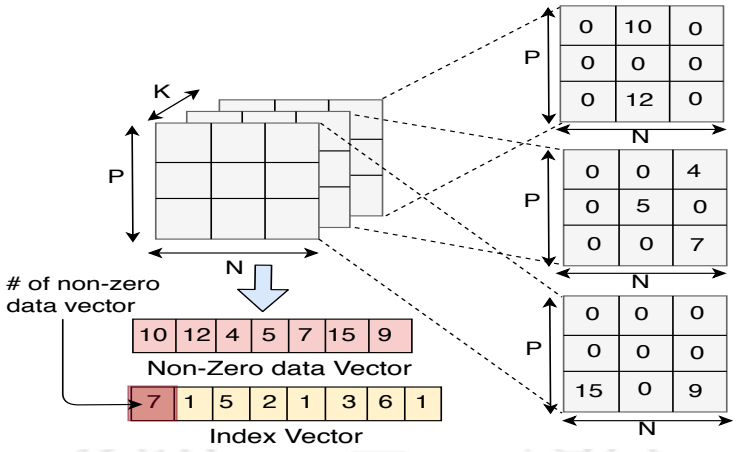


Figure 4.4: Illustration of Compression technique.

4.3.3 Data Compression

The sparsity of both the weights and activations is harnessed by the support of a compressed data format. Figure 4.4 represents the compression technique used in this *nZESPA* dataflow. N, P, and K have the same meaning, as shown in Figure 1.4. We adopted a variant of the previously proposed compression algorithm [106], [142]. The compressed format of data assists the *nZESPA* dataflow in eliminating the zero-valued computations associated with both the weights and activations. In Figure 4.4, we have shown the compression technique for a single weight matrix. The same technique is followed for the compression of all the weight and activation matrices. From Figure 4.4, it can be seen that the number of $K \times P \times N$ (where K=P=N=3) filter elements (weights) of a filter reduced to the number of non-zero elements (7 in this case) which are stored in non-zero data vector. The first entry (shown in red in Figure 4.4) of the index vector stores the number of non-zero elements of a filter/activation matrix, and it is followed by the number of zeros before each non-zero element. We considered 4bit/entry for the index vector, which allows up to 15 consecutive zeros between any two non-zero elements. If the two non-zero elements are further apart from each other, then a zero-value place holder can be inserted without any notable loss in the compression efficiency. As the dimensions of the filters and input activation layers are known, we can decrypt the compressed data with the help of the above two vectors.

The *nZESPA* controller is designed to perform the inference operations on this specific format of data. At each neuron position (as shown in Figure 1.4) for the output activations, the *nZESPA* controller only fetches non-zero input activations from the non-zero data vector. Upon receiving the non-zero input activations, the *nZESPA* controller checks their validity for the corresponding neuron position with the help of the index vector. For the valid non-zero activations, the controller also searches the corresponding non-zero weights.

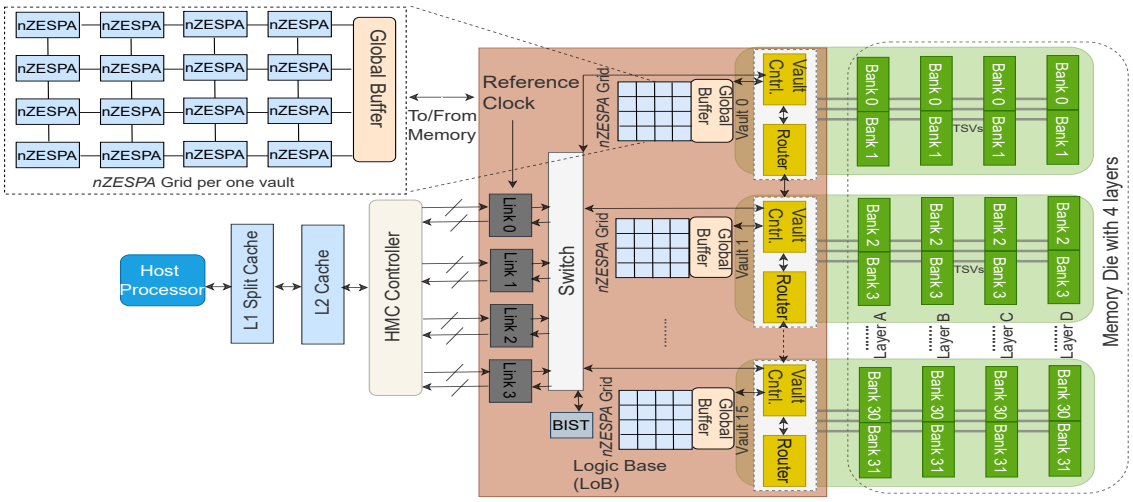


Figure 4.5: The proposed NMP-fully-sparse architecture.

Upon a successful search, the activations and their corresponding weights are multiplied and accumulated. Consequently, all zero-valued computations are skipped as only non-zero elements participate in the MAC operations.

4.4 System Architecture

The architecture, shown in Figure 4.5, is an abstract view of the proposed NMP-fully-sparse architecture. The system consists of a host processor and a two-level cache hierarchy (L1 split cache and shared L2). Instead of traditional DDRx memory, we choose Micron's HMC as the main memory for NMP. The reason is: 3D memories are a promising technology for near-memory integration of logic. We can easily integrate the custom accelerators in the logic layer (LoB) of the device without sacrificing the expensive memory area. The on-chip HMC controller manages all the global memory references. The HMC, considered in our setup, consists of four memory layers (A, B, C, and D), each having a size of 4-Gbit, which sum up to 2 GB of memory in total. The two banks from each layer constitute a vault that has a separate vault controller. The vault controllers work independently and handle all the memory references local to the corresponding vault. Vault controllers exchange the non-local data with the other vaults with the help of routers, connected through a 2D mesh network-on-chip (NoC). The detailed NoC circuitry is omitted in Figure 4.5 for simplicity. The LoB layer of the original HMC device primarily consists of vault controllers, a crossbar network to interconnect the vaults, the off-chip link SerDes, and other testing circuitry [40]. To integrate the grid of *nZESPA*, we replaced the crossbar network with the 2D mesh network-on-chip

(NoC), similar to the previous studies [3], [16].

16 *nZESPA* modules are connected through NoC to form a grid, as shown in the top-left part of Figure 4.5. Each grid is connected to the individual vault controllers. The logic die of the HMC device that we have considered permits approximately 50-60 mm^2 in total or 3.5 mm^2 per vault of additional space for near-memory logic apart from the existing circuitry [40]. This clearly manifests that there is a strict area budget for a larger array of logic. For our experiment, we placed a grid of 16 *nZESPA* modules in each vault as it easily fits within the available space of 3.5 mm^2 in each vault. On UMC 90 nm technology, the entire grid of 16 such *nZESPA* modules incurs an area overhead of 2.56 mm^2, which is lesser than the area budget of 3.5 mm^2. On the other hand, it is difficult to integrate larger logic arrays in the logic layer of 3D memory. For example, state-of-the-art like [120] incurs an area overhead of 16 mm^2 on TSMC 65 nm technology for larger processing elements (PEs) array, thus making such designs suitable for on-chip rather than near-memory integration.

In addition to the traditional cache hierarchy, the proposed NMP-fully-sparse architecture includes local buffers (weight and activation buffer) in the *nZESPA* modules, a shared global buffer for the *nZESPA* grid, and DRAM banks. These buffers are incorporated since accessing the data from the DRAM memory die is costlier than the rest of the two. The NMP-fully-sparse architecture discussed above delivers threefold benefits: (1) the concurrent

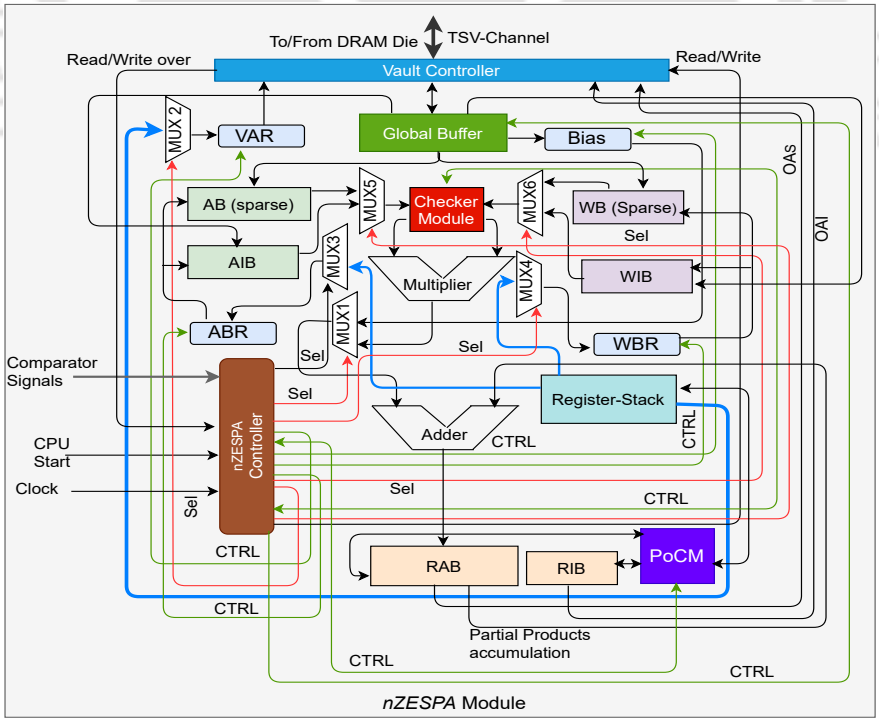


Figure 4.6: The abstract view of *nZESPA* architecture.

processing of the inference operations through intra and inter vault parallelism, (2) exploitation of both static and dynamic sparsity through *nZESPA* dataflow on the compressed data, and (3) faster memory accesses due to the NMP capability. Towards achieving these features, we have proposed the *nZESPA* dataflow and designed the *nZESPA* hardware that resembles the *nZESPA* dataflow. The proposed *nZESPA* module is synthesized in hardware using the Cadence tool-set (described in Section 4.6.2.1).

4.5 nZESPA Architecture and Operations

The complete microarchitecture of *nZESPA*, being complex and difficult to represent by a diagram, we have presented an abstract overview of the designed unit in Figure 4.6.

- The *nZESPA* controller, being the master of the entire circuitry, drives the whole module. The *nZESPA* controller fetches all the required data of a tile (shown in Figure 4.3) from the DRAM die through the TSVs with the help of the vault controller. The global buffer within the individual vaults is shared among the *nZESPA* modules of a grid belonging to that vault. The data are initially brought into the global buffer and then transferred to the respective buffers with the help of the *nZESPA* controller. The vault address register (VAR) holds the address of the memory that needs to be fetched. The host processor initiates the execution by sending a start signal to the *nZESPA* controllers. Additional information like dimensions and number of channels of weight/activation matrices, initial addresses of the metadata (like activations, weights, strides, and bias) are offloaded to the register-stacks by the host processor. The bias register holds the bias value, which is added only once at each neuron position. The *nZESPA* modules can handle any stride value and thereby execute CNNs of diverse shapes and sizes.

- The hardware includes separate buffers for storing non-zero data vectors and index vectors of the compressed-sparse format, shown in Section 4.3.3. The activation data and indices are stored in the activation buffer (AB) and activation index buffer (AIB), respectively. The weight buffer (WB) and weight index buffer (WIB) store the non-zero weights and their indices, respectively. The activation buffer register (ABR) and weight buffer register (WBR) are used to access the respective buffers since these registers hold the addresses for the corresponding buffer.

- Each unit includes the 16-bit multiplier and adder to perform the MAC operations. The checker module plays a crucial role in achieving sparsity-aware inference operations. It searches the AB efficiently for the valid input activations at each neuron

Table 4.1: Configuration parameters.

nZESPA Parameters	Value	NMP-fully-sparse Parameters	Value
		# *nZESPA*s / grid	16
Multiplier width	16 bits	# grids / vault	1
AB size	32 KB	# grids in LoB	16
AIB size	8 KB	# Multiplier	256
WB size	256 B	Total AB data	512 KB
WIB size	64 B	Total AIB data	128 KB
RAB size	32 KB	Total WB data	4 KB
RIB size	8 KB	Total WIB data	1 KB
		Total RAB data	512 KB
		Total RIB data	128 KB

position. As the AB only contains non-zero activations, the zero-valued activations are never fetched. It also finds the corresponding weights in the WB. Upon a successful search, the MAC operation is triggered. Note that WB also contains non-zero weights. Consequently, zero-valued weights are never fetched, and hence it possesses the ability to exploit both static and dynamic sparsity (fully-sparse). The output activations are accumulated in the result accumulation buffer (RAB).

- After the generation of the output activation channels for all the filters, the post convolution module (PoCM) applies the non-linear activation function (ReLU) on the output activation channels. On completion of ReLU, PoCM performs the pooling operations and encodes the results in the specific compressed-sparse format for the processing of the next activation layer. PoCM updates the RAB with only non-zero output activation vectors and writes the corresponding index vector in the result index buffer (RIB). Finally, the output activations (OAs) from RAB and output activation indices (OAI) from RIB are updated in the DRAM die through the vault controller and TSVs.

Note that the multiple inputs of a MUX (e.g., MUX2-MUX6) are replaced by one thick arrow (blue lines) for simplicity. Similarly, the multiple control signals between any two modules within the *nZESPA* are replaced by a single CTRL signal (green lines). The select lines of the multiplexers are shown in red lines. The configuration parameters of *nZESPA* and the NMP-fully-sparse architecture are presented in Table 4.1. The *nZESPA* design, mentioned above, preserves the *nZESPA* dataflow at every state of the state machine to leverage the sparsity close to the memory.

4.6 Experimental Methodology

We evaluate all four architectures by executing widely-used state-of-the-art benchmarks from industries and academia. The selected benchmarks stand in the top in terms of reduced *'top-*

Table 4.2: Benchmark Details.

ConvNets/Networks	AlexNet	VGG-16	ResNet-34
Year	2012	2014	2015
Place	1st	2nd	1st
Top-5 error	15.3%	7.5%	5.7%
# of CONV layer	5	13	33
# of parameters	61 M	138 M	21 M
Parameter Size	116.54 MB	263.90 MB	41.58 MB
Total Size	121.015 MB	373.58 MB	90 MB

5 *error rate'* in the ImageNet Large Scale Visual Recognition Challenge (ILSVRC) [130]. Table 4.2 lists the attributes of all the three **benchmarks**: AlexNet [116], VGG-16 [118], and ResNet-34 [48]. The *top-5 error* of the networks has reduced substantially over time. The number of CONV layers has also increased, which makes the networks deeper and accurate. We have shown the parameter size and the total size of the model separately in Table 4.2. The specification summary that includes shapes and sizes of the networks is extracted from the PyTorch framework [133].

4.6.1 Particulars of the Architectures

We compare four different architectures of a system: ❶ traditional-dense, ❷ traditional-fully-sparse, ❸ NMP-dense, and ❹ NMP-fully-sparse (proposed). The detailed specifications of these architectures are listed in Table 4.3. All the architectures include a similar quad-core host processor with a similar cache hierarchy, as shown in Table 4.3. HMC is used as the main memory module for all the architectures. The traditional systems have 16 grids of dense-accelerators, and 16 grids of *nZESPA* modules for the respective systems (❶ and ❷) integrated close to the on-chip main memory controller through a 2D mesh network-on-chip (similar place of integration as shown in [57]). Conversely, a grid of dense-accelerators is integrated with each vault of HMC in its LoB for NMP-dense architecture (❸). And the grid of *nZESPA* modules is integrated with each vault of HMC in the proposed NMP-fully-sparse architecture (❹). Consequently, compared to the NMP based architectures, all traditional architectures suffer from longer data access latency and higher per bit energy cost due to the high off-chip communication cost. The HMC parameters, capabilities, working frequency of dense-accelerators / *nZESPA* modules, and area overhead of the individual architecture are also mentioned in Table 4.3. Note that dense-accelerators are comparatively superior to the *nZESPA* modules in terms of area overhead and working frequency because sparse architectures require additional hardware and have a longer critical path for maintaining the sparse

Table 4.3: Specification of the architectures.

Host Processor (common for all architecture)	
Core	x86-64
Core Count	4
Core Frequency	2 GHz
Cache Memory (common for all architecture)	
L1 i-cache	32 KB, private, 4 way associative, 64B blocks
L1 d-cache	32 KB, private, 8 way associative, 64B blocks
L2 cache	256 KB, shared, 8 way associative, 64B blocks
HMC (common for all architecture)	
Timings	t_{RP} = 7.7ns, tt_{CCD} = 3.3ns, t_{RCD} = 10.2ns, t_{CL} = 9.9ns, t_{WR} = 15ns, t_{RAS} = 21.6ns
Energy [40]	3.7 pJ/bit (DRAM read), 6.78 pJ/bit (SerDes hop)
Power[132]	11.08W
Size	2 GB
Logic Die	90 nm, dimension $27 \times 27 mm^2$
traditional-dense / traditional-fully-sparse	
# of grids	16 (for both architectures)
# of logic units (dense-accelerator/*nZESPA*) per grid	16 (for both architectures)
# of multipliers	256 (for both architectures)
Area overhead per grid	1.96 mm^2 / 2.56 mm^2
Frequency of dense-accelerator / *nZESPA* module	∼1.2 GHz / ∼1 GHz on UMC 90 nm
Capabilities on data	Parallel processing on dense / compressed-sparse
Area of integration	Near the on-chip memory controller, similar to [57]
NMP-dense / NMP-fully-sparse (proposed)	
# of grids	16 (for both architectures)
# of logic units (dense-accelerator/*nZESPA*) per grid	16 (for both architectures)
# of multipliers	256 (for both architectures)
Area overhead per grid	1.96 mm^2 / 2.56 mm^2
Frequency of dense-accelerator / *nZESPA* module	∼1.2 GHz / ∼1 GHz on UMC 90 nm
Capabilities on data	Parallel + NMP on dense / compressed-sparse
Area of integration	LoB of the HMC (for both architectures)

encoding. However, the property of sparsity neutralizes the effect, as shown in Section 4.6.3. The dense-accelerator follows a similar data partitioning scheme, like the data partitioning scheme of the proposed *nZESPA* dataflow. However, unlike *nZESPA*, it works on dense data, and no zero-valued computations are skipped. All these architectures are primarily designed to optimize the convolution layers because state-of-the-art CNNs are primarily dominated by these compute-intensive layers [125], [123]. However, these architectures include dedicated logic for the non-linear operation (ReLU) and pooling.

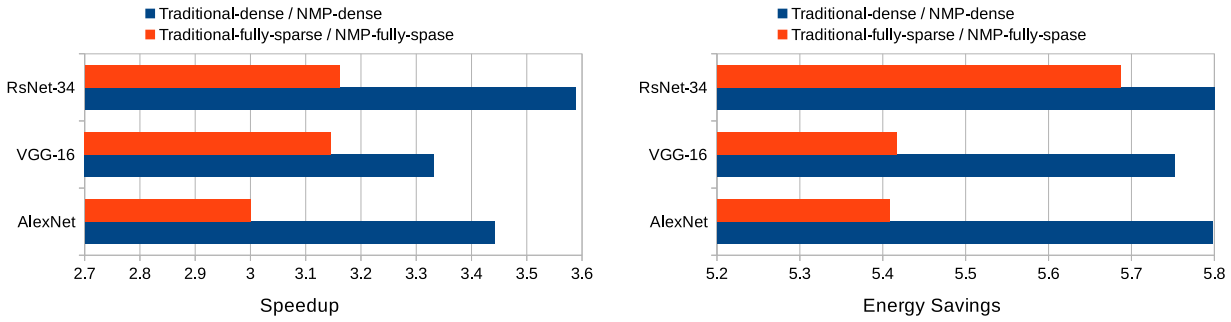
4.6.2 Measuring Performance and Power

4.6.2.1 Performance

We implement the designs of both the dense-accelerator and proposed *nZESPA* module in synthesizable Verilog hardware description language (HDL). We then perform placement-aware logic synthesis in Genus Synthesis Solution (version 15.21) from Cadence. We use UMC 90 *nm* technology library and obtain the operating frequency around 1.2 GHz and 1 GHz, respectively, for the dense-accelerator and proposed *nZESPA* module. We develop an in-house cycle level simulator that can be configured to resemble the state machine of the dense-accelerator and proposed *nZESPA* module. The simulator is parametrizable and can be fed with the frequency obtained from the hardware synthesis of the respective hardware. The execution time is measured from the simulator by layer-wise execution of the respective ConvNets. Note that the reported execution time by the simulator includes the data processing time in the grid of accelerators (dense/*nZESPA*) as well as the time to fetch/store results in the memory through the memory hierarchy of the respective architectures. Here, additional latency for off-chip communications in traditional systems is also taken into account. The timing parameters to fetch/store data in HMC are shown in Table 4.3.

4.6.2.2 Power Analysis

The total power consumption of the dense-accelerator and *nZESPA* module is also obtained from the Genus Synthesis Solution (version 15.21) from Cadence after the logic synthesis. The power and energy parameters of the HMC device are obtained from the existing state-of-the-art (shown in Table 4.3). We use McPAT [134] to measure the on-chip power values. While measuring the energy consumptions of each architecture during the execution of the ConvNets, we consider the energy consumed by the logic units (dense/*nZESPA*) as well as the energy required to fetch/store data through the memory hierarchy of the respective systems. Note that the area analysis for the dense-accelerator has been done in a similar fashion, as explained in Section 4.6.3.5.



(a) Speedup of dense and sparse systems due to NMP.

(b) Energy savings on both dense and sparse systems due to NMP.

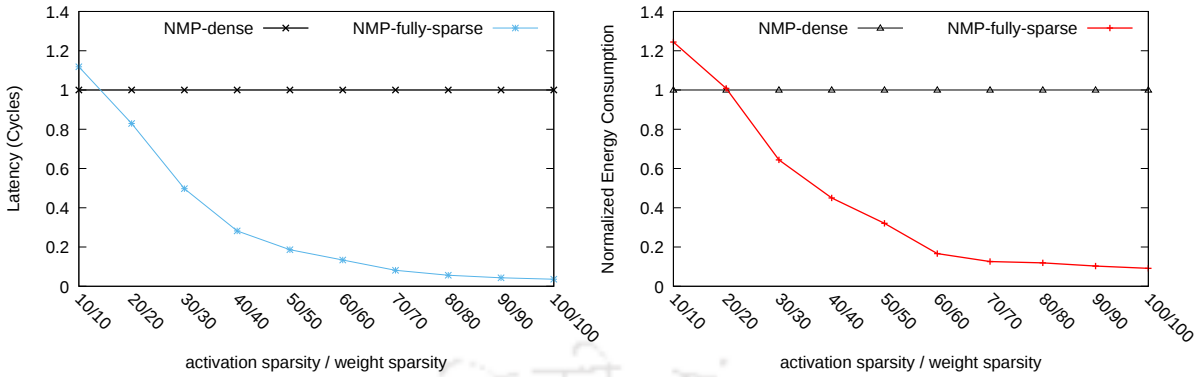
Figure 4.7: Effectiveness towards NMP capability.

4.6.3 Evaluation

We have evaluated the proposed system (NMP-fully-sparse) based on its performance, energy consumption, and area overhead. The proposed system substantially outperforms all the baselines while being energy efficient. In this section, we first evaluate the effectiveness of NMP capability and sparsity for a system. Then we measure the system's throughputs for all the architectures.

4.6.3.1 Effectiveness of NMP Capability

Figure 4.7 manifests the effectiveness of the NMP approach. The NMP capability benefits both the dense and sparse systems. In Figure 4.7a, we have shown the speedup achieved by the NMP-dense and NMP-fully-sparse over the traditional-dense and traditional-fully-sparse system, respectively, for all the networks. This speedup is obtained as the traditional systems (both dense and sparse) have longer off-chip memory access latency than the NMP-based systems. Note that, the speedup (traditional-dense / NMP-dense) achieved in the NMP-dense system due to NMP capability is higher than the speedup (traditional-fully-sparse / NMP-fully-sparse) obtained in the NMP-fully-sparse system. The reason is; the number of data blocks that need to be accessed by the dense systems is higher than the sparse systems. Consequently, the amount of off-chip latencies, saved in the NMP-dense system, is higher than the NMP-fully-sparse system. In Figure 4.7b, a similar effect can be seen in the energy savings due to the NMP capability. Both the NMP-dense and NMP-fully-sparse have achieved significant savings in the energy compared to their respective traditional systems. The NMP-dense architecture leverages more benefits from the NMP approach than the NMP-fully-sparse architecture due to its comparatively high data requirements.



(a) Performance. (b) Energy.

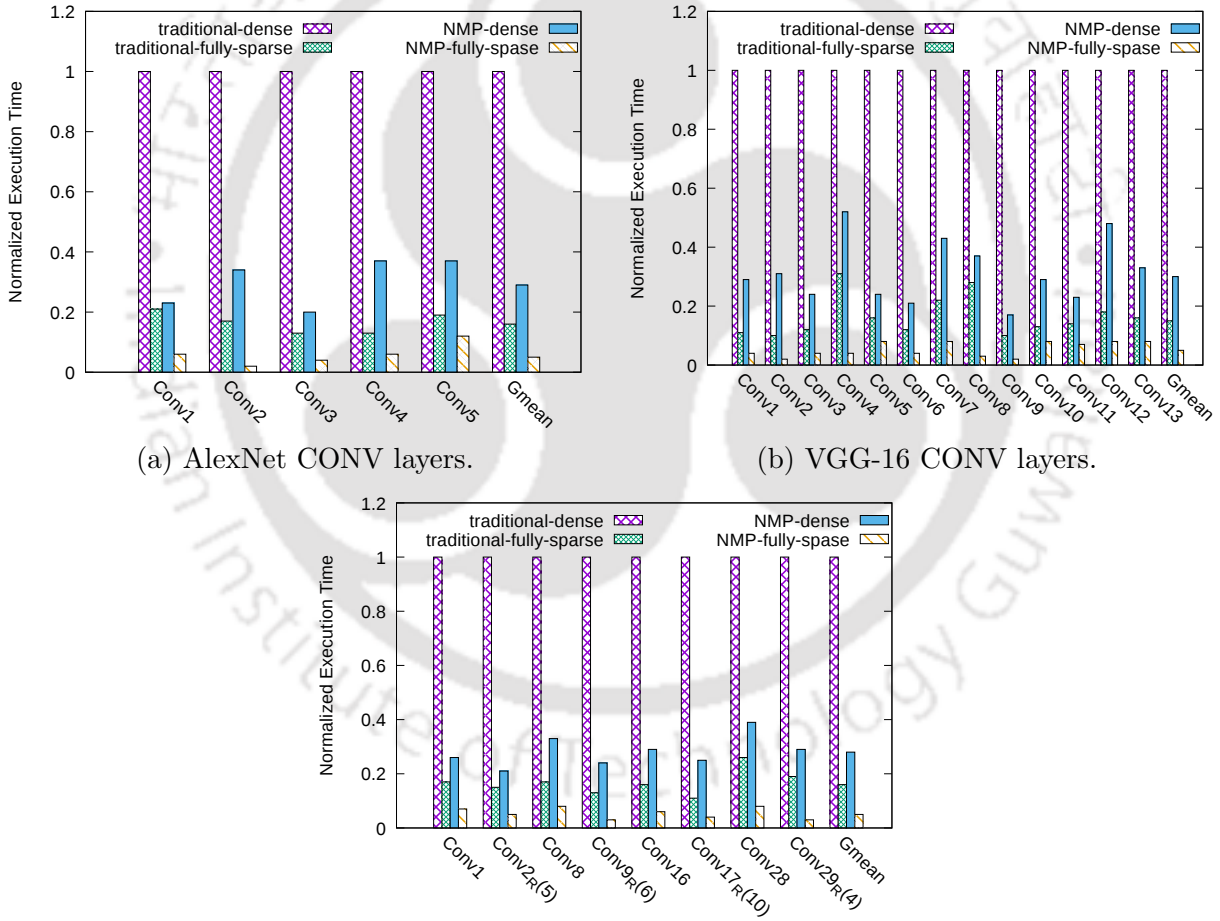
Figure 4.8: ResNet-34 performance and energy vs. sparsity.

4.6.3.2 Effectiveness of Sparsity

While executing the ResNet-34, we compare the performance and energy efficiency of the NMP-fully-sparse with NMP-dense architecture while varying the activation and weight sparsity. We extract the data from PyTorch [133] and artificially wipe the weights and activations (similar to the process of [57]) in ResNet's layers to vary the sparsity in the range of 10-100%. Along the X-axis of Figure 4.8, we plot activation/weight sparsity, which scales simultaneously. For example, the point 50/50 represents 50% weight and 50% activation sparsity. Figure 4.8a shows that at 10% (10/10 point) sparsity, NMP-fully-sparse achieves about 89% of the performance of NMP-dense architecture because of the complexity of maintaining the sparse encoding of the *nZESPA* dataflow. However, NMP-fully-sparse starts performing better than the NMP-dense with an increasing amount of sparsity. At point 20/20, NMP-fully-sparse obtains 1.2x better performance than NMP-dense, which reaches up to 27x at point 100/100. A similar trend is also observed in the case of energy consumption in Figure 4.8b. At point 10/10, NMP-fully-sparse achieves only 80% of the energy efficiency of NMP-dense architecture due to the additional hardware overhead to maintain sparse encoding. With the increasing amount of sparsity, the NMP-fully-sparse starts obtaining better energy efficiency, which reaches up to 10x at point 100/100. Certainly, increasing the sparsity in weights and activations affects the classification accuracy. For ResNet-34 at point 10/10, we observe that the accuracy is very close to the accuracy of the original dense network. Here the loss of accuracy is minimal (below 1%). At point 50/50, this loss reaches only up to 10%. However, it can reach to complete loss of accuracy at point 100/100. Note that point 100/100 is a hypothetical situation and doesn't occur typically in the practical scenario.

Table 4.4: Average speedup over respective baselines.

Networks	Speedup over baselines		
	traditional-dense	traditional-fully-sparse	NMP-dense
AlexNet	18.48x	3x	5.37x
VGG-16	20.30x	3.14x	6.09x
ResNet-34	19.44x	3.16x	5.41x



(a) AlexNet CONV layers.

(b) VGG-16 CONV layers.

(c) ResNet-34 CONV layers.

Figure 4.9: Performance analysis of ConvNets/Networks; normalized w.r.t. traditional-dense architecture.

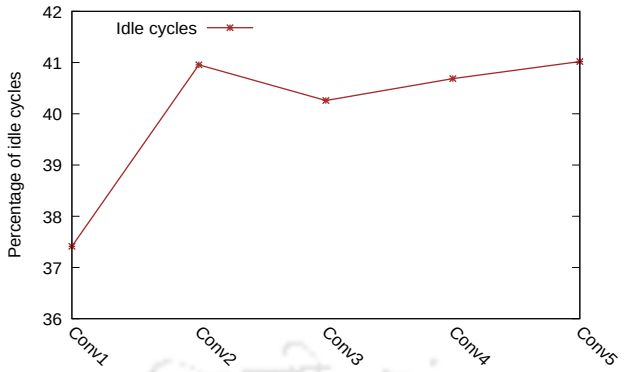
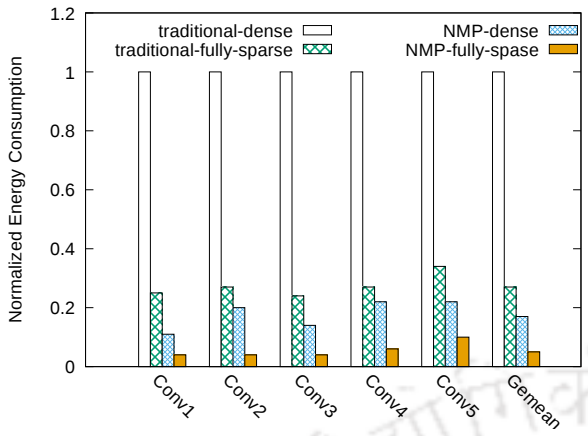


Figure 4.10: Percentage of Idle cycles for multipliers (AlexNet CONV layers).

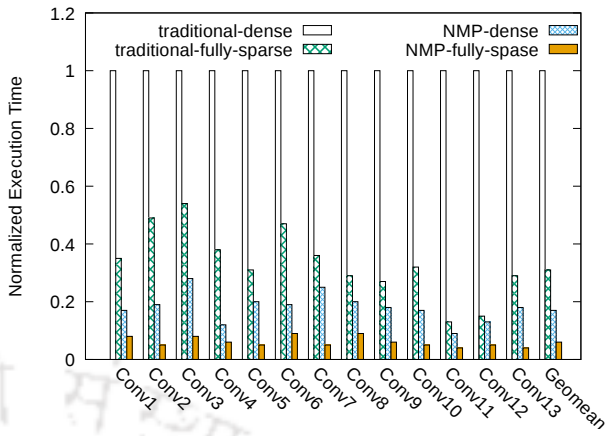
4.6.3.3 Performance Analysis

We compare the performance of the proposed NMP-fully-sparse architecture with all three baselines discussed in Section 4.6.1. In Figure 4.9, the normalized execution times w.r.t traditional-dense architecture are plotted along the Y-axis while all the CONV layers of the respective networks are shown on the X-axis. Separate layer-wise performance analysis for AlexNet, VGG-16, and ResNet-34 are presented in 4.9a, 4.9b, and 4.9c, respectively. The proposed NMP-fully-sparse system outperforms all the baselines in terms of execution time. To measure the performance, we consider both the execution time as well as data fetching/storing time through the memory hierarchy of the respective systems. Table 4.4 shows the average speedup achieved by the proposed system over the respective baselines. We have achieved the maximum speedup over traditional-dense architecture (see the first column of Table 4.4). The reasons are: 1) traditional-dense architecture works on dense data. Consequently, no zero-valued computations are skipped here, 2) it does not own the NMP capability and hence suffers from high off-chip latency. The gain over the traditional-fully-sparse architecture (the second column of Table 4.4) is comparatively less as it owns the similar property of exploiting sparsity for both the weights and activations. The gain over this architecture stems out only from the additional NMP capability of the proposed system. Compared to traditional-fully-sparse, the proposed system provides relatively higher speedup in the case of NMP-dense (the third column of Table 4.4) as it is dense, and no computations are eliminated in NMP-dense architecture. However, this NMP-dense architecture performs better than the traditional-dense since it possesses the NMP capability like the proposed system. Note that, ResNet-34, shown in Figure 4.9c, has 33 CONV layers. We have only shown the representative CONV layers. For example, the term, $\text{Conv2}_R(5)$, is the representative of the subsequent five layers.

