Declaration

I hereby declare that except where specific reference is made to the work of others, the contents of this dissertation are original and have not been submitted in whole or in part for consideration for any other degree or qualification in this, or any other university. It is not substantially the same as any that I have submitted, or am concurrently submitting, for a degree or diploma or other qualification at the University of Cambridge or any other University or similar institution except as declared in the Preface and specified in the text. This dissertation does not exceed the prescribed limit of 60 000 words.

Aida Miralaei
January 2024
Acknowledgements

I have reserved the writing of this acknowledgment section for the final stage of my dissertation due to the deep emotional significance it holds for me. The journey to complete my PhD has been profoundly challenging—testing every facet of my resolve—yet it has been equally rewarding due to the personal growth I’ve experienced and the incredible people who have supported and enriched this journey in ways beyond my imagination.

I am deeply thankful to many people, from friends and mentors to the dedicated staff who support both this institution and the broader community. While it is impossible to fully express my gratitude to everyone, I must highlight a few who have been particularly influential.

Firstly, I extend my sincere appreciation to my supervisor Prof. Timothy M. Jones, for his unwavering support and guidance. Especially his commitment throughout all these years to have regular meetings to discuss ideas and addressing the challenges of implementing them has been invaluable.

My heartfelt thanks go to Emmanuel College for shaping my experience in Cambridge in the most supportive and enriching way possible. To Dame Fiona Reynolds, the former master of Emmanuel I owe immense gratitude for her love, support, and mentorship from the outset of this journey. Her unwavering belief in me, despite the obstacles that arose, her encouragement to always look beyond the current state of events and her wisdom in navigating challenges have been profoundly influential in my life. I also extend my gratitude to the college staff, the tutorial office, graduate tutors, the porters, MCR, and the groundskeeper of Emmanuel College. Thank you for the many years of support, the community, and the gardens that never fail to uplift one’s spirit regardless of the season.

I am equally indebted to the Faculty of Computer Science and technology. To Lise Gough, our Postgraduate Education manager, I owe tremendous thanks for always have the door of her office open and ready to listen and advise at every step. Whether I was stressed about some paperwork or life in general as a postgraduate student, the countless times I left her office filled with hope and reassurance have been pivotal in my journey. To my peers in the Computer Architecture group, the women@CL community, and the graduate forum, each of you has taught me invaluable lessons through our interactions.
To one of my sister’s best friends, Dr Jyothish Soman, who from day one made sure that I know I have a big brother in Cambridge whom I can always count on. Whether it was for having a chat to brainstorm some technical questions or asking for logistic help during difficult times like the lockdowns, I am forever grateful for his support.

To my dearest friend Melanie, I am forever grateful that we embarked on our PhD journey together! From our matriculation day until now, you have been my rock every step of the way. While many people say they owe their life to their friends as a figure of speech, I genuinely mean it when I say I wouldn’t have survived the first wave of COVID without you. During those difficult times when I struggled to breathe properly and couldn’t even get out of bed, you selflessly cooked meals for me every day for an entire month and left your house each day in the peak of the pandemic just to leave fresh food by my door. Your sisterly love, inspiration, joy, strength, and wisdom have been a constant source of comfort and support in my life.

I feel incredibly fortunate to have crossed paths with numerous remarkable individuals during my time in Cambridge. Their presence has enriched my life beyond measure. Among them are the amazing souls from my years of living in Park Lodge, Sturton, and Cutter Ferry House, who have left an indelible mark on my heart. I’m forever grateful for their friendship, joyous memories, laughter, and the unwavering support that carried me through the challenges of my PhD journey, especially during the final year and writing process.

To all my friends who checked in on me and kept my spirits up, whether near or far, your kindness and support meant the world to me. Whether it was a comforting call from home or the online presence of a friend during late-night writing sessions, each gesture was a guiding light through the darkest nights. Thank you all for being there for me; I am eternally grateful.

The successful completion of my PhD would not be possible without the generous financial support of: ARM Ltd, Entrepreneur First Ltd (EF), Trinity-Henry Barlow scholarship, Faculty of Computer Science and technology and Tim and Grace Freshwater from Emmanuel College. Thank you for believing in my potential and my research. I am deeply grateful for the profound impact your support has had on my academic growth and professional development. I would also like to express my gratitude to my examiners for their time and effort in reviewing this thesis. Their insightful feedback has been instrumental in strengthening my work.

In the end, I dedicate this thesis to my late sister, Negar, who tragically left us too soon before completing her own PhD dissertation in the Faculty of Computer Science and Technology at Emmanuel College. We first dreamed this future together when we were just kids. I vividly recall that summer afternoon, sitting on the terrace of my late grandmother’s house in Iran, as if it were yesterday. It stands out in my memory as clearly as the day Negar
received her admission to Cambridge. It was then that we made a promise to each other: against all odds, we would strive together to attend one of the world’s best universities and make a tangible, positive impact. I never foresaw that "together" would mean she would start this journey and I would complete it... Negar, you were my first and best friend, teacher, and mentor, and you continue to be my greatest source of inspiration, love, and strength. And to my parents, Zohreh and Mohammad, thank you for everything. Your unconditional love and support have given us both our wings to fly, taught us resilience, and reminded us that no challenge is too great when faced with courage and confidence.
Abstract

Deep neural networks (DNNs) provide many application domains with state-of-the-art performance and accuracy. However, they are compute-heavy and data-intensive which makes deploying them on resource-constrained edge devices challenging. Transforming the real-valued parameters of a DNN into a minimised-bit-width approximated version through quantising or binarising lowers their accuracy to an extent but significantly reduces their computation complexity and storage space. Moreover, it makes them good candidates to be considered for near-memory processing due to these criteria. On the other hand, when it comes to designing a hardware module (either for near-memory processing or not), the more specialised the hardware the better the performance. The downside with over-specialised modules then becomes their inability to adapt, which can be problematic in a fast-evolving field such as deep learning. To this end, designing a module with reasonable performance, area overhead and energy consumption while maintaining a good balance between the physical limitations of the design and its flexibility is a challenge.

The contribution of this thesis is to introduce a processing-near-memory module (PNM) for low-precision convolution neural networks. The placement of this module is near the main memory (DDR4 DRAM) and the memory controller, with a data layout that does not require rearrangement either before or after the convolution operations. Here, two distinct design modes for the PNM module are presented: mode S, which focuses on minimizing area overhead, and mode T, aimed at optimizing data transfers between the module and DRAM. The performance, area, and energy costs of these designs were thoroughly assessed through both analytical and practical analysis. These evaluations highlighted the impact of varying filters, hardware replicas, and bit-widths for each model on the stated criteria, leading to recommended design choices tailored to specific use cases’ demands and constraints. An evaluation using a BCNN based on AlexNet indicates that for mode S, a configuration of one hardware replica with 16 filters per replica offers an optimal balance between area, runtime, and energy. In contrast, for mode T, the best configuration comprises 16 replicas and 32 filters. Comparatively, mode T surpasses mode S by a factor of 6.39 in performance, while mode S achieves greater savings in area and energy, by factors of 3.56 and 15.99, respectively.
# Table of contents

List of figures  

1 Introduction  
1.1 Research questions and hypotheses .......................... 2  
1.2 Contributions .................................................. 3  
1.3 Outline ......................................................... 3  

2 Background  
2.1 Artificial Neural Networks and Deep Neural Networks ........... 5  
2.2 Convolutional Neural Networks .................................. 7  
2.2.1 Fully Connected Layers .................................... 8  
2.2.2 Convolutional Layers ...................................... 9  
2.2.3 Pooling Layers ........................................... 11  
2.3 Quantised Convolutional Neural Networks ......................... 12  
2.3.1 Uniform Affine Quantisation ................................ 13  
2.3.2 Quantisation Granularity .................................. 14  
2.4 Binarised Convolutional Neural Networks ......................... 15  
2.4.1 Training a BCNN ............................................ 15  
2.4.2 Running a BCNN ........................................... 16  
2.5 Inference on edge devices .................................... 17  
2.6 Low-precision Networks and Accuracy ............................ 20  
2.7 DRAM and Processing Near-Memory ............................. 21  
2.7.1 Processing In or Near Memory .............................. 21  
2.7.2 DRAM Architecture ........................................ 22  
2.8 Summary ......................................................... 24  

3 Literature Review  
3.1 Algorithmic Approach ........................................... 26
| 3.1.1 | Quantisation | 27 |
| 3.1.2 | Pruning | 30 |
| 3.1.3 | Tensor Decomposition | 31 |
| 3.1.4 | Knowledge Distillation | 31 |
| 3.1.5 | Network Architecture Search (NAS) | 32 |
| 3.2 | Hardware Approach | 32 |
| 3.2.1 | Hardware Architectures for DNNs | 34 |
| 3.2.2 | Processing In or Near Memory | 36 |
| 3.3 | Summary | 37 |
| 4 | A BCNN PNM Module | 39 |
| 4.1 | A Design Overview | 39 |
| 4.2 | BCNN Computations | 41 |
| 4.3 | PNM Module Design | 43 |
| 4.3.1 | Terminologies and Definitions | 43 |
| 4.3.2 | Processing Data | 44 |
| 4.3.3 | Replicas in the PNM Module | 46 |
| 4.3.4 | Mode S: Processing to Minimize Space | 50 |
| 4.3.5 | Mode T: Processing to Minimize Data Transfers | 52 |
| 4.4 | Estimating Expected Performance in Theory | 55 |
| 4.4.1 | Computation Time Estimation | 56 |
| 4.4.2 | Communication Time Estimation | 59 |
| 4.4.3 | Communication and computation overlaps | 62 |
| 4.5 | Hardware Cost Estimation | 65 |
| 4.5.1 | Full-precision AlexNet | 65 |
| 4.6 | Instruction Set Architecture | 68 |
| 4.6.1 | Instructions in the ISA | 69 |
| 4.6.2 | Architectural implementation using the ISA | 70 |
| 4.6.3 | Complex DAGs within the ISA | 71 |
| 4.7 | Summary | 72 |
| 5 | Evaluation | 73 |
| 5.1 | PNM Module Evaluation Setup | 74 |
| 5.2 | Performance of the Inference Operation | 76 |
| 5.2.1 | Increasing the Number of Filters per Replica | 76 |
| 5.2.2 | Altering Bit-width | 80 |
| 5.3 | Comparison | 82 |
Table of contents

5.3.1 Different Datasets ................................................. 82
5.3.2 Different CNN Architectures .................................... 83
5.3.3 Different Data Layouts ........................................... 85
5.3.4 Area and Energy Analysis ....................................... 86
5.3.5 Expected vs. Actual Performance of Binary AlexNet ........ 94
5.3.6 Non-determinism .................................................. 96
5.4 Comparison of Area and Performance Trade-offs .................. 97
  5.4.1 Comparison with CPU and BRein Near-Memory Accelerator .... 101
5.5 Summary ............................................................ 104

6 Conclusion ............................................................. 105
  6.1 Future Research ..................................................... 107
    6.1.1 Supporting Complex DAGs ................................... 107
    6.1.2 Low Precision Integer-Only Inference ...................... 108
    6.1.3 Graph Neural Networks ...................................... 108
    6.1.4 Network Architecture Search ................................ 109
    6.1.5 Publication plans ............................................. 109
  6.2 Summary .......................................................... 109

Bibliography ............................................................ 111
List of figures

2.2 An example deep neural network with convolutional and fully connected layers 7
2.4 An example of a 2-dimensional convolution operation (adapted from https://github.com/janosh/tikz) ........................................ 10
2.7 Edge computing architecture for inference ................................. 18
2.10 Row access. (Figure is extracted from Jacob et al. [1].) .............. 23
2.11 Column access. (Figure is extracted from Jacob et al. [1].) ........... 23

4.1 Possible positions of the PNM module ...................................... 40
4.4 Different cases of the Overlap Between Communication and Computation. 47
4.6 An example of a PNM module with multiple replicas. .................. 48
4.9 Multiple replicas of the PNM processing hardware with several filters per replica. ................................................................. 52
4.12 A comparison between the order of computations taking place in modes S and T. .............................................................. 55
4.13 A 3D demonstration of how different computations proceed in modes S and T. 56
4.14 Timeline of a convolutional neural network inference. ................. 63
4.15 Different cases of the overlap between communication and computation. 64
4.16 The abstract architecture of a full-precision PNM module considering one input feature-map and one filter. .............................. 67

5.10 The performance of running a binary AlexNet in both modes S and T across various datasets including MNIST, CIFAR10, ImageNet and audio. . . . 84
5.11 The performance of running a binary AlexNet, VGGNet16 and MobileNet.v1 in both modes S and T on the PNM module. .......................... 85
5.13 Displaying the PNM module’s area in mode S against the number of filters with separate lines representing different numbers of replicas. Each line illustrates how variations in filter count impact the module’s area for a specific replica configuration. ............................. 89
5.14 Displaying the PNM module’s area in mode T, plotting the number of filters against the module’s area with separate lines representing different numbers of replicas. Each line illustrates how variations in filter count impact the module’s area for a specific replica configuration. ........................................... 90
5.15 The area breakdown for 32-bit PNM module for mode S. .................. 90
5.16 The area breakdown for binary PNM module for mode S. ................. 91
5.17 The energy consumption of the PNM module in mode T for different numbers of filters and replicas. ................................................................. 92
5.18 The energy consumption of the PNM module in mode S for different numbers of filters and replicas. ................................................................. 92
5.19 Predicted vs. Real Inference Run-time for binary AlexNet on ImageNet. ... 94
5.20 Comparison of computation and communication times versus actual time when there is one replica in Mode T .................................................. 95
5.21 Comparison of computation and communication times versus actual time when there are 64 replicas in Mode T .............................................. 96
5.22 Comparison of computation and communication times versus actual time when there is 1 replica in Mode S ..................................................... 96
5.23 Comparison of computation and communication times versus actual time when there are 64 replicas in Mode S .............................................. 97
5.24 The performance of the PNM module in a non-deterministic environment. 98
5.25 The Area-Performance product for Mode T ......................................... 99
5.26 The Area-Performance product for Mode S ......................................... 99
5.27 The Area-Performance-Energy product for Mode T ............................ 100
5.28 The Area-Performance-Energy product for Mode S ............................ 101
Chapter 1

Introduction

Deep neural networks (DNNs) are now the foundation of problem-solving techniques used in many artificial intelligence (AI) application domains. These applications include a wide range of real-life problems, from autonomous driving, mobile devices, wearable medical devices, and cancer detection to, more recently, helping scientists make breakthroughs in finding new antibiotics or amino acids [2, 3]. In all these applications, one or more techniques are needed to solve the problems in the target area, such as image recognition, decision-making, face detection, speech recognition, and natural language processing. While earlier approaches relied on hand-crafted features and rules decided by experts, nowadays, the accuracy of DNNs exceeds human accuracy in many of the aforementioned fields.

The high accuracy of DNNs comes at the cost of their expensive computational complexity and storage. Although the appearance of fast graphics processing units (GPUs) played an essential role in satisfying their computational and storage needs, with them used as general-purpose computing devices, or GPGPUs, deploying DNNs in portable devices and the Internet of Things (IoT) end-nodes with limited hardware resources (e.g. memory, bandwidth, and energy) remained a challenge for researchers [4].

One method for overcoming the resource limitation of edge devices is to have the computationally heavy process of DNNs, especially the training phase, run on servers or cloud-based infrastructure and only run the inference on the lower-resource systems. This scheme is particularly popular in training and running convolutional neural networks (CNNs). However, the inference operations can still be computationally and memory intensive. Migrating the inference operations to cloud-based infrastructure is also not a promising option, especially when it comes to applications that are interactive and need real-time decision-making. Not to mention the high cost of data movement and the security risks of relying on the cloud infrastructure [5].
To reduce the memory and computation requirements of DNN inference while maintaining an acceptable accuracy and performance, several efforts have been made, both in algorithmic and hardware aspects. One of the most effective methods for reducing the inference cost of neural networks is to reduce the precision of computation. Quantising or binarising network weights and activations reduces the precision of arithmetic operations in a CNN, but will significantly reduce the power consumption. Moreover, computations in CNNs with binary weights and activations have a very good potential to be simplified into bitwise operations, which can make them even more energy efficient. Recently, researchers demonstrated that Binary Convolutional Neural Networks (BCNNs) can achieve satisfying accuracy on image recognition datasets such as MNIST, CIFAR-10, SVHN and ImageNet [6–8].

Orthogonal to this, we can take advantage of the idea of in or near-memory processing and bring all or part of the data-intensive computations closer to memory to avoid the communication costs and, as such, positively impact performance and energy consumption. In near-memory computing, processing units still remain distinct from memory arrays. However, in-memory computing integrates logic operations into memory arrays, fundamentally blurring the distinction between processing and memory [9].

### 1.1 Research questions and hypotheses

In this dissertation, I address the following research questions:

- How bringing computation closer to memory without over-specialising the implementation does not lead to loss of generalisation or design flexibility.

- How the data layout of the input feature-maps and filters’ data can help reduce the hardware cost of the module.

- How changing the order of producing results can impact the DRAM access frequency by the module.

- What are the specific advantages and limitations of running inference in near-memory isolated environments, especially when using binarised networks?

- Can a Pareto-optimal analysis identify the most efficient configuration for the PNM module, and what criteria define this efficiency?
1.2 Contributions

- In what ways can a conceptualized Instruction Set Architecture (ISA) be crafted and leveraged to abstractly evaluate and showcase the potential and adaptability of a memory-centric processing module’s design options?

- How to design a mathematical model to predict and assess a memory-centric processing module’s performance across diverse design setups.

My hypothesis is that a near-memory accelerator for processing binary convolutional neural networks can provide high-performance inference at low hardware cost, targeting edge devices with limited hardware resources and energy budget.

1.2 Contributions

The contribution of this thesis is introducing a processing-near-memory module for binary convolution neural networks. This module is placed close to the main memory and the memory controller (MC) and requires no change in the architecture of either of them. The data layout on DRAM is designed in a way that there is no data redundancy, no need for rearranging the data either before or after the convolution and data are streamed into the PNM module row by row in the same way as it is received from DRAM. This module is flexible and not specialised to any specific CNN architecture and so, considering the fast evolution of CNN architectures, it is expected to remain practical for a longer period of time in comparison to many state-of-the-art BCNN in-memory designs. The design of this module is directed towards the inference phase of BCNNs on edge devices, where there are energy and hardware resource limitations along with restrictions on the size of the input feature-maps and filters. A thorough analysis of the various design choices within the scope of the hypothesis is done and the results are presented.

1.3 Outline

The rest of this thesis is organised as follows: Chapter 2 provides background knowledge on artificial neural networks (ANNs) and DNNs in general, with more focus on CNNs, their different layers and their corresponding computations. Later in this chapter, I explain Quantised and Binarised CNNs, and I conclude this chapter by explaining DRAM memory architecture and processing in and near memory.

Chapter 3 discusses the related work regarding accelerating DNNs using algorithmic and hardware-based approaches.
Chapter 4 discusses the foundations of PNM design for running BCNNs and two different designs of processing convolution in the PNM module. Chapter 5 evaluates the performance of the proposed PNM module. In Chapter 6, I conclude the dissertation and provide a few suggestions on possible future research on designing more efficient near-memory modules for a wider range of low-precision DNNs.
Chapter 2

Background

This chapter provides theoretical background knowledge of the main concepts discussed in this thesis. I first start with a brief description of Artificial Neural Networks (ANNs) and Deep Neural Networks (DNNs) in Section 2.1. Then the focus will be more on Convolutional Neural Networks (CNNs) in Section 2.2, their computations, and the potential gains in energy efficiency and performance of quantising in Section 2.3 and binarising the parameters of these networks in Section 2.4. In Section 2.5, I discuss the importance of inference on edge devices and the challenges associated with it. Finally, I talk about DRAM, its architecture, and the context of processing in or near memory in Section 2.7.

2.1 Artificial Neural Networks and Deep Neural Networks

Artificial neural networks (ANN) are machine-learning models that are inspired by the biological neural networks of brains. The foundational unit of the human brain is the neuron. It is optimized to receive information from other neurons, process this information in a unique way, and send its result to other neurons through connections called synapses. Likewise, a neural network is a set of modelled neurons that are connected to each other. Neurons receive information from their predecessors along their input edges (synapses), process the information, and send it to their output edges. The information that a neuron receives in an ANN is a signal containing a real-valued number. Each edge has a weight associated with it that is adjusted during the training phase of creating the neural network. Each neuron computes a non-linear function of the weighted sum of its inputs (i.e. the sum of edge weights multiplied by the edge value) added to a constant value called bias, and this becomes the output of the neuron. The bias is used in the activation function formula to offset the result to the more positive or negative side, adding more flexibility and better generalisation to the neural network. The weight of each connection increases or decreases the strength of the
Background

Fig. 2.1 Schematic of a neuron in a neural network (figure adapted from Li et al. [12])

signal at a connection, and therefore, it’s the strength of each connection that determines the contribution of their respective inputs to the neuron’s output. Figure 2.1 shows the schematic of a neuron in neural networks and its connections, and how it resembles the function of a neuron in the brain.

However, as the field of machine learning has evolved, the design and functionality of ANNs have significantly diverged from their biological counterparts. Particularly notable is the emergence of architectures like Transformers [10] and Graph Neural Networks (GNNs) [11]. Transformers, which rely on self-attention mechanisms, allow a network to weigh the importance of different parts of an input sequence, irrespective of their distance from each other. This capability, while effective for tasks like language translation and understanding, doesn’t have a direct analogue in human neural processing. On the other hand, GNNs are designed to handle graph-structured data, capturing relationships between nodes in a network. While the human brain does process relational information, the methodical and explicit way GNNs handle and propagate information in graphs isn’t mirrored in our neural circuits. These advancements highlight how ANNs, while rooted in biological inspiration, have taken paths uniquely tailored to computational needs and challenges, often venturing into territories not directly reflected in human cognition.

Building on the divergence of Artificial Neural Networks from their biological inspirations, it’s essential to understand their foundational structure. Within these networks, neurons are methodically organized into layers, with each layer tasked with specific transformations on the data it receives. Information flows from an initial input layer through intermediate layers—potentially multiple times—before reaching the final output layer. A notable subtype of ANNs is the Deep Neural Network (DNN), characterized by numerous layers nestled between the input and output. The depth of DNNs allows them to discern intricate, high-level features with a degree of abstraction unmatched by their shallower counterparts. This capability is especially evident in visual data processing. For instance, when an image’s data
2.2 Convolutional Neural Networks

Fig. 2.2 An example deep neural network with convolutional and fully connected layers

enters a DNN, the initial layers might detect rudimentary features like lines and edges. As the data progresses, these lines merge to form shapes, and these shapes coalesce into more complex patterns. By the end, the DNN can gauge the likelihood that these sophisticated features represent specific objects or scenes.

2.2 Convolutional Neural Networks

Building upon the foundation of DNNs, Convolutional Neural Networks (CNNs) are a specialized type of ANN structured through a series of layers. Each layer in a CNN modifies a set of activations using a distinct differentiable function. What sets CNNs apart from the broader category of DNNs is the specific use of convolution operations to facilitate these transformations. When constructing CNN architectures, three primary layers come into play: convolutional layers, pooling layers, and fully connected layers. Figure 2.2 shows a simple CNN. In the subsequent sections, we will delve into the intricate computations associated with each of these layers.

CNNs (Convolutional Neural Networks) are a specialized type of neural network, distinct from general-purpose neural networks. They are designed exclusively for processing grid-like data, such as the three-dimensional structure of images, effectively leveraging spatial relationships and patterns. CNN architectures have the flexibility to take in inputs with any number of dimensions (e.g. 1D, 2D, 3D). However, the most common use case is 2D images, with the third dimension representing the colour channels. CNN architectures are hence designed to process inputs with a grid-like topology. The discussion in this section will be limited to 2D CNNs, but the concepts can be extended to higher dimensions. This assumption allows us to constrain the architecture we design in this thesis by encoding specific properties into the architecture. Particularly, neurons in all convolutional layers, including the fully
Fig. 2.3 The layer containing $s_1$ and $s_2$ neurons is a fully connected layer as all its neurons ($s_1$ and $s_2$) are connected to all the neurons in its previous layer ($a_1$, $a_2$ and $a_3$). Here $b$ represents the bias value.

Connected layers, are arranged in three dimensions: width, height, and depth. The word depth here refers to the third dimension of an activation’s 3D matrix, not a neural network’s depth.

2.2.1 Fully Connected Layers

A fully connected layer, also known as a dense layer, is a fundamental component in neural networks, particularly prominent in the architecture of deep learning models. In a fully connected layer (FC), all neurons have connections to all neurons in the previous layer. Therefore, activations in each layer can be computed with a matrix multiplication followed by a bias offset. Based on Figure 2.3, we can interpret the output of each layer as:

$$
\begin{align*}
    s_1 &= [(w_{11}.a_1) + (w_{12}.a_2) + (w_{13}.a_3)] + b_1 \\
    s_2 &= [(w_{21}.a_1) + (w_{22}.a_2) + (w_{23}.a_3)] + b_2
\end{align*}
$$

(2.1)

Vectorizing equation 2.1 leads to:

$$
\begin{bmatrix}
w_{11} & w_{12} & w_{13} \\
w_{21} & w_{22} & w_{23}
\end{bmatrix}
\begin{bmatrix}
a_1 \\
a_2 \\
a_3
\end{bmatrix} +
\begin{bmatrix}
b_1 \\
b_2
\end{bmatrix} =
\begin{bmatrix}
s_1 \\
s_2
\end{bmatrix}
$$

(2.2)
The matrix representation of which is as below, where $B^T$ and $W^T$ are the transposed matrices of biases and weights:

$$(W.A) + B^T = S(A)$$

$$(W^T.A) + B = S(A) \quad (2.3)$$

The linear layer is typically followed by a non-linear activation function, such as the Rectified Linear Unit (ReLU) \[13\] function, which is defined as follows:

$$f(x) = \max(0, x) \quad (2.4)$$

The ReLU function is a non-linear function that is used to introduce non-linearity into the network. This allows for modelling more complex functions. Further discussion on non-linearity is beyond the scope of this thesis.

When a CNN is used to analyze an image, for each pixel in that input image, the pixel’s importance is encoded as the value for a corresponding neuron in the input layer. For the 28 $\times$ 28 pixel images of the MNIST dataset, for example, this means the network has 784 (= 28 $\times$ 28) input neurons depicted as a vertical line of neurons considering the grey-scale colouring of MNIST input images.

Considering a fully connected layer with $n$ input neurons and $m$ output neurons, the number of weights will be $n \times m$. Additionally, having a bias for each output neuron results in $(n + 1) \times m$ parameters to be learned. Hence, assuming $l$ is the number of layers, $n$ is the number of neurons in the previous layer, $w$ is the corresponding weight for each neuron and $m$ is the number of neurons in a layer, the total computations needed will be $m \times \sum_{i=1}^{l} (n \times w + 1)$.

### 2.2.2 Convolutional Layers

The major difference between fully connected and convolutional layers is that the neurons in each convolutional layer are connected only to a subset of the neurons in the previous layer in the input instead of all of the neurons in a fully connected manner. The spatial extent of this connectivity is a hyperparameter called the receptive field of the neuron (equivalently this is the filter size) \[14\].

The core of the convolutional layer is the convolution operator. The convolutional operator performs element-wise multiplication of a filter (or kernel) with a portion of the input data, followed by a summation, effectively extracting localized features. Figure 2.4 shows a convolutional filter operating on a matrix. It is notable that the convolutional kernel ($K$) traverses across the input matrix ($I$). This is a feature of the convolutional layer, that
allows the model size to be small even while operating on very large input sizes. This makes convolutions especially useful for computational environments with restricted memory and processing.

There are four hyperparameters that are used to configure a convolution layer:
1. Filter and filter size \( (K) \): A filter is essentially a feature/pattern detector. Filters represent combinations of connections that get replicated across the entirety of the input [16]. For gaining more accuracy, a smaller filter size is better. However, if we set the filter size of the first convolution layer to small, it will take a lot of memory (as less spatial down-sampling occurs). In the context of convolution neural networks, filters are also referred to as kernels.

2. Stride \( (S) \): This determines how many pixels the kernel window will slide over the input per iteration.

3. Zero padding \( (P) \): This feature allows us to control the spatial size of the output volumes. By putting zeros on the image border, it is possible for the convolution output size to be the same as the input size.

4. Number of filters \( (F) \): We described earlier that filters are feature/pattern detectors. So the number of filters is actually the number of patterns that the convolutional layer will look for.

Let’s consider we have configured a convolutional layer, which means that we have already decided the values of the weights of the network. The next step is to apply the kernel across the input and add a bias to the result. Each pixel of the output will be the result of element-wise matrix multiplication of the kernel and input matrix. The output calculation of a 2D convolution without zero padding is shown in Figure 2.5.

### 2.2.3 Pooling Layers

The core operation in a pooling layer is down-sampling the input feature-map by sliding a two-dimensional filter over the whole input feature-map matrix and producing an output for each sliding window that is the summarised information of that window. What we mean
here from summarizing the information in terms of mathematical operation, is either using a MAX function (finding the maximum value of a function), an average function, or using a global approach, which is mapping a group of nodes to one specific value. Figure 2.6 shows an example of a max-pooling layer in operation; here, the max-pooling is happening over a matrix of size $2 \times 2$.

Inserting a pooling layer after convolutional layers in a CNN architecture, specifically after a layer whose nodes were produced from a non-linear function (e.g. ReLU function) is a common approach that helps reduce the spatial size of the output representations and the number of parameters, and therefore controls over fitting.

A pooling layer requires two hyperparameters: the spatial extent ($F$) and the stride ($S$). It produces a volume of size $H_{out} \times W_{out} \times D_{out}$ where:

$$H_{out} = \frac{H_{in} - F}{S} + 1$$
$$W_{out} = \frac{W_{in} - F}{S} + 1$$
$$D_{out} = D_{in} \quad (2.5)$$

In practice, there are only two commonly seen variations of the max-pooling layer: a pooling layer with $F=3$, $S=2$ (also called overlapping pooling), and a pooling layer with $F=2$, $S=2$, of which the latter is more common. If the size of a pooling layer is larger, it becomes too destructive. This can be seen in the discussion on the effect of pooling layers in Wu and Gu [17]. The drop probability of a value is related to the size of the pooling layer. They discussed the effect of the size of the pooling layer on the performance of the network.

2.3 Quantised Convolutional Neural Networks

To perform efficient inference on complex networks, several techniques have been proposed. These include improved network designs and network search, network pruning, and network quantisation. Quantisation encodes the floating-point values of the network’s weights and activations to a reduced bit-width. Depending on how much smaller the new bit-width of the parameters is, the total memory footprint of the network would be reduced by the same factor. Using fixed-point-encoded values rather than the original floating-point values of network parameters is also a type of quantisation. Here, the computations can be performed using integer values instead of floating-point; therefore, the operations’ complexity and
the hardware used to implement them would be simpler, which would suit a low-area and low-power system.

### 2.3.1 Uniform Affine Quantisation

Three quantisation parameters, namely the scale factor $S$, the zero-point $Z$, and the bit-width $b$ define the uniform affine quantisation function. This function takes real values in floating-point and maps them to an integer grid using the scale factor and the zero-point. The size of the integer grid depends on the bit-width. A popular choice for a quantisation function is as follows:

$$Q(r) = \text{Int}(r/S) - Z$$  \hspace{1cm} (2.6)

where $Q$ is the quantisation operator, $r$ is a real-valued input (activation or weight), $S$ is a real-valued scaling factor, and $Z$ is an integer zero-point. The $\text{Int}$ function maps a real value to an integer value through a rounding operation (e.g. round to nearest and truncate). As the resulting quantised values (also known as quantisation levels) are uniformly spaced, this method of quantisation is known as uniform quantisation. There are also non-uniform quantisation methods whose quantised values are not necessarily uniformly spaced. Generally, non-uniform quantisation enables us to better capture the signal information, by assigning bits and discretising the range of parameters non-uniformly. However, non-uniform quantisation schemes are typically difficult to deploy efficiently on general computation hardware, e.g. GPU and CPU, because of their more mathematically complex operations.