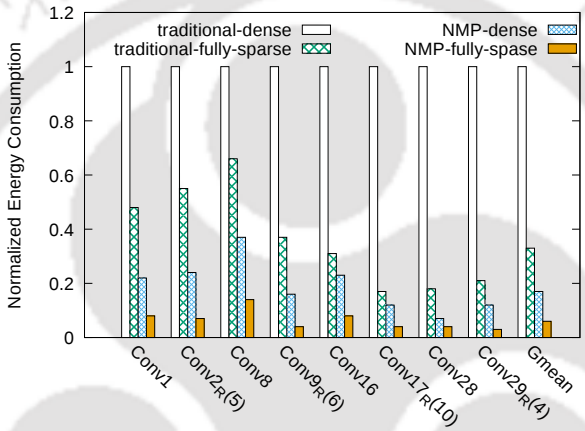
The idle cycles for the multipliers is an overhead for the sparse accelerators. Figure 4.10



(a) AlexNet CONV layers.



(b) VGG-16 CONV layers.



(c) ResNet-34 CONV layers.

Figure 4.11: Energy efficiency comparison of ConvNets/Networks; normalized w.r.t. traditional-dense architecture.

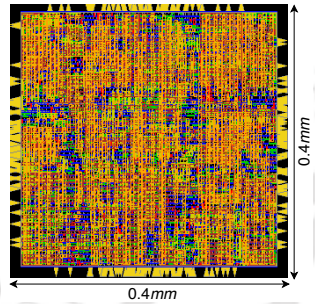
quantitatively shows the percentage of idle cycles for the CONV layers of the AlexNet. The Y-axis of Figure 4.10 shows the percentage of idle cycles for the multipliers of the *nZESPA* modules, while the X-axis corresponds to the CONV layers of the AlexNet. The idle cycles for the multipliers are primarily consumed by the checker modules and the controllers of the *nZESPA* modules before it sends the weight and activation pairs to the multipliers. Compared to the dense accelerators, this overhead is higher in the case of sparse accelerators because of the complexity of maintaining the sparse encoding.

4.6.3.4 Energy Efficiency

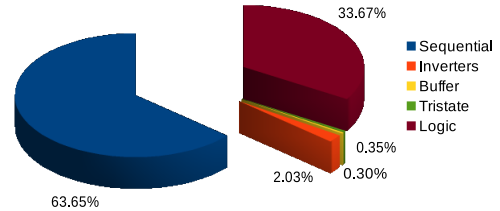
Figure 4.11 presents the comparison of the four accelerated architectures across the layers of the ConvNets. In Figure 4.11, the normalized energy consumptions w.r.t traditional-dense architecture are plotted along the Y-axis, and the CONV layers of the individual networks

Table 4.5: Average savings in energy over respective baselines.

Networks	Energy efficiency over baselines		
	traditional-dense	traditional-fully-sparse	NMP-dense
AlexNet	19.9x	5.4x	3.44x
VGG-16	17.35x	5.41x	3.01x
ResNet-34	17.49x	5.69x	3.02x



(a) *nZESPA*-Chip Layout.



(b) Area breakdown of the designed *nZESPA*.

Figure 4.12: Area analysis.

are shown on the X-axis. The proposed NMP-fully-sparse architecture harnesses substantial energy efficiency over all three baselines. We consider the total energy consumption of each system. The total energy consumption includes the energy consumption of each unit as well as the energy required to fetch/store data through the memory hierarchy of the respective systems. Table 4.5 illustrates the average improvement in energy consumption achieved by the proposed system over the respective baselines. The Maximum saving in energy is obtained over traditional-dense (shown in the first column of Table 4.5) architecture as it neither exploits the sparsity nor the NMP approach. The improvement in the energy efficiency over traditional-fully-sparse architecture is achieved because of the lack of NMP capability in it, while the energy efficiency over NMP-dense architecture is obtained as it does not exploit the sparsity (shown in the second and third column of Table 4.5). Note that, similar to Figure 4.9c, we show only the representative CONV layers in Figure 4.11c.

4.6.3.5 Area Analysis

Memory industries are area-sensitive. Consequently, area overhead plays a crucial role in the in-memory integration of logic. We implement the *nZESPA* hardware in Verilog HDL. We use the Innovus (from Cadence toolset) to obtain post-synthesis area estimates. Figure 4.12a depicts the layout of the designed *nZESPA*, where the standard cells are placed. Our *nZESPA* net-list gets placed on $0.4mm \times 0.4mm$ (shown in Figure 4.12a) square box at UMC

$90nm$ technology. This leads to an area overhead of $2.56mm^2$ for a single grid, resulting in a 5.61% of overhead for all 16 grids in total. Figure 4.12b shows the area breakups of a single *nZESPA* module. Note that the sequential elements, including AB, AIB, WB, WIB, RAB, and RIB of the *nZESPA* module (shown in Figure 4.6), consume the maximum area, which is around 63.65%. The second highest consumer of the area is the logic elements, which incur an overhead of approximately 33.67%.

4.6.3.6 Thermal Feasibility

The designed *nZESPA* modules and their specification provide us with ease from the thermal concerns. In the previous studies [3], it has been investigated that an HMC with 4-stacked DRAM die with a processing cluster clocked up to 5 GHz can be integrated into the logic die. This integration fits within the thermal limit of HMC [143] as this integration increases the temperature up to 76 $^{\circ}C$. Apart from that, thermal feasibility in the case of near-memory integration of 8.5 W processors is also manifested in [144]. The total power budget of *nZESPA* modules is 2.85 W, which is less than the power budget (3.41 W) of [3] in a similar technology node. This budget is also substantially less than that of 8.5 W. Similar to [15], [145], we can also safely conclude that our proposed architecture is thermally feasible as it has a lower power budget and frequency.

4.6.4 Comparison with Previous Accelerators

In this section, we compare the proposed NMP-fully-sparse architecture with the architectures that exploit the static (NMP-static-sparse) and dynamic sparsity (NMP-dynamic-sparse) in isolation. The existing state-of-the-art architectures like Cnvlutin (dynamic sparsity) [104] and Cambricon-X (static sparsity) [105] have leveraged the benefits of exploiting static and dynamic sparsity in isolation. Analogous to the principles of Cambricon-X [105],

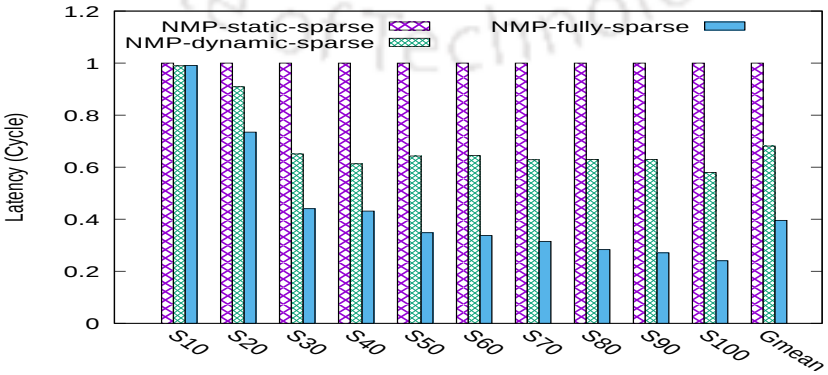


Figure 4.13: Performance Comparison.

we implement the NMP-static-sparse architecture, which utilizes the static sparsity property during the execution of CNN tasks near the HMC. Analogous to the principles of Cnvlutin [104], we also design NMP-dynamic-sparse that exploits the dynamic sparsity property during the execution of CNNs in the LoB of the HMC. Due to the substantial differences in the organization/buffering size, design and dataflow, and design choices like the use of eDRAM, the implemented architecture can not precisely replicate the state-of-the-art. However, we keep the spirit of NMP-static-sparse and NMP-dynamic-sparse the same as the Cambricon-X and Cnvlutin, respectively. A similar approach of comparison has also been performed in SCNN [57].

Figure 4.13 shows the comparison of the performance while executing ResNet-34 in the three different architectures. We plot the varying sparsity along the X-axis. For example, the point S10 represents 10% static sparsity in NMP-static-sparse, 10% dynamic sparsity in NMP-dynamic-sparse, and 10% sparsity in both the weights and activations for the proposed NMP-fully-sparse architecture. The proposed system performs well in all the points. On average, the NMP-fully-sparse achieves 2.5x and 1.7x better performance than the NMP-static-sparse and NMP-dynamic-sparse architecture, respectively. Note that this gain is initially less due to the minimum amount of sparsity, and the gain increases for the higher sparsity values. Analogous to the performance comparison, we observe a similar trend in the comparison of energy consumption by these systems. Note that our proposal will perform even better when static-sparse and dynamic-sparse systems are integrated into traditional (on-chip) setting instead of near-memory.

Interestingly, the benefits of the proposed NMP-fully-sparse architecture can also be seen in the following observation. There are differences between the architecture of SCNN [57] and the proposed one: (1) Though SCNN targets both static and dynamic sparsity like us, they implement on-chip accelerators while the proposed one is a near-memory solution. (2) SCNN uses a Cartesian product-based solution with an assumption of unit strides for convolution, when unit stride may not always be true for all the hidden layers of various ConvNets. This Cartesian-product strategy restricts SCNNs applicability only to CNNs with unit-stride convolutions. However, the proposed architecture relies on the dot-products and can be used to accelerate any CNNs with any strides. (3) SCNN keeps the input_activations in the processing elements (PEs) while it broadcasts the filter elements to all the PEs. As different input_activations inevitably have different sparsities, and because all input maps are multiplied by the same filter, the PEs with denser maps would lag behind those with sparser maps by the next broadcast of the filter. This approach increases the systematic load imbalance while providing filter reuse benefits. Unlike the filter broadcast, the proposed *nZESPA* dataflow keeps the filter in the local buffers (WB, WIB of Figure 4.6) of the *nZESPA*

modules and consequently survives from such a load imbalance issue in exchange for a loss of filter reuse. It can be observed that the proposed NMP-fully-sparse architecture achieves several times more performance gain over the traditional-dense architecture compared to SCNN. While SCNN delivers 2.37x and 3.52x performance again over the traditional-dense system for AlexNet and VGGNet, respectively, the proposed NMP-fully-sparse architecture obtains 18.48x and 20.30x performance gain over the traditional-dense system for the same networks. This additional gain is achieved due to the minimum memory access latency of near-memory processing as well as for the efficiency of *nZESPA* dataflow.

4.7 Summary

In this chapter[1], we propose hardware design of *nZESPA* and integrate it into a versatile architecture, NMP-fully-sparse, which possesses capabilities like exploiting parallelism, utilizing the sparsity of both weights as well as activations, and near-3D-memory processing. The cumulative outcome from all these features makes this architecture aggressive to outperform all other baselines while executing state-of-the-art benchmarks like AlexNet, VGG-16, and ResNet-34. On average, the proposed system obtains a maximum of up to 20.30x, 3.16x, and 6.09x improvement in the performance over the traditional-dense, traditional-fully-sparse, and NMP-dense architecture, respectively. Additionally, on average, the proposed system achieves a maximum of up to 19.9x, 5.69x, and 3.44x energy efficiency, compared to the traditional-dense, traditional-fully-sparse, and NMP-dense architecture, respectively. On UMC 90 *nm* technology, the combined area overhead for all the 16 grids of *nZESPA* modules is only 5.61%. The obtained results make the proposed NMP-fully-sparse architecture attractive for practical implementation. This chapter achieves the objectives like reducing data movement (objective1), obtaining high performance and energy efficiency through accelerated architecture (objective2), providing support for various applications (objective3), and delivering high throughput by applying optimization to reduce computations (objective4).

[1]The publication details related to this chapter are as follows.
(1) Das, Palash, and Hemangee K. Kapoor. "nZESPA: A near-3D-memory zero skipping parallel accelerator for CNNs." IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems 40.8 (2020): 1573-1585.

5

ALAMNI: Adaptive LookAside Memory based Near-memory Inference engine for eliminating Multiplications in Real-time

This chapter introduces another optimization that skips the redundant inference computations through a search-based technique. Multiple lookaside memories (LAMs) are used in the proposed system to store the repeating weight-activation $\langle W, A, M \rangle$ pairs along with their multiplication results (M). The costly multiplication operations are skipped for the pre-stored results once there are hits for the repeating $\langle W, A \rangle$ pairs in the LAM modules. We also include a bit-masking technique to increase the hit rates in the LAMs and further amortize the number of computations involved in an inference phase. We build custom hardware to implement the mentioned technique and integrate multiple such instances within the HMC memory to exploit intra/inter vault parallelism. The proposed system leverages the benefits of parallelism, sparsity, and NMP while outperforming the baseline and state-of-the-art in terms of performance and energy savings.

5.1 Introduction

In recent years, accelerated architectures have played a crucial role in enhancing the performance of CNNs inference while being energy efficient. Most of the accelerators aim to obtain a good trade-off between improving computation efficiency and alleviating memory constraints. The CNN accelerators often suffer from the limited off-chip memory bandwidth and on-chip capacity constraints, leading to *memory-wall* [110] problem. Several approaches viz. keeping weights and activations in on-chip eDRAM [45], leveraging sparsity of CNNs

[106], etc., have been explored to amortize the costly off-chip communications. Another viable solution to this problem is near-memory processing (NMP), where the computations are offloaded to the processing units, placed near to the memory. Deep neural networks (DNNs) have been accelerated in prior works using the NMP concept [3, 15, 16]. Apart from addressing the memory constraints, researchers have also exploited several optimization techniques to skip certain CNN computations with minimal loss of accuracy in order to enhance the CNN performance [57, 80, 81, 146, 147]. Thankfully, NNs and other machine learning algorithms are inherently tolerant to minor errors without notably affecting the intended classification accuracy [81, 146]. It is also evident from the fact that multimedia applications can produce outputs with small audio-visual artifacts without being noticeable to a human user. The error-tolerant phenomenon of NNs can be realized in the form of approximate computing. The challenge in approximate computing is to find the tradeoff between application accuracy and overall improvement in performance while being energy efficient. The system's performance and energy consumption can be improved by computational reuse through associative memories [148]. The redundant arithmetic operations can be skipped by computing once and storing the operands and results in the associative memories for future use.

In this work, we design custom hardware, ALAMNI, an Adaptive LookAside Memory based Near-memory Inference engine that focuses on two essential requirements of CNN inference: ❶ improving the performance and energy efficiency of the CNN inference by skipping the redundant computations, ❷ removing the memory constraints by avoiding the costly off-chip accesses through near-memory processing. The CNN algorithms are highly packed with the dot-products of two matrices (input feature maps and weight matrix) at each neuron position. The floating-point matrix-vector multiplications are computationally complex operations of any CNN algorithm. Consequently, bypassing these multiplications can bring substantial speedup and energy savings for any system while executing the CNN algorithm. We propose to store the most common inputs and outputs for each network in order to readily use the pre-computed outputs for future matching inputs. For faster search, we use content addressable memory to store the inputs and outputs and term it as lookaside memory (LAM). A triplet $\langle W, A, M \rangle$ of weight (W), activation (A), and their multiplication result (M) are cached into the LAMs to skip future redundant multiplications of the CNN inference. We integrate LAMs with our proposed ALAMNI units by placing them adjacent to the MAC modules. During MAC operations, if a matching $\langle W, A \rangle$ pair is found, then the product (M) available in LAM is used, and the multiplication operation is skipped. We observe that there is a considerable percentage of reuse of the precomputed results in CNNs. If we approximate the value of $\langle W, A \rangle$ pairs, we can further increase the

hit percentage in the LAM. The higher amount of approximation increases the hit rate on LAMs substantially and saves the recomputing overhead significantly in the ALAMNI units while executing the CNNs. However, high approximation leads to accuracy loss. We study (Section 5.4) the relation between them (approximation, accuracy loss) to decide the amount of approximation. In particular, we use bit-masking to derive the approximated values of the inputs. These approximated values are used for caching and searching in the LAM.

Our proposed ALAMNI approach updates the LAMs in real-time by collecting the most frequent $\langle W, A, M \rangle$ triplets. Consequently, the proposed policy does not depend on any data pre-profiling steps. During the initial layer, the ALAMNI policy provides a low hit rate in the LAMs due to compulsory misses. However, the proposed approach quickly escalates the hit rate primarily because of two reasons. ❶ The ALAMNI units are adaptive, and it starts keeping the most critical entries in the LAMs as it traverses through the hidden layers. ❷ We set the optimal size (64 entries) for each LAM, which provides the best hit-rate without much hampering the classification accuracy.

Apart from the above optimization of amortizing the number of multiplications, our proposed ALAMNI policy also leverages the benefits of near-memory processing (NMP) to avoid costly off-chip accesses. We integrate our ALAMNI units with the individual vaults (shown in Section 5.3.1) of a 3D-memory, specifically Hybrid Memory Cube (HMC) [40]. The parallel architecture and high bandwidth of the HMC help in exploiting the intra- and inter-vault parallelism while executing the inherently parallel CNN algorithms. The data and the NN models are partitioned across the vaults of the HMC (Section 5.3.2) for the parallel executions on the proposed ALAMNI units. In summary, the main contributions of this chapter are as follows.

1. We design dedicated hardware, ALAMNI, and integrate those units with the individual vaults in of the HMC to alleviate off-chip memory accesses using the NMP concept.

2. In ALAMNI units, we integrate lookaside memories or LAMs, which store the triplet $\langle W, A, M \rangle$. In the initial layer of CNNs, the LAMs face compulsory misses for the inputs of $\langle W, A \rangle$ pairs. However, it recovers soon due to its adaptive policy of keeping only the frequent occurring triplets.

3. The ALAMNI policy completely removes the data pre-profiling overheads and can provide good system throughput in real-time on unseen data.

4. This work studies the relation between the amount of bit-mask and the loss of accuracy for popular CNN workloads like VGG-16 [118], VGG-19 [118], and ResNet-34 [48] on

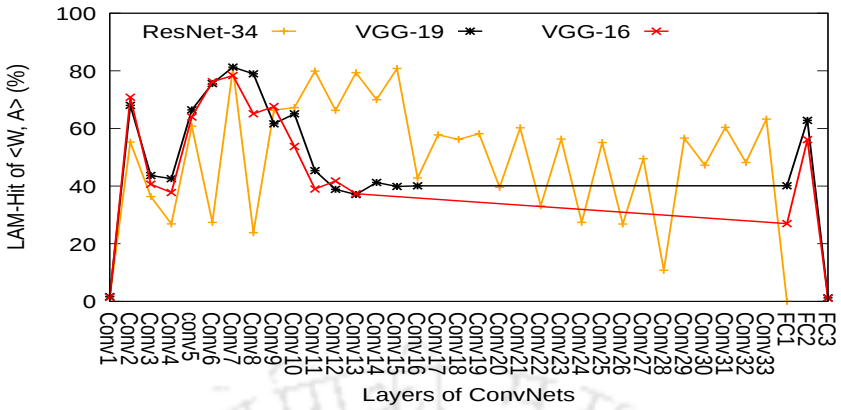


Figure 5.1: Redundancy across the layers of ConvNets.

CIFAR-10 dataset [149]. The amount of bitmasking is reconfigurable in our hardware and can be adjusted based on the desired accuracy.

5. We also employ data partitioning to leverage intra- and inter- vault parallelism for the concurrent execution of the CNN tasks in multiple ALAMNI units.

6. To quantify the ability of the proposed systems, we compare three architectures: ⓐ system having NMP capability without LAM (No_LAM_NMP) search, ⓑ the proposed ALAMNI without bitmasking, and ⓒ the proposed ALAMNI with 5-bit masking on $\langle W, A\rangle$ pairs (ALAMNI_Opt). We also compare our work with the popular NMP-based CNN accelerators [3, 15].

5.2 Motivation

The CNNs are highly packed with multiplications of weights (W) and activations (A). For example, AlexNet comprises around 85%-90% MAC (multiplication-and-accumulation) operations in the inference phase [150]. Reducing the number of costly multiplications without significantly hampering the classification accuracy can lead to better speedup and energy savings for any ConvNet. Among these multiplications, several of them are redundant since the CNN algorithms encounter decent temporal locality in the weight-activation, $\langle W, A\rangle$ pairs. We conduct experiments to assess the temporal locality of $\langle W, A\rangle$ pairs for state-of-the-art CNN benchmarks: ResNet-34, VGG-19, and VGG-16, on CIFAR-10 dataset [149]. Figure 5.1 depicts the redundancy of unmasked $\langle W, A\rangle$ pairs in the form of LAM hits across the layers of three ConvNets. The Y-axis represents the percentage of hits in LAMs for the $\langle W, A\rangle$ pairs, and the X-axis shows the layers of the ConvNets. On average, we observe around 43.45%, 35.86%, and 32.14% redundancy of the $\langle W, A\rangle$ pairs across the layers of

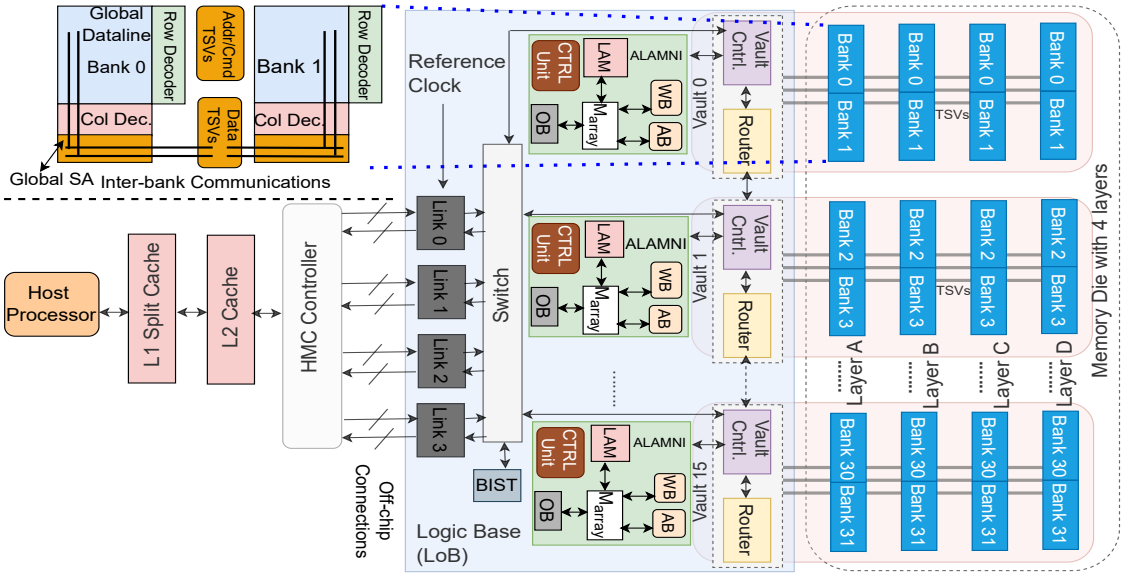


Figure 5.2: The Proposed NMP architecture.

ResNet-34, VGG-19, and VGG-16, respectively. If we use bit-masking to approximate the search, we can further increase the hit percentage in LAM. We use small 64-entry LAMs with a batch size of 4 images for this experiment. The detailed analysis in this context is shown in Section 5.4. This gives us insight for skipping multiplications by the effective LAM searches in the proposed ALAMNI units. Note that we store the ⟨W, A⟩ pair and its corresponding multiplication result (M) in the LAM. The lookaside memory (LAM) has been implemented in the form of content addressable memory to improve the performance and energy efficiency of parallel processors [151, 152]. The search operation in LAM is a single cycle operation that improves the proposed ALAMNI's throughput. Apart from optimizing the computations, the ALAMNI architecture also addresses the memory constraints of the CNNs through NMP processing [3, 15, 107]. The exploitable redundancy of $\langle W, A \rangle$ pairs are lucrative and appropriate for designing custom hardware for CNN.

5.3 System Architecture

5.3.1 The Proposed Near-Memory Architecture

Figure 5.2 shows an abstract view of the proposed ALAMNI architecture. The host side includes a host processor with a two-level cache hierarchy (L1 split and L2 shared). We choose Micron's HMC as the main memory for near-memory integration of the proposed

ALAMNI units. The benefits of using HMC instead of DDRx is multidimensional: (1) The logic layer (LoB) provides additional space for ALAMNI logic integration without sacrificing the memory area, (2) TSVs provide an order of magnitude higher bandwidth (160 to 250 GBps) than the conventional DDRx memory [40]. We integrate a 4-layer (A, B, C, and D), 16-vault configuration HMC device. The two banks (Bank 0 and 1) from each layer form a vault, shown in Figure 5.2. Each vault is equipped with a separate vault controller that can work independently. The vault controllers can communicate through the router connected in a 2D mesh network-on-chip (NoC). The ALAMNI units, shown in green boxes in Figure 5.2, are integrated with the individual vault controllers of the HMC. We incorporate 16 ALAMNI units (1 per vault) to leverage the inter-vault parallelism. The unmodified LoB layer primarily contains vault controllers, a crossbar network to interconnect the vaults, the off-chip link SerDes, and other testing circuitry [40]. We replace the crossbar network with the 2D mesh network-on-chip (NoC) to ingrate the ALAMNI units, similar to the previous studies in [3, 16]. The top left of Figure 5.2 shows the inter-bank communication through the global sense amplifiers (Global SAs) and channel TSV data bus. The detailed circuitry of the proposed system is omitted for simplicity. One challenge in near-memory integration of additional logic is the strict area budget in the LoB of the HMC device. The HMC, we select, permits approximately 50-60 mm^2 in total or 3.5 mm^2 per vault of additional space for near-memory logic apart from the existing circuitry [40]. On UMC 90 nm, we find that one ALAMNI unit accounts for 1.88 mm^2 area, which is lesser than per vault area budget of HMC.

5.3.2 Dataflow

The proposed NMP architecture exploits both the intra- and inter-vault parallelism to maximize the system's throughput. The data distribution and its granularity of execution are represented in Figure 5.3. The upper portion of Figure 5.3 explains the inter-vault parallelism. Each input channel of a hidden layer is sent to one ALAMNI unit for the convolution (CONV)/pool operations at each neuron position. All ALAMNI units may not get a channel to be processed for the initial layer because of the lesser input channels in the initial layer of CNNs. However, for remaining all hidden layers, a batch of 16 channels can be processed concurrently by 16 ALAMNIs, one at each vault. Note that our architectural setup uses the 16-vault configuration of the HMC. Increasing the number of ALAMNIs by increasing the number of vaults as in HMC 2.1 can further increase the system's performance due to the additional inter-vault parallelism. However, it is a design choice as additional parallelism by integrating more processing units requires extra power.

The lower portion of Figure 5.3 shows the intra-vault parallelism within one ALAMNI unit. During execution, an operation is done either by the MAC or the result is picked up from the LAM (in the case of hit). We term this combination as a LAM-MAC unit. Each ALAMNI unit has 32 LAM-MAC units, LAM-MAC$_0$, LAM-MAC$_1$, ..., LAM-MAC$_{31}$, as shown in Figure 5.3. These 32 LAM-MAC units perform the CONV operations at 32 neuron positions of a channel in parallel, leading to intra-vault parallelism. The ALAMNI controller assigns the work to each LAM-MAC$_i$ unit based on the available neuron positions to be processed. Each LAM-MAC$_i$ unit has a small stream buffer that holds the $\langle W, A \rangle$ pairs to be processed. The ALAMNI controller always keeps these buffers operation-ready by filling them with the required $\langle W, A \rangle$ pairs to be processed by the corresponding LAM-MAC$_i$ units. Hence, the buffer filling time is overlapped with the MAC operations. During the filling of streaming buffers with the $\langle W, A \rangle$ pairs, the ALAMNI controller reduces the number of fetch operations by fetching the data once and broadcasting them to all the appropriate streaming buffers as shown in Figure 5.3. The combination of intra-vault parallelism, inter-vault parallelism, and the reduction of multiplications by effective use of LAM makes ALAMNI highly efficient in terms of the system's throughput.

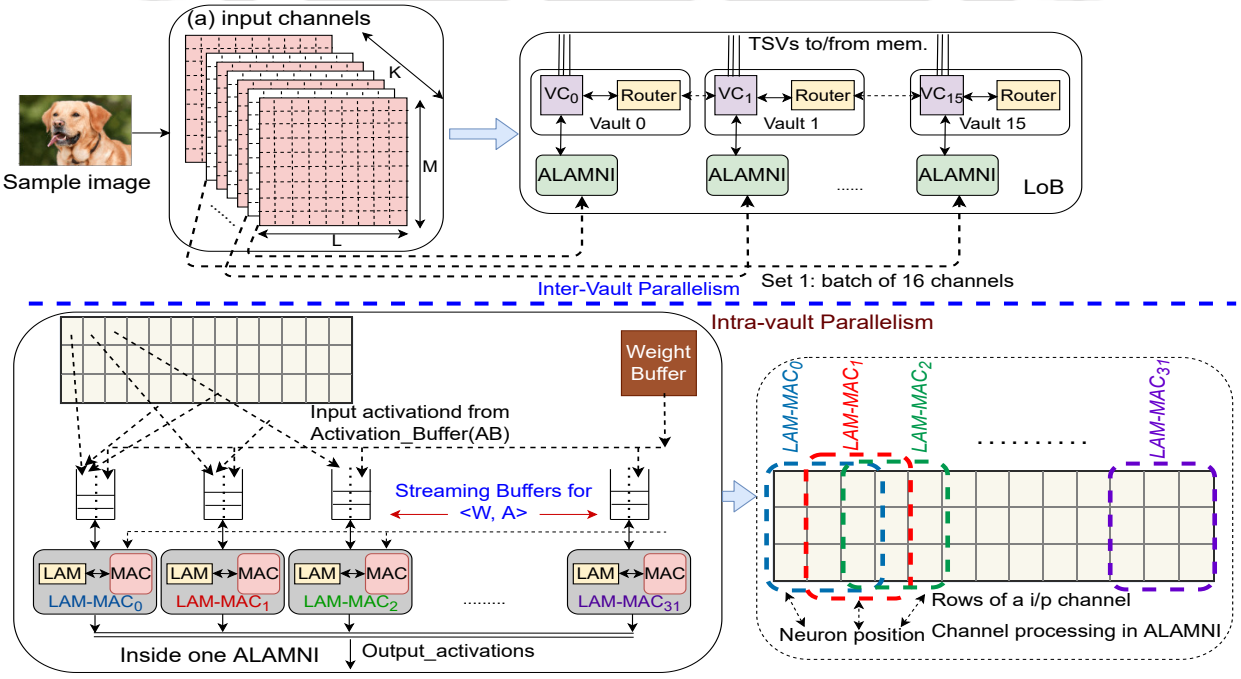


Figure 5.3: Dataflow to leverage parallelism.

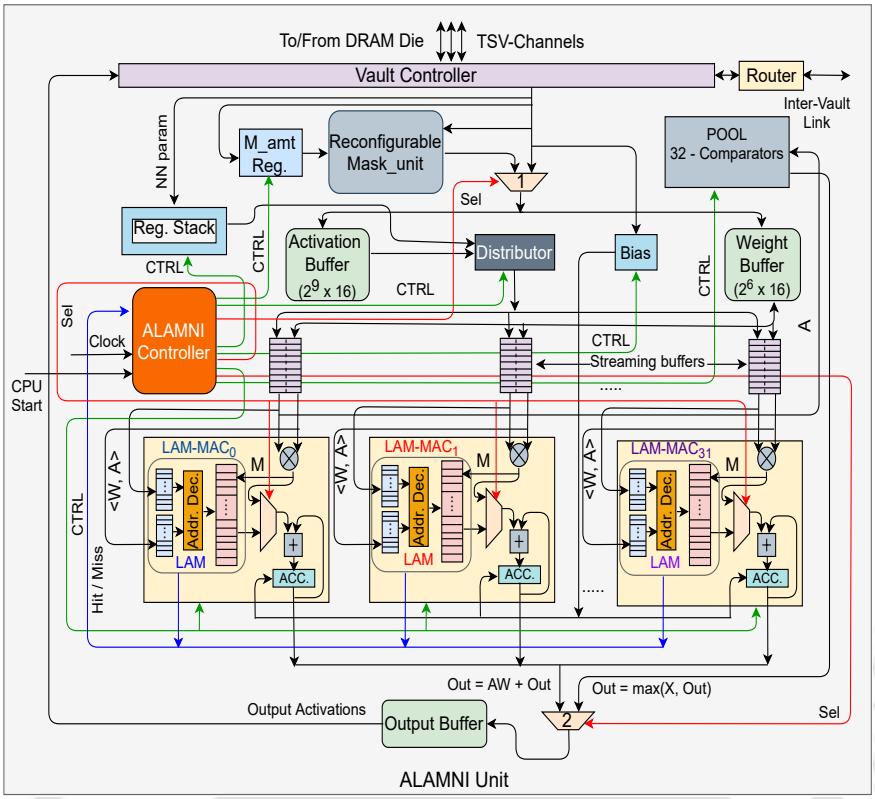


Figure 5.4: The Proposed ALAMNI unit.

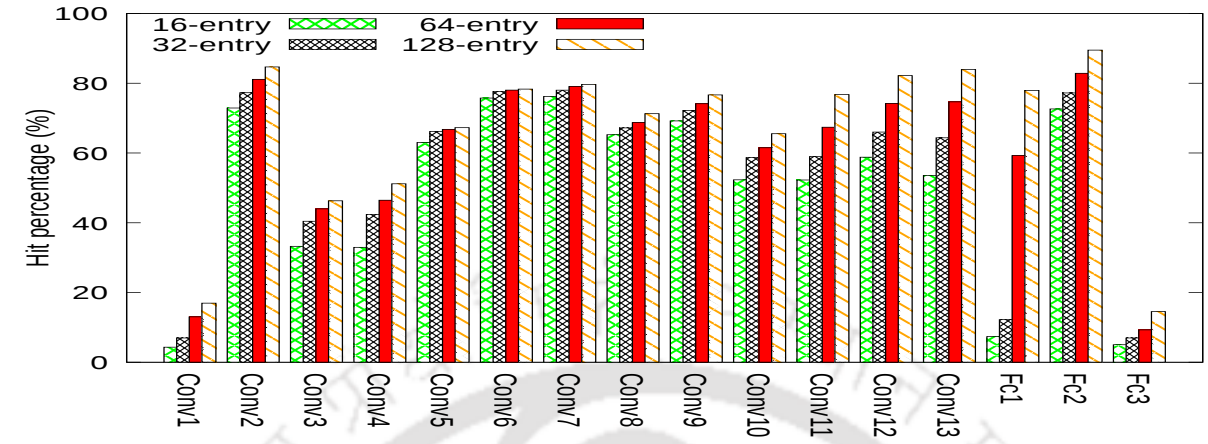
5.3.3 The ALAMNI Unit

As the microarchitecture of the ALAMNI units is complex and difficult to represent by the diagram, we show an abstract view of the designed hardware consisting of the primary modules in Figure 5.4. The primary components of the ALAMNI units are the ALAMNI controller, activation buffer (AB), weight buffer (WB), an output buffer (OB), 32 small streaming buffers, 32 LAM-MAC$_i$ units, register stack, bias register, reconfigurable mask unit, and a pool unit with 32 comparators. Apart from these units, there is also a result accumulation unit with an integrated linear function, which is not shown in the figure for simplicity. Note that we choose multiple LAMs instead of one shared LAM to facilitate the parallel searching of $\langle W, A \rangle$ pairs. The ALAMNI controller, being the master of the entire circuit, drives all the modules to accomplish the proposed policy. The host processor initiates the inference by sending a special start signal to the ALAMNI controllers. The meta-data information like hyperparameters of the ConvNets is also loaded into the register stack by the host side. Each channel of input maps and kernels is loaded into the AB and WB respectively by the controller. The ALAMNI units have been configured to operate on FP16. However, our design can be easily adapted for other precisions. The ALAMNI provides a

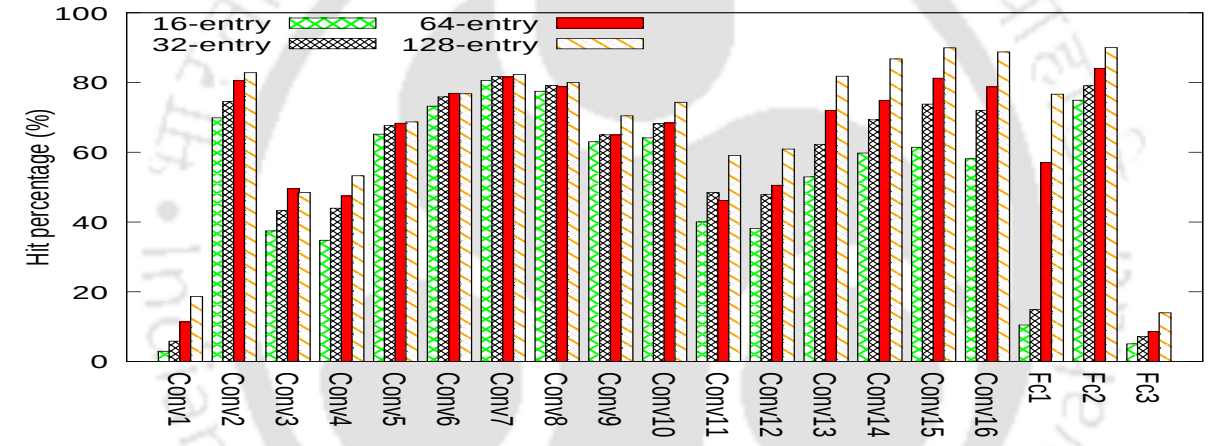
special feature of masking the inputs and weight parameters. The use of the masking feature is optional and entirely depends on users and the applications. The user can take the benefits of bit-masking, whenever accuracy can be sacrificed to some extent. In the applications where the prediction accuracy is highly critical, the bit-masking can be completely skipped while still getting decent benefits from the ALAMNI units. In bit-masking, the desired amount of the least significant bits (LSB) can be made to zero to increase the temporal locality of data by approximation. The reconfigurable mask unit, shown in Figure 5.4, performs a parallel bitwise AND operation with the bit-mask vector stored in the M_amt register. For example, a bit-mask vector $\langle 1111111111100000 \rangle$ can be used to achieve 5 bit-masking of data as the 5 LSBs of the vector are zero. The user of the ALAMNI can also adjust the amount of bit-masking by simply changing the content of the M_amt register. The AB and WB can be loaded either with the masked or unmasked data by sending a proper select signal to MUX-1. The masking latency can also be overlapped with the data loading cycles required for loading activation and weight buffer. Masked values increase the hit rate in LAM of the LAM-MAC$_i$ modules and help in skipping more multiplications compared to unmasked data. The masking also leads to an accuracy loss. A detailed study on the approximation-loss behavior is presented in the subsequent section. The ALAMNI controller continuously loads the streaming buffers by the $\langle W, A \rangle$ pairs with the help of the distributor module. The bias register holds the bias value that is added only once to each neuron position.

The LAM-MAC$_i$ units are another important component of the ALAMNI. The LAM consists of two 64-entry content addressable memories (CAM) and a result buffer (64-entry). The first CAM caches the activations (A), and the second CAM caches the weights (W) from the most frequent $\langle W, A \rangle$ pairs. The associated result buffer stores the multiplication result (M) for the given $\langle W, A \rangle$ pair that hits in the CAMs. In the LAM-MAC$_i$ unit, the $\langle W, A \rangle$ pair is parallelly sent to CAMs and the multiplier. Upon a hit in the CAMs, the multiplication result from the result buffer is immediately sent to the adder for accumulation without waiting for the output from the multiplier. The multiplication operation is then aborted by the ALAMNI controller. Upon a miss on the $\langle W, A \rangle$ pair, the multiplication result from the multiplier is sent to the adder as well as updated in the LAM using the least recently used (LRU) policy. This runtime collection of $\langle W, A \rangle$ pairs aids in skipping more multiplications in the future as the LAM adapts itself with the best data values for the given ConvNets. The ALAMNI controller updates the LAM in parallel to the addition operation, leading to no additional latency. We have shown the green lines and red lines to represent the control signals (CTRL) to different modules and the select signals (Sel) of the MUXes, respectively. MUX-2 is used to select the results between the CONV layer and pooling layer. After the channel accumulation and linearization, the final results are sent to the memory die using

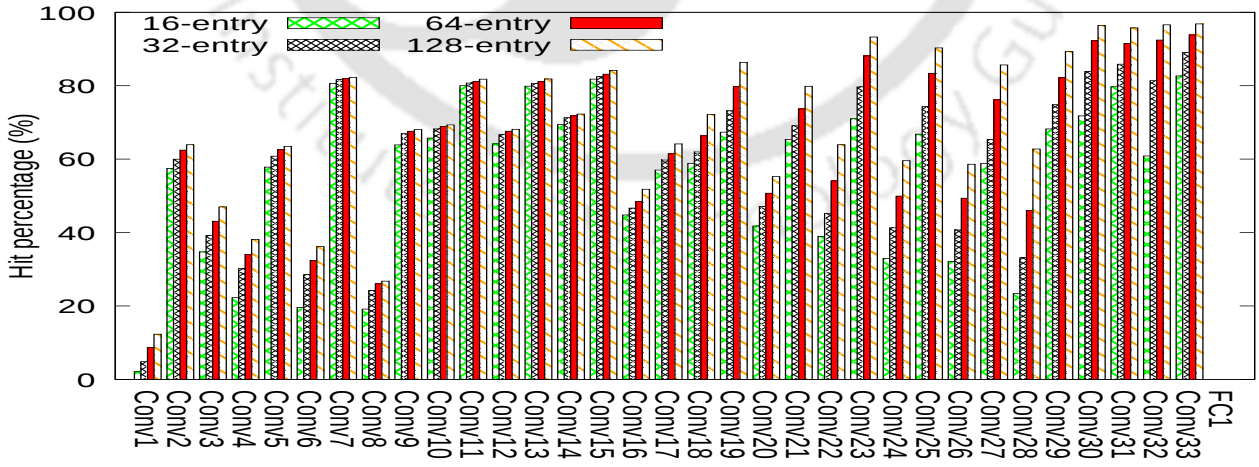
the vault controller and the TSVs.



(a) VGG-16



(b) VGG-19



(c) ResNet-34

Figure 5.5: Comparative study of Hit percentage vs. LAM size.

Table 5.1: Percentage loss in accuracy compared to original (unmasked) data for different values of bit masking.

ConvNets	3-bit	4-bit	5-bit	6-bit	7-bit
VGG-16	1.93	2.46	4.13	8.4	22.8
VGG-19	0.4	1.26	2.13	6.4	19.53
ResNet-34	0.46	1	2.4	6.2	17.8
Avg.	0.71	1.46	2.76	6.93	19.94

5.4 The ALAMNI Policy Validation

To validate the ALAMNI policy, we perform a case study of three popular state-of-the-art networks, VGG-16, VGG-19, and ResNet-34, on the CIFAR-10 dataset. The study focuses on the three major aspects: (1) Deciding the amount of allowable bit-mask for inference, (2) Finding the optimal number of LAM entries, and (3) Obtaining the percentage of LAM hits at different amounts of bit-masking on the optimal LAM size.

We conduct experiments in PyTorch framework [133] to address all three aspects. We integrate the idea and design of ALAMNI with the PyTorch framework to test the efficacy of the proposed idea. For all the above three networks, we use models that are pre-trained on the CIFAR-10 dataset. To find the maximum allowable bit-mask, we perform multiple inference runs with different amounts of bit-masking on $\langle W, A \rangle$ pairs with the different LAM sizes on the whole test-set. The test-set includes 10000 sample images (32×32) from 10 classes. We decide the maximum allowable bit-masking by observing the accuracy loss between the inferences runs with unmasked and masked $\langle W, A \rangle$ pairs having the same hyperparameters. Table 5.1 lists the accuracy loss compared to its related unmasked inference run for all three networks. On average, the accuracy loss is less than 1 % for 3 bit-masking, which reaches around 20 % for 7 bit-masking. We find the bit-masking up to 5-bits to be optimal as accuracy loss is within 3 % on an average across the networks. However, one can choose any settings while executing the NNs on ALAMNI as per the requirement.

To identify the optimal size of LAM, we conduct experiments by changing the number of entries in LAM and observe its effect on the hit rate. However, there is a tradeoff between better hit rate and area/power overheads of larger LAM. Integrating larger LAMs improves the LAM hit rates in exchange for an additional cost of area and power. Figure 5.5 presents a comparative study of hit percentage with the various LAM sizes for all three networks. We keep 16, 32, 64, and 128 entries for each LAM to measure the hit percentage with different LAM sizes. Table 5.2 lists the average hit percentage of all three networks with various LAM sizes. We choose 64-entry LAMs in our design as this design configuration provides substantial hits without much overhead on the system. Additionally, the area and power of

Table 5.2: Average hit percentage with various LAM size (%).

ConvNets	16-entry	32-entry	64-entry	128-entry
VGG-16	37.13	43.64	53.68	59.90
VGG-19	39.54	46.82	54.77	62.75
ResNet-34	47.60	54.62	60.13	64.98

Table 5.3: Percentage of hits at different bit-mask (LAM size = 64 entries).

Bit-mask	VGG-16	VGG-19	ResNet-34
0-bit	32.14	35.86	43.45
5-bit	53.68	54.77	60.13
10-bit	95.90	96.42	96.48

the LAM modules increase by 97 % and 96%, respectively, while migrating 64-entry LAMs to 128-entry LAMs on UMC 15 nm technology. While migrating from 64-entry LAMs to 128-entry LAMs, the total power of the proposed system increases by 0.94 W (at 15 nm), which is around 63 % of the total power (1.5 W) of all ALAMNI units.

We perform experiments to see the effect of bit-masking on the hit rates in LAM (of size 64). In Table 5.3, we list the hit rates in LAMs (64 entries) for unmasked (0-bit), 5-bit mask, and 10-bit mask data ($\langle W, A \rangle$ pairs) for all three networks. With the increasing mask amounts, the hit rates increase linearly as the locality of data increases by the approximation. Here, the benefits of 0-bit mask come from LAMs and the data locality without any optimization. From the above study, we choose 64-entry LAMs for the ALAMNI units. We evaluate our hardware both for the unmasked and the 5-bit masking of data in the subsequent section.

5.5 Experimental Methodology

We perform experiments on the popular benchmarks from industries and academia. Table 5.4 lists the details of the workloads used for the evaluation. We have shown the number of CONV layers, the number of parameters, and the model size of each network. The details of the networks, shown in Table 5.4, are based on the pre-trained models of the CIFAR-10

Table 5.4: Workload Details.

ConvNets	VGG-16	VGG-19	ResNet-34
CONV layers	13	16	33
No. Params	33.647 M	38.959 M	21.282 M
Model Size	129 MB	149 MB	82 MB

Table 5.5: Specification of the architectures.

Host Processor (Common for all systems)	
Core	x86-64, 4 core, 2 GHz
Cache Memory (Common for all systems)	
L1 i-cache	32 KB, private, 4 way, 64 B blocks
L1 d-cache	32 KB, private, 8 way, 64 B blocks
L2 cache	256 KB, shared, 8 way, 64 B blocks
HMC (Common for all systems)	
Timings	t_{RP} = 7.7 ns, tt_{CCD} = 3.3 ns, t_{RCD} = 10.2 ns, t_{CL} = 9.9 ns, t_{WR} = 15 ns, t_{RAS} = 21.6ns
Energy [40]	3.7 pJ/bit (DRAM read), 6.78 pJ/bit (SerDes hop)
Power[132]	11.08 W
Size, # of vaults	2 GB, 16
Logic Die	90 nm, dimension $27 \times 27mm^2$
NLN (Baseline)	
# of NLN units	16
# of MACs / NLN	32
Activation buffer size / PE	1 KB (FP16), 512 B (INT8)
Weight buffer size / PE	128 B (FP16), 64 B (INT8)
Power (15 nm)	1.4 W (FP16), 0.81 W (INT8)
Area (total), 15 nm	1.68 mm^2, 1.02 mm^2 (INT8)
Frequency (15 nm)	∼3 GHz (FP16), ∼3.5 GHz (INT8)
Precision	FP16 / INT8
Proposed ALAMNI	
# of ALAMNI units	16
# of MACs / ALAMNI	32
Activation buffer size / PE	1 KB (FP16), 512 B (INT8)
Weight buffer size / PE	128 B (FP16), 64 B (INT8)
Power (15 nm)	1.5 W (FP16), 0.90 W (INT8)
Area (total), 15 nm	1.94 mm^2 (FP16), 1.2 mm^2 (INT8)
Frequency (15 nm)	∼3 GHz (FP16), ∼3.5 GHz (INT8)
MAC latency	16 Cycles (FP16) / 5 Cycles (INT8)
LAM-MAC latency	3 Cycles
Precision	FP16 / INT8

dataset.

5.5.1 Particulars of Architectures

To measure the efficacy of the proposed system, we evaluate three NMP-based architectures.

1. **NLN:** System with hardware acceleration without LAM search termed as No_LAM_NMP or NLN. This architecture does not skip any multiplication operations.
2. **ALAMNI:** System with ALAMNI units that skip multiplications using LAM hits, without using any bit-masking approximation.
3. **ALAMNI-Opt:** The system with ALAMNI units that skip multiplications using LAM hits with the approximation of using 5-bit masking before performing LAM search.

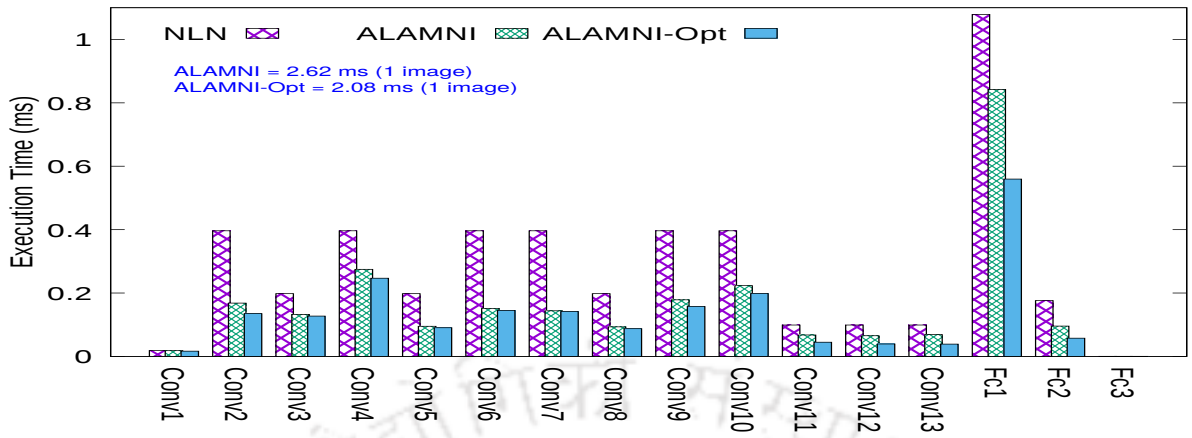
For all three architectures, we keep a common host system. The host processor is quad-core with two levels of the cache hierarchy. The detailed configurations of all the architectures are listed in Table 5.5. The HMC is used as the main memory for all the systems. To perform a fair comparison, both the NLN and ALAMNI hardware are integrated in a similar fashion with the individual vault controllers of the HMC. We also keep the number of processing units (NLN/ALAMNI), dataflow, number of MACs/unit, and the precision same for both the hardware. The NLN hardware does not contain any LAM with the MAC units; hence no operations are skipped in this architecture. We use the same ALAMNI unit for the evaluation of both ALAMNI and ALAMNI-Opt architecture. However, in ALAMNI, we have not leveraged the benefits of bit-masking. The benefits in ALAMNI arise only from the temporal locality of unmasked data cached into the LAMs as we have processed the unmasked $\langle W, A \rangle$ pairs in ALAMNI. In ALAMNI-Opt, we take the benefits of bit-masking and mask the $\langle W, A \rangle$ pairs with bit-vector $\langle 1111111111100000 \rangle$ to achieve 5-bit masking.

5.5.2 Evaluation

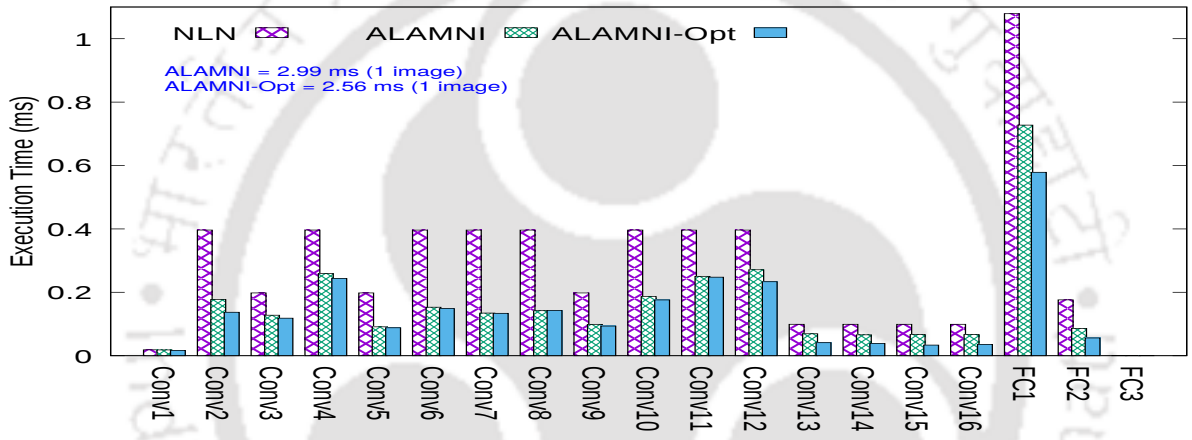
We evaluate all the systems based on performance and energy consumption. While investigating the NMP concept for accelerating CNNs, the area of the proposed hardware is also a critical parameter to be measured for near-memory logic integration. Thus, we perform a detailed area analysis of the proposed ALAMNI and find the overhead significantly low for near-memory integration.

5.5.2.1 Performance Analysis

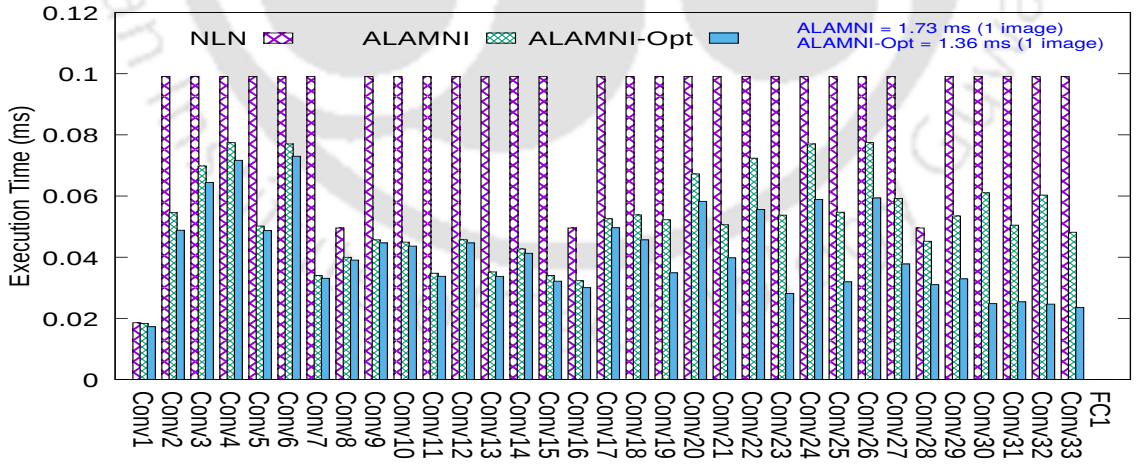
We implement both the NLN and ALAMNI hardware in Verilog hardware description language (HDL). We then perform the placement-aware logic synthesis in Genus Synthesis



(a) VGG-16



(b) VGG-19



(c) ResNet-34

Figure 5.6: Performance analysis of ConvNets/Networks.

Solution (version 15.21) from Cadence. At 15 *nm* technology, we obtain the operating frequency of both the hardware around 3 GHz, as shown in Table 5.5. Like in [3], we also

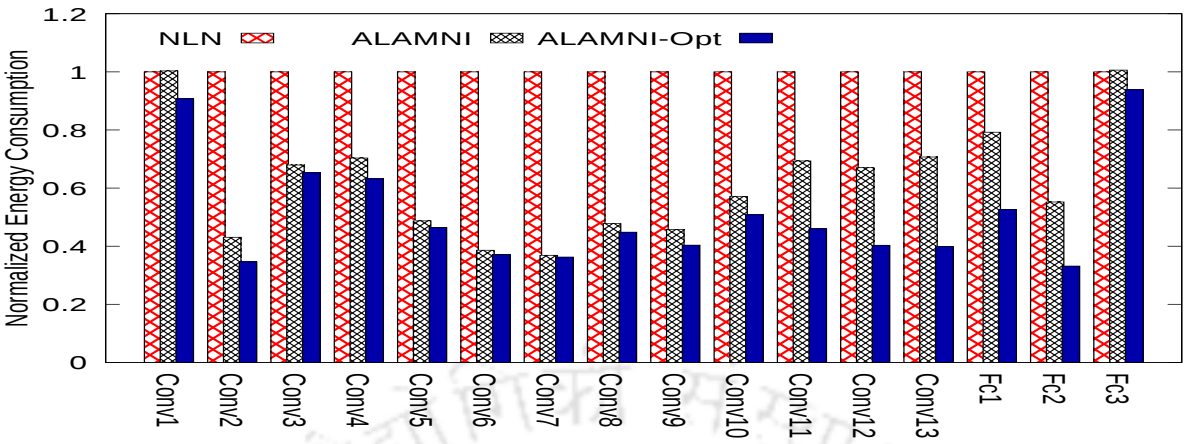
Table 5.6: Percentage of performance improvements over NLN architecture.

ConvNets	ALAMNI	ALAMNI-Opt
VGG-16	42.33	54.07
VGG-19	44.92	52.83
ResNet-34	43.22	55.19
Avg.	43.48	54.02

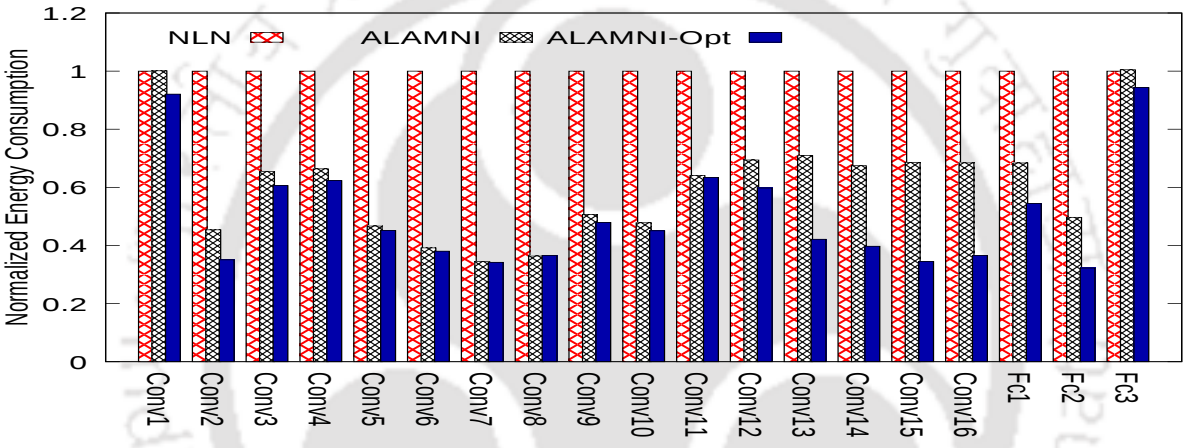
develop an in-house cycle-accurate simulator that resembles the state machines of the design hardware, NLN, and ALAMNI. The simulator is parametrizable and can be fed with the frequency obtained from the respective hardware synthesis. The execution times of each layer are measured from the simulator by using the corresponding state machine and the obtained frequency from the synthesis. The reported execution time by the simulator includes the data processing time in the accelerators and the time to fetch/store results in the memory through their respective memory hierarchy. The timing parameters to fetch/store data in HMC are shown in Table 5.5. Figure 5.6 shows the comparison of layer-wise execution times for all three networks in NLN, ALAMNI, and ALAMNI-Opt. Here, the Y-axis represents the execution time in milliseconds (ms), and the X-axis shows the layers of the network. The NLN architecture performs the worst among all three architectures for all the networks. The reason is: NLN has neither got the benefits of LAM nor the bit-masking. ALAMNI performs better than the traditional CNN accelerator (NLN), but not the best as it only leveraged the advantage of the temporal locality from the unmasked data through LAM search. Consequently, a lesser amount of multiplications are skipped here compared to the ALAMNI-Opt. The ALAMNI-Opt stands out to be the best in terms of performance because of the increased hits in LAMs due to bit-masking. Hence, more multiplications are skipped in ALAMNI-Opt compared to the ALAMNI. In Table 5.6, we list the percentage of improvements of the proposed ALAMNI and ALAMNI-Opt over the NLN architecture. On average, ALAMNI achieves around 43.48% improvements in the performance over the baseline NLN, and ALAMNI-Opt further increases it up to 54.02%. The execution times in milliseconds (ms) are also shown in Figure 5.6.

Table 5.7: Improvements (%) in energy savings over NLN.

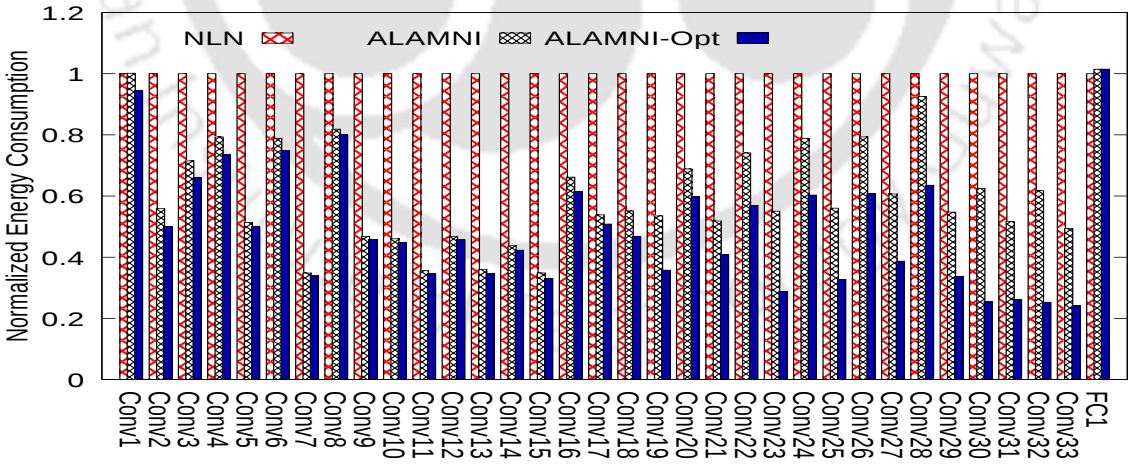
ConvNets	ALAMNI	ALAMNI-Opt
VGG-16	41.49	53.40
VGG-19	44.12	52.14
ResNet-34	42.39	54.54
Avg.	42.65	53.35



(a) VGG-16



(b) VGG-19



(c) ResNet-34

Figure 5.7: Energy efficiency comparison of ConvNets/Networks.

5.5.2.2 Energy Savings

Apart from the performance, we also aim to design an energy-efficient system. We find the proposed ALAMNI to be substantially energy efficient while accelerating the CNNs. To measure the power consumption, we perform synthesis on Genus from the Cadence tool-set. At 15 *nm* technology, we find the power consumption to be 1.5 W and 1.4 W, respectively, for ALAMNI and NLN modules. ALAMNI consumes slightly higher power because of the additional LAM modules. Figure 5.7 presents the comparative analysis of the energy consumptions across the layers of networks for all the systems. Here, the Y-axis represents normalized energy consumptions, and the X-axis shows the layers of the network. Both the ALAMNI and ALAMNI-Opt are substantially energy efficient compared to NLN architecture. Table 5.7 lists the percentage of energy savings of the proposed ALAMNI and ALAMNI-Opt over the NLN architecture. On average, ALAMNI achieves around 42.65% improvements in energy savings over the baseline NLN, which reaches up to 53.35% for the ALAMNI-Opt. Though the ALAMNI hardware module consumes slightly higher power compared to the NLN hardware module, the savings in energy for the proposed systems are obtained from the savings in execution cycles due to the skipping of multiplication operations through LAM search. Compared to ALAMNI, ALAMNI-Opt manifests better energy efficiency as more operations are skipped here due to the bit-masking. The effect of compulsory misses in the LAMs is also visible in the CONV1 layer for all the networks, as shown in Figure 5.7. ALAMNI does not provide much benefits in terms of latency in the CONV1 layer as the LAMs suffer from compulsory misses in the initial layer (CONV1). Additionally, the power consumption of ALAMNI units is also slightly higher than NLN units. As a result, ALAMNI provides almost equivalent energy efficiency for the first layer (CONV1) w.r.t NLN architecture. However, ALAMNI also recovers quickly (in the subsequent layers) as it starts getting high hit rates for the remaining layers. The ALAMNI-Opt performs well even in the initial layer (CONV1) because of the bit-masking. The last FC layer of all the networks, being substantially small, the caching effect in the LAMs is also very low in that particular layer. The same can be verified from Figure 5.7. Table 5.8 lists the performance gain, energy savings, and accuracy loss for the different bit-masks of VGG-16. The improvements are achieved over the baseline NLN architecture, which does not skip the multiplication operations. The power breakups of both the hardware (one chip) are shown in Table 5.9. Note that the LAM-MAC unit consumes higher power than the MAC unit because of the additional LAMs integrated with the MACs. However, the execution time of each image on ALAMNI is substantially lower than NLN. This neutralizes the effect of additional power and brings energy savings for the proposed system.

Table 5.8: Performance gain, energy savings, and accuracy loss (VGG-16). All values are in percentage.

Bit-mask	Perf. gain	Energy savings	Acc. loss
3-bit	52.56	51.87	1.93
4-bit	53.14	52.45	2.46
5-bit	54.07	53.40	4.13
6-bit	55.98	55.33	8.4
7-bit	59.32	58.72	22.8

Table 5.9: Power breakups of both NLN and ALAMNI (15 nm).

Modules	NLN (W)	ALAMNI (W)
Activation buff.	1.31E-03	1.31E-03
Weight buff.	2.14E-04	2.14E-04
Streaming buffs.	8.10E-03	8.10E-03
Output buff.	1.18E-02	1.18E-02
MAC/LAM-MAC	2.87E-03	5.02E-03
Controller and datapath	6.50E-02	6.42E-02

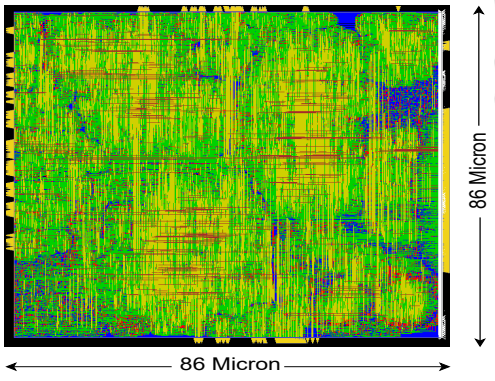


Figure 5.8: ALAMNI-Chip Layout from Innovus.

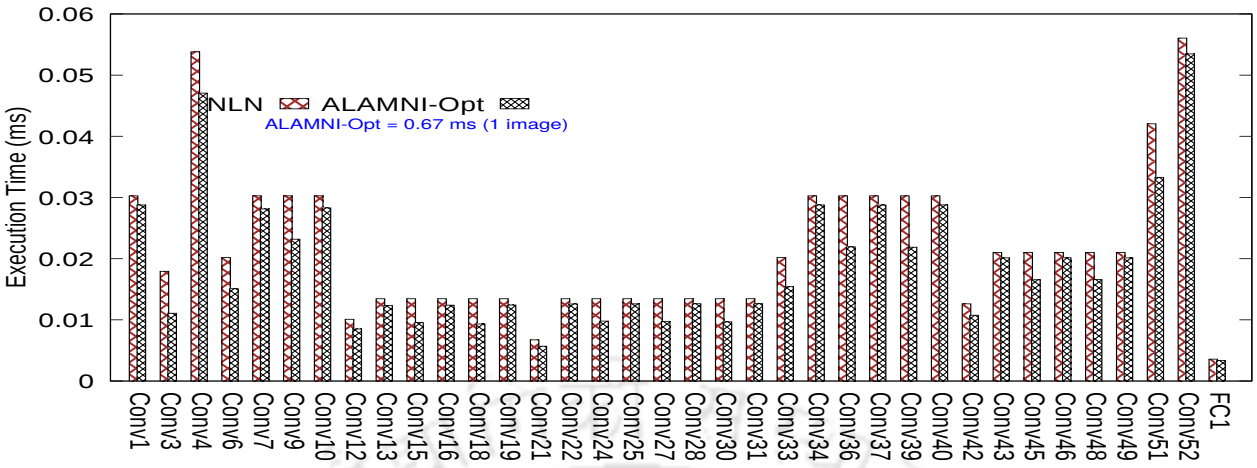


Figure 5.9: Performance on reduced precision (MobileNet).