The scaling factor $S$ is important in uniform quantisation as it breaks down a given range of value $r$ into a number of partitions:

$$S = \frac{\beta - \alpha}{2^b - 1}$$  \hspace{1cm} (2.7)

where $[\alpha, \beta]$ denotes the clipping range, and $b$ is the quantisation bit-width. Based on this equation, choosing the clipping range (also known as the calibration process) is the first thing to do for defining the scaling factor. One approach for the calibration process is a straightforward choice of $\alpha = r_{\text{min}}$ and $\beta = r_{\text{max}}$. This approach is an asymmetric quantisation.

It is also possible to choose a symmetric clipping range where $\alpha = -\beta$. Another straightforward choice for this scenario is also using the $r_{\text{min}}$ and $r_{\text{max}}$ of the signal,
such that $\alpha = -\beta = \max(|r_{min}|, |r_{max}|)$. Using symmetric quantisation helps simplify the quantisation function in equation 2.6 by replacing the zero point with $Z = 0$:

$$Q(r) = \text{Int}(r/S)$$

Therefore, this is widely adopted in practice because removing the zero point can lead to a significant reduction in computational cost during inference. However, it is sub-optimal for cases where the range could be skewed and not symmetric. In those situations, using asymmetric quantisation is preferred [18].

### 2.3.2 Quantisation Granularity

In CNNs, each activation input to a layer goes through a set of different convolutional filters. The value of each of these filters can have a range. Therefore, what level of network granularity we choose to perform the weights’ calibration process is another differentiator in quantisation methods. As such, quantisation methods can also be categorised as follows:

1. **Layer-wise Quantisation**: In layer-wise quantisation, the same clipping range is applied to all filters that belong to the same layer.

2. **Channel-wise Quantisation**: Channel-wise quantisation dedicates different clipping ranges to different channels.

3. **Group-wise Quantisation**: If one group’s multiple different channels inside a layer to calculate the clipping range (of either activations or convolution kernels), it is called Group-wise quantisation.

4. **Sub-Channel-wise Quantisation**: If the clipping range is determined with respect to any groups of parameters in a convolution or fully connected layer, it is called sub-channel-wise quantisation. This approach could add considerable overhead since the different scaling factors need to be taken into account when processing a single convolution or fully connected layer.

Currently, the standard method used for quantising convolutional kernels is channel-wise quantisation. The reason, as explained by Garg et al. [19], is how this strategy adjusts each kernel’s clipping range with negligible overhead. While on the other hand, sub-channel-wise quantisation may result in significant overhead.
2.4 Binarised Convolutional Neural Networks

This section explains binarised convolutional neural networks (BCNNs) as introduced in BinaryNet [20]. A binarised CNN, in general, is an extreme form of quantised CNN where the bit-width of the network parameters is set to one. In a binarised CNN, the neural network has binary weights and activations at both run time and train time; however, at train time, the real-valued gradients of the weights are accumulated in real-valued variables. In a trained BCNN, the weight matrix only contains two values, either +1 or -1, while -1 is denoted as 0 in the computations. Also, the binarisation function used for the weights is a deterministic function, while for the activations it is a stochastic function. The stochastic binarization function is more appealing than the corresponding deterministic function (the sign function) but harder to implement as it requires “random bits generation” hardware when quantising. The deterministic binarisation function is:

\[ x^b = \text{Sign}(x) = \begin{cases} +1 & \text{if } x \geq 0, \\ -1 & \text{otherwise.} \end{cases} \]  

(2.9)

where \( x^b \) is the binarised variable and \( x \) the real-valued variable. The stochastic binarisation function is:

\[ x^b = \begin{cases} +1 & \text{with probability } p = 1 - \sigma(x), \\ -1 & \text{with probability } 1 - p. \end{cases} \]  

(2.10)

where the “hard sigmoid” function is:

\[ \sigma(x) = \text{clip} \left( \frac{x+1}{2}, 0, 1 \right) = \max \left( 0, \min \left( 1, \frac{x+1}{2} \right) \right) \]  

(2.11)

2.4.1 Training a BCNN

The BinaryNet training algorithm consists of three major phases: “forward propagation”, “backward propagation” and “accumulating the gradients of the parameters”. The binarisation function is only applied to weights and activations in the forward propagation phase. Also, batch normalisation is applied to activations before binarising them to solve the problem of internal covariate shifts. These parameters are then used to calculate gradients in backward propagation; however, the gradients themselves are not binary.

It is worth noting that the exact gradient of the cost concerning weights or activations would be zero as the derivative of the sign function is zero almost everywhere. This remains true even with using the stochastic binarisation function instead of the sign function. To solve
Background

this problem, Bengio et al. [21] studied the question of estimating or propagating gradients through stochastic discrete neurons. Based on their experiments, the fastest training was obtained when using the “straight-through estimator”. In BinaryNet, a version of the straight-through estimator is used that takes into account the saturation effect and uses deterministic sampling of the bit.

If the sign function is:

\[ q = \text{Sign}(r) \]

and assuming an estimator \( g_q \) and the gradient \( \frac{\partial C}{\partial q} \) with the straight-through estimator; then, the straight-through estimator of \( \frac{\partial C}{\partial r} \) would be:

\[ g_r = g_q \cdot \frac{1}{|r|} \cdot 1 \]

2.4.2 Running a BCNN

As mentioned earlier, in a trained BCNN, the weight matrix only contains two values, either +1 or -1, while -1 is denoted as 0 in the computations. Therefore, all dot operations between the inputs of each layer and the weights in the inference phase can be easily replaced by bitwise XNOR, with the exception of the first layer. The inference operation is shown as equation 2.12 and is followed by batch normalization. In this equation, A stands for the result matrix of the previous activation layer, W stands for the weight matrix and B is the bias matrix. As all the other layers would be fed by binarised values from previous layers, the output of this equation would be a count of the number of 1s and 0s and deciding which one is dominant. This makes it possible to replace the additions with hardware for counting the number of 1s and 0s.

\[ A' = f\left(\sum W \times A + B\right) \quad (2.12) \]

For the first layer, considering that it has real-valued inputs instead of dot operations, only a sign change will be sufficient. However, the effect of computation overhead of the first layer in comparison with the other layers is not a major issue as the first layer of the CNN is often the smallest convolution layer, both in terms of parameters and computations [22]. Courbariaux et al. [20] proposed a shift-based batch normalization (SBN) technique, and showed through experiments that using SBN rather than normal batch normalization will not result in accuracy loss. Based on their method for calculating batch normalization, SBN can approximate it almost without multiplication.

In that algorithm, batch normalization of the activation neurons is calculated twice. However, in 2017, Yonekawa and Nakahara [23] showed that because in binary neu-
2.5 Inference on edge devices

An edge device is a computational unit designed to process data at or near the point of data generation, characterized by its ability to perform autonomous operations within the constraints of limited power availability, computational resources, and environmental conditions. Its role is pivotal in distributed computing networks, where it reduces latency, conserves bandwidth, and enhances data privacy by locally analyzing and acting upon data without necessitating constant communication with centralized data centres. Power availability is a critical defining aspect, as edge devices are often deployed in scenarios with strict energy constraints, necessitating highly efficient hardware architectures and energy management strategies to optimize performance and operational longevity. The efficiency of an edge device is thus not only measured in computational speed and accuracy but also in its capacity to maximize the utility of each watt of power consumed, adapting its precision and processing strategies to balance energy use with the demands of its application environment. It is notable that the power budget, memory, and computational capacity of edge devices are much lower than those of data-centre servers, as shown in Table 2.1. Also, the power budget is dependent on the availability of power source close to the device, which is not always the case. For instance, a sensor node in a remote location may have a limited power source, such as a battery or a solar panel, which may not be able to provide a continuous power supply. In
Table 2.1 Comparison of different devices for power, memory and compute.

<table>
<thead>
<tr>
<th>Device</th>
<th>Power</th>
<th>Memory</th>
<th>Compute</th>
</tr>
</thead>
<tbody>
<tr>
<td>Nvidia Jetson Xavier</td>
<td>30W</td>
<td>8GB</td>
<td>512 CUDA cores</td>
</tr>
<tr>
<td>Raspberry Pi 4</td>
<td>15W</td>
<td>8GB</td>
<td>4x 1.5GHz ARM</td>
</tr>
<tr>
<td>Nvidia Jetson Nano</td>
<td>5W</td>
<td>4GB</td>
<td>128 CUDA cores</td>
</tr>
<tr>
<td>Arduino Nano</td>
<td>0.5W</td>
<td>2KB</td>
<td>16MHz ARM</td>
</tr>
<tr>
<td>Nvidia DGX H100</td>
<td>11KW</td>
<td>2TB</td>
<td>16,384 CUDA cores</td>
</tr>
<tr>
<td>Nvidia DGX A100</td>
<td>6KW</td>
<td>2TB</td>
<td>6,912 CUDA cores</td>
</tr>
<tr>
<td>Cerebras CS-1</td>
<td>20KW</td>
<td>12TB</td>
<td>400,000 cores</td>
</tr>
</tbody>
</table>

such cases, the power budget is even more constrained. On the other hand, a device that is
collected to a power source can afford to have a higher power budget.

This thesis is interested in the inference phase of neural networks on edge devices. In
the context of a neural network life-cycle, training is the process of learning the parameters/weights of a model to make accurate predictions on a given training dataset. Inference, on the other hand, uses a trained model to make predictions on new data. Typically, training is a computationally intense process and needs to be performed only once. Once a model is trained, it can be deployed to make predictions on new data.

![Fig. 2.7 Edge computing architecture for inference](image)

The table 2.1 compares some low-power devices with a typical DNN data-centre server. Nvidia and Cerebras provide the most powerful GPU and ASIC servers, respectively, while Arduino and Raspberry Pi are examples of low-power devices. Nvidia Jetson Xavier and Jetson Nano are examples of embedded GPU devices. It is visibly clear that the power budget,
2.5 Inference on edge devices

memory and computational capacity of low power and embedded devices is much lower than a server.

Inference is computationally less intensive and is performed more frequently than training. Inference on edge devices presents interesting challenges as these devices are typically resource-constrained. For example, the Arduino Nano 33 BLE Sense, a popular microcontroller board, has an Arm Cortex-M4F CPU running at 64 MHz, 1 MB of flash memory and 256 KB of RAM [24]. It draws a maximum of 19.5 mA of current at 3.3 V, and has a maximum power consumption of 64.35 mW. In contrast, a DGX H100 server from NVIDIA has 8 NVIDIA H100 GPUs, each with 80 GB of memory, 2,048 GB of CPU memory, and 30 TB of NVMe SSD storage [25]. It can draw up to 11 kW of power. Thus inference on the edge has to be performed with limited computational resources and power budget as compared to a data-center. Training of ML models typically happens in the cloud, and inference on edge devices is performed locally on the device. Figure 2.7 shows the edge computing architecture typically used.

While local inference on edge devices offers lower latency to obtain results than cloud-based alternatives, it presents its own set of challenges due to the devices’ limited computational resources. Consequently, striking a balance between model size, performance and speed becomes imperative for successful deployment on edge devices, a topic we will explore in this thesis.

In recent years, there has been a significant surge in the proliferation of edge devices, characterized by their small form factor, reliance on battery power, and constrained computational capabilities. These devices, such as smartphones, smartwatches, smart home devices, and sensors, are often interconnected and capable of communication through the Internet. Home devices such as televisions, refrigerators, and thermostats are also becoming increasingly connected to the Internet. An intermediate power category is that of vehicles, which are becoming increasingly connected and autonomous. Vehicles cannot afford to rely on the cloud for inference, as they need to make real-time decisions. The risk of a network failure or a network delay is too high for autonomous vehicles. For such devices, performing inference locally on the device is the only viable option. Another example is the Internet of Things (IoT) devices, which are typically small, low-power devices that are capable of sensing, actuating, and communicating.

Many popular applications are examples of edge computing. For example, SwiftKey is a popular keyboard app that uses machine learning to predict the next word a user will type. It uses a neural network model that is trained on the user’s typing history and is deployed on the user’s device [26, 27]. Google’s Translate app also uses a neural network model to
translate text in real-time [28]. These applications are clearly examples of edge computing, where data is processed locally on edge devices rather than in the cloud.

### 2.6 Low-precision Networks and Accuracy

As discussed earlier, accuracy is a concern for low precision models. However, it’s important to note that accuracy is more a function of the model architecture than the hardware architecture. This is evident from the results of BitNet b1.58 [29]. BitNet b1.58 uses a ternary system for its parameters, set to -1, 0, or 1, effectively requiring an average of only 1.58 bits per weight. This approach allows BitNet to achieve substantial efficiency gains over traditional binary systems. Impressively, despite its lower precision, BitNet b1.58 matches the performance of full-precision large language models in terms of perplexity and overall effectiveness. This is a significant achievement, given that maintaining high accuracy levels is a challenge for most low-precision architectures. BitNet’s success suggests that future efficient neural network designs will combine the advantages of low-precision computing with the high accuracy standards required for deep learning applications.

My work is an attempt in exploring hardware architectures that would enable low precision deep learning architectures. Methods such as BitNet provide a direction for the hardware designers to optimise along. Our method is implemented for Binary networks, but can be extended for BitNet as well.

Market pressure for larger models with less inference time and cost is driving a trend towards more efficient hardware for deep learning inference. The table 2.2 illustrates the decreasing precision of Nvidia’s data-centre GPUs over time, a trend likely to continue. The trend is clearly towards lower precision, with the B200 architecture expected to support 4-bit floating point precision. The trend can be extrapolated to 2 bit or ternary precision in the future. Insights from architectures like BitNet are guiding such hardware development. Our work as hardware designers aims to inform algorithm developers about the hardware advancements supporting low precision deep learning models.

<table>
<thead>
<tr>
<th>Model</th>
<th>Lowest Precision</th>
<th>Release Year</th>
</tr>
</thead>
<tbody>
<tr>
<td>A100</td>
<td>FP16</td>
<td>2020</td>
</tr>
<tr>
<td>H100</td>
<td>FP8</td>
<td>2022</td>
</tr>
<tr>
<td>B200</td>
<td>FP4</td>
<td>2024</td>
</tr>
<tr>
<td>Future architecture</td>
<td>Ternary?</td>
<td>2026?</td>
</tr>
</tbody>
</table>
2.7 DRAM and Processing Near-Memory

In 1995, Wulf and McKee [30] predicted the situation where improvements to processor speed would be masked by the much slower improvement in dynamic random access memory (DRAM) speed, even in the situation of having ideal caches. They referred to the gap between processor and memory performance as the Memory Wall and determined that it would become far worse within the subsequent 5 to 10 years. They concluded that future computing systems would unambiguously be dominated by the technological progress in memory performance.

While Wulf and McKee’s work concentrated on the performance gap between processor and memory technologies, Kagi et al. [31] investigated the off-chip bandwidth between the processor package pins and the memory package pins. They declared that by improving processors’ performance, more data and instructions are required, and, as a result, more cache lines will have to be moved from the memory through the processors’ cache hierarchy. Considering the possibility that each cache line might be underutilized and would be replaced with more important cache lines, this would lead to a significant data-transfer overhead and demand for even higher bandwidth. This issue, referred to as the bandwidth wall, would be worsened by running multiple threads on multiple cores within the same package.

Besides the latter two issues, new processor architectures will have higher leakage power as a result of technology scaling, growing transistor density, and rising clock speed/frequency, which would lead to the power wall. This high power dissipation (Watts per cm$^2$) would be increased dramatically due to higher power leakage, steering towards thermal and hence cooling constraints, resulting in the potential of melting the computer chips [32]. Both processor architectures and memories are typically associated with the power wall.