5.5.2.3 Area Analysis

The area overhead plays a critical role in the near-memory integration of additional logic since memory industries are extremely area-sensitive. We implement the ALAMNI unit in Verilog HDL. We then use the Innovus from Cadence tool-set to obtain the post-synthesis area estimates. Each ALAMNI unit got placed on 86 μm $\times$ 86 μm square box at 15 nm technology. This leads to a negligible area overhead of less than 1% (15 nm) in total for 16 ALAMNI units on the logic die of HMC. Figure 5.8 shows one ALAMNI chip's layout where the standard cells are placed.

5.5.2.4 Reduced-Precision Analysis

We also examine the performance of ALAMNI policy for the low precision inference, specifically 8-bit integer (INT8). The lower precision increases the input similarity, which helps in improving the LAM's hit rate. On average, we observed around 10% increase in the hit rate for lowering the precision. A similar trend is also observed in [153]. For versatility, we use a larger dataset like ImageNet [119] for this experiment. The ALAMNI policy is more effective for larger models with larger layers. The LAMs get less scope to recover from compulsory misses using its adaptive replacement policy in small layers. Hence, to conduct more aggressive testing of ALAMNI, we choose a smaller model, MobileNet (pre-trained on ImageNet) [154]. Figure 5.9 shows the performance analysis of ALAMNI-Opt architecture on MobileNet. Note that we have not shown a few layers of MobileNet, as the values are significantly small to be represented in the graph compared to the other layers. The ALAMNI-opt (INT8 configuration) achieves around 14.15% (on average) performance boost over NLN (INT8 configuration). For ALAMNI, we observe around 11.73% improve-

ment (not shown in Figure 5.9 for readability) over the baseline NLN (INT8). We configure both the NLN and ALAMNI hardware to INT8. Consequently, the power consumption of both the hardware, NLN and ALAMNI, is reduced by 42% and 40%, respectively. The area overhead is also reduced by 39% (NLN) and 38% (ALAMNI). The reduction in power and area is primarily because of the less complex hardware in INT8 configuration. On average, ALAMNI and ALAMNI-Opt achieve around 4% and 7% energy savings over NLN. Although ALAMNI consumes slightly more power, ALAMNI still provides energy saving over NLN as the execution time (ms) is significantly less for ALAMNI and ALAMNI-Opt compared to baseline NLN. The percentage of improvements is competitively less in the reduced-precision ALAMNI approach mainly because of two reasons: (1) multiplier (INT8) latency is less compared to FP16 multiplier, and (2) some layers of MobileNet have significantly less number of $\langle W, A \rangle$ pairs to be processed. Consequently, LAMs get minimum opportunity to recover from compulsory misses. The experimental setup is identical to the preceding section. The detailed parameters regarding the performance and power consumption are mentioned in Table 5.5. Note that we find an accuracy loss of around 1% in ALAMNI-Opt for MobileNet model on ImageNet dataset.

5.5.3 Comparison with Previous NMP-based Accelerators without LAM Search

We compare our architecture with two popular NMP-based architectures: (1) NeuroStream (NS) [15], and (2) NeuroCube (NC) [3]. In NeuroCube, the DNN operations have been parallelized using a data partitioning scheme. NeuroCube needs a specialized state machine integrated with each vault controller to drive the data into the processing elements. Their experiment is limited to the small ConvNets having only 6 layers. Apart from that, NeuroCube does not skip the multiplications by any LAM search. In NeuroStream, DNN operations are performed in the logic layer of the 3D memory. Here, RISC-V cores and streaming coprocessors have been used as the NMP logic units. NeuroStream does not implement any LAM search-based technique to skip operations. As both the state-of-the-art architectures are near-memory (similar to us), we compare our ALAMNI with them to manifest the efficacy of the proposed ALAMNI approach. Among the ALAMNI and ALAMNI-Opt, we choose the ALAMNI one as it is comparatively less efficient than the other.

Table 5.10 summarizes the key design specifications of all the NMP-based architectures. The comparison has been made on similar technology nodes and precisions for fairness. The proposed ALAMNI achieves a peak throughput of 0.67 TFLOPS and 3.13 TFLOPS over 0.24 TFLOPS of NeuroStream and 0.13 TFLOPS of NeuroCube, respectively, as shown in Table

Table 5.10: Comparison with previous near-memory DNN accelerators without LAM search.

Node (nm)	*Precision*	SOA	Peak (TFLOPS)	Power (W)	Efficiency (GFLOPS/W)	*Area (mm^2)*
28 nm	FP32	NS [15]	0.24	11	22.5	8.3
		ALAMNI (no bit-masking)	0.67	6.4	104.69	5.7
15 nm	FX16	NC [3]	0.13	3.4	38.8	0.98
		ALAMNI (no bit-masking)	3.13	2.1	1490.48	2.2

5.10. Compared to both, ALAMNI also consumes substantially less power. The efficiency of 104.69 GFLOPS/W and 1490.48 GFLOPS/W are also appreciably higher compared to both architectures.

5.6 Summary

In this chapter, we propose a near-memory architecture, ALAMNI, that reduces the number of operations in CNNs. ALAMNI avoids the recomputation of similar inputs by caching the inputs and outputs in the small Lookaside memories (LAMs) for future computations. Our proposal collects the operand pairs in the LAM at runtime and hence adapts very well to a given ConvNet. In that, it needs no pre-processing. Being near-memory, the proposed architecture survives from off-chip memory constraints. ALAMNI also provides a unique feature to its users. The user can leverage the benefits of bit-masking as an approximation as per their requirement. This feature helps in skipping more computations and hence increases the performance and energy efficiency of the proposed system. We evaluate our system on the popular CNN workloads like VGG-16, VGG-19, and ResNet-34. On average, ALAMNI achieves around 43.48% performance gain and 42.65% energy savings compared to a near-memory accelerator without any LAM (NLN). On a 5 bit-masking, the performance gain and energy savings can rise up to 54.02% and 53.35%, respectively, over the NLN. Our proposed ALAMNI also outperforms other existing NMP-based architectures in terms of performance, power consumption, and efficiency. Moreover, at 15 *nm* technology, the additional hardware overhead is also below 1%. Certainly, the obtained results and the additional feature make this system an attractive solution to address the issues of real-time inference. The thesis's objectives like reducing data movement (objective1), obtaining high performance and energy efficiency through accelerated architecture (objective2), providing support for various applications (objective3), and delivering high throughput by applying optimization to reduce computations (objective4) are coved in this chapter.

भारतीय प्रौद्योगिकी संस्थान गुवाहाटी
Indian Institute of Technology Guwahati

6

Exploring Other Avenues for NMP Processing

In this chapter, we perform further experiments to explore multiple avenues related to NMP processing for detailed comprehension. The previous chapters of this thesis focus on the integration of logic close to 3D memories. However, 2D memories are also prevalently used in almost every type of computing system. The exploration of NMP's efficacy remains partial if NMP is not explored with 2D memories like DRAM or hybrid memory. Additionally, it is also essential to investigate the NMP's efficiency with another application since we accelerated only CNNs using the NMP until this point. Towards this end, we perform the following experiments in this chapter.

1. In section 6.1, we design CNN inference accelerators and integrate those units inside DRAM-PCM-based hybrid main memory to estimate NMP's efficiency with hybrid main memory-based system.

2. In section 6.2, we explore design space in terms of performance, power consumption, and area overhead of a system while integrating additional logic inside DRAM for inference acceleration.

3. We also manifest the benefits of the NMP approach for another application like database operations in section 6.3.

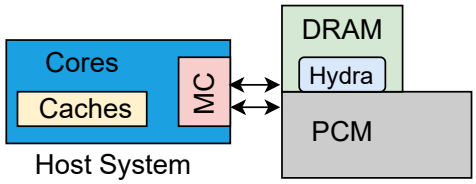


Figure 6.1: A high-level view of the proposed system.

6.1 Hydra: A near Hybrid Memory Accelerator for CNN Inference

6.1.1 Introduction

We already discussed that the key limitations faced by many of the CNN accelerators are the limited off-chip memory bandwidth or on-chip capacity. Here, we aim to address this limitation through NMP integration in hybrid main memory. CNN inference has been accelerated in prior works using the NMP concept [3, 15, 16]. These works integrate custom hardware near 3D DRAM-based memories for the acceleration of CNNs. Efforts have also been spent to implement traditional DRAM-based accelerators that use processing-in-memory (PIM) for CNNs [18, 37]. However, these works avoid crucial multiplications using binary/ternary neural networks, as implementing multiplication in PIM is challenging. Avoiding multiplications by exploiting binary/ternary quantized networks leads to an energy-efficient solution for mobile/IoT devices. However, there is a non-negligible inference error of around 7% for ImageNet top-1 accuracy, between binary neural network [155] and the best case [156].

Emerging data-intensive applications demand high memory capacity. Limitation in DRAM scaling is unable to meet the growing memory requirement. Non-volatile memories (NVM) such as phase-change memory (PCM) has emerged as a potential candidate for main memory. The NVMs offer high memory density, low cost per bit, and near-zero standby power consumption in exchange for low performance and limited endurance [68, 69]. Though NVMs, like PCM, provide several advantages, they cannot entirely replace the DRAM. Instead, it is more practical to use NVMs in conjunction with DRAM to form a hybrid memory system [68, 69, 157]. The NMP has been extensively explored for accelerating CNNs [3, 15, 16, 18, 37]. However, the efficacy/feasibility of NMP is not investigated for the emerging hybrid main memory while executing the CNNs.

In this work, we aim to build a system with NMP capability in hybrid memory that benefits inference operations in terms of performance and energy efficiency. We choose one of the most promising main memory subsystems; specifically, hybrid memory [68, 158, 159]. Figure 6.1 presents a conceptual view of the proposed system where we integrate

custom hardware, Hydra, closer to DRAM for inference processing. The Hydra units are designed to operate with hybrid main memory as they prefetch model parameters from PCM in parallel to inference execution. We choose the DRAM-PCM combination (shown in Figure 6.1) among the various DRAM-NVM combinations available in hybrid memory architectures as PCM is one of the most mature candidates for main memory [70]. However, our proposed hardware is memory-technology independent and can be integrated with any DRAM/NVM-based memory. The reason is; unlike [18, 37], we rely entirely on the proposed hardware (Hydra) for all the CNN computations, and we do not modify the DRAM or PCM memory cells for any computation. We use these memory cells only for traditional load/store operations, making the design less complex for practical implementation. Consequently, unlike [18, 37], our architecture is not restricted to quantized binary/ternary models as Hydra has the capability of executing the multiplications. We place the Hydra units close to the memory to reduce the data access cost in terms of access latency and energy consumption without any significant change in the memory sub-arrays.

We design custom hardware, *Hydra, a near hybrid memory accelerator* for CNN inference, and integrate 16 Hydra units (one per chip) inside the DRAM DIMM (dual in-line memory module) in the hybrid memory subsystem. These 16 Hydra modules can work in parallel, and all the multiply-accumulate (MAC) units of each Hydra module can also concurrently execute the inference tasks, leading to inter- and intra-chip parallelism, respectively. We integrate the Hydra module inside DRAM as a design choice in the DRAM-PCM hybrid memory subsystem. This design choice helps in eliminating the intermediate writes of the CNN's hidden layers in the PCM's cell array, leading to an enhanced lifetime for the PCM device [160]. While executing inference close to the DRAM, we choose to keep models in PCM and the data generated during the inference in the DRAM. The reasons for keeping the models in denser PCM are: ❶ Models are usually heavier (in size) than a sample under classification, ❷ In inference, the model's parameters are not updated, and hence no PCM writes are required, ❸ The read latency of both PCM and DRAM is equivalent, and ❹ After the initial prefetching of data, page migration latency is overlapped by the CNN's execution time as we keep buffers in Hydra to store both the input activations and weights. The primary contributions of this work are as follows.

1. We design dedicated hardware, Hydra, and integrate those units inside the DRAM's chips in the hybrid main memory subsystem. This integration helps in alleviating the off-chip memory accesses using the NMP concept.

2. We also employ data partitioning to leverage intra- and inter-chip parallelism inside DRAM for the concurrent execution of the CNN tasks using multiple Hydra units.

3. The Hydra controller prefetches the models from PCM (only read) into its local buffers and overlaps the page migration latency by the execution latency. The Hydra does not use its PCM module for CNN's intermediate write operations and thus enhances the lifetime of PCM.

The proposed system outperforms the state-of-the-art while accelerating inference. Unlike DRAM-based inference accelerators, our architecture does not restrict the execution of non-quantized models. The throughput of the proposed system is evaluated with popular networks like VGG-16 [118], VGG-19 [118], and ResNet-34 [48], pre-trained on ImageNet dataset [119].

6.1.2 Background: Hybrid Main Memory Subsystem

Several works on hybrid main memory [68, 159] exploit the benefits of both DRAM and NVM (like PCM). The DRAM-PCM-based hybrid memory architecture can be classified into two categories: (1) DRAM-PCM parallel architecture, and (2) DRAM as a cache for PCM [159]. We consider DRAM-PCM parallel architecture with Hydra integrated into the DRAM chips. The benefits of using a hybrid memory are multi-dimensional. The hybrid main memory subsystem exploits the benefits of both DRAM and PCM (high capacity, low standby power, non-volatility of PCM + high performance, better active power of DRAM) while minimizing the negative impact of both memories [161].

Modern DRAMs are typically multi-banked, each serving requests independently from others. Each bank is composed of mats or the 2D-subarray of memory cells. An entire row of data is brought to the row buffer with the help of control wires like wordlines and bitlines. The row buffers primarily include a set of sense amplifiers that amplify the small change of voltage to a stable voltage level. From the row buffer, a few bits are sent to the I/O pads by the column decoder.

PCM is a chalcogenide glass that has large resistance contrast between crystalline and amorphous states [162]. The difference in resistance is used for representing 1 and 0, or even multi-state/cell. The heating process drives the phase transitions between the two states of PCM. Unlike DRAM, PCM does not require refresh operation. Despite all the benefits, the heating processes involved in changing PCM states are long and power-consuming. Consequently, writing operations on PCM is costly in terms of latency and power consumption than a DRAM write. Additionally, PCM is prone to wearing out due to multiple writes on account of its low endurance. Our aim is to reduce writes to PCM so that its lifetime increases. Similar to DRAM, the PCM is also hierarchically organized with channels, ranks, chips, and banks in our proposed system [163]. The detailed parameters of the DRAM-PCM

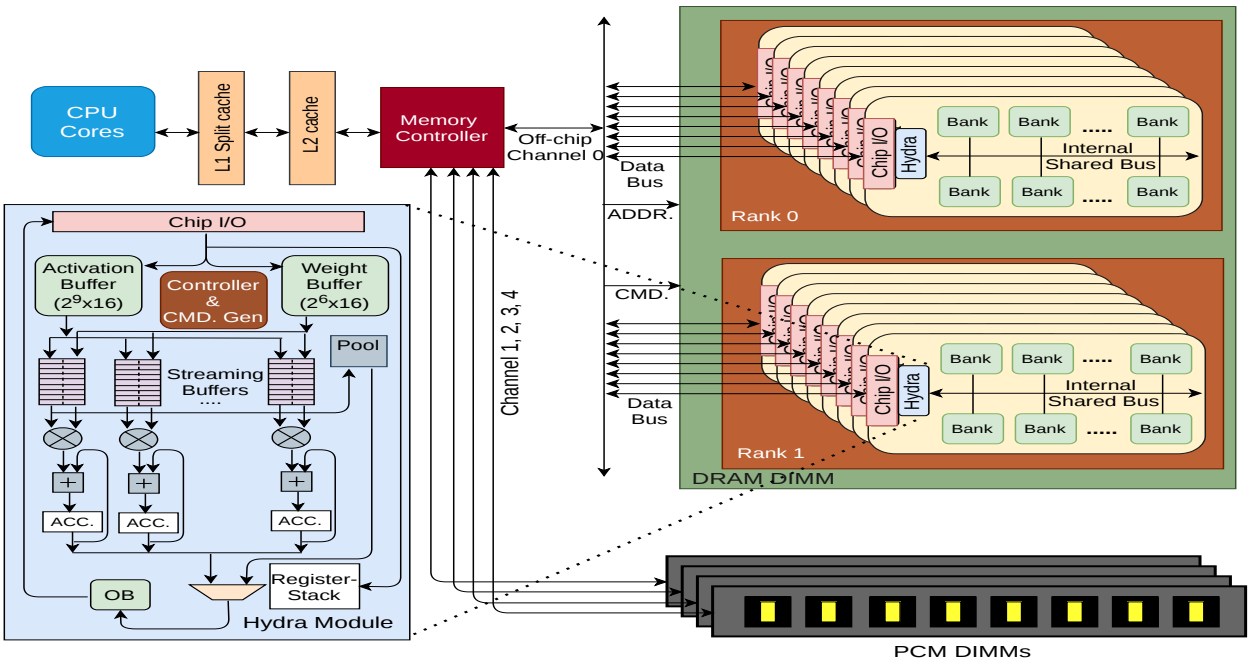


Figure 6.2: The proposed system.

hybrid memory are shown in Table 6.1.

6.1.3 Proposed Architecture

6.1.3.1 Overall System Architecture

Figure 6.2 shows the overall system architecture. The host side includes a quad-core host processor with a two-level cache hierarchy (L1 split and L2 shared). We choose PCIe interface-based DRAM-PCM parallel hybrid memory for the near-memory integration of the proposed Hydra units. Similar to [158], we use one channel for DRAM and the other four channels for PCM memory, as shown in Figure 6.2. A rank is composed of multiple chips to increase the width of the memory channel. To leverage the chip-level parallelism available via rank, we integrate 16 Hydra units, one per DRAM chip (chip-level integration), inside the DRAM module (shown in Figure 6.2). This chip-level integration is more area efficient than a bank-level integration, as the number of hardware (Hydra) is less in chip-level integration compared to the bank-level integration. The same has been manifested in [2]. Within each DRAM chip, the Hydra module is connected with the internal bus that is shared by all banks. The CNN inference data, like weights, input activations, are fetched into the local buffers of Hydra modules through the arbitration of the internal shared bus.

6.1.3.2 Hydra Microarchitecture

The abstract micro-architecture of the Hydra unit is also shown in Figure 6.2 (left blue box). The primary components of a Hydra are a Hydra controller and command generator, weight buffer (WB), activation buffer (AB), an output buffer (OB), 32 small streaming buffers, 32 fast and efficient MAC units, register-stack, and pool unit with 32 comparators. Additionally, a result accumulation unit with an integrated linear function is also integrated with the memory controller. The host processor initiates the inference process and sends the meta-data information like hyperparameters of the ConvNets to the register-stack with the help of the memory controller. Hydra controller drives all the components to accomplish the inference tasks. Each Hydra controller loads one channel of input maps and kernels into its local AB and WB, respectively, from DRAM through the internal shared bus. To save the command bandwidth, basic command generation can also be done by the Hydra controller, similar to [2]. The filling of AB and WB is done by the WRITE command. The Hydra architecture has been configured to operate on 16-bit fixed-point precision. However, our design can be easily adapted for other precisions. The Hydra controller continuously loads the streaming buffers by the weight-activation, $\langle W, A \rangle$, pairs to be processed by the MAC units. The bias register in the register stack holds the bias value that is added only once to each neuron position. After MAC operations, the Hydra controller stores the results in the OB of the Hydra modules. Finally, the results from the OB can also be read by the memory controller using the READ commands for accumulation in the result accumulation unit. Note that we have not shown all the connections in Figure 6.2 for simplicity and readability.

6.1.3.3 Dataflow

The proposed architecture with integrated Hydra exploits both the intra- and inter-chip parallelism to maximize the system's throughput. Figure 6.3 represents the data distribution and its granularity of execution. The upper portion of Figure 6.3 shows the inter-chip parallelism. Each channel of inputs and weights of a hidden layer is sent to one Hydra unit of a chip for the parallel execution of convolution (CONV)/pool operations. For the initial layer, all Hydra units may not get a channel to be processed because of the lesser input/weight channels in the initial layer of CNNs. However, for remaining all hidden layers, a batch of 16 channels can be processed concurrently by 16 Hydra units, one at each chip. Note that we consider a dual-rank, 8 chip/rank, 16 bank/rank configuration of DRAM (2Rx8). Increasing the number of Hydras by increasing the number of chips or ranks can further increase the system's performance due to the additional inter-chip parallelism.

The intra-chip parallelism within one Hydra unit is represented in the lower portion of

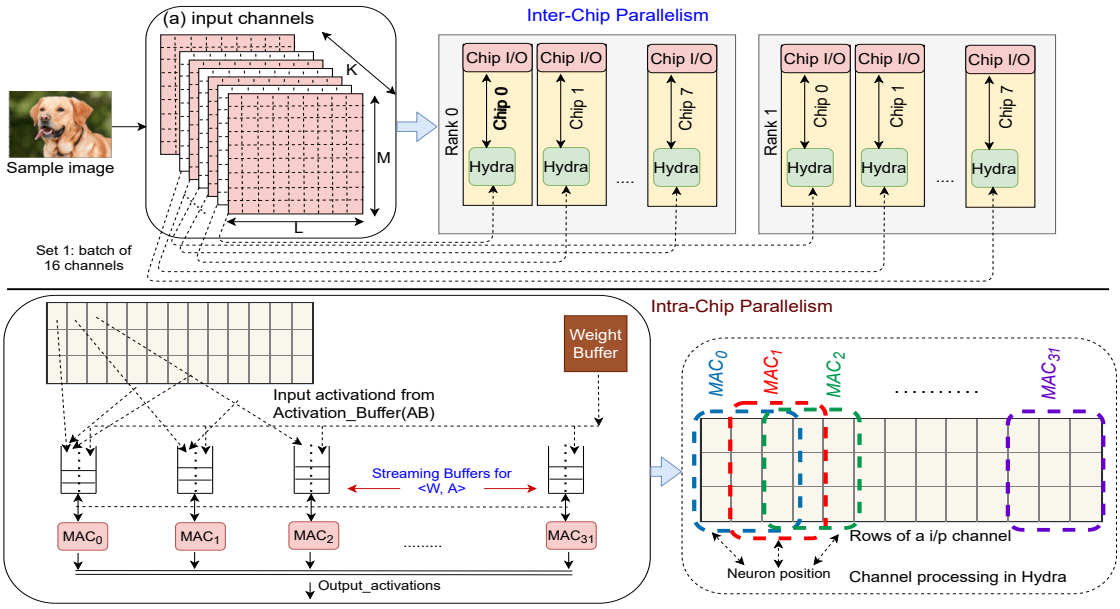


Figure 6.3: Dataflow to leverage parallelism.

Figure 6.3. The 32 MAC units of a Hydra can perform the CONV/pool operations at 32 neuron positions of a channel in parallel to others, leading to intra-chip parallelism. The Hydra controller schedules the CNN tasks to each MAC, based on the available neuron positions to be processed. Each MAC is associated with a small stream buffer that holds the $\langle W, A \rangle$ pairs to be processed. The Hydra controller always keeps these buffers operation-ready by filling them with the appropriate $\langle W, A \rangle$ pairs to be processed by the corresponding MAC units. The buffer filling time is overlapped with the MAC operations. The Hydra controller reduces the number of fetch operations by fetching data once and broadcasting them to all the appropriate streaming buffers during the filling of the streaming buffers with the $\langle W, A \rangle$ pairs, as shown in Figure 6.3. The model parameters can be prefetched from PCM banks through the data bus and memory controller in parallel to inference tasks. The combination of intra- and inter-chip parallelism enables Hydra to achieve high performance.

6.1.4 Evaluation

6.1.4.1 Evaluation Methodology

To measure efficacy of the proposed system, we conduct experiments using the popular CNN workloads like VGG-16 [118], VGG-19 [118], and ResNet-34 [48], pre-trained on ImageNet dataset [119]. The data and network parameters are extracted from the experiments in the PyTorch framework [133]. The detailed configuration of the proposed architecture is

Table 6.1: Specification of the architectures.

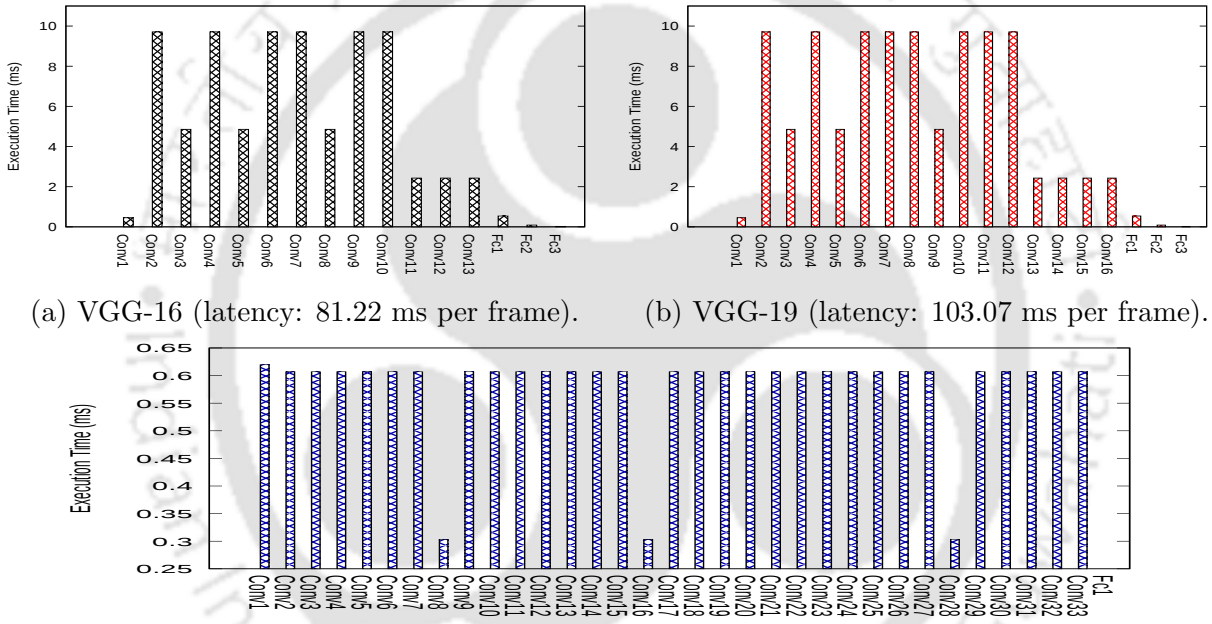
Host Processor (Common for all systems)	
Core	x86-64, 4 core, 4 GHz
Cache Memory (Common for all systems)	
L1 i-cache	32 KB, private, 4 way, 64 B blocks
L1 d-cache	32 KB, private, 8 way, 64 B blocks
L2 cache	256 KB, shared, 8 way, 64 B blocks
DRAM	
Timings [164]	t_{RP} = 5 cy, t_{CCD} = 4 cy, t_{RCD} = 5 cy, t_{CL} = 5 cy, t_{WL} = 4 cy, t_{WR} = 6 cy, t_{RTP} = 3 cy
Energy [164]	1.17 pJ/bit (array read), 0.39 pJ/bit (array write) 0.93 pJ/bit (buffer read), 1.02 pJ/bit (buffer write)
Configuration	8GB, 1 channel, 2 rank, 8 chip, 16 banks, FR-FCFS request scheduling
PCM	
Timings [164]	t_{RP} = 60 cy, t_{CCD} = 4 cy, t_{RCD} = 22 cy, t_{CL} = 5 cy, t_{WL} = 4 cy, t_{WR} = 6 cy, t_{RTP} = 3 cy
Energy [164]	2.47 pJ/bit (array read), 16.82 pJ/bit (array write) 0.93 pJ/bit (buffer read), 1.02 pJ/bit (buffer write)
Configuration	32GB, 4 channel, 8 rank, 8 banks/rank, FR-FCFS request scheduling
Proposed Hydra	
# of Hydra units	16
# of MACs / Hydra	32
Activation buffer size / PE	1 KB
Weight buffer size / PE	128 B
Power (15 *nm* CMOS tech.)	1.4 W
Area (total), 15 *nm* CMOS tech.	1.6 mm^2
Frequency (15 *nm* CMOS tech.)	∼3 GHz
Precision	FX16

mentioned in Table 6.1. We evaluate our proposed system with integrated Hydra based on its performance and energy consumption. While incorporating the NMP concept in hybrid main memory, the area of the proposed units is highly critical. Hence, we also perform a detailed area analysis of the Hydra modules in the subsequent section.

6.1.4.2 Performance and Energy Analysis

We implement the Hydra module in Verilog hardware description language. We use Genus Synthesis Solution (version 15.21) from Cadence and 15 *nm* CMOS technology for the placement-aware logic synthesis of the designed Hydra. After synthesis, we obtain an operating frequency of around 3 *GHz*, as shown in Table 6.1. We develop an in-house cycle-accurate

simulator that resembles the state machines of the designed hardware, similar to [3]. Our simulator is parametrizable and can be fed with the frequency obtained from synthesis. We obtain the network's layer-wise execution times (ms) from the simulator, which include both data processing and load/store latency. Figure 6.4 shows the layer-wise latency in milliseconds (ms) for all networks in the Hydra architecture. The total execution latency per frame is also shown in Figure 6.4 for all the networks. Note that VGG-16 and VGG-19 have a similar trend in the execution latency as they have similar CNN architecture. However, VGG-19 has more Conv layers than VGG-16, leading to an additional latency of 21.85 ms per frame in Hydra. The ResNet-34 takes less time (ms) to process one frame as the number of parameters in ResNet-34 is less compared to VGG-16/VGG-19 network.



(a) VGG-16 (latency: 81.22 ms per frame). (b) VGG-19 (latency: 103.07 ms per frame).

(c) ResNet-34 (latency: 19.13 ms per frame).

Figure 6.4: Performance of CNN inference in Hydra.

For a comparative study on the system's throughput, we also integrate our hardware with a pure DRAM (no PCM attached) and a pure PCM (no DRAM attached) based system. We keep the number of hardware units, place of integration, and dataflow the same for all systems. We obtain an equivalent performance for all systems (pure DRAM, pure PCM, and Hydra) though the memory access latencies are different. The reason is: Hydra is specially designed with additional buffers to overlap data access latency with its execution latency. Additionally, execution latency is the same as we integrate the same hardware for all the systems. Consequently, we obtain equivalent performance for all the systems.

However, we observe a significant impact on the data transfer energy across all three

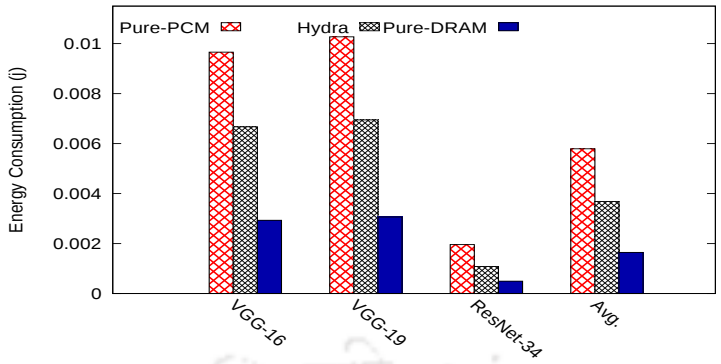


Figure 6.5: Comparison of data transfer energy per frame.

systems. We measure the post-synthesis power consumption of the proposed hardware on Genus from the Cadence. At 15 *nm* CMOS technology, we find the power consumption to be 1.4 W in total for all 16 hardware units. Figure 6.5 shows the comparison of data transfer energy (joule) for 1 frame in all the systems. On average, Hydra consumes 1.6x lower energy than the pure PCM-based system. However, the pure DRAM-based system stands out to be the best (3.5x lower energy than pure PCM-based system) among these systems. The reason is: the array read, write energies of PCM are respectively 2.1x and 43.13x more than DRAM [164].

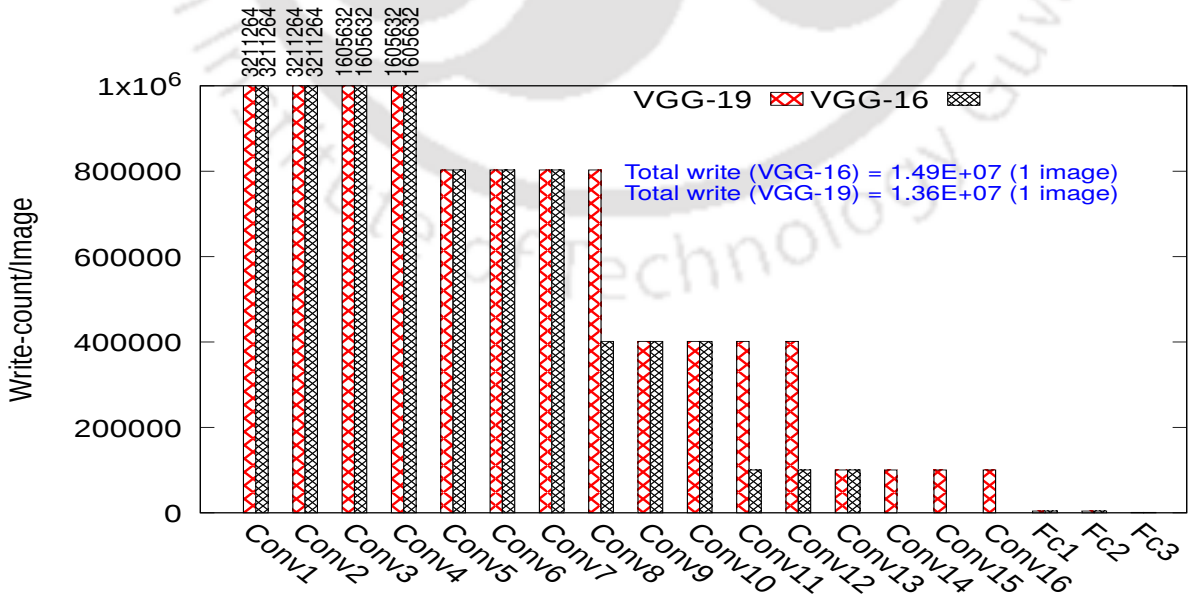


Figure 6.6: Write reduction on PCM as all the intermediate values are written in DRAM.

6.1.4.3 Write Reduction on PCM

We integrate Hydra modules into DRAM chips. This integration aims to eliminate intermediate writes on the PCM while executing the hidden layers of CNNs. Figure 6.6 presents the layer-wise estimates of write reduction on the PCM modules for one frame in VGG-16 and VGG-19. A total of *1.49E+07* and *1.36E+07* numbers of writes per frame are avoided in VGG-19 and VGG-16, respectively, for the PCM memory. For one inference epoch on ImageNet test-set [119], the savings on the number of writes are *1.49E+12* and *1.36E+12* for VGG-19 and VGG-16, respectively, leading to increased lifetime for PCM [160].

6.1.4.4 Area Analysis

Since the memory industry is extremely area-sensitive, the area overhead needs to be considered when integrating additional logic near the memory. After implementing the Hydra module in Verilog, we use the Innovus from Cadence tool-set to obtain the post-synthesis area estimates. Each Hydra unit got placed on a 0.32 mm $\times$ 0.32 mm square box at 15 nm technology. This leads to an area overhead of 1.6 mm^2 in total for all 16 Hydra units. Compared to the 80 mm^2 area of a DRAM [100], this area overhead is only around 2%.

6.1.4.5 Comparison with Previous Accelerators

We compare our Hydra architecture with two popular DRAM-based processing-in-memory frameworks, DRISA [18], and DrAcc [37]. These architectures aim to modify the sub-array of commodity DRAM to exploit bit operation capability. Special rows are introduced to perform shift, NOT, and XOR operations in DrACC. In contrast, DRISA proposes to reorganize the bank and sub-array dimensions of DRAM for in-memory operations. Both DRISA and DrACC adapt the concept of concurrently activating multiple rows to realize the in-DRAM operations. Two limitations in this concept are: (1) the process variation in the sense amplifiers and (2) high energy consumption stemming from the multi-row activations. The process variations often lead to logical operation failure. A 25% process variation can lead to around 26% failure in the logical operations [27]. Unlike DrAcc, DRISA addresses the problem by restructuring the sub-array that is different from the highly optimized commodity DRAM sub-array design. However, we use the sub-array only for traditional load-store operations, leading to no sub-array-level modification for the proposed work. The entire computations are done in a separate module, Hydra, integrated near the chip I/O. Multiplication operations (MUL) are crucial as they occupy the major portion of inference computation. It is also feasible to adopt quantized models to eliminate MULs in mobile/IoT devices. However, there is a non-negligible classification error of around 7% for ImageNet top-1 accuracy,

Table 6.2: Comparison with previous DRAM-based accelerators.

	DrAcc [37]	DRISA [18]	Hydra
Perf. (FPS/ms)	0.3/3282 (VGG-16) 0.25/3933 (VGG-19)	0.3/3283 (VGG-16) 0.25/3933 (VGG-19)	6.61/151.32 (VGG-16) 5.21/192.03 (VGG-19)
Power (W)	2	98	3.63
Area (mm^2)	0.01	65.2	2.7
Efficiency (Fr./watt)	0.15	0.0031	1.82

between binary neural network [155] and the best case [156]. It is challenging to support the important multiplication operations in the PIM design. DrAcc skips the multiplications using ternary weight neural networks, while DRISA uses 1-bit weights for the inference. Instead of skipping multiplications, we perform them in MAC units, making this suitable for the requirement of high accuracy. As shown in Table 6.2, we obtain around 20x performance gain compared to both the state-of-the-art works because of the additional MAC-based logic used in Hydra. Additionally, the proposed system also leverages two-level of parallelism: (1) inter-chip and (2) intra-chip. The parallelisms and the additional MAC units provide high performance for the proposed Hydra-based system. Table 6.2 summarizes the quantitative key design specifications of all the accelerated architectures. The reported results for the state-of-the-art works are obtained from DrAcc [37]. On the comparable CMOS technology node, Hydra is more power and area efficient compared to DRISA, as shown in Table 6.2. The area/power overhead of Hydra is higher than DrAcc because of the additional logic for processing. However, the efficiency (Frame/watt) of Hydra is substantially higher than both DRISA [18] and DrAcc [37], as shown in Table 6.2.

6.1.5 Summary

We propose Hydra, a near hybrid memory accelerator for inference. We store the models in the denser PCM banks and prefetch them into the local buffers in parallel to the inference tasks. We place one Hydra module per DRAM chip to leverage inter-chip parallelism. We also exploit intra-chip parallelism by performing CONV/pool operations at 32 neuron positions of a channel in parallel to others. The placement of Hydra eliminates the intermediate writes of inference on the PCM, leading to enhanced lifetime. We also outperform the state-of-the-art works in terms of performance. Additionally, our system does not restrict the execution of non-quantized networks, unlike the previous works.

6.2 Exploring the Design Space for Near-DRAM MAC-based Inference Engine

6.2.1 Introduction

Continuing the discussion to integrate logics closer to conventional 2D-DRAM-based main memory, in this section, we explore the impact of performance/power consumption by placing the CNN accelerator at various levels of DRAM DIMM. The neural network acceleration has also been achieved through in-DRAM processing [18, 37]. Unlike our multiple-accumulate (MAC) based acceleration, as discussed before, these works primarily rely on the bit operation capability achieved by modifying the DRAM cell arrays. However, with in-DRAM processing, it is challenging to implement multiplication operations which are the primitive operations of any inference. Consequently, these works avoid the crucial multiplications by choosing quantized binary or ternary neural networks.

In this work, to accelerate CNN inference, we aim to design near-DRAM processing-based systems that are not restricted to executing only binary/ternary quantized CNNs, unlike the previous in-DRAM processing-based works [18, 37]. We choose the most popular main memory system, traditional DRAM, for the integration of additional logic inside the DRAM's chip. While in previous DRAM-based designs [18, 37], the CNN's computations depend on the computing capability of memory cells; our design entirely relies on the proposed hardware and its MAC units for the processing of the inference tasks. We do not drastically modify the DRAM cell arrays or the sense amplifier circuits of the DRAM's data path and use them only for traditional fetch/store operations. We design our proposed hardware, near DRAM inference accelerator (DiA) and integrate multiple such instances in the dual in-line memory module (DIMM) of DRAM. As the DRAM has a strict area and power budget [37], we also explore the design space based on the parameters of interest like power consumption, area overhead, and performance while integrating the proposed hardware (DiA) inside the DRAM's chip. We identify two locations where the designed hardware (DiA units) for inference can be integrated seamlessly without significantly changing the existing DRAM circuitry. Similar places of near-DRAM logic integration have been identified in [2]. However, they [2] target to accelerate database operations. The two sites for hardware integration in the DIMMs are as follows.

1. **Chip-level integration:** Here, we integrate only one DiA unit per chip.

2. **Bank-level integration:** In this integration, one DiA unit is integrated with every bank of a chip.

These integrations lead to two proposed architectures that have integrated DiA units in them. Note that the number of chips or banks can vary across different DRAMs. Consequently, the chip or bank-level integration of DiA in different DRAM configurations can result in different system throughput with varying area/power overheads. In our proposed system, we choose a dual-rank, 8 chips/rank, 16 banks/rank configuration of DRAM (2Rx8). It can be observed that the number of DiA units is more in bank-level integration (32 in total) compared to the chip-level integration (16 in total) in the case of our proposed systems. As a result, bank-level integration provides higher performance in exchange for additional area/power overhead compared to the chip-level integration of DiA units. All the DiA units can work in parallel to others leading to exploitation of chip- or bank-level parallelism. Additionally, each DiA unit comprises multiple MAC units that can also work in parallel to others, leading to intra-chip or intra-bank parallelism for chip-level or bank-level integration of DiA units, respectively. The memory controller takes the role of data distribution and accumulation. Considering the area/power constraints in DRAM, we also propose two optimizations of the designed DiA modules, namely DiA-light1 and DiA-light2.

1. In DiA-light1, we reduce the computation capability of each hardware (DiA) by minimizing the computational resources like MAC units. This optimization reduces the performance in exchange for better area/power efficiency compared to DiA. As bank-level integration of DiA is a heavier design than the chip-level integration of DiA units, we prefer to integrate optimized DiA-light1 as a bank-level integration and compare the three architectures in terms of the system's throughput.

2. As inference is also popular in reduced precision like the 8-bit integer (INT8), we optimize the DiA-light1 further by reducing its precision from the 16-bit fixed point (FX16) to INT8 in DiA-light2. We measure the performance, power consumption, and area overhead of DiA-light2 both for chip-level and bank-level integration while accelerating the inference.

In summary, the primary contributions of the work are as follows.

1. We design dedicated hardware, DiA, and integrate those units inside the DRAM's chip using the NMP concept.

2. To explore the design space for near-DRAM MAC-based accelerator, DiA, we integrate one DiA unit per chip or per bank and leverage the chip-level or bank-level parallelism, respectively.

3. We design a data partitioning scheme to leverage both intra- and inter-chip (or bank) level parallelism for the concurrent execution of the CNN tasks in multiple DiA units.

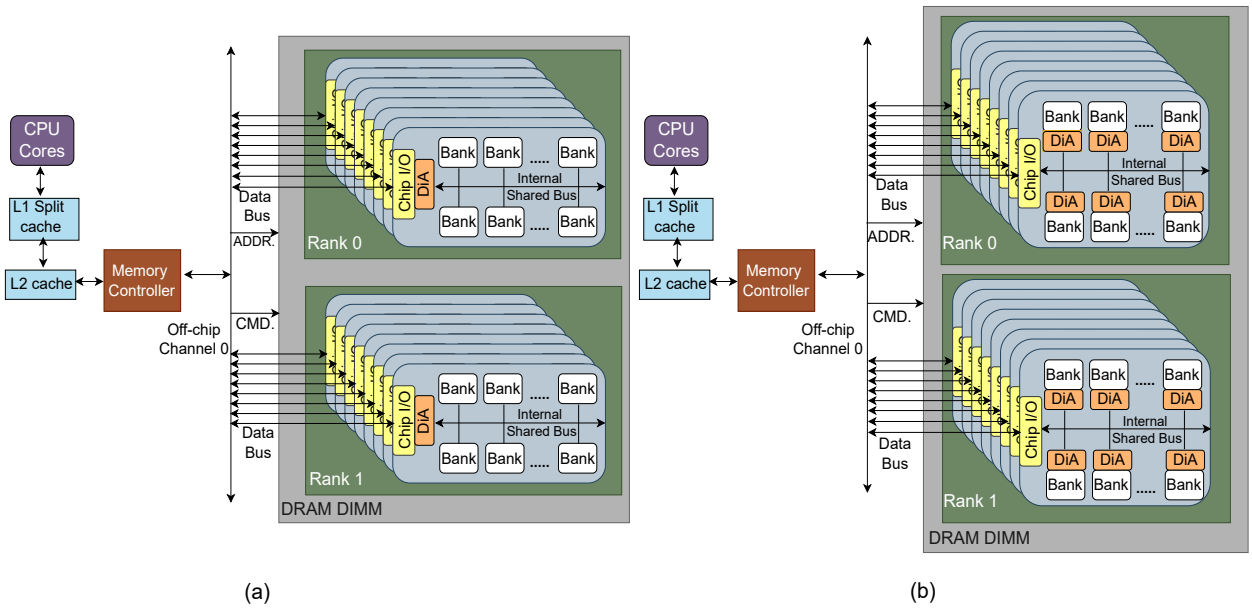


Figure 6.7: The proposed architecture of (a) Chip-level integration of DiA, and (b) bank-level integration of DiA.

4. We also employ two optimizations over the DiA module in the form of DiA-light1 and DiA-light2 and measure their impact on the system's throughput.

The design space exploration based on power consumption, area overhead, and performance, for all the hardware (DiA, DiA-light1, and DiA-light2) are measured with popular CNN workloads like VGG-16 [118], VGG-19 [118], ResNet-34 [48], and MobileNet [154] on ImageNet dataset [119]. The proposed system also outperforms the state-of-the-art works while accelerating inference.

6.2.2 System Architecture

6.2.2.1 The Proposed System Architectures

Figure 6.7 shows the abstract views of the proposed systems. The chip-level integration of the proposed hardware (DiA) is shown in Figure 6.7(a) while the bank-level integration is presented in Figure 6.7(b). Both the architectures include a quad-core host-processor with a two-level cache hierarchy (L1 split and L2 shared). To explore the design space impartially, we use a similar DRAM organization for both architectures. A single memory channel is equipped with the DIMM for both architectures. A rank is composed of multiple chips to increase the width of the memory channel. For the chip-level integration, we incorporate one DiA unit (shown in orange boxes in Figure 6.7) per chip, leading to 16 DiAs in total. Note that each DIMM has two ranks with 8 chips per rank (2Rx8). The DiA unit is connected

with the internal bus that is shared by all banks within a DRAM chip. The data (weights / activations) required for inference are fetched into the local buffers of DiA modules through the arbitration of the internal shared bus. In the bank-level integration, we put one DiA unit per bank, which leads to 32 DiA units in total as there are 16 banks per rank in our proposed dual-ranked DIMM. All the DiA units in both the architectures can execute the inference operations in parallel to the other. In the bank-level integration, one DiA unit is deployed near the global sense amplifiers or bank I/O of a bank, and the data regarding the inference are fetched using the global datalines. The architecture of bank-level integration has comparatively more processing units, DiAs, than the architecture of chip-level integration. Consequently, the system's performance in bank-level integration is also substantially high compared to the other one in exchange for additional area/power overhead. The detailed analysis of this trade-off is shown in the subsequent section.

6.2.2.2 Dataflow

Figure 6.8 presents the dataflow to exploit intra-chip/bank and inter-chip/bank parallelism in both the architectures. The data distribution and the granularity of execution are also shown in Figure 6.8. The inter-chip and inter-bank parallelism is shown in the upper portion of Figure 6.8(a) and Figure 6.8(b), respectively. Each input channel of a hidden layer is sent to one DiA unit of a chip or bank for the concurrent execution of the convolution (CONV)/pool operations. Note that all the DiA modules from both the architecture may not get an input channel to be processed in the initial inference layer. However, a batch of 16 channels at chip-level and a batch of 32 channels at bank-level can be processed simultaneously by the DiA units of both the architectures. Note that DRAM with a different number of chips or banks can have different degrees of parallelism with significant changes in the area/power overhead.

The intra-chip and intra-bank parallelism within one DiA unit is presented in the lower portion of Figure 6.8(a) and Figure 6.8(b), respectively. The 32 MAC units of each DiA can simultaneously execute the CONV operations at 32 neuron positions of an input channel, leading to intra-chip or intra-bank parallelism for the respective systems. The controller of DiA distributes the inference tasks among the MAC units based on the available neuron positions to be processed. A small streaming buffer is associated with each MAC unit. This streaming buffer holds the weight-activation pairs before they are processed by the corresponding MAC unit. The buffer filling latency is overlapped with the execution latency in the MAC unit, resulting in savings in the total execution cycles for inference. The DiA controller amortizes the number of fetch operations by bringing the data once from the local buffers (weight and activation buffer) and broadcasting them to all the appropriate

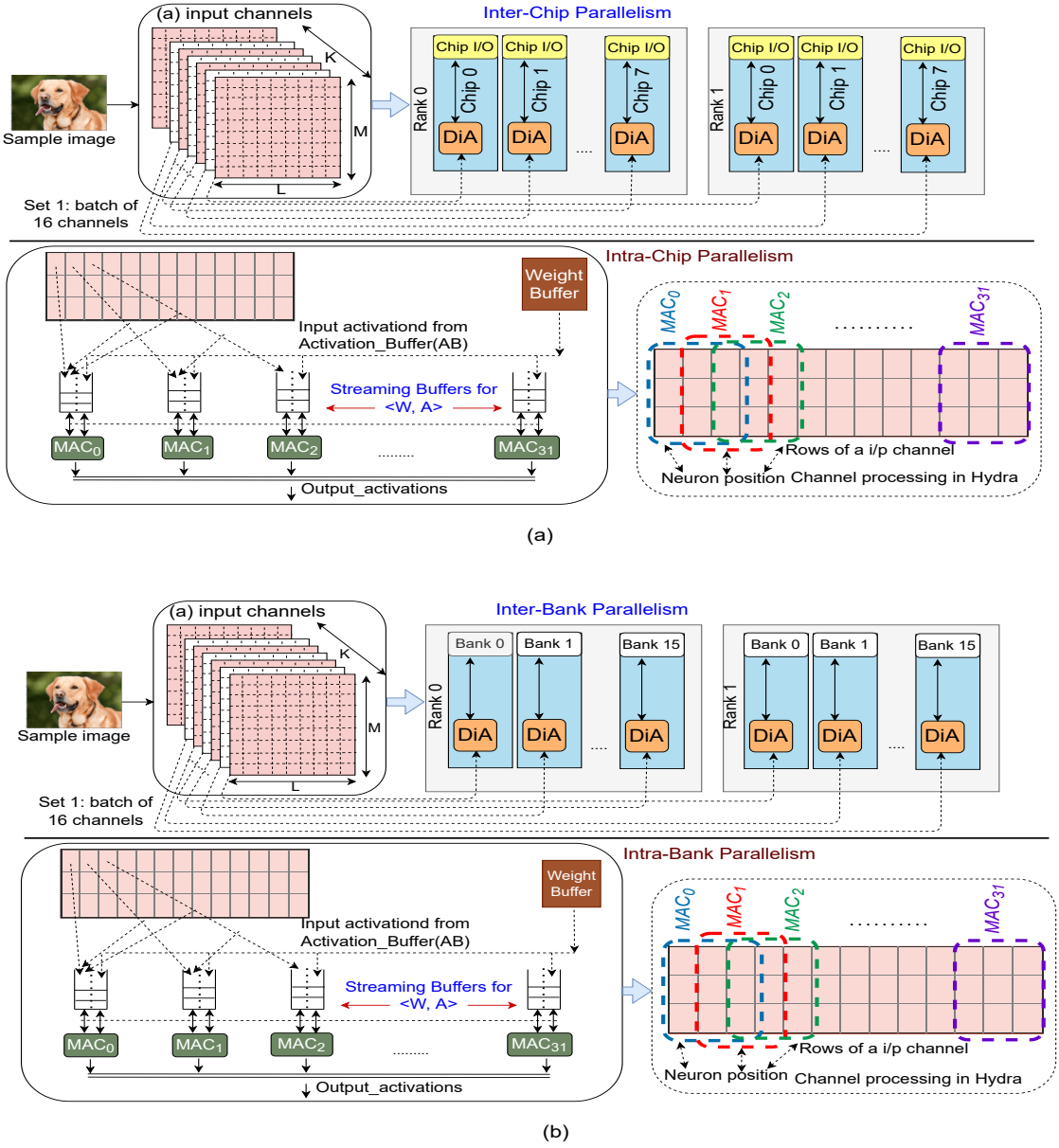


Figure 6.8: The dataflow in (a) chip-level integration of DiA units, and (b) bank-level integration of DiA modules to leverage intra-chip/bank and inter-chip/bank parallelism.

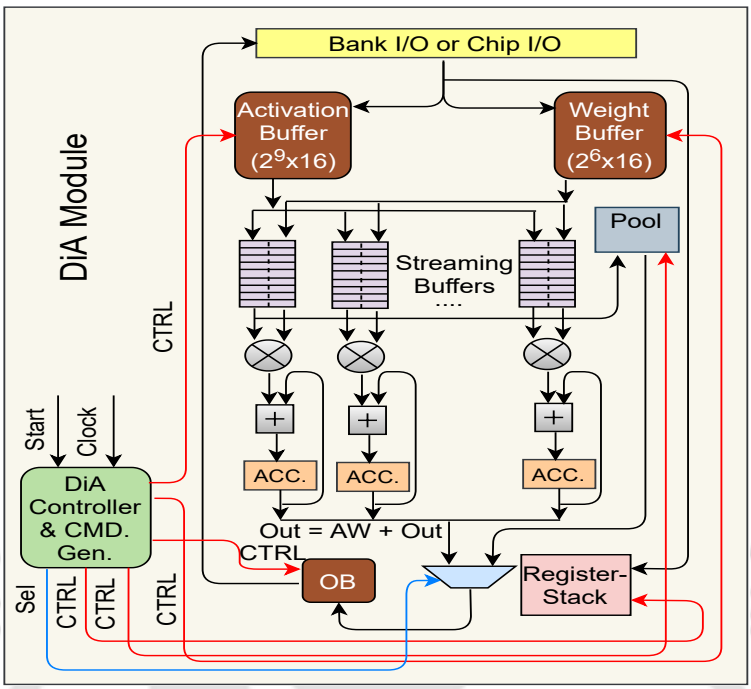


Figure 6.9: The proposed DiA unit.

streaming buffers during the filling of the streaming buffers with the appropriate weight-activation pairs, as shown in Figure 6.8. The performance of both systems is substantially improved due to the combined effect of intra-chip/bank and inter-chip/bank parallelism. Note that the dataflow works in a similar fashion for both the DiA-light1 and DiA-light2 with reduced intra-chip/bank parallelism because of the lesser number of MAC units (16).

6.2.2.3 The DiA Unit

An abstract micro-architecture of the proposed DiA unit is shown in Figure 6.9. The primary components of each module are weight buffer (WB), activation buffer (AB), DiA controller and command generator, 32 small streaming buffers, an output buffer (OB), register-stack, 32 mac units (16 in DiA-light1 and DiA-light2), and pool unit with 32 comparators. Additionally, the memory controller is also integrated with a result accumulation unit with an integrated linear function. The DiA controller can initiate the inference process once it receives the start signal from the host processor. The host processor offloads the meta-data information like hyper-parameters of the network to the register-stack, shown in Figure 6.9. The DiA controller plays a crucial role in driving all the modules to accomplish the inference tasks. Each channel of input maps and weights are loaded into the local AB and WB, respectively, by the DiA controller. To fetch the data into the AB and WB, the DiA controller uses the internal shared bus for the chip-level integration and global datalines for

bank-level integration. Similar to [2], the DiA controller possesses the capability of generating the basic READ/WRITE commands, leading to savings in the command bandwidth. The local buffers like AB and WB can be filled by using the WRITE command. The DiA is configured to work on 16-bit fixed-point (FX16) precision. However, we also evaluate DiA on low precision (INT8), as shown in the subsequent section. The DiA controller continuously loads the streaming buffers with the appropriate weight-activation pairs to be processed by the respective MACs. The bias register in the register-stack stores the bias value that is added only once at each neuron position. The results are stored in the output buffers of the DiA unit with the help of the DiA controller. Finally, The results from the OB are read by the memory controller with the help of READ commands for accumulation in the result accumulation unit. Note that, for simplicity and readability, we have not presented all the modules and connections in Figure 6.9. The DiA-light1 and DiA-light2 have a similar architecture to the DiA with minimal changes in the number of resources and precision.

6.2.3 Evaluation

To explore the design space for MAC-based accelerators like DiA near the DRAM, we conduct experiments on state-of-art CNN benchmarks with diverse shapes and sizes. We use the models like VGG-16 [118], VGG-19 [118], ResNet-34 [48], and MobileNet [154], pre-trained on ImageNet dataset [119]. We use PyTorch framework [133] to extract the data and network parameters. Table 6.3 lists the detailed configurations of the proposed architectures for both chip-level and bank-level integration of DiA's variants. The particulars of the architectures are explained in the following subsection.

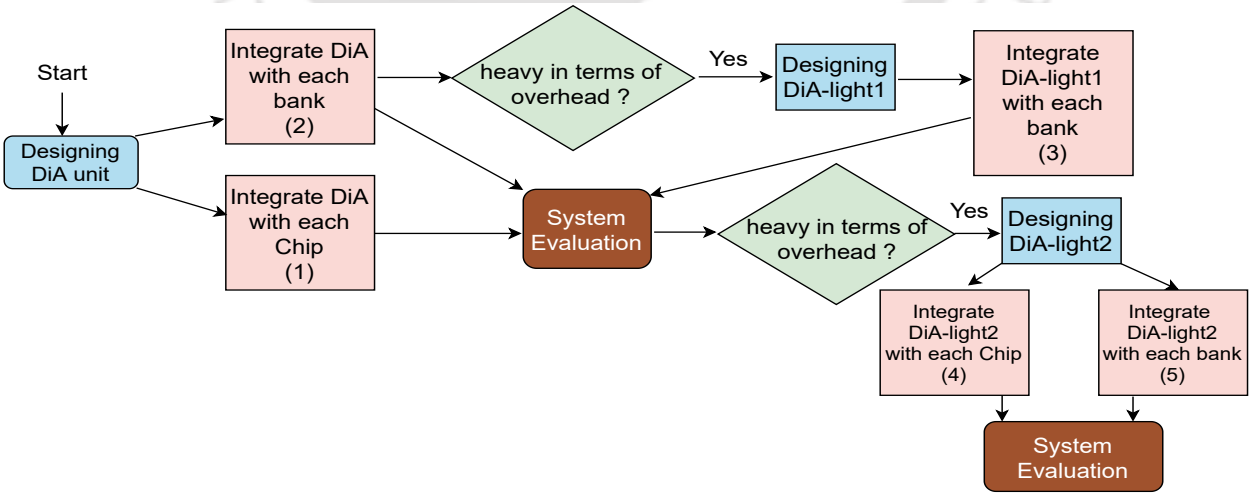


Figure 6.10: The steps followed for design space exploration.

Table 6.3: Specification of the architectures.

Host Processor (Common for all systems)	
Core	x86-64, 4 core, 4 GHz
Cache Memory (Common for all systems)	
L1 i-cache	32 KB, private, 4 way, 64 B blocks
L1 d-cache	32 KB, private, 8 way, 64 B blocks
L2 cache	256 KB, shared, 8 way, 64 B blocks
DRAM	
Timings [164]	t_{RP} = 5 cy, t_{CCD} = 4 cy, t_{RCD} = 5 cy, t_{CL} = 5 cy, t_{WL} = 4 cy, t_{WR} = 6 cy, t_{RTP} = 3 cy
Energy [164]	1.17 pJ/bit (array read), 0.39 pJ/bit (array write) 0.93 pJ/bit (buffer read), 1.02 pJ/bit (buffer write)
Configuration	8GB, 1 channel, 2 rank, 8 chip, 16 banks, FR-FCFS request scheduling
Proposed DiA architecture	
# of DiA units	16 (chip-level), 32 (bank-level)
# of MACs / DiA	32, 16 (DiA-light1 and DiA-light2)
Activation buffer size / DiA	1 KB
Weight buffer size / DiA	128 B
Power (15 *nm* CMOS tech.)	2.08 W (DiA@chip-level), 4.18 W (DiA@bank-level), 3.46 W (DiA-light1@bank-level), 1.62 W (DiA-light2@bank-level), and 0.81 W (DiA-light2@chip-level)
Area (total), 15 *nm* CMOS tech.	2.23 mm^2 (DiA@chip-level), 4.46 mm^2 (DiA@bank-level), 3.67 mm^2 (DiA-light1@bank-level), 1.98 mm^2 (DiA-light2@bank-level), and 0.99 mm^2 (DiA-light2@chip-level)
Frequency (15 *nm* CMOS tech.)	∼2.97 GHz (DiA), ∼3.3 GHz (DiA-light1), and ∼3.96 GHz (DiA-light2)
Precision	FX16 (DiA and DiA-light1), and INT8 (DiA-light2)

6.2.3.1 Particulars of Architectures

The steps that we follow to explore the design space (DSE) are shown in Figure 6.10. The DSE near the DRAM has been investigated with five architectures. The details of the proposed architectures are as follows.

1. **DiA at chip-level integration:** In this architecture, we integrate DiA units with each chip of a DIMM. In total, there are 16 DiA units that work in parallel.

2. **DiA at bank-level integration:** Here, we integrate the same DiA module with each bank, leading to 32 DiAs working in parallel.

3. **DiA-light1 at bank-level integration:** As 32 DiAs in bank-level integration increases area and power overhead, we optimize the DiA in this architecture by reducing the computational resources like MAC units. The 32 MACs in each DiA are reduced to 16 for area/power efficiency. However, the number processing elements (DiA-light1) and the place of integration are kept similar to the previous architecture.

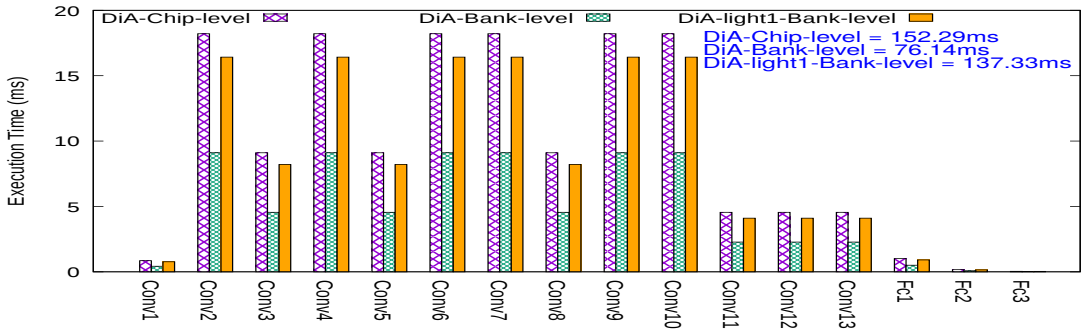
4. **DiA-light2 at chip-level integration:** To make the NMP design even lighter, DiA-light2 is designed. Here, the DiA-light1 is configured to work on low precision, specifically INT8. In this architecture, we integrate DiA-light2 units with each chip of a DIMM. Consequently, 16 DiA-light2 units operate in parallel to others.

5. **DiA-light2 at bank-level integration:** In this architecture, DiA-light2 is integrated with the individual banks of the DIMM. As a result, 32 DiA-light2 units work in parallel to others.

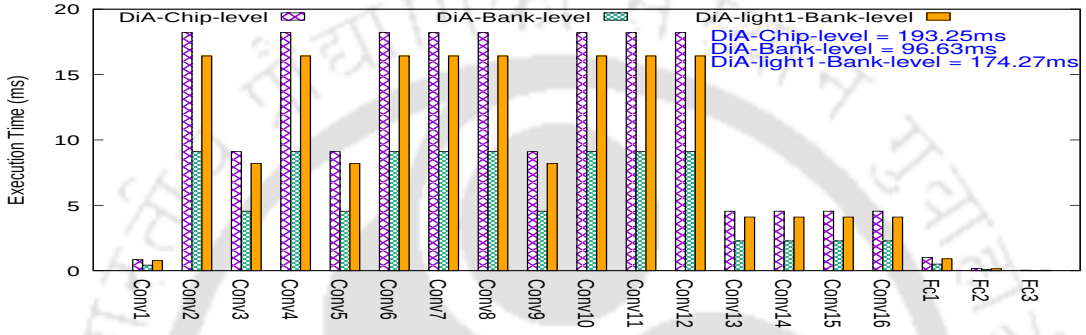
As DRAM industries are highly susceptible to the area/power overhead [37], comparing all of them in terms of performance, energy/power consumption, and area overhead provides comprehensive insights for the practical implementation of near-DRAM processing. Towards achieving the design trade-offs, we compare the architectures with similar precision. The design space exploration based on the performance, power consumption, and area overhead for low precision (INT8) hardware (DiA-light2) is done in a separate section (6.2.3.4).

6.2.3.2 Performance Analysis

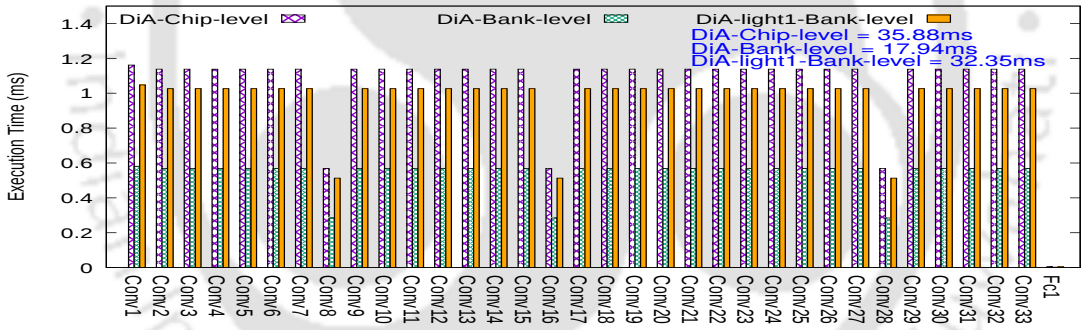
We design both the DiA and DiA-light1 hardware in Verilog hardware description language (HDL). The placement-aware logic synthesis has also been done on the designed hardware units using the Genus Synthesis Solution (version 15.21) from Cadence. We obtain an operating frequency of around 3-4GHz at 15nm technology for the variants of DiA units, as shown in Table 6.3. We develop an in-house cycle-accurate simulator that resembles the state machines of the design hardware units, DiA, DiA-light1, and DiA-light2. As the simulator is parameterizable, the post-synthesis frequencies of the hardware can be fed to the simulator. For each of the systems, the performance in terms of execution time (ms) of each network is measured from the simulator by using the corresponding state machine and the obtained frequency from the synthesis. The execution times for each layer of all the networks consider both the data processing time in the accelerator as well as the time to fetch/store data in the memory. In Table 6.3, we show the timing parameters to fetch/store data in DRAM. Figure 6.11 presents a comparison of layer-wise execution times for all four networks on the three architectures, namely DiA at chip-level integration, DiA at bank-level integration,



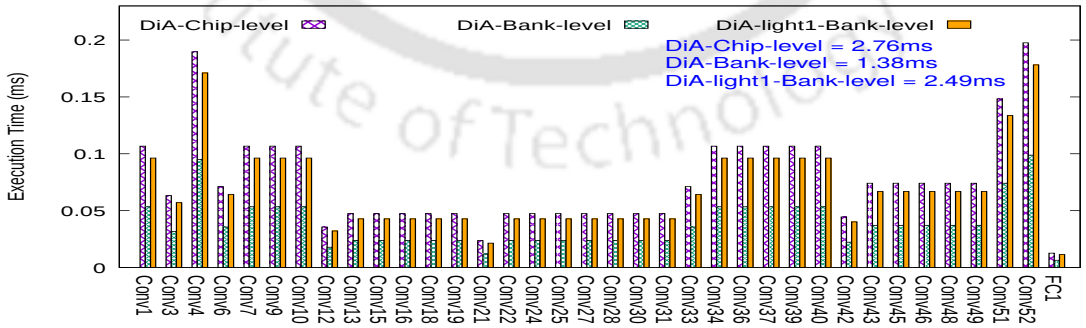
(a) VGG-16



(b) VGG-19

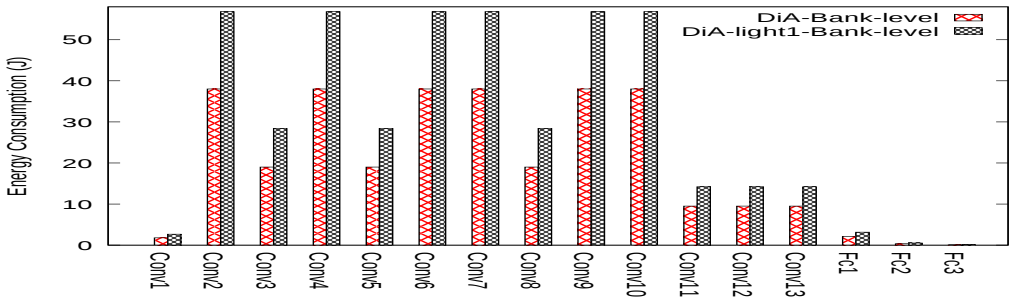


(c) ResNet-34

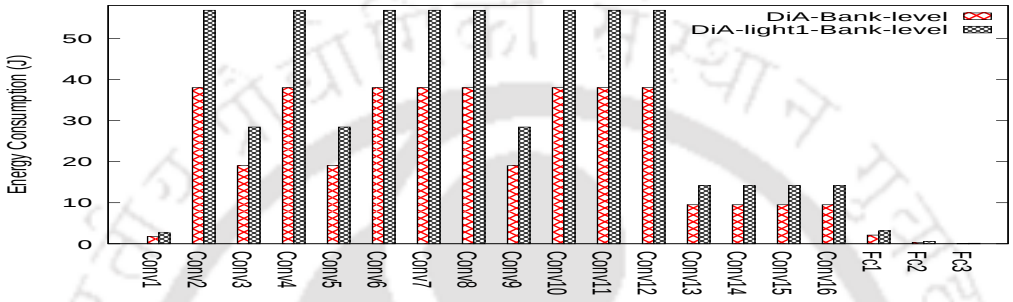


(d) MobileNet

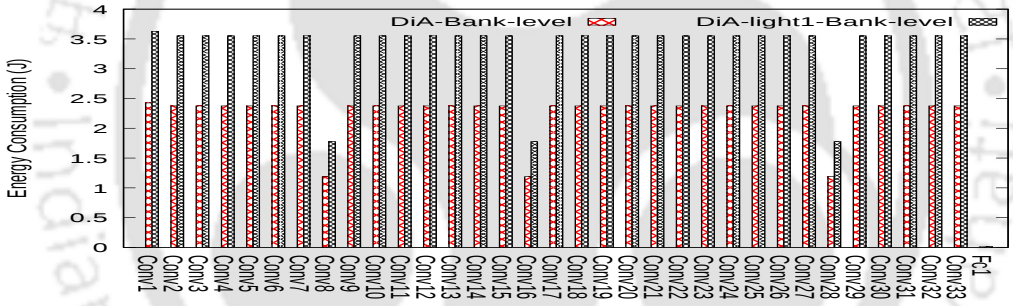
Figure 6.11: Performance of ConvNets/Networks.