2.7.1 Processing In or Near Memory

The aforementioned bottlenecks motivated the idea of bringing all or part of the data-intensive computations nearer to memory (so-called near-data computation). This concept is not new and has been studied since the 1990s [33–35]. However, none of the schemes developed at that time have been successfully applied due to two reasons. First, it was practically difficult to integrate computation logic with DRAM due to incompatible fabrication processes. Second, the lack of data-centric killer applications prevented the industry from investing in the adoption of such techniques.

Since that time, many other techniques, such as prefetching, multithreading or having hierarchical memory, have tried to mitigate these inefficiencies to some extent, but a traditional von Neumann computing architecture has its intrinsic limitation on data access
due to the latency inherent in the hierarchical memory and interconnect architecture. Even in modern systems, the data transfer between CPUs and off-chip memory consumes two orders of magnitude more energy than a floating-point operation [36]. On the other hand, the constraints on the near data computations idea have been resolved to a reasonable extent in recent years. State-of-the-art memory stacking techniques [37] answered the problem of its fabrication process difficulty, and the dawn of the big-data era caused a bloom in data-centric applications. As a result, offloading computation to memory or closer integration of logic and memory is being studied again under different names, such as near-memory processing, logic-in-memory, computing-in-memory, or processing-in-memory (PIM).

These efforts may be classified into two categories: moving logic closer to memory or near memory computing [34, 38–41] and performing computations within memory structures, or in-memory computing [42–45]. Near-memory computing refers to bringing logic or processing units closer to memory. Notwithstanding the closer integration, processing units still remain distinct from memory arrays. In-memory computing integrates logic operations into memory arrays, fundamentally blurring the distinction between processing and memory [9].

2.7.2 DRAM Architecture

At a high level, a DRAM chip consists of multiple banks (shown in Figure 2.8) that can be accessed in several different ways. They can be designed in a way that acts in unison, they can act completely independently, or in a manner that is somewhere between the two [1]. In general, they are assumed to work independently. Each bank is further divided into multiple memory arrays, the organization of which is illustrated in Figure 2.9. Each memory array
consists of multiple rows of DRAM cells connected to a row of sense amplifiers. Each row of DRAM cells shares a word line that controls the connection between the cells of that row and the sense amplifiers. Similarly, each column of DRAM cells shares a bit line that connects those cells to the corresponding sense amplifier.

To demonstrate the steps involved in a DRAM cell access, let’s consider a simple architecture, including a processing unit (CPU), memory controller, and DRAM device. The DRAM device connects indirectly to the CPU through a memory controller; the CPU connects to the memory controller through some form of network (bus, point-to-point, crossbar, etc.), and the memory controller connects to the DRAM through another network.

Once the request arrives at the memory controller, it is queued until the DRAM is ready and all previous and/or higher-priority requests have been handled. The memory controller must decompose the provided data address into rank, bank, row, and column addresses. Once
they are identified, the sense amplifiers first PRECHARGE the appropriate bit-lines to 12Vdd. Then cell access is triggered by the ACTIVATE command to the corresponding DRAM row; the row address and bank identifier will be sent over the address bus and signalling the DRAM’s RAS pin. This tells the DRAM to send an entire row of data (thousands of bits) into the DRAM’s sense amplifiers. After the sense amplification, data can be accessed from the sense amplifiers using the READ/WRITE command to the corresponding column of data within the DRAM row (sending the column address and bank identifier over the address bus and signalling the DRAM’s CAS pin). Once the data is accessed, the memory array can be taken back to the PRECHARGE state by issuing a PRECHARGE command. This process is shown in Figures 2.10 and 2.11.

2.8 Summary

This chapter has described the background of my thesis, including neural network quantisation and how DRAM works. The next chapter performs a review of salient works from the literature that improve CNNs and design hardware to process them.
Chapter 3

Literature Review

Over the past decade, deep neural networks (DNNs) have been widely deployed in many fields and have shown growing success in accuracy and performance both in academia and industry. More recently, the number of DNN layers in state-of-the-art models has been increasing and can be as high as tens of thousands of layers with billions of parameters, making them costly both in terms of computation and storage [46]. These characteristics lead to two main issues running large DNNs: huge memory bandwidths and computational complexity. While the appearance of fast graphics processing units (GPUs) played an essential role in satisfying these needs, deploying DNNs in portable devices and IoT end-nodes with limited hardware resources (e.g. memory, bandwidth, and energy) remained a challenge for researchers.

To obtain a good balance between the accuracy of DNNs and their resource limitations, one typical scenario is to train them on more powerful systems like servers and only run the inference on the lower-resource systems. This scheme is particularly popular in training and running CNNs. However, the inference phase itself is still quite computationally and memory-intensive. To reduce the memory and computation requirements of DNN inference while maintaining acceptable accuracy and performance, several efforts have been made. These efforts can be categorized into algorithmic approaches, hardware approaches, and a combination of these two methods (software and hardware co-design).

The design space for algorithmic approaches for inference in edge computing is quite large. Edge devices are constrained by the power budget, computational resources, and memory capacity. DNN inference as discussed in Chapter 2 is a compute-intensive task, and the memory access patterns are also quite complex.

The main goal of these approaches is to reduce the space overhead and power consumption of DNNs while maintaining their accuracy and performance. Computational complexity and memory access are the two main factors that affect the performance of DNNs. The memory access issue is addressed by designing a new memory architecture or by bringing...
Table 3.1 Comparison of different approaches for accelerating CNNs.

<table>
<thead>
<tr>
<th>Approach</th>
<th>Algorithmic</th>
<th>Hardware</th>
<th>Co-design possibility</th>
</tr>
</thead>
<tbody>
<tr>
<td>Quantisation</td>
<td>✔</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Pruning</td>
<td>✔</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Tensor Decomposition</td>
<td>✔</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Knowledge Distillation</td>
<td>✔</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Network Architecture Search</td>
<td>✔</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ASIC</td>
<td></td>
<td>✔</td>
<td></td>
</tr>
<tr>
<td>FPGA</td>
<td>✔</td>
<td>✔</td>
<td></td>
</tr>
<tr>
<td>Co-Processor</td>
<td>✔</td>
<td>✔</td>
<td></td>
</tr>
<tr>
<td>Programmable In-memory</td>
<td>✔</td>
<td></td>
<td>Limited</td>
</tr>
<tr>
<td>Programmable Near-memory*</td>
<td>✔</td>
<td>✔</td>
<td></td>
</tr>
</tbody>
</table>

the computation closer to memory. The computational complexity issue is addressed by designing a new hardware architecture for DNNs or using algorithmic approaches to reduce the computational complexity. Such systems require deep hardware and software co-design to be able to exploit the parallelism in the system. Table 3.1 shows a comparison of different approaches for accelerating CNNs. In the table, the possibility of a co-design is shown in the last column. Co-design is not possible for ASIC-based designs, as the hardware is designed for a specific algorithm. For other architectures, the hardware implementation can be designed to suit the software implementation and vice versa. The design of the programmable near-memory approach is the focus of this thesis.

At the algorithmic level, the goal is to make DNNs run more efficiently by directly designing or deriving a more efficient neural network with or without using a base model. On the other hand, in the hardware approach, the goal is to design the hardware in a way that makes running DNN computations more efficient. The interaction between the hardware architecture and software algorithm is also a present trend in designing efficient DNN inference systems, which is addressed as a co-design approach in this chapter.

3.1 Algorithmic Approach

In this approach, designers either manually create a specifically designed CNN (e.g. MobileNet, SqueezNet and ShuffleNet) or automatically find an efficient base model design by searching in a predefined search pool, or they try to modify a given design to obtain an efficient compressed model. Well-established approaches like quantisation of network parameters and pruning, and more recent approaches like tensor decomposition and knowledge distillation, fall into the latter category. They use the given base model and try to
reduce its memory accesses and computations. On the other hand, more recent approaches, such as Network Architecture Search (NAS), programmatically search for highly efficient neural network structures. In this section, I look through the quantisation, in-memory, and near-memory approaches in more detail as they are the focus of this thesis, and briefly go through the other methods.

### 3.1.1 Quantisation

The main idea of DNN quantisation is to transform the real-valued parameters of a network into a minimised-bit-width approximated version of them by applying a quantisation function to these values. Minimising the bit-width of network data leads to simpler computations and less storage space. However, it comes at the cost of accuracy loss due to some network information getting lost during this transformation. The precision of a network with quantised parameters varies based on the length of its data bit-width. Over the past decade, there has been a great amount of research on finding the best quantisation function and the optimal bit-width for specific DNN architecture, all with the target goal of obtaining a competitive accuracy compared to the pre-quantisation networks.

Two prominent approaches to DNN quantisation are Post-Training Quantization (PTQ) [48] and Quantization Aware Training (QAT) [49]. There are other methods such as Zero shot quantisation [50], which quantises the model without retraining, and Quantisation with fine-tuning [51], which quantises the model and then fine-tunes it to recover the accuracy. But we are focusing on PTQ and QAT as they are the most widely used methods. PTQ involves quantising the model after it has been fully trained with floating-point precision. While this method is straightforward and doesn’t require retraining the model, there can be a noticeable accuracy drop due to the abrupt change in precision. On the other hand, QAT incorporates quantisation into the training process itself. By simulating the effects of quantisation during training, the model becomes inherently robust to the reduced precision. This often results in a quantised model that maintains closer accuracy to its original floating-point counterpart.
However, QAT might necessitate modifications to the standard training procedure, making it a bit more involved than PTQ. Both PTQ and QAT aim to strike a balance between computational efficiency and model performance, and the choice between them often depends on the specific requirements and constraints of a given application. Figure 3.1 shows the different mechanisms followed by QAT and PTQ for the quantisation of a Neural Network. It is notable that in PTQ, there is no requirement to re-train the model, but the training process in QAT requires fine-tuning the model (restricted training) to quantise the model. Such retraining has an effect on model performance as shown in [49, 47].

A comparison of the effect of quantisation using QAT and PTQ is provided in [47] using the Nvidia TensorRT library. It shows that for a restricted set of examples, QAT-based models were able to achieve close to full model qualitative performance during inference, yet the time for computing was reduced by 8 to 10 times the baseline. Such results are useful in environments where power and computational power are restricted in comparison to a data centre environment which allows thousands of kilowatts (KW) per server. An example is the Nvidia Jetson appliance [52], which is specifically designed for edge computing. Such devices use around 30 watts, whereas an H100 DGX GPU server can use up to 11KW [25]. It is notable that ML training is a slow and costly process, with the recent GPT4 costing $65 million to train [53]. The cost is not restricted to transformer-based models, convolution-based models also do well when trained at scale [54]. Such costly training is only possible in a data centre environment and is not suitable for edge devices. In summary, training models with methods such as QAT in a data centre allows for highly accurate models at scale, and deploying such models in a restricted environment such as edge computing is a valid and commonly used mechanism.

Following the exploration of training mechanisms for quantisation and their practical deployment in hardware environments with varying computational resources, we now transition to a distinct yet related topic - Binarisation. This technique further epitomizes the trade-off between computational efficiency and model performance. By reducing the precision of the model parameters to the extreme of single-bit binary values, binarisation offers significant savings in memory and computational requirements, enabling the deployment of deep learning models in even the most resource-constrained environments.

**Binarisation**

Courbariaux et al. [55] proposed binarising the weights of deep neural networks. During training, the weights are stored and updated with full precision but binarised for forward and backward propagation. This is an example of QAT. Further, they show that binarising weights when computing the parameters’ gradients provides a form of regularization that can help to
3.1 Algorithmic Approach

generalize better (as previously shown with variational weight noise in Dropout [56]) and helps their algorithm to gain state-of-the-art results on data sets such as MNIST, CIFAR-10 and SVHN. In their approach, called BinaryConnect, they used a deterministic binarisation function rather than a stochastic function. In both formulas, as shown below, a “hard sigmoid” function $\sigma$ is used to determine the probability distribution.

$$x_{det}^b = \text{Sign}(x) = \begin{cases} +1 & \text{if } x \geq 0, \\ -1 & \text{otherwise} \end{cases} \quad (3.1)$$

$$x_{sto}^b = \begin{cases} +1 & \text{with probability } p = 1 - \sigma(x), \\ -1 & \text{with probability } 1 - p. \end{cases}$$

$$\sigma(x) = \text{clip} \left( \frac{x + 1}{2}, 0, 1 \right) = \max \left( 0, \min \left( 1, \frac{x + 1}{2} \right) \right) \quad (3.2)$$

In a follow-up work [57], the same authors extended their idea and proposed quantising the inputs to the layers in backward propagation to 3 or 4 bits and replacing the multiplications with shift-add operations. They showed that their CNN even outperforms the full-precision version in terms of accuracy.

Following this trend, BinaryNet [20] and XNOR-NET [58] were proposed. Both of these works consider the idea of binarisation of the layer inputs, such that the proposed algorithms can be implemented using only XNOR operations. Courbariaux et al. [20] binarised both weights and activations. The binarisation functions used in this approach were the same deterministic and stochastic binarisation functions as described earlier in BinaryConnect [55]. However, the stochastic function was only used for activations at train time, but for other implementations, the deterministic binarisation function (sign function) was used. The reason for the limited use of stochastic binarisation is its required hardware for generating random bits when quantising.

Rastegari et al. [58] presented Binary Weight Networks (BWNs) and XNOR-Net, two approximations to standard CNNs, which were shown to outperform BinaryConnect and BinaryNet by large margins in terms of accuracy on ImageNet. In BWNs the convolutional filters are approximated with binary values (-1,1); in XNOR-Networks, both filters and input to convolutional layers are binary. What makes BWN different from BinaryConnect and BinaryNet is its different method of binarisation and also different network structure: more precisely, in BWN the output channels are scaled by the mean of the real-valued weights, gaining better accuracy on the ImageNet dataset when using AlexNet [59]. In XNOR-Net, by
also binarizing the input images, they achieved an accuracy of 69.2% in the top-5 measure, in comparison with the 80.2% of the setup in BWN.

Based on the XNOR-NET algorithm, Wu et al. [60] proposed another approach that improved the accuracy up to 81% using log-loss with soft-max pooling. Their results showed that their approach could even outperform the accuracy of results on AlexNet. However, the XNOR-based approach is still not mature enough as it has only been proven on a few networks by a small research community.

The aforementioned algorithms considerably reduce both model size and computation complexity by using binarized weights and activations. Therefore, they enable acceleration of the forward pass of the neural network with a bit convolution kernel. However, the backward pass of binarized networks still requires convolutions between floating-point gradients and weights, and as a result, most of the training time of BNN and XNOR-Net is spent in the backward pass.

DoReFa-Net [61] was the first work to succeed in quantising gradients to 6-bit numbers during the backward pass, and still achieve a comparable prediction accuracy. In this approach, weights and activations of the CNN are quantised deterministically, while for the gradients a stochastic function is used. The authors showed that gradients, in general, require larger bit-widths than activations, and activations, in general, require larger bit-widths than weights, to lessen the degradation of prediction accuracy compared to 32-bit precision counterparts. They derived a DoReFa-Net from AlexNet with 1-bit weights, and 2-bit activations that can be trained from scratch using 6-bit gradients and obtained 46.1% top-1 accuracy on the ImageNet validation set.

### 3.1.2 Pruning

Pruning [50] is a pivotal technique employed to minimize the size and computational demands of neural networks by eradicating less vital parameters or connections. There are two main categories: structured [62] and unstructured pruning [63]. Unstructured pruning removes individual parameters, leading to a sparse network, while structured pruning eliminates entire neurons, filters, or channels, maintaining a dense but smaller network.