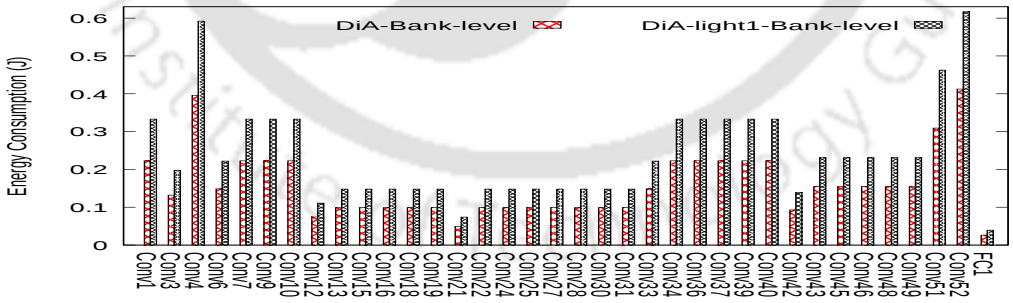
(a) VGG-16



(b) VGG-19



(c) ResNet-34



(d) MobileNet

Figure 6.12: Energy efficiency comparison of ConvNets/Networks.

and DiA-light1 at bank-level integration. Here, the Y-axis represents the execution time in milliseconds (ms), and the X-axis shows the layers of the networks. Note that we have not shown a few layers of MobileNet, as the values are significantly small to be represented in

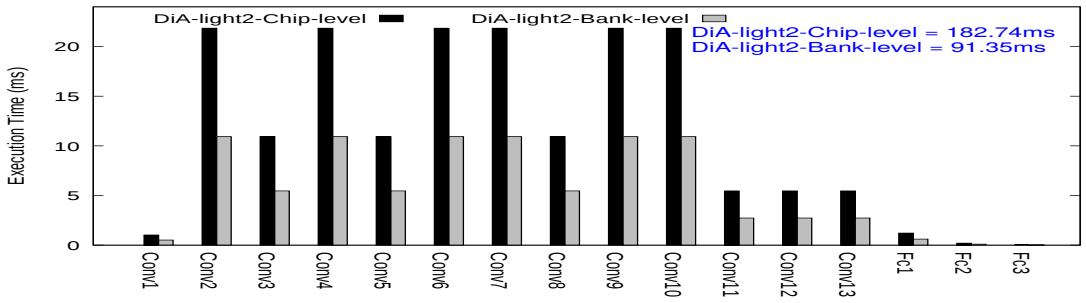
the graph compared to the other layers. On average, DiA at bank-level and DiA-light1 at bank-level integration achieves around 50% and 9.82% performance benefits, respectively, over the DiA at chip-level integration for all the networks. DiA at bank-level integration stands out to be the best for all the networks primarily because of the additional parallelism stemming out of 32 DiA units compared to the 16 in chip-level integration. The DiA-light1 at bank-level integration provides less performance than the DiA at the bank-level as the lighter version has fewer MAC units, specifically 16, as shown in Table 6.3. However, DiA-light1 provides better power/area efficiency due to lesser computing resources compared to DiA, as explained in the subsequent section. The total execution time (ms) per image in each architecture is also shown in Figure 6.11 for all the networks.

6.2.3.3 Energy Savings

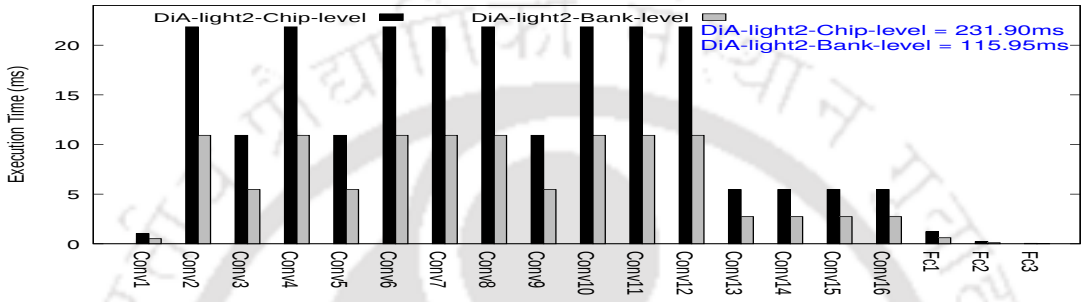
Apart from the performance, we also perform the energy analysis of the systems. The power consumption is measured after the synthesis of the hardware using Genus from the Cadence tool-set. At $15nm$ technology, we find the total power consumption for the additional logic is in the range of around $0.81W$-$4.18W$ (Shown in Table 6.3) for the different architectures proposed in this work. Figure 6.12 shows the comparison of energy efficiency across the layers of the networks for DiA at bank-level and DiA-light1 at bank-level integration. Note that we do not show the energy consumption for DiA at chip-level integration in Figure 6.12 as the energy consumption (joule) for DiA at chip-level and DiA at bank-level is almost similar. The energy consumption depends on two factors: (1) power consumption of the system and (2) the execution time of workload in the respective systems. Compared to the DiA at bank-level integration, the reduction in power consumption and the increase in the execution time of workload for the DiA at chip-level integration are of a similar order, leading to nearly equivalent energy consumption for both systems. The Y-axis in Figure 6.12 represents energy consumption (joule), and the X-axis shows the layers of the networks. On average, the DiA-light1 at bank-level is around 1.2x power efficient but 1.8x slow in overall execution latency, resulting in 1.5x less energy efficiency compared to DiA at bank-level integration.

6.2.3.4 Reduced-Precision Analysis

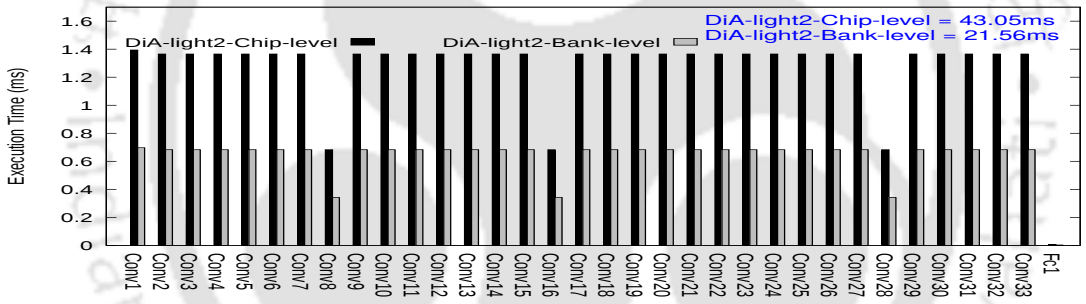
Towards implementing even lighter hardware than DiA-light1, we implement DiA-light2 that processes the data with reduced precision, specifically INT8. We integrate the DiA-light2 at chip-level as well as at bank-level to measure the efficacy of both architectures. Similar to the previous hardware, we implement the DiA-light2 in Verilog HDL and synthesize it



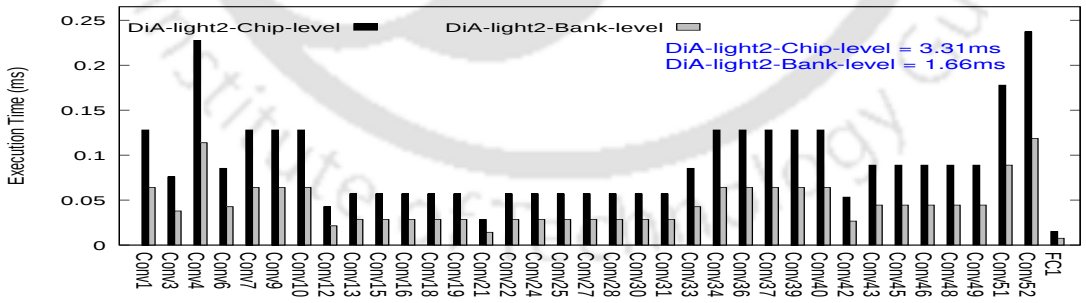
(a) VGG-16



(b) VGG-19



(c) ResNet-34



(d) MobileNet

Figure 6.13: Performance analysis on reduced precision (INT8) for the ConvNets/Networks.

using the Genus from Cadence tool-set at 15nm CMOS technology. Figure 6.13 shows the performance analysis of both the architectures at reduced precision for all four networks. The Y-axis represents the execution time in milliseconds, and the X-axis shows the layers of the

Table 6.4: Power breakups of one hardware unit ($15nm$).

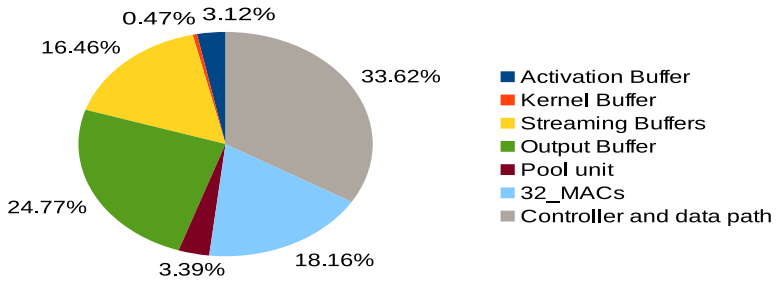
Modules	DiA (nW)	DiA-light1 (nW)	DiA-light2 (nW)
Activation Buff.	1.32E+06	1.45E+06	1.01E+06
Kernel Buff.	2.11E+05	2.34E+05	1.55E+05
Input Buffs.	8.09E+06	4.27E+06	2.85E+06
Output Buff.	1.16E+07	1.22E+07	8.17E+06
Pool unit	1.87E+06	9.55E+05	8.84E+05
MACs	1.48E+07	2.98E+06	1.54E+06
Controller and data path	9.26E+07	8.61E+07	3.60E+07
Total	**1.30E+08**	**1.08E+08**	**5.06E+07**

networks. For all the networks, the total execution time (ms) per image is also reported in Figure 6.13 for both the architectures. On average, the bank-level integration of DiA-light2 achieves a 2x speedup over the chip-level integration for all four networks. The speedup is obtained primarily because of the additional parallelism due to higher numbers (32) of DiA-light2 units, attached with each bank of a rank compared to the 16 DiA-light2 units of chip-level integration.

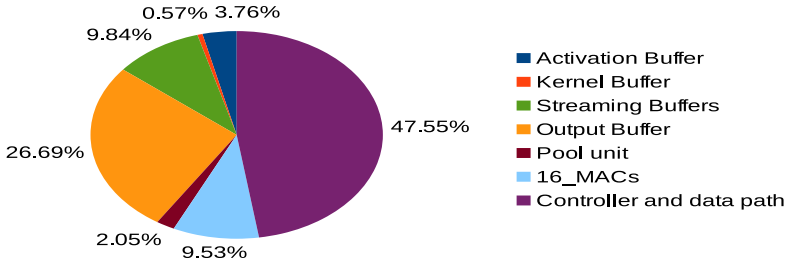
At $15nm$, the power breakups for one unit of each hardware are shown in Table 6.4. One DiA-light1 unit and one DiA-light2 unit is around 17.10 % and 61.20% power-efficient, respectively, compared to the DiA unit, as shown in Table 6.4.

6.2.3.5 Area Analysis

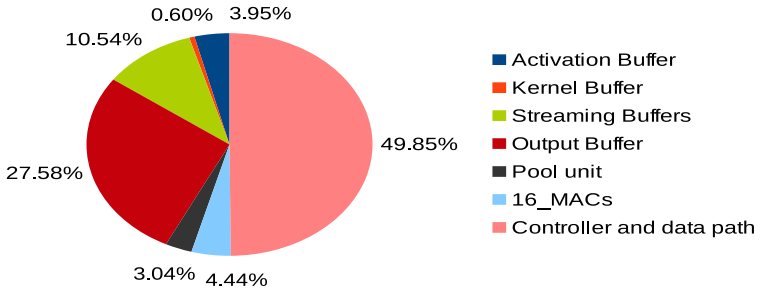
As the DRAM industries are extremely area-sensitive, a detailed area analysis for all the architectures is required for near-DRAM logic integration. We implement all three hardware, DiA, DiA-light1, and DiA-light2, in Verilog HDL. We then find the post-synthesis area estimates for all the hardware in Innovus from the cadence tool-set. The proposed hardware modules, DiA, DiA-light1, and DiA-light2 got placed on 11.81 μm $\times$ 11.81 μm, 10.71 μm $\times$ 10.71 μm, and 7.87 μm $\times$ 7.87 μm square boxes, respectively, at 15 nm technology. Figure 6.14 shows the area breakdown of each designed hardware. The controller and datapath consume the maximum percentage of area for all the hardware, as shown in Figure 6.14. Note that the percentage of area for MAC units has reduced significantly across the hardware due to the optimizations. The total area overhead (shown in Table 6.3) of DiA at chip-level, DiA at bank-level, and DiA-light2 at bank-level integration are 2.23 mm^2, 4.46 mm^2, and 3.67 mm^2, respectively, at 15 nm technology. The total area overhead has reduced to 1.98 mm^2, and 0.99 mm^2 for DiA-light2 at bank-level and DiA-light2 at chip-level integration, respectively, in similar technology nodes. Compared to the 80 mm^2 area of a DRAM [100], the maximum area overhead is around 5.56% for DiA at bank-level integration, and this



(a) DiA unit.



(b) DiA-light1 unit.



(c) DiA-light2 unit.

Figure 6.14: Area breakdown for each hardware.

overhead reduces to around 1.24% for DiA-light2 at chip-level integration.

Table 6.5 summarizes the whole design space explored in our experiments. The parameters of interest related to all five architectures are shown in Table 6.5. Note that the performance in terms of frames per second (FPS) can vary because of the different sizes and shapes of the networks used for the evaluation of the proposed systems. The maximum FPS is obtained for the MobileNet, while the minimum FPS is achieved in VGG-19.

Table 6.5: Summary of design space exploration.

Design Space	DiA @chip-level	DiA @bank-level	DiA-light1 @bank-level	DiA-light2 @chip-level	DiA-light2 @bank-level
Performance (FPS) (Min-Max)	5.2-362.32	10.35-724.64	5.74-401.61	4.31-302.11	8.62-602.41
Power (W), 15 nm	2.08	4.18	3.46	1.62	0.81
Area (mm^2), 15 nm	2.23	4.46	3.67	1.98	0.99
Place of integration	Chip-I/O	Bank-I/O	Bank-I/O	Chip-I/O	Bank-I/O
Number of units	16	32	32	16	32
Number of MACs/ unit	32	32	16	16	16
Precision	FX16	FX16	FX16	INT8	INT8

6.2.4 Comparison with Previous Accelerators

Among our proposed architectures, we choose the most aggressive DiA at bank-level integration and the lightest DiA-light2 at chip-level integration to compare them with two popular DRAM-based state-of-the-art inference accelerators, DrAcc [37], and DRISA [18]. The sub-arrays of DRAM are modified to exploit bit operation capability in these works [18, 37]. DrACC introduces special rows to perform NOT, XOR, and shift operations. The bank and sub-array dimensions of DRAM are reorganized for in-memory operations in DRISA. The concept of concurrently activating multiple rows to realize the in-DRAM operations are adapted in both works. Two major limitations in this concept are (1) high energy consumption stemming from the multi-row activations and (2) the process variation in the sense amplifiers. The process variation often causes logical operation failure. A 25% process variation can lead to around 26% failure in the logical operations as observed in [27]. Unlike DrAcc, DRISA addresses this issue by restructuring the sub-array that is different from the highly optimized commodity DRAM's sub-array. While both the works modify sub-arrays to exploit bit operation capability, we use the sub-array only for traditional load-store operations, leading to no sub-array-level modification for the proposed work. The entire computations are done in the separate modules, namely DiA, DiA-light1, and DiA-light2, integrated inside the DRAM's chip. Additionally, it is challenging to implement the multiplication operations with the in-DRAM processing. DrAcc skips the multiplications using ternary weight neural networks, while DRISA uses 1-bit weights for the inference. Though it is feasible to adopt quantized models to eliminate MULs in mobile/IoT devices, there is a non-negligible classification error of around 7% for ImageNet top-1 accuracy, between binary neural network [155] and the best case [156]. Instead of skipping multiplications, we perform them in our MAC units, making our architecture suitable for the requirement of high accuracy.

Table 6.6: Comparison with previous DRAM-based accelerators.

System's parameters		DrAcc	DRISA	DiA@bank-level	DiA-light2@chip-level
		FPS/ms	FPS/ms	FPS/ms	FPS/ms
Performance	VGG-16	0.3/3282	0.3/3283	13.13/76.14	5.47/182.74
	VGG-19	0.25/3933	0.25/3933	10.35/96.63	4.31/231.90
Power (W)		2	98	4.18	0.81
Area (mm^2)		0.01	65.2	4.46	0.99
Efficiency (Fr./watt)		0.15	0.0031	3.14	6.75

Table 6.6 summarizes the key design specifications for all the architectures. The reported results for the state-of-the-art works are obtained from DrAcc [37]. For all the systems shown in Table 6.6, we have shown the system's performance in terms of two metrics, frames per second and execution time (ms) for a single frame. Our proposed system outperforms both the DrAcc [37], and DRISA [18] in terms of performance, as shown in Table 6.6. DiA at bank-level integration provides around 41x-43x speedup over both the state-of-the-art, while DiA-light2 at chip-level integration delivers around 17x speedup over both DrAcc and DRISA. We have also shown the comparison of power and area overhead of the respective systems. Both the area and power overhead of DrAcc is lesser than the DiA at bank-level integration because of the additional MAC units used in our proposed system. However, the efficiency (frames/watt) of our proposed systems is substantially higher than both state-of-the-art. Note that, though DiA at bank-level integration provides the best performance, DiA-light2 at chip-level integration delivers better efficacy (Fr./watt) among the systems, shown in Table 6.6.

6.2.5 Summary

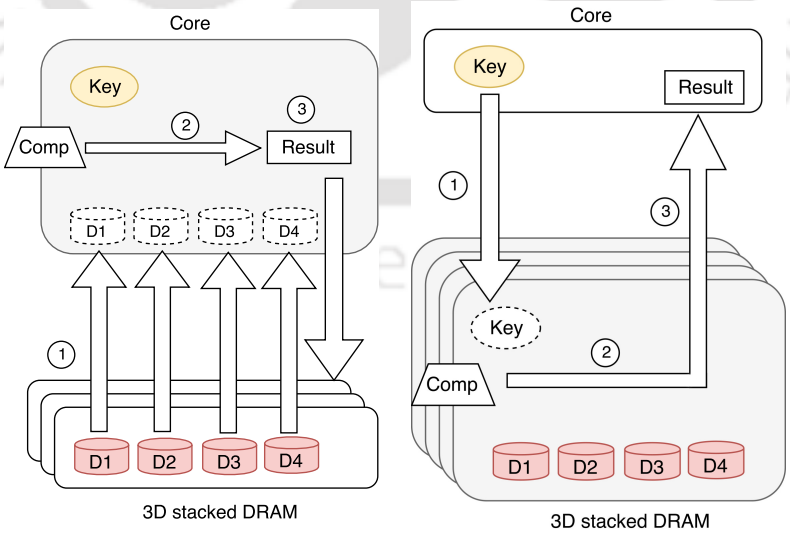
This section discusses the CNN accelerator, DiA, and its two optimized versions, DiA-light1 and DiA-light2. We integrate these hardware units with each chip or bank of a DRAM. We also explore intra-chip/bank and inter-chip/bank parallelism through our data partition scheme for the parallel execution of the inference tasks. These integrations help in studying the design space for near-DRAM logic integration. We evaluate the proposed systems based on their performance and energy efficiency. We also measure the additional power and area required for these integrations within the DRAM's strict area and power budget. Our proposed DRAM-based systems also outperform the prior DRAM-based inference accelerators in terms of the system's performance and efficiency. Additionally, our systems do not restrict the execution of non-quantize networks, unlike the previous works. At 15 nm CMOS technology, the area overhead of the proposed architectures can vary in the range of 1.24-5.56% based on the hardware and its place of integration. During the practical implementation of the near-DRAM inference accelerator, this study can be helpful in deciding the number of

hardware units, place of integration, and precision as per the design constraints.

6.3 Exploring NMP for basic Operations in Database

6.3.1 Introduction

As a continuation of our exploration of NMP's efficacy for other applications, we propose a custom design for popular database operations in this section. As databases are growing in size and real-time results are in demand, we can integrate logic closer to the memory rather than bringing contents closer to the CPU. To keep the NMP logic option small and generic, we explore the implementation of common search routines closer to the memory. Flexibility, being an issue with the application specific custom hardware, we have targeted widely used operations like *'compare-n-op'* which is primitive to many applications, typically in a database. The *'compare-n-op'* compares two parameters, and based on the result of the comparison, some additional simple operations like read, write, finding multiple occurrences, counting the number of occurrences, or finding the maximum value, etc., can be performed on a huge dataset. Almost similar operations have been targeted in [2]. Our work is also motivated by the use of hardware for the acceleration of the database operations as in [165]. However, they have followed an FPGA-based approach which is different from our NMP-based approach. In this work, we aim to exploit the high unused internal bandwidth of the hybrid memory cube (HMC) for *'compare-n-op'* on a huge amount of data with



(a) Conventional system. (b) NMP based system.

Figure 6.15: Conventional system vs. NMP based system.

minimal changes to the existing ecosystem of the HMC. Though our proposed model leverages minimum computing capability by keeping complex and unbounded operations on to the host processor, it helps the host processor by behaving like a slave to the master. The applications that comprise of compare operations on a huge dataset can be benefited substantially from this model. We have evaluated our system on the database applications as it comprises of maximum number of *'compare-n-op'*.

Figure 6.15 depicts two systems: (a) Conventional Systems, and (b) Systems having NMP capability. Conventional Systems include ① fetching chunk of data one by one from memory, ② comparing key with the fetched data, and ③ storing the results back to the memory. This process uses the external bandwidth heavily, with its heavy off-chip communication resulting in low system's throughput. However, the second approach needs ① sending the key to the memory side processing unit using off-chip connection, ② comparing the key with target data close to DRAM itself, and ③ sending the result back to the host processor for further high-level processing using the off-chip connection. The second approach is superior to the previous one in terms of system's throughput because of its highly reduced off-chip communications (only two in this case). The internal bandwidth being extremely high in a multi-vault HMC, the requirement of using external bandwidth can easily be compensated with its internal bandwidth, leading to high-performance computing. This section makes the following contributions:

- We use NMP to improve the performance of various *'compare-n-op'*s. We present the design of a near-data compare unit (NDCU) which performs *'compare-n-hit'*, *'compare-n-count'*, and *'compare-n-max'* operations near the memory.
- We have proposed two full-system NMP architectures. One has the benefits of NMP with no parallelism (NNP), while the other has vault-level parallelism with NMP (NVLP).
- The difference with existing state-of-the-art is the use of 3D-stacked memory environment, dedicated logic units, and exploiting vault-level data parallelism.

6.3.2 Background and Motivation

This section discusses contributions related to database operations near the memory. While talking about simple 2D DRAM, building PIM can broadly be classified into two categories. One is integrating logical units on top of the DRAM chip, while the second is modifying the DRAM itself to provide computational capabilities. NDA [100], Buffered Compares [2], and JAFAR [38] belong to the first category. In NDA, Coarse-Grained Reconfigurable

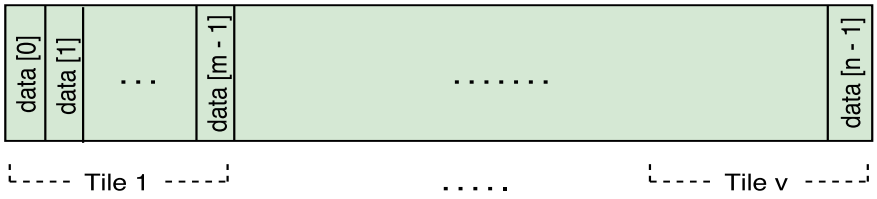


Figure 6.16: Tiled Data layout for parallel processing.

Architecture (CGRA) units are stacked on top of DRAM modules to perform near-memory computation. In [2], a Buffered Compare Unit (BCU) has been made for compare operation. In [38], select operations in databases are accelerated. JAFAR sends only qualifying data to the processor whenever a select request is sent by the host processor, reducing the traffic between processor and DRAM. Both these works have portrayed a similar approach, but their idea of NMP has been implemented with the traditional 2D DRAM. However, our approach of NMP has been implemented in a completely different setup of 3D memory, specifically, HMC [40], and it also exploits data parallelism. Experimental results show evidence of the fact that 3D memories are more desirable candidates for NMP.

6.3.2.1 Table-Scan

Table-scan is predominantly used in database operations. It is a critical operation where the processors have to waste a lot of unnecessary efforts while fetching the data for comparison. It can be performed on column-store databases (TSC) as well as on row-store databases (TSR). Table-scan is specifically efficient for column-store databases [166] as it restricts the fetching of unnecessary data. As a result, the cache hierarchy performs effectively in the case of TSC. In NMP, both TSC, as well as TSR databases can be used. Our proposed NDCU performs basic operations close to the data and sends results to the host processor making any type of data storage easy to use.

6.3.3 System Architecture and Operational Steps

The philosophy behind the design has got twofold advantages: (1) leveraging the novelty of NMP by placing the logical units as close to the data as possible (2) harnessing the hidden parallelism in the scan operation on huge datasets. To exploit (1), we place our NDCU on the HMC's logic layer, close to the data. Here, all memory references are managed by the host memory controller instead of individual vault controllers. This architecture is named NNP, i.e., NMP with no-parallelism. In order to exploit (2), i.e., data parallelism, we propose to place NDCUs with every vault. In particular, each NDCU will parallelly operate on

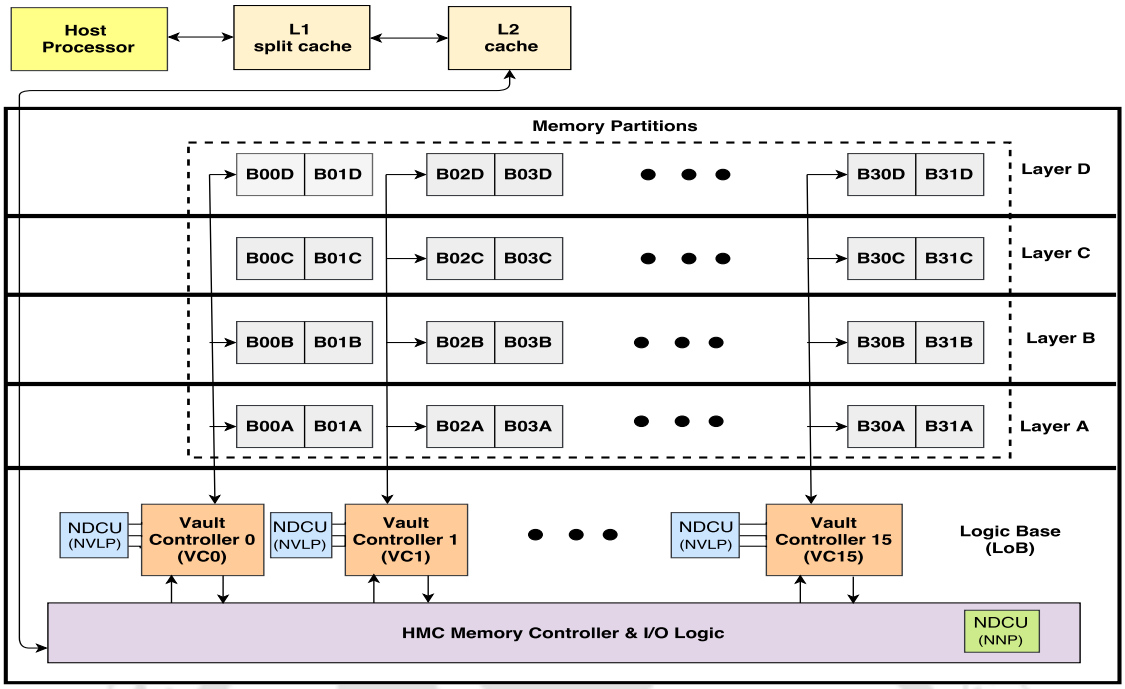


Figure 6.17: A conceptual view of proposed full system architectures (NNP and NVLP).

data (tile) from its corresponding vault. In our experimental setup, we have 16 vaults and, therefore 16 NDCUs. Memory references, in this case, are handled by the individual vault controllers. Our proposed idea is based on the observation of having inherent parallelism in the scan operation. While scanning huge datasets, the entire data can be partitioned into tiles (relatively small pieces of datasets) as shown in Figure 6.16. The entire dataset of size n has been divided into v number of tiles where each tile has the approximately same amount of data (size of m). Here, v is the number of vaults in the HMC. These tiles are distributed by the host processor over different vaults of the HMC module (1 tile / vault) for parallel scanning to obtain maximum speedup, leading to our proposed second architecture (NVLP). However, this maximum speedup cannot be obtained in cases when the tiles are mapped unevenly by the host across the vaults. In such cases, some NDCUs may take a longer time compared to others which will have an impact on achievable speedup. Figure

```
opcode 00: Compare_n_Hit (key, D[n]) {
        for i = 0 to n-1 {
            result = compare (D[ i ], key)
            if (result == "D[i] equals key") {
                hit reg = 1
                return } } }
```

```
opcode 01 : Compare_n_Count (key, D[ n ]) {
        for i = 0 to n-1 {
            result = compare (D[i], key)
            if (result == "D[i] equals key")
                count = count + 1
        } return }
```

```
opcode 10 : Compare_n_Max (key, D[n]) {
        item = key
        for i = 0 to n-1 {
            result = compare (D[i], item)
            if (result == "D[i] greater than item")
                item = D[i]
        } return }
```

Figure 6.18: Specialized pseudocode of *'compare-n-op'*.

6.17 shows both the architectures: NNP and NVLP. In NNP architecture, the system has only 1 NDCU component (marked in green), whereas, in NVLP, the system has 16 NDCUs (marked in blue). This logic unit (NDCU) has the capability of scanning both row-store and column-store databases. Additionally, it can perform basic arithmetic operations while performing the following three variants of *'compare-n-op'*. The pseudocodes for all the three operations, shown in Figure 6.18, are explained below.

- The *'compare-n-hit'* operation is used to search a key, sent by the host processor, on an attribute of the schema. If that key is found in the target data (D[i]), the NDCU informs the host processor about the presence of the corresponding key by setting the hit register.
- In *'compare-n-count'* operation, the number of occurrences of a particular key (sent by the host processor) in the target data (D[i]), is to be counted. The result is sent to the host processor for high-level processing.
- In *'compare-n-max'*, the maximum value (can be found in *'item'* register) of any attribute of the schema is sent to the host processor.

All these operations can be executed on both row-store and column-store databases. The generalized *'compare-n-op'* function becomes a specialized one by the respective opcodes sent by the host processor. The 2-bit opcodes for hit, count, and max operations are shown in Figure 6.18. NDCU remains in inactive mode when opcode is set to '11'. Initially, the host processor sets the opcode based on its requirement and sends the key to be processed, along with some additional information, like, initial address and range of data to be processed, to the NDCU. Upon offloading the workload to the slave NDCUs, the host processor becomes free to start some other works. This, indeed, increases the parallelism, and thereby the throughput of the proposed system becomes high.

6.3.3.1 Operational Steps

The abstract data-path of the NDCU is shown in Figure 6.19. NDCU controller is the centroid of the entire data path. For simplicity, all the communication channels between any two units in the data path are shown with a single connector. After the processor's intervention, the NDCU controller receives all the relevant information like opcode, initial data address, range of data to be processed, etc., from the host processor and stores it in the register stack for further processing. In the next step, the operand register is assigned with the key sent by the host processor. With the help of the vault controller in the case of NVLP or the HMC host memory controller in case of NNP, the data items are brought by the NDCU controller one by one to the data register for *'compare-n-op'*. The ALU acts as

a processing element that compares the values. Based on the ALU output and the opcode given by the processor, the NDCU controller invokes the corresponding function. Depending on the function to be performed and the ALU output, the NDCU controller sets the hit register for hit operation, whereas it updates the count register in case of count operation. For max operation, the NDCU controller transfers the content of the data register to the operand register if the item (the content of the operand register) is less than the value of the data register. NDCU controller has separate connections for sending reset and write signals to all the registers of the entire data path, which are not shown in Figure 6.19 for simplicity.