The seminal “Optimal Brain Damage” paper by LeCun et al [63] provides a foundational approach for pruning, aiming to reduce network size without significantly impacting performance. This early work laid the groundwork for modern pruning strategies, showing how pruning could lead to more computationally efficient networks. This work is an example of unstructured pruning, where weights are removed irrespective of the structure of the resulting network.
Adding a dimension to pruning is the Lottery Ticket Hypothesis [64], suggesting that smaller, sparser subnetworks within larger networks, dubbed “winning tickets”, when properly initialized, can attain comparable or superior performance to the original network. This hypothesis underscores the potential of pruning in discovering highly efficient and performant network architectures, thus forming a collection of strategies for optimizing neural networks. This is an example of structured pruning, where the pruning is aware of the topology of the network.

### 3.1.3 Tensor Decomposition

A tensor is a type of data structure commonly used in DNN computations as one of its main building blocks. Generally, a tensor is an array of numbers arranged on a regular grid with a variable number of axes [65]. So, in other words, it is a multidimensional array. Based on this definition of tensors, vectors are one-dimensional, and arrays are two-dimensional tensors. Therefore, if the designers compress tensors of a DNN by using tensor decomposition, it will result in a smaller model size, and also, it will also simplify the tensor operations. In tensor decomposition, the goal is to preserve the structure of the network while using a low-rank approximation to obtain a better-compressed model. Low-rank parameters and layers within a network are the ones that contain less significant information and, therefore, can be replaced by a low-rank approximation of them. For example, in computer vision, a low-rank weight matrix accounts for the smooth components of an image, while a sparse matrix represents fine details such as edges. There are two classes of tensor decomposition algorithms that have been the main focus of research over the last few years: low-rank matrix decomposition and tensorised decomposition.

### 3.1.4 Knowledge Distillation

Hinton et al. [66] was the first to bring the concept of knowledge distillation to the mainstream in DNNs, and the implementation of the distiller is based on this work; however, its idea was originally introduced in 2006. Knowledge distillation is extracting information from a large neural network model or an ensemble of models and transferring this knowledge to a new, simpler, and compact model such that it results in competitive or better accuracy. By doing so, the distilled model will also have a better runtime performance during inference due to its smaller model size and fewer parameters. The large model is also known as the teacher model, and the distilled model is called the student model. To gain a comparable or better accuracy in classification, a loss function is used to compare the labels produced by the teacher model with the predicted labels produced by the student model.
3.1.5 Network Architecture Search (NAS)

Although manually designed CNNs with tuned and fewer parameters and smaller model sizes have shown high accuracy in CNNs such as MobileNet [67], SqueezeNet [68] and ShuffleNet [69], due to the inevitable limitations that human knowledge has, methods that automatically explore a predefined search space for efficient models like Network Architecture Search (NAS) seem a more logical choice. In fact, NAS can be seen as a subfield of automated machine learning (AutoML). AutoML refers to the idea of automating the entire pipeline of machine learning (ML). The AutoML pipeline mainly consists of four processes: data preparation, feature engineering, model generation, and model evaluation [70]. In NAS —if seen as a subfield of AutoML— the focus is only on model generation and model estimation. What makes NAS different from the previously mentioned methods is that it automatically searches for an efficient DNN without depending on the base model.

The challenges in this field are generally an attempt to answer one of these questions so that it results in finding an optimal DNN model: 1) how to properly define the search space; 2) what search strategy should be taken; 3) what is the best performance evaluation methodology?

Answering the first question is important because if there is no restriction on the number of network elements or their connections, the search space grows exponentially [71]. A generic solution was to represent a neural architecture as a directed acyclic graph (DAG) consisting of ordered nodes and run the search algorithm on such a graph. The problem was, that having a large number of layers within a DNN and a great number of possible operations on each layer results in a very large representation. Consequently, search operations in this extremely large DAG become costly both in terms of time and computation. However, if the search space is restricted to only a finite number of small-cell structures, it can lead to a successful cell. By constructing the final network structure as a sequence of these repeated cells we can obtain an optimal model. This strategy is known as a cell-based architecture. Some examples of the works that have been done using this strategy are namely DARTs [72], NASNet [73], ENAS [74] and AutoDispNet [75]. On the other hand, there is a different methodology that relies on a pre-built successful network such as ResNets or MobileNets. These are developed using a supernet-based approach and this strategy has been the core idea of another group of works in this field [76–78].

3.2 Hardware Approach

There has been considerable work done in the hardware community to address the issue of running DNNs on low-resource devices. These are typically focused towards reducing
the memory access issues and computational complexity of DNNs. The memory access issue is addressed by designing a new memory architecture or by bringing the computation closer to memory. The computational complexity issue is addressed by designing a new hardware architecture for DNNs. Memory issues are because of DNNs hitting the Memory Wall issue. These challenges are especially important for edge computing, where the devices are constrained by the power budget, computational resources, and memory capacity.

The table 2.1 compares some low-power devices and data-centre GPUs. The limited power and computational resources of low-power and embedded devices, in stark contrast to traditional DNN servers, necessitate the design and development of specialized hardware architectures. These architectures must prioritize energy efficiency while retaining the flexibility required to meet the varied demands of DNN applications. This need is particularly pronounced in edge computing scenarios, where devices operate under stringent power constraints yet are expected to execute complex computational tasks with high reliability. Therefore, the focus shifts towards the innovation of DNN accelerators engineered to deliver optimal performance within the severe energy limitations characteristic of embedded systems. Such accelerators are imperative for facilitating the deployment of DNNs in environments where traditional computing resources are either inadequate or infeasible. Achieving a balance between the high energy efficiency provided by ASICs and FPGAs, and the broad programming accessibility and flexibility offered by GPUs, is a key objective. Current approaches aim to reconcile the demands for raw computational power and energy-sensitive, adaptable computing solutions, marking a pivotal development in the broader adoption and scalability of DNN technologies across diverse hardware platforms. In the later chapters, we show that the proposed design is able to handle multiple DNNs with different sizes and complexities, and is able to achieve high energy efficiency and performance. Thus, the proposed design is able to address the challenges of running DNNs on low-power devices. Additionally, the design can be incorporated into ASICs and FPGAs, making it suitable for edge computing scenarios.

Across the hardware community, there have been several efforts to design efficient hardware architectures for DNNs. In general, the efforts can be categorized into two main categories:

1. Designing new hardware architectures for DNNs, and
2. Processing in or near memory for DNNs.

In this section, I will go through the most prominent works in these two categories.
Table 3.2 Comparison of different hardware architectures for DNNs.

<table>
<thead>
<tr>
<th>Features</th>
<th>CPU/GPU</th>
<th>ASICs</th>
<th>FPGA</th>
<th>PNM</th>
</tr>
</thead>
<tbody>
<tr>
<td>Memory</td>
<td>Off-chip</td>
<td>On-chip</td>
<td>On-chip</td>
<td>Near-module</td>
</tr>
<tr>
<td>Programmability</td>
<td>High</td>
<td>Low</td>
<td>Medium</td>
<td>High</td>
</tr>
<tr>
<td>Energy Efficiency</td>
<td>Low</td>
<td>High</td>
<td>Medium</td>
<td>High</td>
</tr>
<tr>
<td>Area</td>
<td>Depends on device</td>
<td>High</td>
<td>Medium</td>
<td>Low</td>
</tr>
<tr>
<td>Cost</td>
<td>Low</td>
<td>High</td>
<td>Medium</td>
<td>Low</td>
</tr>
<tr>
<td>Speed</td>
<td>Low</td>
<td>High</td>
<td>Medium</td>
<td>High</td>
</tr>
</tbody>
</table>

3.2.1 Hardware Architectures for DNNs

In this section, I will go through the most prominent works in designing hardware architectures for DNNs. If we categorize hardware architectures into two main categories — temporal and spatial — CPUs and GPUs fall under the category of temporal hardware architecture, while many ASIC-based and FPGA-based modules are considered spatial architectures. An overview of ASIC-based architectures for DNNs is presented in [79]. A similar discussion on FPGAs is presented in [80]. Temporal architectures split the computation into multiple time steps and execute them sequentially, while spatial architectures split the computation into multiple spatial regions and execute them in parallel. In temporal architectures, the data is stored in a single memory and the computation is done in a separate unit. In spatial architectures, the data is stored in multiple memories and the computation is done in the same unit as the memory. GraphCore [81] and Cerebras [82] are examples of temporal architecture, while Google’s TPU [83] is an example of a spatial architecture. This thesis proposes a new temporal architecture for DNNs.

In general, the following design parameters are considered in designing hardware architectures for DNNs:

1. **Memory**: The memory capacity and bandwidth are important parameters in designing hardware architectures for DNNs.

2. **Programmability**: The programmability allows the hardware to be used for different applications.

3. **Energy Efficiency**: The energy efficiency allows the hardware to be used in low-power devices.

4. **Area**: Low area is critical in low-power devices, as the area directly affects energy efficiency.
5. **Cost**: The total cost of ownership affects usability in commercial applications.

6. **Computational speed**: The computational complexity of DNNs is high, and designing hardware architectures that can efficiently execute these computations is important. This is a side-effect of software and hardware architecture.

Accuracy is an important concern in designing hardware architectures for DNNs, but it is not considered in this thesis. It is a function of the software and hardware architecture, and it is not a hardware design parameter. Table 3.2 compares different hardware architectures for DNNs on the above parameters.

YodaNN [84] was the first ASIC accelerator designed and optimized for BCNNs. In this design, the authors exploit the BinaryConnect approach to store all the weights in binary format and reduce the complexity of inference to two-complement operations so they could save more area and energy. For greater energy optimization, they try to lower the supply voltage. They support seven different kernel sizes to add flexibility to their design, which introduces a 29% loss in energy efficiency, but still, the overall energy consumption of YodaNN is better than a BCNN without any optimization.

Tang et al. [85] proposed a crossbar accelerator based on resistive random access memory (ReRAM) for the forward process of BCNN. The overall structure of their design consists of a series of convolution layers cascaded by a series of FC and pooling layers. Each of these layers has its own input buffer and computing circuit. For FC and convolution layers, the convolver circuit is made up of the PRAM crossbar, while the pooling layer circuit is implemented as a multi-input “OR” gate. This work takes advantage of the high area density of ReRAM and all convolution kernels are mapped onto the crossbar. However, if the network scale increases, the convolution computing unit has to employ matrix splitting, which is not energy efficient. Another significant overhead of this design is the buffers and DAC/ADC, which contribute more than 85% of area and energy consumption.

Fan and Angizi [86] present a convolution-in-memory accelerator using a Spin-Orbit Torque Magnetic Random Access Memory (SOT-MRAM) device architecture. Their design consists of a sub-array of SOT-MRAM cells with modifications on the row decoders; it can have both memory read-write and multi-line AND/OR logic operations. In their design, any two cells in identical columns could be selected and sensed simultaneously, and then the equivalent resistance of this parallel connected SOT-MRAM is compared with a specific reference by a sense amplifier to realize bulk AND operation. Then, the data will be read out and sent to a following bit-counter module to finish the convolution computation. This architecture has a considerably lower overhead in comparison with the previous work [85]. Still, spintronic-based memories have several challenges that need to be addressed before
Literature Review

widespread commercial utilization. For instance, a relatively higher write access energy and latency in comparison with CMOS-based memory technology which can easily lead to read disturb and other degradation issues such as Time Dependent Dielectric Breakdown (TDDDB) [87].

The project from Ando et al. [88], which is the closest to my work, has implemented a near-memory accelerator for binarised and ternarised DNNs using an array of tightly coupled SRAM-logic modules. In their approach, all network parameters are stored in on-chip SRAMs, and computations are completed in logic units located nearby using the intra-word bit-parallel readout and inter-word-wise serial access of SRAM. For computations, they have developed a processing-in-memory module that can house a 3-layer binary neural network with a constraint on the depth of each layer. They have prototyped a 13-layer fully connected binary version of their architecture as well as built a neural network trainer and emulator software tuned to their customized memory (BRein memory). My design, on the other hand, will benefit from a more general near-DRAM implementation rather than a very tuned single on-chip memory that can only be used for the inference phase of training CNNs.

Jain and Gupta proposed TiM-DNN [89], a programmable in-memory accelerator that is specifically designed for ternary DNNs. TiM-DNN uses a new CMOS-based bit-cell called TPC (Ternary Processing Cell) that enables in-memory signed multiplication with ternary values. Using TCP, they have designed Tim Tiles, which are specialised memory arrays that execute signed ternary dot-product operations. Other than ternary multiplication, writing is the other operation that TPC can do. The advantage of this method is its high level of parallelism, as all tiles compute signed vector–matrix multiplications, which is the core operation of CNNs in parallel. However, the downside of this architecture is dramatically changing the fundamental architecture of a memory cell which may make exploiting this design less desirable for industry. Also, these new memory cells will not be able to operate read (only write and multiplication), so all the memory arrays made of these tiles can no longer operate as commodity memory when no computation is needed to be done.

3.2.2 Processing In or Near Memory

Bandwidth constraints and the energy required for communication motivated the idea of moving data-intensive computations near to memory (near data computation). This concept is not new and has been studied since the 1990’s [33–35]. However, these were not successful due to two reasons. First, it was practically difficult to integrate computation logic with DRAM due to incompatible fabrication processes. Second, the lack of data-centric killer applications prevented the industry from investing in these techniques.
Since then, many other techniques, such as prefetching, multithreading or having hierarchical memory, have tried to mitigate these inefficiencies to some extent, but traditional von Neumann compute architecture has an intrinsic limitation on data access due to the latency inherent in hierarchical memory and interconnect. Even in modern systems, the data transfer between CPUs and off-chip memory consumes two orders of magnitude more energy than a floating point operation [36].

On the other hand, constraints affecting near-data computation have been resolved to a reasonable extent in recent years. State-of-the-art memory stacking techniques [37, 90] addressed fabrication issues, and the introduction of the big data era caused a bloom in data-centric applications. As a result, offloading computation to memory or closer integration of logic and memory is being studied again under different names, such as near-memory processing, logic-in-memory, computing-in-memory, or processing-in-memory (PIM).

These efforts may be classified into two categories: moving logic closer to memory or near memory computing [40, 34, 38, 39, 41] and performing computation within memory structures, or in-memory computing [42–45]. Near-memory computing refers to bringing logic or processing units closer to memory. Notwithstanding the closer integration, processing units still remain distinct from memory arrays. In-memory computing integrates logic operations into memory arrays, fundamentally blurring the distinction between processing and memory [9].

3.3 Summary

In this chapter, I reviewed the different approaches used for accelerating convolutional neural networks in algorithmic and hardware domains. In the algorithmic domain, I reviewed the quantisation, pruning, tensor decomposition, knowledge distillation and network architecture search approaches. In the hardware domain, I reviewed the temporal and spatial hardware architectures for DNNs, and the processing in or near memory approaches. In the next chapter, I will present my proposed approach for accelerating CNNs using a co-design approach.
Chapter 4

A BCNN PNM Module

This chapter presents a BCNN PNM module, a processing-near-memory module that performs inference operations close to DDR4 DRAM memory. On receiving a command to start execution, it interacts with the memory controller to bring in data needed for inference and to write back results. In this chapter, I will first present an overview of the PNM design in Section 4.1, indicating where the possible location for such a module is and where it is located in this design, and how it will communicate with the main memory and processing units. In Section 4.2, I will explain the computations that are needed for the inference phase of a BCNN. Then in Section 4.3 I will present a detailed description of two architectural designs for this module, explaining the way data is mapped on DRAM and read into the PNM module, how the convolution operations take place, followed by a discussion in Section 4.4 on the effect that the number of module replicas, and their buffers and sizing have on DRAM’s access frequency and performance of the inference process. In Section 4.5, I will present an estimation of the hardware cost of the PNM module, and finally, in Section 4.6, I will introduce the instruction set architecture (ISA) for this module and how it can be used to model different layer types.

4.1 A Design Overview

There are four different positions within or near DRAM memory where such a computing module can be placed. Figure 4.1 shows these different potential placements, considering the architecture of a DDR4 DRAM memory. The highest level of abstraction that can be considered for placement of a PNM module is shown in Figure 4.1a where the PNM module directly interacts with the memory controller (MC) and has one or more ranks of the DRAM allocated to its data. It is important to note that in comparison with the other placements, this one has the highest latency as it cannot take advantage of the higher degree
A BCNN PNM Module

(a) MC level

(b) Device/Bank Group level

(c) Bank level

(d) Memory array level

Fig. 4.1 Possible positions of the PNM module
of parallelism that is available in lower levels of DRAM. However, it is the least complicated in terms of data placement, addressing policy and micromanaging the timings of each request and, consequently, the one with the highest level of flexibility and ease of integration with existing devices. Respectively, the next possible locations of the processing module would be between bank groups (Figure 4.1b), in each bank group between the banks (Figure 4.1c), and finally in each bank between the memory arrays (Figure 4.1d). In these scenarios, the design would no longer fall into the category of near-memory computing but would be a processing-in-memory design.

Among all these possible locations, Figure 4.1a with the highest level of abstraction was chosen for the placement of the module due to its flexibility and lack of disruption to existing DRAM devices. Depending on the neural network’s size, one or more ranks could be allocated for storing the network data.

For the PNM module to start inference operations, first, a command should be sent from the processor/memory-controller to the module, containing the start address of the data for the network, which is laid out as described in Section 4.3, and the parameters of the network, such as size of filters and input feature-maps. The module will then start processing by instructing the memory controller to fetch the first part of the required data.

### 4.2 BCNN Computations

Figure 4.2 shows an overview of the operations required for the inference phase in a BCNN. Having binary weights and activations in BCNNs allows us to replace the MAC calculation, which is the core of a convolution operation, with a bitwise XNOR of the weight and input feature-map activation vectors (“multiplying”), followed by a popcount for counting the number of set bits in the XNORed intermediate bit vector (“adding”). If the number of set bits in the vector dominates the number of zeros, the result of a MAC operation will be “1”, otherwise it will be “0”. Knowing the size of the vector, it is sufficient to compare the number of set bits to a constant value that is equal to the value of 1/2 the vector size to decide whether the number of set bits is dominant or not. However, in BinaryNet, the inference phase calculations also include shift-base batch normalization and applying the Relu activation function (sign function) to the result.

Earlier in Section 2.4.2, we discussed that all calculations needed for batch normalization during run-time can be skipped and replaced with only adding one constant bit [23], like having an extra bit for bias. Moreover, the comparator plays the same role of the Relu activation function as it results in “1” if the number of set bits is dominant (equivalent to having a positive real-valued partial sum) and “0” otherwise (declaring that the result of a
partial sum is a negative number and therefore should be represented by “0”). As a result, all calculations regarding applying a filter vector on one part of its corresponding feature-map vector can be performed in three consecutive cycles in the PNM module upon receiving a command from the memory controller. It is worth noticing that the popcount module has been fully pipelined.

The computations for calculating an activation node in one layer are completely independent of computations that have to be done to calculate other activation nodes in that layer. Figure 4.2 shows how the inference phase is broken into five simple stages. Here, it is assumed that the batch normalization constant has been pre-calculated and accumulated with the bias. Therefore, the bias bit is the result of accumulating both the neural network bias and the pre-calculated batch normalization constant value. In the final implementation of the PNM module, the intermediate result of the popcount for each channel of one filter will be accumulated with the popcount result of its other channels until all the channels of that specific filter have been applied on all their corresponding input feature-map channels.

Fig. 4.2 Near-memory computations of the BCNN inference phase.
4.3 PNM Module Design

In designing energy-efficient modules for either near-memory or in-memory computations, each building block’s area and energy consumption is very important. Going through different methods of implementing the convolution operation in DNN accelerators, there was a characteristic that most had in common: the need for rearranging the input feature-map and filter’s data structures by flattening their matrices to gain their equivalent Toeplitz matrix and, therefore, turning the matrix-vector operation into a matrix-matrix multiplication. This consequently results in a lot more data redundancy, too. As a result, by following this implementation of the convolution operation, extra hardware would be needed to rearrange the input feature-map and filter data before and after the convolution operations while also using a large proportion of the memory to store multiple replicas of data.

To this end, the design I introduce for my PNM module does not require any data rearrangement for either the input or output matrices. It is notable that typical convolution operations followed by linear layers need to rearrange the input feature-map and filter data structures by flattening their matrices to gain their equivalent Toeplitz matrix and, therefore, turn the matrix-vector operation into a matrix-matrix multiplication. We work around this, by remapping linear layer as a convolution operation, and therefore, we can avoid the need for data rearrangement. Based on this approach, the input feature-maps’ and filters’ matrices need to be read row by row into buffers within the PNM module and, therefore, should have been stored in the DRAM in the first place accordingly.

4.3.1 Terminologies and Definitions

In this section, I will summarise and define some terminologies that will be used throughout the rest of this chapter.

- **PNM Module**: The Processing Near Memory module that is designed to perform the convolution, maxpooling and fully-connected operations near memory.

- **Mode S**: The mode of operation in the PNM module where a set of filters are applied to all the input feature-map before moving on to the next set of filters.

- **Mode T**: The mode of operation in the PNM module where all filters are applied to an input channel before moving on to the next input channel.

- **Replica**: A single processing unit in the PNM module that is capable of performing the convolution operation on one input feature-map and its corresponding filters.
• **Filters per Replica:** The maximum number of filters that are applied in parallel to the input feature-map in one replica of the processing unit.

• **Layer:** A single neural network layer that consists of a single operation such as convolution, maxpooling or fully-connected.

### 4.3.2 Processing Data

For calculating the activations without rearranging data, a method which builds upon the “row-stationary” data-flow that was introduced in the Eyeriss DNN accelerator [91] for implementing the kernel of convolution, the MAC operation, was developed. Based on this data-flow, the rows of filter weights are kept stationary inside the register file (RF) of each processing engine (PE), and then the input feature-map activations are streamed into the PEs. The PE does the MAC operation for each sliding window at a time, which uses just one memory space for the accumulation of each partial sum. Since there are overlaps of input feature-map activations between different sliding windows, the input feature-map activations can then be kept in the RF and get reused. Going through all the sliding windows in the row completes the 1-D convolution and maximises the data reuse and local accumulation of data in that row. A big advantage that this method has for computing convolutions in memory is that it needs no rearrangement of the input feature-map, filter or output feature-map vectors either before or after the operations. As a result, filter and input feature-map vectors can be stored in the main memory and read into their corresponding buffers row by row, while each pair of filter and input feature-map is processed independently of the computations on other pairs of filter and input feature-map rows. Finally, the summation of these partial results will give us the final output of convolution.

To this end, the design I introduce for my PNM module does not require any data rearrangement for either the input or output matrices. Based on this approach, the matrices of input feature-maps and filters need to be read row-by-row into buffers within the PNM module and, therefore, should have been stored in the DRAM in the first place accordingly.

However, in my approach, I do not use several processing elements in parallel like Eyeriss to optimise the performance of running CNNs. Instead, in my scheme, the input feature-maps’ data is streamed row by row into the PNM module to perform the computations in a pipeline while avoiding repetitive reads of the same data from memory. Figure 4.3 shows an abstraction of how applying a sliding window of a 2-D (size $3 \times 3$) filter on a 2-D input feature-map (size $5 \times 5$) looks in their equivalent flattened matrices. Each row of the filter has been aligned with each row of the input feature-map. For sliding the filter on an input
feature-map, filter rows will stay stationary; however, the vector containing input feature-map bits will be shifted to the right by the number of bits in the stride per cycle.

Considering Figure 4.3 in more detail, in cycle $i$ the filter and input feature-map areas are aligned such that the filter is logically placed in the top-left corner of the input feature-map. This will create the partial sum for the top-left cell in the output feature-map. The rows of the filters are aligned with the start of the rows of the input feature-map. All data for the filter and all data for the same number of rows from the input feature-map have been loaded into the buffers. Notice that computation will only be performed on the first three elements of each input feature-map row initially (e.g. elements $a$, $b$ and $c$), but other elements from the same row have been loaded (i.e. $d$ and $e$). In addition, the data is shown backwards, so each row starts on the right, although this is just to aid visual presentation.

At the end of cycle $i$, ready for cycle $i+1$, the data for each row of the input feature-map is shifted right by the stride parameter (which, in this case, is one element). This logically lines the filter up with the top middle of the input feature-map. In cycle $i+1$ therefore, the filter can be applied to produce the next partial sum.

Cycle $i+2$ (not shown) is the same, but at the end of this cycle, the input feature-map needs to be shifted by more elements to line the filter up with the next row of the input feature-map. In this example, with stride 1, we need to shift by only three elements. Had the stride been 2, we would have needed to shift by $3 + 5 = 8$ elements to skip a row of the input feature-map. Processing can then continue as before in cycle $i+3$, as shown.
Data Layout in Memory

The choice of data layout influences how effectively it can be read from memory into the processing module, and therefore, it affects how efficiently operations can be performed. Earlier in 4.3.3, I discussed that one big advantage that my module has in comparison to commodity BCNN implementations is that there will be no need to write redundant data in the memory as instead of writing the Toeplitz matrix of input feature-maps and filters, I write every row of their matrix only once. The way data should be read into the PNM module is row by row, however the order each row of input feature-map or filter can be read may differ based on our layout.

Here, I have explored three different data layouts of how rows of input and filter can be stored and their effect on the PNM module operations. For the first mapping, the order in which the rows of input feature-maps and filters are mapped to addresses within DRAM is shown in Figure 4.4a. Here, $I_n$ is the data of an input feature-map unrolled matrix at a specific channel followed by the unrolled matrices of its corresponding filters. The filters placed between two consecutive inputs, which have been marked $F_n^1$ to $F_n^m$, are all the filters that need to be applied to $I_n$.

In the second data layout, as shown in Figure 4.4b, for each layer, first, we will have the data of all input feature-maps of that layer, followed by all the filters that the layer contains. It is important to notice that even though there is no longer a distinction between different channels of inputs and filters, each layer’s data is mapped separately.

In Figure 4.4c, the order of placement for input data and filters is shown. For a network, first, all the inputs of all layers will be written in the memory, and then all the filters’ data of those layers will be written subsequently.

Figure 4.5 shows an abstract overview of how data is transferred between my system design’s three main building blocks, namely DRAM, Memory Controller (MC) and PNM module.

4.3.3 Replicas in the PNM Module

For performing more work in parallel per layer, the PNM module can be designed to work with multiple replicas of the input feature-map and filter pairs, as shown in Figure 4.6. The mechanism of how the computations are split across the replicas depend on the mode of operation, which is discussed in the following sections. A single PNM module can have multiple replicas, each replica working with a different unit of work. The number of replicas is a design parameter, and the effect of having different numbers of replicas on the performance of the inference process will be discussed in Section 4.4. For parallel processing,
there are hence two concepts, the number of replicas and the number of filters per replica. This also has an added effect of reducing the time taken to complete, but the effect is not linear, as the number of replicas increases, the time taken to complete the inference process decreases, but the rate of decrease slows down as the number of replicas increases. This is due to contention for the memory controller and the DRAM itself.
PNM Architecture

Figure 4.7 shows an abstraction of my proposed PNM module when only one filter is applied to one input feature-map and how the convolution function is calculated in this module. In this architecture, as in Figure 4.3, you can see the rows of an input feature-map being streamed at each clock cycle into the input feature-map buffer. This buffer contains the rows of one channel of that input feature-map. For each input feature-map buffer, there is a filter buffer in the PNM module. With a filter size of \((S \times S)\), the convolution operations cannot take place unless \(S\) input feature-map and filter rows have been read into their corresponding buffers. After initialisation, when the computations are taking place, the PNM module will keep reading data from the DRAM into the buffers; if, at any point in time, there are fewer than \(S\) input feature-map and filter rows, the computations will be stalled until enough data is read into the buffers.
Before the computations start, the PNM module needs to be initialised with four important values based on the BCNN architecture parameters. These could be sent via special instructions direct from the CPU to the PNM module or by mapping the registers holding these values within the PNM into memory in the same manner as memory-mapped IO, allowing the CPU to simply store the values to the relevant addresses in memory as they are.

The first parameter is the value that will be sent later to the n-bit shifter unit, indicating how many bits should be shifted to the right and eventually erased from the input feature-map buffer when the filter window has covered the length of the input feature-map matrix. This value differs for each layer of convolution. For a convolution layer with a filter size of \((S \times S)\) and feature-map size of \((H \times H)\), this value will be \((H - S)\) and the n-bit shifter will count \((H - S)\) cycles and then shift the input feature-map buffer \(S\) bits to the right. If the stride is \(ST\), then the value of the n-bit shifter will be \((H - S)/ST\).

The second value is a constant number that is sent to the comparator and is the value to which the number of set bits gained from the popcount unit at each stage should be compared. Its result will indicate whether the final activation node is “1” or “0”. If the number of set bits is higher than this constant number in the early stages of the computation (i.e. before the convolution is complete for that input feature-map), then the final output activation node is a set bit, no matter what the result of the rest of the computation is. Therefore, the comparator could send a signal to the local controller unit to indicate the early end of computation for
that input feature-map and set the output activation node. This could save valuable cycles and improve performance.

The third and fourth values are the number of input and output channels for each convolution layer. Later, in Section 5.1, we will discuss the effect of having multiple replicas of {input feature-map, filter} pairs on the performance of the inference process. To make the best use of the hardware, the number of replicas should be set to a common factor of the value of input channels, although this is not a hard constraint, and the PNM module will work with an arbitrary number of replicas. Knowing the number of input channels is crucial for setting the number of times that filter buffers need to be renewed to compute one output activation node. The number of output channels, on the other hand, indicates the number of different filters we need to apply on the input feature-map.

At each clock cycle, rows of the filter will be XNORed with their corresponding rows of the input feature-map, and the intermediate result will be sent to the popcount module. At the same time, the input feature-map buffer will be shifted one bit to the right. The popcount module, which has been pipelined itself, will produce an integer value that shows the number of set bits in the output vector of the XNOR operation.

This integer value is stored in a register, as depicted in the figure labeled “sum reg”. This register is then connected to a comparator, which determines the output activation node’s state. If the number of set bits exceeds the constant value pre-stored in the comparator (equivalent to half the vector’s length), the output activation node is set to “1”. Conversely, if the number of set bits is less, the output is set to “0”. Figure 4.5 shows an abstract overview of how data is transferred between my system design’s three main building blocks, namely DRAM, Memory Controller (MC) and PNM module.

Two basic modes of operation, S and T are defined in the following sections, which differ in the way the input feature-maps and filters are processed and how the intermediate results are stored. It is important to note that the PNM module can be designed to work in either mode, and the choice of mode will depend on the specific requirements of the network being processed. Both modes are capable of processing the various layers discussed in Section 2.2. In the following sections, I will discuss the two modes of operation and the effect of having different numbers of replicas of the PNM module on the performance of the inference process.

4.3.4 Mode S: Processing to Minimize Space

Figure 4.8 shows the complete architecture of the initial PNM design. Taking the architecture shown already in Figure 4.7, the hardware for processing one filter on one input feature-map has been replicated so that three sets of processing can occur simultaneously. In this diagram,
4.3 PNM Module Design

only one filter is shown per input feature-map (hardware replica), but the design supports multiple filters being processed simultaneously (and therefore producing values for multiple output feature-maps), as shown in Figure 4.9. However, for now, we restrict our discussion initially to the case where there is only one filter per hardware replica.

Figure 4.8 shows the starting point for processing in both modes of operation of the PNM module. In mode S, processing proceeds through the space of input feature-maps and filters by calculating all values necessary for one output feature-map. Having the hardware of popcount, adder and comparator pipelined, this module will produce one output feature-map activation node in each cycle if the number of input channels equals the number of input feature-map and filter sets present in the PNM hardware. Earlier, in Figure 4.7, there was no adder module as we only had one input feature-map and one filter. However, in real examples of CNNs, we always have multiple 3-D filters that should be applied on multiple 3-D input feature-maps, where the third dimension is the depth of the convolution layer or, in other words, the number of input channels. If, for example, the number of input channels is 96 in one layer, it means we need to do the MAC operation for 96 pairs of input feature-maps and filters and add all their partial sums together just to produce one output activation node. Having a limited number of buffers, we need to keep the value of this partial sum in a register.
Fig. 4.9 Multiple replicas of the PNM processing hardware with several filters per replica.

and accumulate it with the result of the popcount coming to the adder in the next cycle to calculate the total number of set bits in the popcount vector.