6.3.4 Experimental Evaluation

We have chosen database application because it comprises of significant amount of *'compare-n-op'* which justifies the efficacy of the proposed system as well as demonstrates its usefulness.

6.3.4.1 Target Applications

Table scan - column store (TSC): Table-scan is having benefits over index-based database operations in terms of storage, as storing indexes needs extra space on the disk. Specifically, in the context of column-store databases where the data is stored in column-wise format, scan is often used. However, it only fetches the necessary information while saving the costly memory bandwidth. We have implemented all the variants of *'compare-n-op'* and tested on the *star schema benchmark* [167]. This benchmark is very efficient in measuring the performance of database products. The *'PARTKEY'* attribute of the 'lineorder' fact table is used for *'compare-n-hit'*, *'SUPPLYCOST'* is used for *'compare-n-max'*, and *'QUANTITY'* is used for *'compare-n-count'* operation.

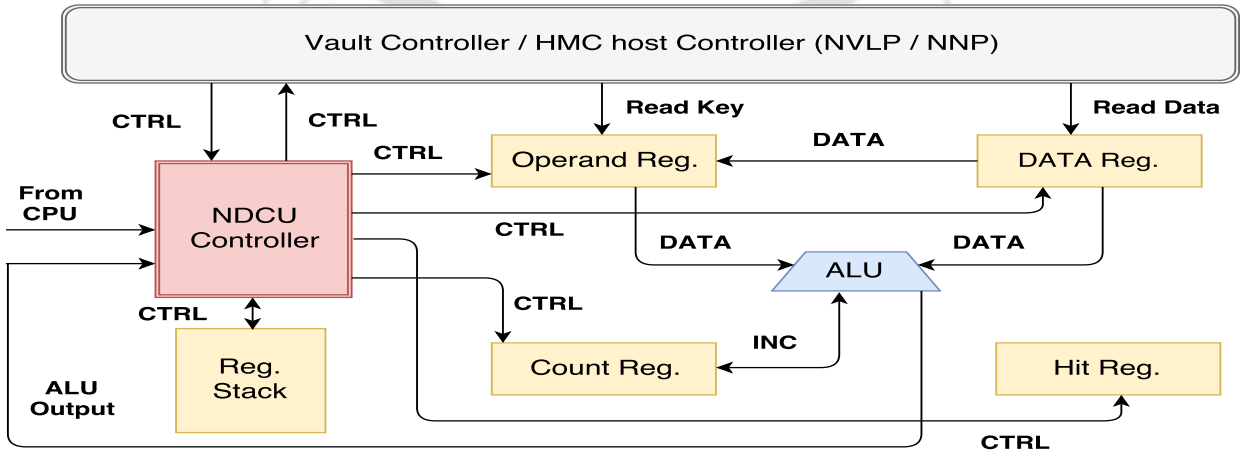


Figure 6.19: Abstract design of NDCU.

Table scan - row store (TSR): The operations same as those in TSC are performed on the same attributes in this case as well. However, the data is stored in row-wise format here. Row-store databases are less efficient as CPU-based processing units fetch unnecessary data leading to wastage of memory bandwidth. From the result section, it can be verified that our architectures (NNP and NVLP) perform much better in the case of TSRs. The reason for this improvement is that in row-store databases, data (target attribute of a schema) does not remain in consecutive memory locations. As a result, cache hierarchy in the conventional CPU-based system cannot deliver performance due to a lack of spatial locality.

All three operations mentioned above have been performed in the typical column-store (TSC) and row-store (TSR) database, leading to the workloads as shown in Table 6.7. During the search operation on a given key-value pair, an item (to be searched) may be found anywhere. To quantify the performance improvement, we categorize these searches. In particular, we model three cases: (i) Hit_TSx_best: when the item is found in the first 25% of the search space, (ii) Hit_TSx_avg: when the item is found within 25%-50% of the search space, and (iii) Hit_TSx_worst: when the item is found after 75% of the search space (where TSx represents both column and row store database).

Table 6.7: Workload Details.

Workloads	Details	Workloads	Details
Hit_TSC_best	Best case hit operation	Hit_TSR_worst	Worst case hit operation
Hit_TSC_avg	Average case hit operation	Count_TSC	Count on column-store database
Hit_TSC_worst	Worst case hit operation	Count_TSR	Count on row-store database
Hit_TSR_best	Best case hit operation	Max_TSC	Finding maximum value
Hit_TSR_avg	Average case hit operation	Max_TSR	Finding maximum value

Table 6.8: Specification of the simulation setup.

Host Processor [GEM5]	
Core	Single core, x86-64, 2 GHz
L1 i-cache	32 KB, private, 4 way associative
L1 d-cache	32 KB, private, 8 way associative
L2 cache	256 KB, shared, 8 way associative
Main Memory	2 GB, DDR3_1600_x64
Proposed logic unit for NMP (NDCU)	
NDCU	1 for NNP, 16 for NVLP. Each has 1 ALU.
Frequency	950MHz on UMC 90 *nm* technology
3D Memory Stack (HMC)	
Timing Parameters	t_{RP} = 7.7ns, tt_{CCD} = 3.3ns, t_{RCD} = 10.2ns, t_{CL} = 9.9ns, t_{WR} = 15ns, t_{RAS} = 21.6ns, t_{CK} = 0.8ns
Energy [40]	3.7 pj/bit for DRAM read, 6.78 pj/bit for SerDes hop
Power[132]	11.08W
Size	2 GB
Logic Die	90 *nm*, dimension 27 X 27 mm^2

6.3.4.2 Baseline CPU based System

We have used Gem5 [168] to model our baseline system. Gem5 is a cycle-accurate simulator that is being extensively used for the full system simulation. We have configured our simulation setup with an x86-64bit processing core having a working frequency of 2 GHz. A two-level cache hierarchy (private L1 and Shared L2) has been implemented with DRAM-based main memory system for full system simulation. The energy modeling of the core has been done using McPAT 1.3 [134], and CACTI 6.5 [169] has been used to obtain the energy consumption of the main memory system.

6.3.4.3 NDCU hardware

NDCU is dedicated hardware, designed using Verilog hardware description language. After the initial design, it has been synthesized by the Genus Synthesis solution from Cadence. We have used UMC 90 *nm* technology library, and the obtained operating frequency of NDCU is around 950 MHz. This NDCU can be placed in the logic layer of HMC. Specification parameters for both baseline and proposed systems are given in Table 6.8.

6.3.4.4 Results

We have obtained motivating results in terms of performance and energy, which instigate the concept of near-memory processing. Area, being a critical aspect for integrating logic on the same memory chip, we have analyzed the area overhead for practical implementation of the proposed architectures.

Performance: For both the architectures (NNP and NVLP), we get improvement in the execution time/speedup over the conventional system for all the workloads. While the

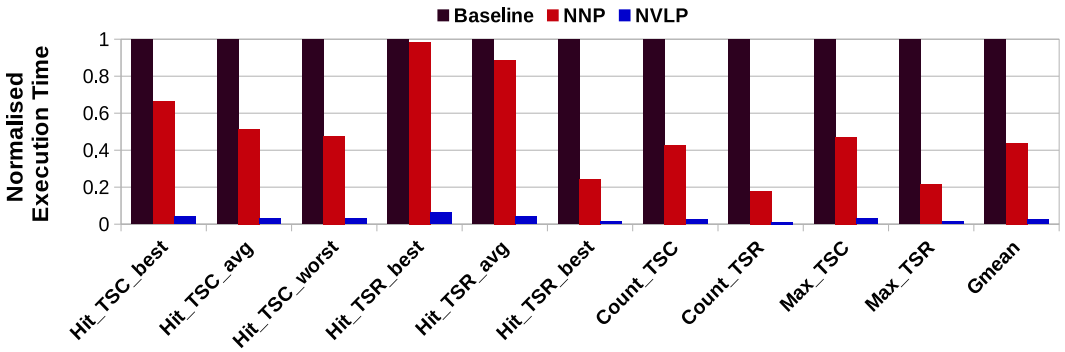
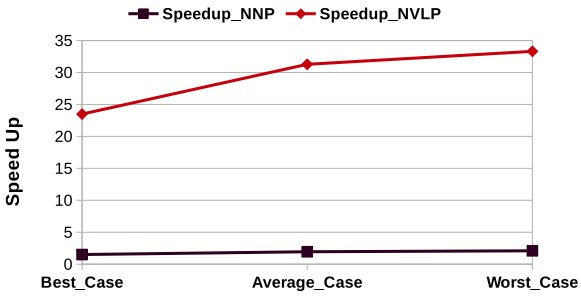
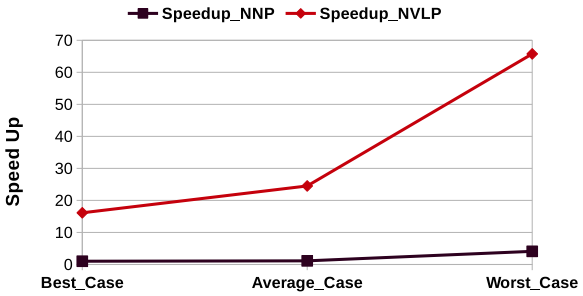


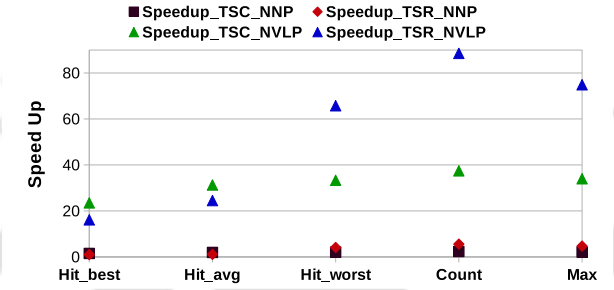
Figure 6.20: Performance comparison of proposed NMP architecture; normalized w.r.t baseline CPU based model.



(a) Speed up comparison for TSC.



(b) Speed up comparison for TSR.



(c) Performance comparison of TSC vs TSR.

Figure 6.21: Analysis of the proposed system performance on typical cases.

NVLP is extremely fast due to its massively parallel architecture, NNP proves its significance in terms of reduced power consumption with negligible area overhead and improved performance compared to baseline. From Figure 6.20, we can see substantial improvement in the performance of all the workloads for both NNP and NVLP architectures. We have evaluated the hit operation for best, average, and worst-case scenarios. Here, the execution time of NNP and NVLP architecture includes both the computation time as well as the data fetching time from the DRAM die to the logic layer. Similarly, in the case of baseline, the execution time comprises the computation time of the CPU and the data fetching time through the conventional memory hierarchy.

The novelty of our approach is that the speedup (in TSC as shown in Figure 6.21a and TSR as shown in Figure 6.21b) is monotonically increasing from the best case to the worst case for both NNP and NVLP architectures. The reason behind this is that, with the increasing number of *'compare-n-op'*, the off-chip communications also increase. The increased off-chip communications pull down the performance for the conventional CPU-based system. However, our NMP-based approach works near the memory. As a result, the effect of costly off-chip communication is avoided, and high internal bandwidth utilization is obtained.

Another important observation is that our proposed approach is doing well under typical data layouts. If we observe Figure 6.21c, we can see that the performance of TSR has

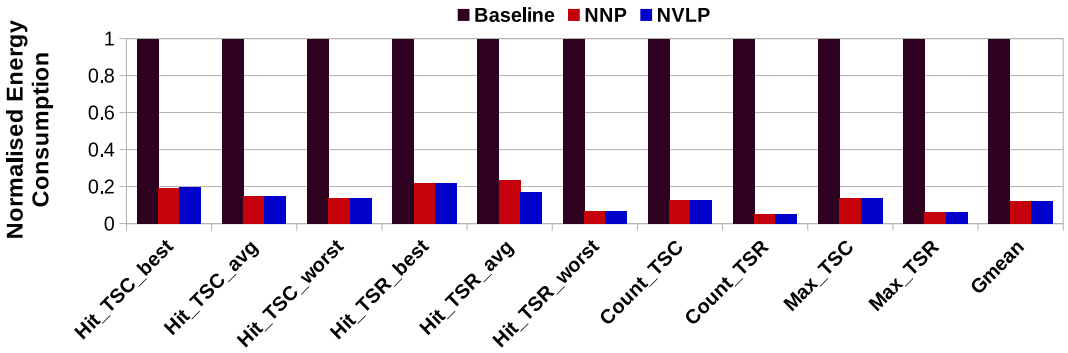


Figure 6.22: Reduction in energy consumption w.r.t baseline CPU based model.

dominated the performance of TSC both for NNP and NVLP architectures with the increasing number of *'compare-n-op'*. Initially, TSC has not been dominated by TSR because the number of *'compare-n-op'* was less. However, with the increasing dataset, the scenario has changed. Conventional CPU-based system suffers from row-store databases. The reason for this is that unnecessary data has to be fetched by the CPU in row-store databases, which not only wastes the memory bandwidth but also increases the off-chip communication. Even in this typical scenario, both the NNP and NVLP architectures leverage the benefits of NMP and minimize the costly off-chip communication. On average, we obtained around 2.3x speedup in NNP and around 37x speed up in NVLP over the baseline.

Energy Savings: Power analysis of baseline has been done using McPAT and CACTI. We analyze the power consumption of the proposed NDCU units in the Genus Synthesis solution from Cadence. Using this value, we analyze the total power consumption of the two proposed systems (NNP and NVLP). Energy consumption depends on the execution time as well as power consumption. Though NNP architecture is more power-efficient, the energy consumption is almost the same as NVLP because the execution time of NNP architecture is longer than NVLP architecture. In both the proposed models, we obtained substantially better results in terms of energy consumption over the baseline. The comparison of energy consumption between the baseline and proposed models is shown in Figure 6.22. On average, we obtained around 8x saving in energy in both NNP and NVLP architecture with respect to the baseline.

Area analysis: Since memory industries are extremely area sensitive, the area overhead of the logical units should be analyzed thoroughly. We perform a detailed area analysis on the NDCU net-list, generated after the post-synthesis simulation done by the Genus. We use Innovus (from Cadence tool-set) and our net-list got placed on 0.18 mm × 0.18 mm square box at 90 nm technology. In NNP architecture, this incorporated the area overhead of 0.032 mm^2, which is only 0.004% of the logic die area, which is negligible. In the case of NVLP

architecture, the total area overhead due to multiple NDCUs is 0.52 mm^2, which is 0.07% of the entire logic die. Even this area overhead is quite reasonable considering its overall throughput.

6.3.5 Summary

Scan or compare operations being primitive in nature can be used in a wide variety of applications. The inherent parallelism in scan operation over the tiled data layout has made it a lucrative candidate for near-memory processing. In this section, we discuss our proposed design of the NDCU hardware unit, dedicatedly built for memory scan operation, while performing some additional tasks like hit, count, and max. We have designed two full system architectures (NNP and data-parallel NVLP) to harness the benefits of NMP. Both have substantial improvements over the CPU-based baseline in terms of performance and energy consumption with a negligible area overhead. While the system can achieve 2.3x and 37x speedup in NNP and NVLP architectures, respectively, it reduces the energy consumption by 8x, on average, for both the architectures. The results of this research encourage the use of NMP for data-parallel applications, like table-scan, by the use of dedicated hardware rather than fully functional processors.

6.4 Summary of the Chapter

This chapter[1] has two primary goals: (1) Estimating NMP's efficacy with various memory technologies like DRAM and hybrid main memory apart from the 3D memory, (2) Estimating NMP's efficacy with other data-intensive operations, namely database operations.

The first two sections illustrate the descriptions related to the first goal. These works perform experiments on DRAM-PCM-based hybrid memory and conventional DRAM, respectively, while accelerating inference through NMP-based accelerated architectures. The last section presents the experiments on accelerating database operations near the 3D stacked HMC memory. The performance and energy efficiency of the proposed systems are substantially better compared to the respective baselines and state-of-the-art. The variety of experiments performed in this chapter manifest NMP's efficiency in a more comprehensive manner for practical implementations.

[1]The publication details related to this chapter are as follows.
(1) Das, Palash, Ajay Joshi, and Hemangee K. Kapoor. "Hydra: A near hybrid memory accelerator for CNN inference" Design, Automation and Test in Europe Conference, DATE 2022 (accepted).
(2) Das, Palash, and Hemangee K. Kapoor. "Towards near-data processing of compare operations in 3D-stacked memory." Proceedings of the 2018 on Great Lakes Symposium on VLSI. 2018.

भारतीय प्रौद्योगिकी संस्थान गुवाहाटी
Indian Institute of Technology Guwahati

7
Conclusion

In this thesis, we aim to design accelerated architectures having the additional capability of near-memory processing (NMP). While designing the accelerators, we primarily focus on the widely used CNN algorithm. The reason is: building such a system can be beneficial for a wide range of emerging applications. However, another goal of this thesis is to verify the effectiveness of NMP capability. To manifest NMP's efficacy, (1) we integrate our accelerators with various memory technologies, and (2) we have also changed our target application to widely used primitive operations, namely database operations. From our experiments, we can conclude that the NMP-based accelerated architectures are able to provide high performance and energy savings in exchange for minimal additional area overhead. Below we briefly present the summary of our four contributions to this thesis.

7.1 Summary of contributions

1. **Exploiting parallelism and NMP:** In the first contribution, we have exploited the parallelism of the CNN algorithm through our proposed dataflow. The proposed hardware, Convolutional Logic Unit (CLU), is designed in such a way that it can implement the proposed dataflow. We integrate multiple CLU modules in the logic layer of HMC memory to leverage the NMP's benefits while accelerating the inference. The multiple CLU and its MAC units can work in parallel to execute the inference tasks, leading to intra- and inter- vault parallelisms. The proposed system has substantially improved performance and energy savings compared to quad-core CPU-based, 64 core CPU-based, and GPU-based baseline with an area overhead of 2.37%. The proposed CLU-based system is also comparable with other existing NMP-based architectures and outperforms them in terms of performance, power consumption, and efficiency.

2. **Exploiting Sparsity of data, NMP, and parallelism:** In the second contribution, we include the additional capability to the designed hardware, Near-3D-memory

Zero Skipping Parallel Accelerator (nZESPA), for accelerating the inference phase. The nZESPA units can exploit data sparsity in both the weights and activations to eliminate ineffectual zero-valued computations, leading to further improvements in performance and energy savings. The contribution deals with compressed/encoded input/output data format. The grids of multiple nZESPA modules are integrated into the logic layer of HMC using the NMP concept, leading to the exploitation of both intra- and inter-vault parallelism. In comparison with the systems that either do not exploit sparsity (NMP-dense) or do not employ NMP (traditional-fully-sparse) or do not include both (traditional-dense), the proposed system has achieved significant gain in both performance and energy saving in exchange for an area overhead of 5.61%.

3. **Exploiting redundant computations, NMP, and parallelism:** In the third contribution, we observe that a substantial amount of computations in inference are redundant and can be eliminated by a lookaside memory (LAM)-based search technique. We design custom hardware, namely Adaptive LookAside Memory based Near-memory Inference engine (ALAMNI) that has integrated LAM modules with the MAC units. The LAMs store the most frequently used weight-activation pairs along with their multiplication results. The ALAMNI controller consults these LAMs for the hits of weight-activation pairs. Upon a hit, the controller skips the costly multiplications with the help of precomputed results in the LAMs during inference, leading to significant improvements in performance and energy savings. We also incorporate an optional bitmasking concept to increase the hit percentage in the LAMs to amortize more computations with an acceptable loss of accuracy. The proposal was extensively validated on real data from CIFAR-10 and ImageNet using the PyTorch framework to find a design point based on the LAM size, hit rate, and accuracy loss. The multiple ALAMNI units are integrated into the logic layer of HMC to harness the benefits of NMP and intra/inter-vault parallelism. Experimental results show a significant improvement in the systems performance and energy efficiency compared to the baseline and state-of-the-art at the expense of around 1% area overhead at 15nm technology.

4. **Exploiting other avenues for NMP processing:** In the fourth contribution, we explore other avenues like (1) Estimating NMP's efficacy with other prominent memory technologies such as conventional DRAM and DRAM-PCM hybrid main memory, (2) Measuring NMP's efficiency with another popular data-intensive operation, namely database operation.

   While integrating custom inference hardware, namely near hybrid memory accelerator (Hydra), near the DRAM-PCM hybrid main memory, we explore intra- and inter-

chip parallelism. The Hydra uses DRAM for intermediate writes while keeping the heavier models inside the PCM. This approach increases the PCM's lifetime as the parameters are only read from the PCM's banks during the inference. The fetching of parameters can also be overlapped with the conventional inference task execution, leading to improved performance for the overall system. The proposed system with integrated Hydra obtains around 20x performance improvements over the previous in-DRAM processing-based works inference. Additionally, unlike its prior works, this system does not eliminate any multiplication operations by using binary or ternary neural networks, making it suitable for quantized and non-quantized networks.

We have also integrated the custom accelerators inside the DRAM's chip for efficient inference execution. We have explored chip-level and bank-level integration in this context. For DRAM, we design dedicated hardware, namely near DRAM inference accelerator (DiA), and integrate multiple such instances inside the DRAM's chip. The proposed design can harness both intra- and inter-chip/bank parallelism while accelerating the inference. As there is a strict area/power budget for additional logic integration in DRAM, we implement multiple variants of DiA having different amounts of resources and precisions. For various proposed architectures, we perform a design-space exploration with these hardware modules and their place of integrations to provide an estimate of performance, power consumption, and area overhead. The proposed architectures are found to be more efficient in terms of performance and energy consumption compared to the state-of-the-art works. Unlike the prior works, the proposed architecture supports the execution of both quantized and non-quantized networks.

Towards measuring the NMP's efficacy with another data-intensive application, we accelerate database operations in the logic layer of the HMC device. We implement a custom hardware, namely near-data compare unit (NDCU), that can accelerate *'compare-n-hit'*, *'compare-n-count'*, and *'compare-n-max'* closer to the memory. In this work, we propose two architectures, one is lighter NMP with no parallelism (NNP), and the second is NMP with vault level parallelism (NVLP). Compared to conventional CPU-based processing, the proposed systems obtain significant performance and energy efficiency improvements because of the accelerated architecture and savings in the costly off-chip communications from the NMP approach.

The overview of the thesis is shown in Figure 7.1 where each contribution is achieved by designing related custom hardware. The aim of each system is to improve the system's performance and energy efficiency while keeping the area overhead as minimum as possible.

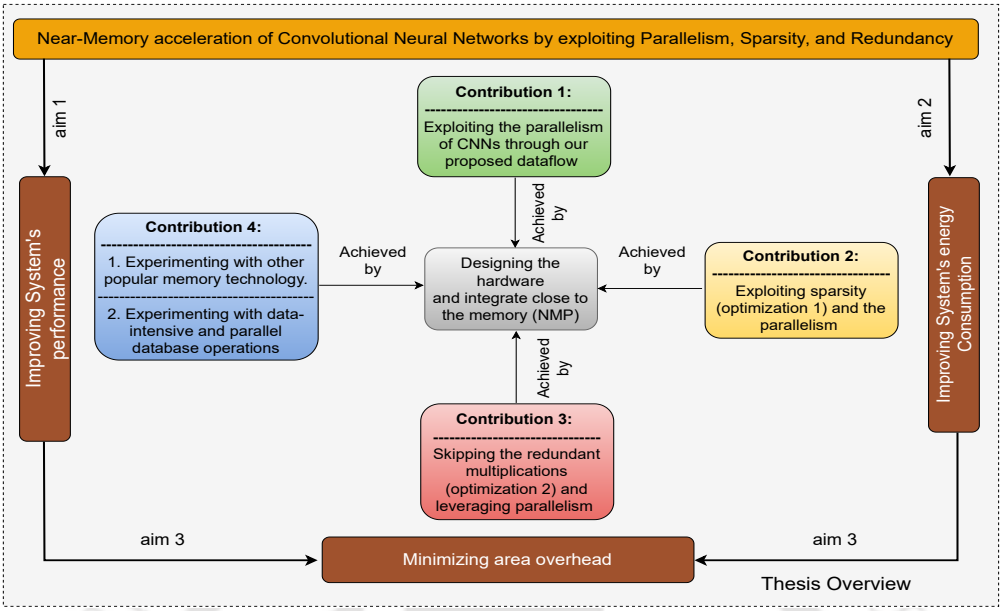


Figure 7.1: Overview of the thesis.

7.2 Scope for Future Work

The contributions of this thesis can be extended in several directions. Some of the possible future research directions are listed below:

1. In this thesis, we incorporate various data and hardware-based techniques to improve the system's throughput during inference. However, techniques and hardware can also be implemented to provide specialized systems for neural network training.

2. The proposed system can also be updated to a new system that can execute heterogeneous neural networks like CNN, recurrent neural network (RNN), and transformer.

3. We have implemented an NMP-based accelerator for the parallel organization of hybrid memory. However, NMP's efficacy on the hierarchical organization of hybrid memory still needs to be explored.

4. The proposed hardware modules are able to work on fixed precision. However, it is common for neural networks to handle different precisions based on the various scenarios. The hardware units can be made reconfigurable to work on the data with various precisions.

Bibliography

[1] Matthias Jung, Christian Weis, Patrick Bertram, Gunnar Braun, and Norbert Wehn. Power modelling of 3d-stacked memories with tlm2. 0 based virtual platforms. In *Synopsys User Group Conference (SNUG)*. Citeseer, 2013.

[2] Jinho Lee, Jongwook Chung, Jung Ho Ahn, and Kiyoung Choi. Excavating the hidden parallelism inside dram architectures with buffered compares. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 25(6):1793–1806, 2017.

[3] Duckhwan Kim, Jaeha Kung, Sek Chai, Sudhakar Yalamanchili, and Saibal Mukhopadhyay. NeuroCube: a programmable digital neuromorphic architecture with high-density 3d memory. *ACM SIGARCH Computer Architecture News*, 44(3):380–392, 2016.

[4] Eric Karl, Zheng Guo, James Conary, Jeffrey Miller, Yong-Gee Ng, Satyanand Nalam, Daeyeon Kim, John Keane, Xiaofei Wang, Uddalak Bhattacharya, et al. A 0.6 v, 1.5 ghz 84 mb sram in 14 nm finfet cmos technology with capacitive charge-sharing write assist circuitry. *IEEE Journal of Solid-State Circuits*, 51(1):222–229, 2015.

[5] Fatih Hamzaoglu, Umut Arslan, Nabhendra Bisnik, Swaroop Ghosh, Manoj B Lal, Nick Lindert, Mesut Meterelliyoz, Randy B Osborne, Joodong Park, Shigeki Tomishima, et al. 13.1 a 1gb 2ghz embedded dram in 22nm tri-gate cmos technology. In *2014 IEEE International Solid-State Circuits Conference Digest of Technical Papers (ISSCC)*, pages 230–231. IEEE, 2014.

[6] Hisaki Makimoto, Moritz Höckmann, Tina Lin, David Glöckner, Shqipe Gerguri, Lukas Clasen, Jan Schmidt, Athena Assadi-Schmidt, Alexandru Bejinariu, Patrick Müller, et al. Performance of a convolutional neural network derived from an ecg database in recognizing myocardial infarction. *Scientific reports*, 10(1):1–9, 2020.

[7] Vivienne Sze, Yu-Hsin Chen, Tien-Ju Yang, and Joel S Emer. Efficient processing of deep neural networks: A tutorial and survey. *Proceedings of the IEEE*, 105(12):2295–2329, 2017.

[8] Mingyu Gao and Christos Kozyrakis. HRL: Efficient and flexible reconfigurable logic for near-data processing. In *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 126–137. IEEE, 2016.

[9] Kevin Siu, Dylan Malone Stuart, Mostafa Mahmoud, and Andreas Moshovos. Memory requirements for convolutional neural network hardware accelerators. In *2018 IEEE International Symposium on Workload Characterization (IISWC)*, pages 111–121. IEEE, 2018.

[10] Gordon E Moore. Cramming more components onto integrated circuits. *Proceedings of the IEEE*, 86(1):82–85, 1998.

[11] Robert H Dennard, Fritz H Gaensslen, Hwa-Nien Yu, V Leo Rideout, Ernest Bassous, and Andre R LeBlanc. Design of ion-implanted mosfet's with very small physical dimensions. *IEEE Journal of solid-state circuits*, 9(5):256–268, 1974.

[12] Hadi Esmaeilzadeh, Emily Blem, Renee St Amant, Karthikeyan Sankaralingam, and Doug Burger. Dark silicon and the end of multicore scaling. In *2011 38th Annual international symposium on computer architecture (ISCA)*, pages 365–376. IEEE, 2011.

[13] Mingyu Gao, Grant Ayers, and Christos Kozyrakis. Practical near-data processing for in-memory analytics frameworks. In *2015 International Conference on Parallel Architecture and Compilation (PACT)*, pages 113–124. IEEE, 2015.

[14] Tianshi Chen, Zidong Du, Ninghui Sun, Jia Wang, Chengyong Wu, Yunji Chen, and Olivier Temam. Diannao: A small-footprint high-throughput accelerator for ubiquitous machine-learning. *ACM SIGARCH Computer Architecture News*, 42(1):269–284, 2014.

[15] Erfan Azarkhish, Davide Rossi, Igor Loi, and Luca Benini. NeuroStream: scalable and energy efficient deep learning with smart memory cubes. *IEEE Transactions on Parallel and Distributed Systems*, 29(2):420–434, 2017.

[16] Mingyu Gao, Jing Pu, Xuan Yang, Mark Horowitz, and Christos Kozyrakis. TETRIS: Scalable and efficient neural network acceleration with 3d memory. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 751–764, 2017.

[17] Ali Shafiee, Anirban Nag, Naveen Muralimanohar, Rajeev Balasubramonian, John Paul Strachan, Miao Hu, R Stanley Williams, and Vivek Srikumar. ISAAC: A convolutional neural network accelerator with in-situ analog arithmetic in crossbars. *ACM SIGARCH Computer Architecture News*, 44(3):14–26, 2016.

[18] Shuangchen Li, Dimin Niu, Krishna T Malladi, Hongzhong Zheng, Bob Brennan, and Yuan Xie. DRISA: A DRAM-based reconfigurable in-situ accelerator. In *2017 50th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 288–301. IEEE, 2017.

[19] Peter M Kogge. EXECUBE-a new architecture for scaleable mpps. In *1994 International Conference on Parallel Processing Vol. 1*, volume 1, pages 77–84. IEEE, 1994.

[20] Pengfei Zuo, Yu Hua, Ming Zhao, Wen Zhou, and Yuncheng Guo. Improving the performance and endurance of encrypted non-volatile main memory through deduplicating writes. In *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 442–454. IEEE, 2018.

[21] Computation structures. https://computationstructures.org/lectures/caches/caches.html.

[22] Shyamkumar Thoziyoor, Jung Ho Ahn, Matteo Monchiero, Jay B Brockman, and Norman P Jouppi. A comprehensive memory modeling tool and its application to the design and analysis of future memory hierarchies. *ACM SIGARCH Computer Architecture News*, 36(3):51–62, 2008.

[23] Vivek Seshadri, Yoongu Kim, Chris Fallin, Donghyuk Lee, Rachata Ausavarungnirun, Gennady Pekhimenko, Yixin Luo, Onur Mutlu, Phillip B Gibbons, Michael A Kozuch, et al. RowClone: fast and energy-efficient in-dram bulk data copy and initialization. In *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 185–197, 2013.

[24] Vivek Seshadri, Kevin Hsieh, Amirali Boroum, Donghyuk Lee, Michael A Kozuch, Onur Mutlu, Phillip B Gibbons, and Todd C Mowry. Fast bulk bitwise AND and OR in DRAM. *IEEE Computer Architecture Letters*, 14(2):127–131, 2015.

[25] Mohsen Imani, Yeseong Kim, and Tajana Rosing. MPIM: Multi-purpose in-memory processing using configurable resistive memory. In *2017 22nd Asia and South Pacific Design Automation Conference (ASP-DAC)*, pages 757–763. IEEE, 2017.

[26] Shuangchen Li, Cong Xu, Qiaosha Zou, Jishen Zhao, Yu Lu, and Yuan Xie. Pinatubo: A processing-in-memory architecture for bulk bitwise operations in emerging non-volatile memories. In *2016 53nd ACM/EDAC/IEEE Design Automation Conference (DAC)*, pages 1–6. IEEE, 2016.

[27] Vivek Seshadri, Donghyuk Lee, Thomas Mullins, Hasan Hassan, Amirali Boroumand, Jeremie Kim, Michael A Kozuch, Onur Mutlu, Phillip B Gibbons, and Todd C Mowry. Ambit: In-memory accelerator for bulk bitwise operations using commodity dram technology. In *2017 50th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 273–287. IEEE, 2017.

[28] Fei Gao, Georgios Tziantzioulis, and David Wentzlaff. ComputeDRAM: In-memory compute using off-the-shelf DRAMs. In *Proceedings of the 52nd annual IEEE/ACM international symposium on microarchitecture*, pages 100–113, 2019.

[29] Shaahin Angizi, Naima Ahmed Fahmi, Wei Zhang, and Deliang Fan. PIM-Assembler: a processing-in-memory platform for genome assembly. In *2020 57th ACM/IEEE Design Automation Conference (DAC)*, pages 1–6. IEEE, 2020.

[30] Sumanth Umesh and Sparsh Mittal. A survey of spintronic architectures for processing-in-memory and neural networks. *Journal of Systems Architecture*, 97:349–372, 2019.

[31] Sparsh Mittal. A survey of reram-based architectures for processing-in-memory and neural networks. *Machine learning and knowledge extraction*, 1(1):75–114, 2019.

[32] Kamil Khan, Sudeep Pasricha, and Ryan Gary Kim. A survey of resource management for processing-in-memory and near-memory processing architectures. *Journal of Low Power Electronics and Applications*, 10(4):30, 2020.

[33] Sukhan Lee, Shin-haeng Kang, Jaehoon Lee, Hyeonsu Kim, Eojin Lee, Seungwoo Seo, Hosang Yoon, Seungwon Lee, Kyounghwan Lim, Hyunsung Shin, et al. Hardware architecture and software stack for PIM based on commercial DRAM technology: Industrial product. In *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*, pages 43–56. IEEE, 2021.

[34] Shaizeen Aga, Nuwan Jayasena, and Mike Ignatowski. Co-ML: a case for co llaborative ml acceleration using near-data processing. In *Proceedings of the International Symposium on Memory Systems*, pages 506–517, 2019.