As the computations for calculating any activation node in one layer of a CNN are completely independent of one another, by having more replicas of the PNM module, we can calculate more activation nodes in parallel. The impact this parallelism has on the overall performance varies drastically according to the number of replicas and the number of filters per replica that we have. As each replica can consist of multiple filters applied to its input feature-map; the larger the number of filters per replica, the more processing can be performed. However, a larger buffer space is also required to hold the partial results of the output feature-maps being calculated.

### 4.3.5 Mode T: Processing to Minimize Data Transfers

The downside of processing in mode S is that input feature-maps have to be read repeatedly, once for every output feature-map produced. However, fetching this data from DRAM is costly, so mode T attempts to reduce this by applying all filters for one input feature-map to it consecutively, then moving on to the next input feature-map. The result is that input feature-maps need only be read once from DRAM. The downside of this approach is that the
partial results of calculating the output feature-maps have to be stored in the PNM module whilst subsequent filters are applied.

In mode T, with $F$ filters being applied on each input feature-map simultaneously, processing starts in exactly the same way as with mode S. The difference is in how the space of input feature-maps and filters is traversed. Like the previous mode, the data of different channels of input are distributed among input feature-map buffers. So, each input feature-map buffer contains the data of one channel of the input, and the $F$ filter buffers assigned to that input buffer will be filled with the data of different filters that need to be applied on that specific channel. Hence, after each cycle, the intermediate results of each filter being applied on each input should be stored separately to be aggregated later with the result of the remaining channels of each filter being applied on the input. These results are saved in a matrix with the same sizing and structure as the output matrix of that layer. However, it is important to notice that only after all the existing filters have been applied on one channel of the input, these results would be valid. Consequently, you can imagine that compared to the previous design, it takes longer for one single output to be produced in this design. Figure 4.10 shows how the intermediate results of different channels of one filter will be processed and stored.

Fig. 4.10 A detailed view of how the intermediate results coming from different channels of one filter are processed and stored.
The delay between the start of processing and an output feature-map being complete will depend on the number of channels each layer of convolution has and, therefore, the number of input-filter replicas the design has. Because for one output to be computed, every filter of every channel should be applied to its corresponding input. On the other hand, after the computation delay for each output is passed, more output nodes will be ready at once. Again, the number of output nodes computed at the end of each channel computation depends on the number of replicas we have.

Figure 4.11 shows an abstraction of the design in mode T for only one input and four filter buffers being applied at a time. Here, instead of accumulating all the intermediate results in one partial summation register as we did in the previous design, we store a matrix
4.4 Estimating Expected Performance in Theory

Fig. 4.12 A comparison between the order of computations taking place in modes S and T.

for each input buffer. These matrices are the same size as the output feature-map matrix and, by the end of the convolution operation, will hold the final output result for that channel. Each element stores the accumulation result of the popcount operation in the previous cycle plus the current value of the popcount. As such, the space usage of this design is significantly higher than the previous.

In general, Figure 4.12 shows the difference between these two modes in the way their computations proceed through the network and, consequently, the order in which their input feature-maps and filter buffers get filled. In this figure, the input feature-map buffers are shown in blue, and the filter buffers are shown in green. If we consider that there are two filter buffers per replica in mode S, the order of computation is first to apply all channels of those two filters on their corresponding input feature-maps in all channels and then move to the next two filters. On the contrary, in mode T, we first apply those two filters on one channel of the input and then only move to do convolution operations on the next channel when these filters have completely covered the first channel of the input feature-map.

For more clarity on how different filters traverse the input feature-map in these two modes, Figure 4.13 shows a 3D demonstration of the above scenario but only for one filter. We can see that in mode S, the filter moves through the depth of the input feature-map first, while in mode T, the filter moves through the length of the input feature-map.

4.4 Estimating Expected Performance in Theory

This section aims to show theoretically how long I expect the inference operation run-time to be in both design modes based on the characteristics of the BCNN architecture and the PNM’s module design parameters. In other words, having the information on the sizing of inputs, outputs and filters and the number of channels, to calculate a model for computation
performance as well, as knowing the amount of time it takes to communicate with DRAM through the memory controller and read and write back data, how long the computation takes to complete. As such, this section is divided into two parts. In the first part, I will show how the computation time for each layer of the network can be calculated based on the network parameters and the configurations of the PNM module. In the second part, I will show how the communication time for each layer of the network can be calculated based on the network parameters and the configurations of the PNM module.

4.4.1 Computation Time Estimation

In this section, I will show how the computation time for each layer of the network can be calculated based on the network parameters and the configurations of the PNM module. We will first go through the calculations for the convolution layer and then for the maxpooling and fully-connected layer. In each of these cases, I will consider the difference that having a different mode of design can make on the computation time formula.

Convolution Layer

We start with mode T. We need to read input and filter data for every replica of the module. Assuming no DRAM contention and a fixed latency for reading data from DRAM, there will be a constant value for the number of cycles needed to read all rows of input and then filters (based on the number of filters each replica has) into the buffers of that one replica. If our module has more than one replica, the number of cycles needed to read data into these
4.4 Estimating Expected Performance in Theory

replicas to calculate their corresponding outputs would be the minimum number of input channels and the number of replicas multiplied by the size of input and filters’ data that is read into each replica. If we call this the initialisation time, it will be computed as follows, where $I_R$ is used for the number of cycles we need to read the inputs into each replica, $D_I$ is the total number of elements of one input feature-map (i.e. the number of rows times columns), $W$ the data width of each of those elements in bits, $R$ is the number of replicas in the PNM module, $C_I$ is the number of input channels that layer of convolution has and $B$ is the data bus width (in bits) between the PNM module and DRAM memory.

\[
I_R = \left\lceil \frac{D_I \times W \times \min(R, C_I)}{B} \right\rceil
\]  

(4.1)

Then, we calculate the number of cycles we need to read the filters, which is shown by $F_R$ in the formula below. $D_F$ is the total number of one filter’s elements and $W$ the bit width of each element, $F_N$ is the number of filters each replica has.

\[
F_R = \left\lceil \frac{D_F \times W \times F_N}{B} \right\rceil
\]  

(4.2)

The number of cycles needed to compute the output is calculated below where $O_{Comp}$ is the number of cycles to compute all elements of output, and $D_O$ is the total number of elements in one output feature-map.

\[
O_{Comp} = D_O
\]  

(4.3)

Therefore, if we calculate the number of cycles each replica takes to compute an output matrix for one given input, it will be as follows. In this equation, $C_O$ represents the number of output channels that the layer of convolution has.

\[
Perf_{rep-T} = I_R + \left\lceil \frac{C_O}{F_N} \right\rceil \times (F_R + O_{Comp})
\]  

(4.4)

As such, the performance of computing convolution for the whole module, including how many replicas it has ($R_N$) will be as follows:
We now consider mode S, where the performance of computing convolution for one replica is a simple sum of components we have already calculated.

\[ \text{Perf}_{\text{rep-S}} = I_R + F_R + O_{\text{Comp}} \tag{4.6} \]

As such, the total number of cycles needed to compute the convolution in mode S will be as follows:

\[ \text{Perf}_{\text{Conv-S}} = \left\lfloor \frac{C_I}{R_N} \right\rfloor \times \left\lfloor \frac{C_O}{F_N} \right\rfloor \times \text{Perf}_{\text{rep-S}} \tag{4.7} \]

Recalling from Figures 4.12 and 4.13, the way the computations proceed through the network in modes S and T is different. In mode T, we only read \( R_N \) number of input feature-maps into our module once, then we wait for \( \left\lfloor \frac{C_O}{F_N} \right\rfloor \) number of cycles (so all the necessary filters were read and applied on all those \( R_N \) input feature-maps) till we can proceed to read new input feature-maps into our module. However, in mode S, we read each input feature-map \( \left( \left\lfloor \frac{C_I}{R_N} \right\rfloor \times \left\lfloor \frac{C_O}{F_N} \right\rfloor \right) \) times as the filter moves through the depth of the input feature-map first.

**Maxpooling Layer**

In calculating maxpooling performance, we need to consider that there are no filters in this layer, so we do not need to account for the time taken to read their data into the PNM module. Therefore, the number of cycles it will take to do the maxpooling operation is equal to the number of cycles needed to read and load the inputs, summed with the number of cycles which the maxpooling operation would take to complete, all multiplied by the number of times this cycle will repeat, meaning the number of input channels divided by the number of replicas we have.

\[ \text{Perf}_{\text{Maxpool}} = \left\lfloor \frac{C_I}{R_N} \right\rfloor \times (O_{\text{Comp}} + I_R) \tag{4.8} \]
4.4 Estimating Expected Performance in Theory

**Fully Connected Layer**

In Fully Connected layers, there is no appliance of an array on a window of the input as they are not concerned with preserving spatial relationships and do not have the notion of local receptive fields. So, there will be an element-wise 1-D matrix multiplication between two arrays. The performance is expected to be the summation of cycles needed to read both arrays plus the number of cycles in which the computation takes place. As all these parameters are independent of the design aspects, this number will always be constant. If we consider the number of cycles needed to read the data of one array as $I_R$ and the number of cycles needed to read the data of the other array as $F_R$, the performance of the fully connected layer will be as follows:

$$\text{Perf}_{FC} = I_R + F_R + O_{Comp} \quad (4.9)$$

**4.4.2 Communication Time Estimation**

In our system, the PNM module communicates with the DRAM controller to read and write data to and from the DRAM. This is assisted by the memory controller, which is responsible for scheduling the memory requests and handling the memory conflicts. The communication overhead is the amount of time spent on communicating with the DRAM through the memory controller. The memory controller has a fixed queue size, and if the queue is full, the memory controller will stall the PNM module until there is space in the queue. This adds a natural delay to the communication overhead, which is the amount of time spent waiting for the queue to have space. Further, in this section, I will present the formula I am using for calculating the communication overhead and the reason behind this choice.

To go into the theory of the subject, we need first to understand the basics of the queueing models relevant to the discussion. The models that are discussed in this chapter are the D/D/c queue and the M/D/c queue. The ‘D’ in D/D/c stands for ‘deterministic’, which means that the time between arrivals is fixed and known in advance. The second ‘D’ in D/D/c means that the service time required by each query is fixed and known in advance. The ‘c’ refers to the number of servers in the system. The ‘M’ in M/D/c stands for ‘Markovian’, which means that the time between arrivals is random and follows a Poisson distribution.

It is notable that the M/D/c queue is a special case of the M/M/c queue, where the service times are also Markovian. The M/M/c queue is a queuing system with c servers, where arrivals are Markovian and service times are also Markovian. The M/M/1 queue is a special
case of the M/M/c queue, where the number of servers is 1. Similarly, D/D/1 is a special case of D/D/c, where the number of servers is 1.

**D/D/1 queue**

To introduce the concepts of queueing theory, we will start with the D/D/1 queue. In a D/D/1 queue, queries arrive at the system according to a fixed schedule, and each query requires a fixed amount of time to be served. The system’s goal is to minimize the time that queries spend waiting in the queue.

One important metric for evaluating the performance of a D/D/1 queue is the average waiting time. This is the amount of time that a query spends waiting in the queue before being served. The average waiting time depends on a number of factors, including the arrival rate of the query, the service time required by each query, and the number of servers in the system.

To optimize the performance of a D/D/1 queue, it is important to balance the arrival rate of queries with the capacity of the system. If the arrival rate is too high, queries will spend a long time waiting in the queue, which can lead to inefficiency. On the other hand, if the arrival rate is too low, the system may be underutilized, which can lead to wasted resources.

D/D/1 (DD1) queue is a model suitable for a simplistic analysis of the system. Based on the mean-field theory, the DD1 queue is a good approximation of the M/M/1 queue, which is a more realistic model of the real world. In DD1 the arrival rate of queries is fixed and is expected to be known in advance, and the service time required by each query is also fixed and known in advance. As the system is restricted to a fixed class of algorithms, the sequence of queries is predictable also since our architecture has a custom memory layout and DRAM architecture, the memory access patterns are predictable.

If, on the contrary, the arrival rate of queries is random, then the M/D/1 queue is a more suitable model for our analysis.

**M/D/1 queue**

In an M/D/1 queue, queries arrive at the system according to a Poisson process, and each query requires a fixed amount of time to be served. As the workloads start to differ in their memory access patterns, which is typical of a system running multiple applications, the arrival rate of queries is no longer fixed and known in advance, and the M/D/c queue is a more suitable model for such analysis. Even though this might seem to be a more realistic model, it is still a simplified model of the real world, as the service time required by each query is still fixed and known in advance.
The average wait time in a M/D/1 system is given by:

\[ w = \frac{\rho}{2\mu(1-\rho)} + \frac{1}{\mu} \]  

(4.10)

Here, \( w \) is the wait time of a single query in the system, \( \rho \) is the ratio of the arrival rate \( \lambda \) and the service rate \( \mu \). The waiting time is the time taken in the queue and the time taken to be served. The first term in the equation is the time taken in the queue, and the second term is the time taken to be served. It is to be noted that the time to communicate the request to the memory controller and the time to communicate the response from the memory controller is not included in the waiting time.

In the case of our module and its operational envelope, the arrival rate of queries and the service time required by each query is dependent on the parameters of the system, the sequence of queries is predictable, the memory access patterns are predictable, the system is restricted to a fixed class of algorithms, and the system is restricted to a fixed class of memory access patterns. Therefore, the M/D/1 queue is a suitable model for our analysis.

We can also use the D/D/1 queue for our analysis, as it is a special case of the M/D/1 queue, where the arrival rate of queries is fixed and known in advance. This is especially useful to analyse the case with one replica and filter.

**Queuing in the BCNN module**

In our system, the PNM module communicates with the DRAM controller to read and write data to and from the DRAM. This is assisted by the memory controller, which is responsible for scheduling the memory requests and handling the memory conflicts. The communication overhead is the amount of time spent communicating with the DRAM through the memory controller. The memory controller has a fixed queue size, and if the queue is full, the memory controller will stall the PNM module until there is space in the queue. This adds a natural delay to the communication overhead, which is the amount of time spent waiting for the queue to have space. In our system, there are multiple replicas within a single PNM module, and each replica can send read and write requests to the memory controller. This can lead to contention in the memory controller, as multiple replicas may try to access the memory controller at the same time. This contention can lead to delays in the communication overhead, as the memory controller may have to wait for the queue to have space before processing the requests. Additionally, due to the multiple requests and the fixed queue size, there is non determinism in the communication overhead, as the memory controller may have to wait for the queue to have space before processing the requests. The formula for
calculating the communication overhead is shown in Equation 4.10. We use the M/D/1 model to model the queue delay.

I predict the communication overhead using an estimation framework, which is trained on the communication overheads of the different models running in various configurations. To simplify the estimation framework, the training is done only on the communication overheads of the convolution layers, as the convolution layers are the most time-consuming layers in the network. Additionally, the statistics of a single layer are used, as making predictions on the statistics of the entire network is more difficult. We use a heuristic approach to estimate the parameters based on experimentally observed values.

The estimation framework could also be a random forest model. The input to the model is the number of replicas, the number of filters per replica, the input dimensions, and the number of bits per element. Additionally, we provide the number of read requests per replica, the number of write requests per replica, and the total number of computation cycles per replica. I also provide the total number of read requests, the total number of write requests, and the total number of computation cycles. The loss function uses the communication overhead as the target, and this is computed using the M/D/1 model. The output of the model is, hence, the parameters for the M/D/1 model, and the communication overhead is calculated using the M/D/1 model.