[35] Leibin Ni, Yuhao Wang, Hao Yu, Wei Yang, Chuliang Weng, and Junfeng Zhao. An energy-efficient matrix multiplication accelerator by distributed in-memory computing on binary RRAM crossbar. In *2016 21st Asia and South Pacific Design Automation Conference (ASP-DAC)*, pages 280–285. IEEE, 2016.

[36] Lixue Xia, Tianqi Tang, Wenqin Huangfu, Ming Cheng, Xiling Yin, Boxun Li, Yu Wang, and Huazhong Yang. Switched by input: Power efficient structure for

rram-based convolutional neural network. In *2016 53nd ACM/EDAC/IEEE Design Automation Conference (DAC)*, pages 1–6. IEEE, 2016.

[37] Quan Deng, Lei Jiang, Youtao Zhang, Minxuan Zhang, and Jun Yang. DrAcc: a DRAM based accelerator for accurate cnn inference. In *Proceedings of the 55th Annual Design Automation Conference*, pages 1–6, 2018.

[38] Sam Likun Xi, Aurelia Augusta, Manos Athanassoulis, and Stratos Idreos. Beyond the wall: Near-data processing for databases. In *Proceedings of the 11th International Workshop on Data Management on New Hardware*, pages 1–10, 2015.

[39] Christian Weis, Norbert Wehn, Loi Igor, and Luca Benini. Design space exploration for 3D-stacked DRAMs. In *2011 Design, Automation & Test in Europe*, pages 1–6. IEEE, 2011.

[40] Joe Jeddeloh and Brent Keeth. Hybrid memory cube new dram architecture increases density and performance. In *2012 symposium on VLSI technology (VLSIT)*, pages 87–88. IEEE, 2012.

[41] JEDEC Standard. High bandwidth memory (HBM) DRAM. *Jesd235*, 2013.

[42] Minsoo Rhu, Natalia Gimelshein, Jason Clemons, Arslan Zulfiqar, and Stephen W Keckler. vDNN: Virtualized deep neural networks for scalable, memory-efficient neural network design. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 1–13. IEEE, 2016.

[43] Gao Huang, Yu Sun, Zhuang Liu, Daniel Sedra, and Kilian Q Weinberger. Deep networks with stochastic depth. In *European conference on computer vision*, pages 646–661. Springer, 2016.

[44] Mostafa Mahmoud, Isak Edo, Ali Hadi Zadeh, Omar Mohamed Awad, Gennady Pekhimenko, Jorge Albericio, and Andreas Moshovos. Tensordash: Exploiting sparsity to accelerate deep neural network training. In *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 781–795. IEEE, 2020.

[45] Yunji Chen, Tao Luo, Shaoli Liu, Shijin Zhang, Liqiang He, Jia Wang, Ling Li, Tianshi Chen, Zhiwei Xu, Ninghui Sun, et al. Dadiannao: A machine-learning supercomputer. In *2014 47th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 609–622. IEEE, 2014.

[46] Xue-Wen Chen and Xiaotong Lin. Big data deep learning: challenges and perspectives. *IEEE access*, 2:514–525, 2014.

[47] Po-Sen Huang, Xiaodong He, Jianfeng Gao, Li Deng, Alex Acero, and Larry Heck. Learning deep structured semantic models for web search using clickthrough data. In *Proceedings of the 22nd ACM international conference on Conference on information & knowledge management*, pages 2333–2338. ACM, 2013.

[48] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.

[49] Yin Fan, Xiangju Lu, Dian Li, and Yuanliu Liu. Video-based emotion recognition using CNN-RNN and C3D hybrid networks. In *Proceedings of the 18th ACM International Conference on Multimodal Interaction*, pages 445–450. ACM, 2016.

[50] Wenpeng Yin, Katharina Kann, Mo Yu, and Hinrich Schütze. Comparative study of CNN and RNN for natural language processing. *arXiv preprint arXiv:1702.01923*, 2017.

[51] Raia Hadsell, Pierre Sermanet, Jan Ben, Ayse Erkan, Marco Scoffier, Koray Kavukcuoglu, Urs Muller, and Yann LeCun. Learning long-range vision for autonomous off-road driving. *Journal of Field Robotics*, 26(2):120–144, 2009.

[52] George E Dahl, Tara N Sainath, and Geoffrey E Hinton. Improving deep neural networks for LVCSR using rectified linear units and dropout. In *Acoustics, Speech and Signal Processing (ICASSP), 2013 IEEE International Conference on*, pages 8609–8613. IEEE, 2013.

[53] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. Tensorflow: A system for large-scale machine learning. In *12th {USENIX} symposium on operating systems design and implementation ({OSDI} 16)*, pages 265–283, 2016.

[54] Sparsh Mittal and Shraiysh Vaishay. A survey of techniques for optimizing deep learning on GPUs. *Journal of Systems Architecture*, 99:101635, 2019.

[55] Tianshi Chen, Zidong Du, Ninghui Sun, Jia Wang, Chengyong Wu, Yunji Chen, and Olivier Temam. A high-throughput neural network accelerator. *IEEE Micro*, 35(3):24–32, 2015.

[56] Lukas Cavigelli and Luca Benini. Origami: A 803-gop/s/w convolutional network accelerator. *IEEE Transactions on Circuits and Systems for Video Technology*, 27(11):2461–2475, 2016.

[57] Angshuman Parashar, Minsoo Rhu, Anurag Mukkara, Antonio Puglielli, Rangharajan Venkatesan, Brucek Khailany, Joel Emer, Stephen W Keckler, and William J Dally. SCNN: An accelerator for compressed-sparse convolutional neural networks. *ACM SIGARCH Computer Architecture News*, 45(2):27–40, 2017.

[58] Nils J Nilsson. *Principles of artificial intelligence*. Springer Science & Business Media, 1982.

[59] Mehryar Mohri, Afshin Rostamizadeh, and Ameet Talwalkar. *Foundations of machine learning*. MIT press, 2018.

[60] Youhui Zhang, Peng Qu, Yu Ji, Weihao Zhang, Guangrong Gao, Guanrui Wang, Sen Song, Guoqi Li, Wenguang Chen, Weimin Zheng, et al. A system hierarchy for brain-inspired computing. *Nature*, 586(7829):378–384, 2020.

[61] Youhui Zhang, Peng Qu, and Weimin Zheng. Towards" general purpose" brain-inspired computing system. *Tsinghua Science and Technology*, 26(5):664–673, 2021.

[62] Stefan Schliebs and Nikola Kasabov. Evolving spiking neural networka survey. *Evolving Systems*, 4(2):87–98, 2013.

[63] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep learning*. MIT press, 2016.

[64] Vinod Nair and Geoffrey E Hinton. Rectified linear units improve restricted boltzmann machines. In *Icml*, 2010.

[65] Andrew L Maas, Awni Y Hannun, Andrew Y Ng, et al. Rectifier nonlinearities improve neural network acoustic models. In *Proc. icml*, volume 30, page 3. Citeseer, 2013.

[66] Hybrid Memory Cube Consortium et al. Hybrid memory cube specification 1.0. *Last Revision Jan*, 2013.

[67] Sung-Kye Park. Technology scaling challenge and future prospects of DRAM and NAND flash memory. In *2015 IEEE International Memory Workshop (IMW)*, pages 1–4. IEEE, 2015.

[68] Gaurav Dhiman, Raid Ayoub, and Tajana Rosing. PDRAM: a hybrid PRAM and DRAM main memory system. In *2009 46th ACM/IEEE Design Automation Conference*, pages 664–669. IEEE, 2009.

[69] Moinuddin K Qureshi, Vijayalakshmi Srinivasan, and Jude A Rivers. Scalable high performance main memory system using phase-change memory technology. In *Proceedings of the 36th annual international symposium on Computer architecture*, pages 24–33, 2009.

[70] Moinuddin K Qureshi, Michele M Franceschini, Ashish Jagmohan, and Luis A Lastras. PreSET: Improving performance of phase change memories by exploiting asymmetry in write times. *ACM SIGARCH Computer Architecture News*, 40(3):380–391, 2012.

[71] Xuanhua Shi, Zhigao Zheng, Yongluan Zhou, Hai Jin, Ligang He, Bo Liu, and Qiang-Sheng Hua. Graph processing on gpus: A survey. *ACM Computing Surveys (CSUR)*, 50(6):1–35, 2018.

[72] Sebastian Breß, Max Heimel, Norbert Siegmund, Ladjel Bellatreche, and Gunter Saake. GPU-accelerated database systems: Survey and open challenges. In *Transactions on Large-Scale Data-and Knowledge-Centered Systems XV*, pages 1–35. Springer, 2014.

[73] Ajay Brahmakshatriya, Yunming Zhang, Changwan Hong, Shoaib Kamil, Julian Shun, and Saman Amarasinghe. Compiling graph applications for GPUs with GraphIt. In *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 248–261. IEEE, 2021.

[74] Yunming Zhang, Mengjiao Yang, Riyadh Baghdadi, Shoaib Kamil, Julian Shun, and Saman Amarasinghe. Graphit: A high-performance graph dsl. *Proceedings of the ACM on Programming Languages*, 2(OOPSLA):1–30, 2018.

[75] Hui Wang, Wan-Lei Zhao, Xiangxiang Zeng, and Jianye Yang. Fast k-nn graph construction by GPU based NN-descent. In *Proceedings of the 30th ACM International Conference on Information & Knowledge Management*, pages 1929–1938, 2021.

[76] Sofoklis Floratos, Mengbai Xiao, Hao Wang, Chengxin Guo, Yuan Yuan, Rubao Lee, and Xiaodong Zhang. NestGPU: Nested query processing on GPU. In *2021 IEEE 37th International Conference on Data Engineering (ICDE)*, pages 1008–1019. IEEE, 2021.

[77] Peter Bakkum and Kevin Skadron. Accelerating SQL database operations on a GPU with CUDA. In *Proceedings of the 3rd workshop on general-purpose computation on graphics processing units*, pages 94–103, 2010.

[78] Rajat Raina, Anand Madhavan, and Andrew Y Ng. Large-scale deep unsupervised learning using graphics processors. In *Proceedings of the 26th annual international conference on machine learning*, pages 873–880. ACM, 2009.

[79] Shi Dong, Xiang Gong, Yifan Sun, Trinayan Baruah, and David Kaeli. Characterizing the microarchitectural implications of a convolutional neural network (CNN) execution on GPUs. In *Proceedings of the 2018 ACM/SPEC International Conference on Performance Engineering*, pages 96–106, 2018.

[80] Mohammad Samragh Razlighi, Mohsen Imani, Farinaz Koushanfar, and Tajana Rosing. LookNN: neural network with no multiplication. In *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2017*, pages 1775–1780. IEEE, 2017.

[81] Daniel Peroni, Mohsen Imani, and Tajana Rosing. ALook: adaptive lookup for gpgpu acceleration. In *Proceedings of the 24th Asia and South Pacific Design Automation Conference*, pages 739–746, 2019.

[82] Sangkug Lym, Donghyuk Lee, Mike O'Connor, Niladrish Chatterjee, and Mattan Erez. DeLTA: GPU performance model for deep learning applications with in-depth memory system traffic analysis. In *2019 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 293–303. IEEE, 2019.

[83] Sparsh Mittal. A survey of FPGA-based accelerators for convolutional neural networks. *Neural computing and applications*, 32(4):1109–1139, 2020.

[84] Kamel Abdelouahab, Maxime Pelcat, Jocelyn Serot, and François Berry. Accelerating CNN inference on FPGAs: A survey. *arXiv preprint arXiv:1806.01683*, 2018.

[85] Maciej Besta, Dimitri Stanojevic, Johannes De Fine Licht, Tal Ben-Nun, and Torsten Hoefler. Graph processing on fpgas: Taxonomy, survey, challenges. *arXiv preprint arXiv:1903.06697*, 2019.

[86] Philippos Papaphilippou and Wayne Luk. Accelerating database systems using FPGAs: A survey. In *2018 28th International Conference on Field Programmable Logic and Applications (FPL)*, pages 125–1255. IEEE, 2018.

[87] Xuan Sun, Chun Jason Xue, Jinghuan Yu, Tei-Wei Kuo, and Xue Liu. Accelerating data filtering for database using FPGA. *Journal of Systems Architecture*, 114:101908, 2021.

[88] Nina Engelhardt and Hayden Kwok-Hay So. Gravf: A vertex-centric distributed graph processing framework on fpgas. In *2016 26th International Conference on Field Programmable Logic and Applications (FPL)*, pages 1–4. IEEE, 2016.

[89] Sang-Woo Jun, Ming Liu, Sungjin Lee, Jamey Hicks, John Ankcorn, Myron King, Shuotao Xu, et al. Bluedbm: An appliance for big data analytics. In *2015 ACM/IEEE 42nd Annual International Symposium on Computer Architecture (ISCA)*, pages 1–13. IEEE, 2015.

[90] Guohao Dai, Tianhao Huang, Yuze Chi, Ningyi Xu, Yu Wang, and Huazhong Yang. Foregraph: Exploring large-scale graph processing on multi-fpga architecture. In *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pages 217–226, 2017.

[91] Ming Xia, Zunkai Huang, Li Tian, Hui Wang, Chang Victor, Yongxin Zhu, and Songlin Feng. SparkNoC: An energy-efficiency FPGA-based accelerator using optimized lightweight CNN for edge computing. *Journal of Systems Architecture*, 115:101991, 2021.

[92] Ru Ding, Guangda Su, Guoqiang Bai, Wei Xu, Nan Su, and Xingjun Wu. A FPGA-based accelerator of convolutional neural network for face feature extraction. In *2019 IEEE International Conference on Electron Devices and Solid-State Circuits (EDSSC)*, pages 1–3. IEEE, 2019.

[93] Chen Zhang, Peng Li, Guangyu Sun, Yijin Guan, Bingjun Xiao, and Jason Cong. Optimizing FPGA-based accelerator design for deep convolutional neural networks. In *Proceedings of the 2015 ACM/SIGDA international symposium on field-programmable gate arrays*, pages 161–170, 2015.

[94] Huimin Li, Xitian Fan, Li Jiao, Wei Cao, Xuegong Zhou, and Lingli Wang. A high performance fpga-based accelerator for large-scale convolutional neural networks. In *2016 26th International Conference on Field Programmable Logic and Applications (FPL)*, pages 1–9. IEEE, 2016.

[95] Xiaocong Lian, Zhenyu Liu, Zhourui Song, Jiwu Dai, Wei Zhou, and Xiangyang Ji. High-performance FPGA-based CNN accelerator with block-floating-point arithmetic. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 27(8):1874–1885, 2019.

[96] Diksha Moolchandani, Anshul Kumar, and Smruti R Sarangi. Accelerating CNN inference on asics: A survey. *Journal of Systems Architecture*, 113:101887, 2021.

[97] Chuang-Yi Gui, Long Zheng, Bingsheng He, Cheng Liu, Xin-Yu Chen, Xiao-Fei Liao, and Hai Jin. A survey on graph processing accelerators: Challenges and opportunities. *Journal of Computer Science and Technology*, 34(2):339–371, 2019.

[98] Zhen Li, Yuqing Wang, Tian Zhi, and Tianshi Chen. A survey of neural network accelerators. *Frontiers of Computer Science*, 11(5):746–761, 2017.

[99] Qiuling Zhu, Berkin Akin, H Ekin Sumbul, Fazle Sadi, James C Hoe, Larry Pileggi, and Franz Franchetti. A 3D-stacked logic-in-memory accelerator for application-specific data intensive computing. In *3D Systems Integration Conference (3DIC), 2013 IEEE International*, pages 1–7. IEEE, 2013.

[100] Amin Farmahini-Farahani, Jung Ho Ahn, Katherine Morrow, and Nam Sung Kim. NDA: Near-DRAM acceleration architecture leveraging commodity DRAM devices and standard memory modules. In *High Performance Computer Architecture (HPCA), 2015 IEEE 21st International Symposium on*, pages 283–295. IEEE, 2015.

[101] Seth H Pugsley, Jeffrey Jestes, Rajeev Balasubramonian, Vijayalakshmi Srinivasan, Alper Buyuktosunoglu, Al Davis, and Feifei Li. Comparing implementations of near-data computing with in-memory mapreduce workloads. *IEEE Micro*, 34(4):44–52, 2014.

[102] Shafiur Rahman, Nael Abu-Ghazaleh, and Rajiv Gupta. GraphPulse: An event-driven hardware accelerator for asynchronous graph processing. In *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 908–921. IEEE, 2020.

[103] Yu-Hsin Chen, Joel Emer, and Vivienne Sze. Eyeriss: A spatial architecture for energy-efficient dataflow for convolutional neural networks. In *ACM SIGARCH Computer Architecture News*, volume 44, pages 367–379. IEEE Press, 2016.

[104] Jorge Albericio, Patrick Judd, Tayler Hetherington, Tor Aamodt, Natalie Enright Jerger, and Andreas Moshovos. Cnvlutin: Ineffectual-neuron-free deep neural network computing. *ACM SIGARCH Computer Architecture News*, 44(3):1–13, 2016.

[105] Shijin Zhang, Zidong Du, Lei Zhang, Huiying Lan, Shaoli Liu, Ling Li, Qi Guo, Tianshi Chen, and Yunji Chen. Cambricon-X: an accelerator for sparse neural networks. In

The 49th Annual IEEE/ACM International Symposium on Microarchitecture, page 20. IEEE Press, 2016.

[106] Song Han, Xingyu Liu, Huizi Mao, Jing Pu, Ardavan Pedram, Mark A Horowitz, and William J Dally. EIE: efficient inference engine on compressed deep neural network. In *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*, pages 243–254. IEEE, 2016.

[107] Duckhwan Kim, Taesik Na, Sudhakar Yalamanchili, and Saibal Mukhopadhyay. DeepTrain: A programmable embedded platform for training deep neural networks. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 37(11):2360–2370, 2018.

[108] Ping Chi, Shuangchen Li, Cong Xu, Tao Zhang, Jishen Zhao, Yongpan Liu, Yu Wang, and Yuan Xie. PRIME: A novel processing-in-memory architecture for neural network computation in ReRAM-based main memory. *ACM SIGARCH Computer Architecture News*, 44(3):27–39, 2016.

[109] Linghao Song, Xuehai Qian, Hai Li, and Yiran Chen. Pipelayer: A pipelined reram-based accelerator for deep learning. In *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 541–552. IEEE, 2017.

[110] Andreas Nowatzyk, Fong Pong, and Ashley Saulsbury. Missing the memory wall: The case for processor/memory integration. In *23rd Annual International Symposium on Computer Architecture (ISCA'96)*, pages 90–90. IEEE, 1996.

[111] Kiran Puttaswamy and Gabriel H Loh. Thermal analysis of a 3D die-stacked high-performance microprocessor. In *GLSVLSI*, pages 19–24. ACM, 2006.

[112] Junwhan Ahn, Sungpack Hong, Sungjoo Yoo, Onur Mutlu, and Kiyoung Choi. A scalable processing-in-memory accelerator for parallel graph processing. *ACM SIGARCH Computer Architecture News*, 43(3):105–117, 2016.

[113] Yi Kang, Wei Huang, Seung-Moon Yoo, Diana Keen, Zhenzhou Ge, Vinh Lam, Pratap Pattnaik, and Josep Torrellas. FlexRAM: Toward an advanced intelligent memory system. In *Computer Design (ICCD), 2012 IEEE 30th International Conference on*, pages 5–14. IEEE, 2012.

[114] Michael Schaffner, Frank K Gürkaynak, Aljoscha Smolic, and Luca Benini. DRAM or no-DRAM?: exploring linear solver architectures for image domain warping in 28 nm

CMOS. In *Proceedings of the 2015 Design, Automation & Test in Europe Conference & Exhibition*, pages 707–712. EDA Consortium, 2015.

[115] Palash Das, Shivam Lakhotia, Prabodh Shetty, and Hemangee K Kapoor. Towards near data processing of convolutional neural networks. In *VLSI Design and 2018 17th International Conference on Embedded Systems (VLSID), 2018 31st International Conference on*, pages 380–385. IEEE, 2018.

[116] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pages 1097–1105, 2012.

[117] Matthew D Zeiler and Rob Fergus. Visualizing and understanding convolutional networks. In *European conference on computer vision*, pages 818–833. Springer, 2014.

[118] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2014.

[119] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. Imagenet: A large-scale hierarchical image database. In *2009 IEEE conference on computer vision and pattern recognition*, pages 248–255. Ieee, 2009.

[120] Yu-Hsin Chen, Tushar Krishna, Joel S Emer, and Vivienne Sze. Eyeriss: An energy-efficient reconfigurable accelerator for deep convolutional neural networks. *IEEE Journal of Solid-State Circuits*, 52(1):127–138, 2017.

[121] Maurice Peemen, Arnaud AA Setio, Bart Mesman, Henk Corporaal, et al. Memory-centric accelerator design for convolutional neural networks. In *ICCD*, volume 2013, pages 13–19, 2013.

[122] Shaahin Angizi, Zhezhi He, Farhana Parveen, and Deliang Fan. IMCE: Energy-efficient bit-wise in-memory convolution engine for deep neural network. In *Proceedings of the 23rd Asia and South Pacific Design Automation Conference*, pages 111–116. IEEE Press, 2018.

[123] Alex Krizhevsky. One weird trick for parallelizing convolutional neural networks. *arXiv preprint arXiv:1404.5997*, 2014.

[124] Olaf Ronneberger, Philipp Fischer, and Thomas Brox. U-net: Convolutional networks for biomedical image segmentation. In *International Conference on Medical image computing and computer-assisted intervention*, pages 234–241. Springer, 2015.

[125] Jonathan Long, Evan Shelhamer, and Trevor Darrell. Fully convolutional networks for semantic segmentation. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 3431–3440, 2015.

[126] N Manohar, YH Sharath Kumar, Radhika Rani, and G Hemantha Kumar. Convolutional neural network with svm for classification of animal images. In *Emerging Research in Electronics, Computer Science and Technology*, pages 527–537. Springer, 2019.

[127] Yichuan Tang. Deep learning using linear support vector machines. *arXiv preprint arXiv:1306.0239*, 2013.

[128] Zhihao Jia, Matei Zaharia, and Alex Aiken. Beyond data and model parallelism for deep neural networks. *arXiv preprint arXiv:1807.05358*, 2018.

[129] Adam Coates, Brody Huval, Tao Wang, David Wu, Bryan Catanzaro, and Ng Andrew. Deep learning with COTS HPC systems. In *International conference on machine learning*, pages 1337–1345, 2013.

[130] Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, et al. Imagenet large scale visual recognition challenge. *International Journal of Computer Vision*, 115(3):211–252, 2015.

[131] Junwhan Ahn, Sungjoo Yoo, Onur Mutlu, and Kiyoung Choi. PIM-enabled instructions: a low-overhead, locality-aware processing-in-memory architecture. In *Computer Architecture (ISCA), 2015 ACM/IEEE 42nd Annual International Symposium on*, pages 336–348. IEEE, 2015.

[132] J Thomas Pawlowski. Hybrid memory cube (HMC). In *Hot Chips 23 Symposium (HCS), 2011 IEEE*, pages 1–24. IEEE, 2011.

[133] Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. Automatic differentiation in pytorch. 2017.

[134] Sheng Li, Jung Ho Ahn, Richard D Strong, Jay B Brockman, Dean M Tullsen, and Norman P Jouppi. McPAT: an integrated power, area, and timing modeling framework for multicore and manycore architectures. In *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 469–480. ACM, 2009.

[135] J Murphy. Deep learning benchmarks of NVIDIA tesla P100 PCIe tesla K80 and tesla M40 GPUs, 2017.

[136] Shaoli Liu, Zidong Du, Jinhua Tao, Dong Han, Tao Luo, Yuan Xie, Yunji Chen, and Tianshi Chen. Cambricon: An instruction set architecture for neural networks. In *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*, pages 393–405. IEEE, 2016.

[137] Norman P Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, et al. In-datacenter performance analysis of a tensor processing unit. In *Proceedings of the 44th Annual International Symposium on Computer Architecture*, pages 1–12, 2017.

[138] Hyunsun Park, Dongyoung Kim, Junwhan Ahn, and Sungjoo Yoo. Zero and data reuse-aware fast convolution for deep neural networks on GPU. In *2016 International Conference on Hardware/Software Codesign and System Synthesis (CODES+ ISSS)*, pages 1–10. IEEE, 2016.

[139] Shaahin Angizi, Zhezhi He, Adnan Siraj Rakin, and Deliang Fan. CMP-PIM: an energy-efficient comparator-based processing-in-memory neural network accelerator. In *Proceedings of the 55th Annual Design Automation Conference*, page 105. ACM, 2018.

[140] Sanchari Sen, Shubham Jain, Swagath Venkataramani, and Anand Raghunathan. SparCE: sparsity aware general-purpose core extensions to accelerate deep neural networks. *IEEE Transactions on Computers*, 68(6):912–925, 2018.

[141] Song Han, Jeff Pool, John Tran, and William Dally. Learning both weights and connections for efficient neural network. In *Advances in neural information processing systems*, pages 1135–1143, 2015.

[142] Richard Wilson Vuduc and James W Demmel. *Automatic performance tuning of sparse matrix kernels*, volume 1. University of California, Berkeley Berkeley, CA, 2003.

[143] HMC specification 2.1, 2015.

[144] Yasuko Eckert, Nuwan Jayasena, and Gabriel H Loh. Thermal feasibility of die-stacked processing in memory. 2014.

[145] Ashutosh Pattnaik, Xulong Tang, Adwait Jog, Onur Kayiran, Asit K Mishra, Mahmut T Kandemir, Onur Mutlu, and Chita R Das. Scheduling techniques for GPU

architectures with processing-in-memory capabilities. In *Proceedings of the 2016 International Conference on Parallel Architectures and Compilation*, pages 31–44. ACM, 2016.

[146] Daniel Peroni, Mohsen Imani, Hamid Nejatollahi, Nikil Dutt, and Tajana Rosing. ARGA: approximate reuse for gpgpu acceleration. In *2019 56th ACM/IEEE Design Automation Conference (DAC)*, pages 1–6. IEEE, 2019.

[147] Hyeong-Ju Kang. Accelerator-aware pruning for convolutional neural networks. *IEEE Transactions on Circuits and Systems for Video Technology*, 30(7):2093–2103, 2019.

[148] Mohsen Imani, Abbas Rahimi, and Tajana S Rosing. Resistive configurable associative memory for approximate computing. In *2016 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 1327–1332. IEEE, 2016.

[149] Alex Krizhevsky, Geoffrey Hinton, et al. Learning multiple layers of features from tiny images. 2009.

[150] Giuseppe Desoli, Nitin Chawla, Thomas Boesch, Surinder-pal Singh, Elio Guidetti, Fabio De Ambroggi, Tommaso Majo, Paolo Zambotti, Manuj Ayodhyawasi, Harvinder Singh, et al. 14.1 a 2.9 tops/w deep convolutional neural network soc in fd-soi 28nm for intelligent embedded systems. In *2017 IEEE International Solid-State Circuits Conference (ISSCC)*, pages 238–239. IEEE, 2017.

[151] Mohsen Imani, Yeseong Kim, Abbas Rahimi, and Tajana Rosing. Acam: Approximate computing based on adaptive associative memory with online learning. In *Proceedings of the 2016 International Symposium on Low Power Electronics and Design*, pages 162–167, 2016.

[152] Jose-Maria Arnau, Joan-Manuel Parcerisa, and Polychronis Xekalakis. Eliminating redundant fragment shader executions on a mobile gpu via hardware memoization. *ACM SIGARCH Computer Architecture News*, 42(3):529–540, 2014.

[153] Marc Riera, Jose-Maria Arnau, and Antonio González. Computation reuse in dnns by exploiting input similarity. In *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, pages 57–68. IEEE, 2018.

[154] Mark Sandler, Andrew Howard, Menglong Zhu, Andrey Zhmoginov, and Liang-Chieh Chen. Mobilenetv2: Inverted residuals and linear bottlenecks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 4510–4520, 2018.

[155] Matthieu Courbariaux, Itay Hubara, Daniel Soudry, Ran El-Yaniv, and Yoshua Bengio. Binarized neural networks: Training deep neural networks with weights and activations constrained to+ 1 or-1. *arXiv preprint arXiv:1602.02830*, 2016.

[156] Aojun Zhou, Anbang Yao, Yiwen Guo, Lin Xu, and Yurong Chen. Incremental network quantization: Towards lossless cnns with low-precision weights. *arXiv preprint arXiv:1702.03044*, 2017.

[157] Wei Wei, Dejun Jiang, Sally A McKee, Jin Xiong, and Mingyu Chen. Exploiting program semantics to place data in hybrid memory. In *2015 International Conference on Parallel Architecture and Compilation (PACT)*, pages 163–173. IEEE, 2015.

[158] Xiaoyuan Wang, Haikun Liu, Xiaofei Liao, Ji Chen, Hai Jin, Yu Zhang, Long Zheng, Bingsheng He, and Song Jiang. Supporting superpages and lightweight page migration in hybrid memory systems. *ACM Transactions on Architecture and Code Optimization (TACO)*, 16(2):1–26, 2019.

[159] Na Niu, Fangfa Fu, Bing Yang, Jiacai Yuan, Fengchang Lai, and Jinxiang Wang. WIRD: an efficiency migration scheme in hybrid DRAM and PCM main memory for image processing applications. *IEEE Access*, 7:35941–35951, 2019.

[160] Alexandre P Ferreira, Miao Zhou, Santiago Bock, Bruce Childers, Rami Melhem, and Daniel Mossé. Increasing pcm main memory lifetime. In *2010 Design, Automation & Test in Europe Conference & Exhibition (DATE 2010)*, pages 914–919. IEEE, 2010.

[161] Hoda Aghaei Khouzani, Chengmo Yang, and Jingtong Hu. Improving performance and lifetime of DRAM-PCM hybrid main memory through a proactive page allocation strategy. In *The 20th Asia and South Pacific Design Automation Conference*, pages 508–513. IEEE, 2015.

[162] Simone Raoux, Geoffrey W Burr, Matthew J Breitwisch, Charles T Rettner, Y-C Chen, Robert M Shelby, Martin Salinga, Daniel Krebs, S-H Chen, H-L Lung, et al. Phase-change random access memory: A scalable technology. *IBM Journal of Research and Development*, 52(4.5):465–479, 2008.

[163] Shihao Song, Anup Das, Onur Mutlu, and Nagarajan Kandasamy. Enabling and exploiting partition-level parallelism (palp) in phase change memories. *ACM Transactions on Embedded Computing Systems (TECS)*, 18(5s):1–25, 2019.

[164] Benjamin C Lee, Engin Ipek, Onur Mutlu, and Doug Burger. Architecting phase change memory as a scalable dram alternative. In *Proceedings of the 36th annual international symposium on Computer architecture*, pages 2–13, 2009.

[165] Jared Casper and Kunle Olukotun. Hardware acceleration of database operations. In *ACM/SIGDA Int. symposium on FPGA*, pages 151–160. ACM, 2014.

[166] Abadi et al. Column-stores vs. row-stores: How different are they really? In *ACM SIGMOD Int. conference on Management of data*, pages 967–980. ACM, 2008.

[167] O'Neil et al. The star schema benchmark and augmented fact table indexing. In *Technology Conference on Performance Evaluation and Benchmarking*, pages 237–252. Springer, 2009.

[168] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R Hower, Tushar Krishna, Somayeh Sardashti, et al. The gem5 simulator. *ACM SIGARCH Computer Architecture News*, 39(2):1–7, 2011.

[169] Muralimanohar et al. CACTI 6.0: A tool to model large caches. *HP laboratories*, pages 22–31, 2009.

Publications

Journals

- **Palash Das** and Hemangee K. Kapoor. "nZESPA: A Near-3D-Memory Zero Skipping Parallel Accelerator for CNNs". IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD), 2020.
- **Palash Das** and Hemangee K. Kapoor. "CLU: A near-memory accelerator exploiting the parallelism in Convolutional Neural Networks". ACM Journal on Emerging Technologies in Computing Systems (JETC), ACM, 2020.
- "ALAMNI: Adaptive LookAside Memory based Near-memory Inference engine for eliminating multiplications in real-time." **(Journal Revision Submitted)**
- "Exploring the design space for near-DRAM MAC-based inference engine." **(Journal Submitted)**

Conferences

- **Palash Das**, Shivam Lakhotia, Prabodh Shetty, and Hemangee K. Kapoor "Towards Near Data Processing of Convolutional Neural Networks" 31st International Conference on VLSI Design (VLSID) 2018, IEEE.
- **Palash Das**, and Hemangee K. Kapoor. "Towards near-data processing of compare operations in 3D-stacked memory." Proceedings of the 2018 on Great Lakes Symposium on VLSI (GLSVLSI), 2018.
- **Palash Das**, Ajay Joshi, and Hemangee K. Kapoor. "Hydra: A near hybrid memory accelerator for CNN inference" Design, Automation and Test in Europe Conference, DATE 2022 (accepted).

Vitae

Palash Das has joined the Ph.D. program in the Department of Computer Science and Engineering (CSE) of the Indian Institute of Technology (IIT) Guwahati, India, in July 2015. During his Ph.D. program, he was affiliated with the SUStainable Multicore Architecture (SUSMA) Lab. of CSE. Prior to joining the Ph.D., he did his Bachelor of Technology in CSE from the Maulana Abul Kalam Azad University of Technology, formerly known as West Bengal University of Technology. He did his Master of Engineering in CSE from the Indian Institute of Engineering Science and Technology, Shibpur. A *certificate of merit* from the government of West Bengal, India, has been awarded in recognition of the high position secured by him in the list of meritorious candidates qualifying for the awards in the secondary examination 2003. He has six years of teaching experience (2009-2015) as an Assistant Professor in the Department of CSE at Dumkal Institute of Engineering and Technology. He received a certificate of appreciation for his outstanding contributions to the college in its various programs during his teaching. He has also received a fellowship grant from VLSID, 2018, for one of his papers at the conference. He was awarded the **Intel India Research Fellowship**, 2020 for his research contributions during his Ph.D. He has a keen interest in pursuing the field of computer architecture. His current research interests include accelerating deep neural networks, near-memory processing, and ASIC design. He enjoys playing guitar and listening to music in his leisure time.

Contact Information

Email	:	palash.das@iitg.ac.in, sanipaul24@gmail.com
Web	:	https://www.iitg.ac.in/stud/palash.das/
Address	:	C/O-Papia Villa, Birnagar, Mahamayasthan, Raiganj, Uttar Dinajpur, Pin-733134 INDIA