### 4.4.3 Communication and computation overlaps

It is notable that the communication and computation overheads are not independent of each other. In fact, there is a significant overlap between the communication and computation overheads. This is because the PNM module can start computing the convolution operations as soon as the data is read from the DRAM, and it does not have to wait for the data to be fully read from the DRAM. Additionally, the memory requests can be sent ahead of time before the PNM module needs the data. This allows for computing and communication to overlap, and the PNM module can compute while the memory requests are being sent to the memory controller. Figure 4.15 shows the overlap between communication and computation. It is notable that the overlap is not 100%, and there is still some time spent on communication and computation separately. There are hence three modes of overlaps,

1. No overlap: The communication and computation overheads are independent of each other.
2. Partial overlap: The communication and computation overheads overlap partially.
3. Full overlap: The communication and computation overheads overlap fully.
**Fig. 4.14 Timeline of a convolutional neural network inference.**
This creates a challenge in predicting the run-time, as the overlap is purely driven by the workload and the hardware parameters. The overlap is also dependent on the number of replicas and filters per replica, as the number of sources of reads increases while the time for total computation decreases. This increases the rate of requests sent and causes the memory controller to stall the PNM module more often. Hence, in a perfect circumstance, the computation should scale with the number of replicas and filters added. Still, the communication should not scale with the number of replicas and filters added. This is another form of the memory wall problem, where the computation is not limited by the computation power but by the memory bandwidth.

We model the overlap as a parameter that is dependent on the queuing delay and the number of replicas and filters per replica. In a perfect situation, the overlap would be 100%, and the communication overhead would be zero. In the worst case, the overlap would be 0%. In our estimation framework, the value of the parameter is set between zero and 0.5. In the case of a single reader, in our experimental setup, the reader has full access to the memory controller, and the overlap is close to 100%. In the case of multiple readers, the overlap is reduced, as the memory controller has to schedule the memory requests from multiple readers.
This is a classic problem in the field of computer science, where there are multiple consumers and a single producer. To simplify the model, I estimate the parameter based on the ratio of the computation time and communication request time without queueing. The higher the ratio, the higher the overlap, and the lower the ratio, the lower the overlap.

We model the parameter using a heuristic approach, where the parameter is estimated based on the ratio of the computation time and communication request time without queueing.

### 4.5 Hardware Cost Estimation

#### 4.5.1 Full-precision AlexNet

In previous sections, I discussed how having binarised and quantised values for network parameters can help simplify the complexity of its inference operations. Based on the simplified operations, I have suggested two different designs for running the inference computations near-memory. To understand how much energy efficiency is improved by having binarised and quantised parameters in comparison to the full-precision network, here I present a theoretical comparison between the energy consumption of the full-precision AlexNet and the binary AlexNet.

Let's consider we want to run the inference computations of a full-precision AlexNet on a near-memory module that is as similar to the proposed PNM module as its computations allow. This means that the order in which the data is streamed into the input feature-map buffer, the order in which data is written back to the main memory, and the order in applying a filter buffer on an input feature-map buffer are exactly the same. However, the inference computations are more complex and, therefore, so is the module's cost in energy and area.

Figure 4.16 shows an abstraction of how a full-precision PNM module would look if we had only one input feature-map and one filter buffer. The data of an input feature-map's rows can be seen streaming into the input feature-map buffer every clock cycle. Similar to the binary PNM module, this buffer contains the rows of one channel of that input feature map, and for each input feature-map buffer, there is a filter buffer. Convolution operations with a filter size of $(S \times S)$ cannot be performed before $S$ input feature-map rows and $S$ filter rows are read into their appropriate buffers. The first difference to be noticed here is that each index of the buffer array is now a 32-bit value. Therefore, the MAC operation of the inference will be performed by each of two parameters streamed into a $32 \times 32$ multiplier and a 32-bit adder. Every two indices of the input feature-map and filter are fed into the multipliers simultaneously, and so is their result to the first column of adders. However, to have the final result of the first batch of MAC operations between the vector of input
A BCNN PNM Module

feature-map and filter ready, the whole path of adders needs to be passed. The orchestration of these adders makes the path look exactly like a reversed full binary tree. As such, the depth of this tree of adders and the number of adder and multiplier modules depend on the size of the filter buffer and are calculated as below.

A full binary tree with $M$ leaves contains $(2M - 1)$ nodes. Here, the number of multipliers is equivalent to the size of the filter buffer, which is $S \times S$. On the other hand, multipliers shape the leaves of this binary tree, and, therefore, the number of adders would be $(S \times S) - 1$

Practically, as discussed earlier, the greatest size of filter we have seen for a CNN is in the first convolution layer of AlexNet with the size of $11 \times 11$; therefore, the number of these adders and multipliers will always be fixed to 121 multipliers and 120 adders.

To estimate the energy consumption of a full-precision equivalent of the binary PNM module, let’s look at Table 4.1. This table shows a comparison of the corresponding energy cost and area cost for different precision for 45nm technology. It is crucial to take into account both the necessary processing and communication costs.

The computational energy consumption for the example above, where the size of the filter buffer is 9, is calculated as follows:

$$9 \times (32b \text{ FP Mult}) + 8 \times (32b \text{ FP Add}) + \text{BN(Inference)} = 96.3pJ$$

where

$$\text{BN(Inference)} = 32b \text{ Add}$$

Similarly, the area cost for the example above is calculated as follows:

$$9 \times (32b \text{ FP Mult}) + 8 \times (32b \text{ FP dd}) + \text{BN(Inference)} = 11 \times 10^4 \mu m^2$$

To formulate what we have just calculated above, the formula for calculating the computational hardware cost of a full-precision AlexNet with the same architecture as the BCNN, without taking into account its communication overhead, is as follows:

$$\text{HW Cost} = F \times (32b \text{ FP Mult}) + (F - 1) \times (32b \text{ FP Add}) + (32b \text{ FP Add})$$

Here, $F$ is the total number of elements in the filter window. In the next chapter, I will discuss how the communication overhead is calculated and how it is added to the hardware cost of the module. Moreover, an architecture-level estimation of the area and energy consumption of my design in both modes is presented in the next chapter.
Fig. 4.16 The abstract architecture of a full-precision PNM module considering one input feature-map and one filter.
Table 4.1 Comparison of the corresponding energy cost and relative area cost for different precision for 45nm technology [50].

<table>
<thead>
<tr>
<th>Operation</th>
<th>Energy (pJ)</th>
<th>Area (µm²)</th>
</tr>
</thead>
<tbody>
<tr>
<td>8b Add</td>
<td>0.03</td>
<td>36</td>
</tr>
<tr>
<td>16b Add</td>
<td>0.05</td>
<td>67</td>
</tr>
<tr>
<td>32b Add</td>
<td>0.1</td>
<td>137</td>
</tr>
<tr>
<td>16b FP Add</td>
<td>0.4</td>
<td>1360</td>
</tr>
<tr>
<td>32b FP Add</td>
<td>0.9</td>
<td>4184</td>
</tr>
<tr>
<td>8b Mult</td>
<td>0.2</td>
<td>282</td>
</tr>
<tr>
<td>32b Mult</td>
<td>3.1</td>
<td>3495</td>
</tr>
<tr>
<td>16b FP Mult</td>
<td>1.1</td>
<td>1640</td>
</tr>
<tr>
<td>32b FP Mult</td>
<td>3.7</td>
<td>7700</td>
</tr>
<tr>
<td>32b SRAM Read (8kb)</td>
<td>5.0</td>
<td>N/A</td>
</tr>
<tr>
<td>32b DRAM Read</td>
<td>640</td>
<td>N/A</td>
</tr>
</tbody>
</table>

4.6 Instruction Set Architecture

In this section, we describe the logical instruction set architecture (ISA) of the proposed accelerator. This is not an implemented ISA but a logical ISA to explain the functionality of the accelerator. This allows us to focus on the functionality of the accelerator whilst abstracting away the implementation details. It is to be noted that this ISA is independent of the mode of operation and is supported by both mode S and mode T. For clarity, a tensor is a mathematical object that is stored as an n-dimensional dense array. A vector is a single-dimensional tensor. In our setup, the following instructions are supported:

1. Reshape/flatten (rs): The reshape operation changes the tensor’s dimension without changing the data’s total size.

2. Reshape with padding (rsp): The rsp operation reshapes the memory but also adds padding to the data as needed.

3. Convolutional multiplication and accumulation (cmac): The cmac operation multiplies one matrix with another matrix and accumulates the result. Note that this is designed to directly map to the convolution operator. The cmac is followed by one of the following operations:

   (a) Compare and store (cas): This is an instruction that compares the result of an operation with an existing value in the memory and stores the result if the result is greater than the existing value. This operation is used to implement the maxpooling operation.
4.6 Instruction Set Architecture

(b) Store (st): This is an instruction that stores the result of an operation in memory.

(c) Accumulation (acc): This is an instruction that accumulates the result of an operation in memory.

4.6.1 Instructions in the ISA

Here, I discuss the instructions that are part of this ISA, and their operation and input requirements.

Reshape/Flatten

For the rs operation, the input is the start and end address of the data, its shape and expected shape. One important aspect of this operation is the layout of the data in memory. Data is stored with priority to the first dimension. So, data of size $4 \times 4 \times 3$ is stored as four elements arranged in groups of four. These sixteen elements are stored in groups of three. Based on the output size requirements, the reshaping happens. For example, if the required dimensions are $4 \times 12$, flattening along the last dimension, then logically there is no change as the layout in memory remains the same. It is to be noted that a transpose operation is not supported.

Reshape with padding

This instruction takes as input a tensor, its dimension, expected dimension and padding size. Reshaping with padding is a necessary operation for convolution. It will be seen later in this discussion that adapting the BCNN-PNM module to various neural network layers is a crucial requirement.

Convolutional multiplication with accumulation

This operation takes as input an input tensor and a collection of filters that need to run on it. Additionally, the stride of operation is also needed. The BCNN-PNM module is specialised to perform this operation. In this operation, a convolution operation is split into a series of vector multiplications and additions. There are variations to this operation, which allows architectural flexibility. These are compare and store, where the output of a single multiplication is compared to the results stored from the last computation. The higher of the values are stored. In store mode the value is directly written back to memory without storing it in a register. In accumulation mode, the results are stored in a register and accumulated. This is the default mode needed for the convolutional operator.
4.6.2 Architectural implementation using the ISA

The operations described above can be used to implement various neural network layers. The following table explains how.

<table>
<thead>
<tr>
<th>Layer</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Convolution</td>
<td>cmac + acc</td>
</tr>
<tr>
<td>Maxpooling</td>
<td>rsp + (cmac + cas) + rs</td>
</tr>
<tr>
<td>Fully connected</td>
<td>rsp + (cmac + acc) + rs</td>
</tr>
<tr>
<td>Average pooling</td>
<td>rsp + (cmac + acc) + rs</td>
</tr>
<tr>
<td>Residual layer</td>
<td>rsp + (cmac + acc) + rs</td>
</tr>
<tr>
<td>Depth wise convolution</td>
<td>rsp + (cmac + acc) + rs</td>
</tr>
</tbody>
</table>

In a reshape operation, the data placed in the memory is reshaped either logically or can be reshaped in place if a logical reshaping is more inefficient than the cost of the next operations. For the sake of simplicity, the implementation details are not visited here. In this thesis, we only perform logical reshaping.

Linear layer

In a linear layer operation, each row of the input matrix is multiplied by the weight matrix and the result is accumulated in the output matrix. To map this to a convolution operation, the last dimension of the input matrix is 1, and the first dimension of the weight matrix is 1. This is achieved by flattening the weight and the input matrix and then performing the convolution operation using the cmac operation.

Convolution layer

The convolution operation is a cmac operation followed by an accumulation operation. For convolution layers that are variants of the convolution operator, we will use a reshape-cmac-reshape (RCR) trick. There are two variants of this: multiplication only and addition only. We will discuss them as part of the implementation of the depth-wise convolution and the residual layer.

Depth-wise convolution (no addition, only multiplication)

In a depth-wise convolution layer, as seen in architectures such as MobileNet, the accumulation operation at the end of the cmac operation is not performed across channels. For
example, if there are three input channels and the input tensor is of size $5 \times 5 \times 3$, With a filter of size $2 \times 2$, the convolution operator will return a tensor of size $4 \times 4$. In the depth-wise convolution layer, the intermediate value is of size $4 \times 4 \times 3$, but the final operation, which sums the intermediate values along the last axis, is not performed. As a cmac operation includes the accumulation operation, depth-wise convolution cannot be performed directly. To work around this issue, the input needs to be resized so that the input is of shape $a \times (b + p)c \times 1$ instead of $a \times b \times c$, where $a$, $b$, and $c$ jointly constitute the size of the input tensor and $p$ is the padding required to perform the operation. We can clearly see that the input having a single channel in the output makes the accumulation operation redundant. The output of the convolution operation also has a single channel, as the number of filters in a depth-wise convolution is 1. The output is then resized and pruned to obtain the desired output shape. This RCR trick makes a matrix multiplication work on the BCNN-PNM architecture.

**Residual layer (no multiplication, only addition)**

Another architecture faces a different issue due to the composite operation of the cmac. In residual networks [92], the outputs of two different previous layers are added together in a residual operation. This operation has the opposite requirement as compared to the depth-wise convolution. The residual layer does not need a multiplication operation but needs an addition operation. This can be achieved by reading the two outputs as two channels of the input and using a convolution filter of size 1, with values [1] in the filter. This will add the values across the channels, and the output will be a single channel. The output is then resized to get the desired output shape. This RCR trick mimics an addition operation.

Given these two workarounds, we can see that the functionalities of the architecture are greatly increased. Addition operations and multiplication-only operations allow us to use much more complex operations.

### 4.6.3 Complex DAGs within the ISA

The module, in its current version, only allows for sequential operations. More complex operations, which can be modelled as complex Directed-Acyclic Graphs (DAGs), are beyond the scope of this thesis as here we focus on a CNN accelerator. Sequentiality reduces the complexity of the architecture, as data lifetimes are automatically managed. The output of one operation is the input of the next operation. Additionally, scheduling a single layer at a time on the entire accelerator removes the need for layer scheduling and sequencing. This greatly reduces the design complexity, which is a necessity in edge devices. In theory, complex DAGs are possible, as we can break them sequentially by allowing layers to read
from the output of any layer that has completed execution. The residual layer discussion above is one such example. We have introduced the feasibility of complex DAGs in the architecture.

A complex DAG support, for example, will open up support for the attention operation [10] in a transformer. It is notable that the attention operation is a collection of matrix multiplication operations. This is possible in the current architecture using the RCR trick shown above. The challenges of maintaining the DAG and the memory management overheads show examples of future extensions to this work. However, due to the complexity of the design, we have not implemented it in this thesis.

4.7 Summary

In this chapter, I have discussed the design of the BCNN-PNM module and the various design choices that were made. A discussion on the trade-off between compute and memory was presented, and the design choices that led to mode S and mode T designs were made. Further, a theoretical discussion on the performance of the module was presented, and the expected performance of the module was calculated. A discussion on the hardware cost of such a module was presented. I have also discussed the ISA of the module and how it can be used to implement various neural network layers.
Chapter 5

Evaluation

In this chapter, I perform a comprehensive exploration of how my module impacts the performance of running low-precision CNNs with regard to their various network parameterisations, datasets, the choice of module design (mode S or T) and the configuration of these designs in terms of the number of replicas and filters. The experiments primarily are focused on the AlexNet architecture, and also provide brief insights from two other well-known CNN architectures: VGGNet and MobileNet. These networks have been chosen for their diverse structures and effectiveness in the field of computer vision and also audio classification [93], which means they are still widely used in many applications and academic experiments. Our objective is to investigate how the combination of different architectural settings with varying data bit-widths, ranging from 1-bit to 32-bit, influences the performance and area and energy overheads. Throughout this chapter, I test the relation between the number of filters and hardware replicas, showing their collective impact on performance measured in terms of inference time. By studying this interplay, I aim to provide valuable insights into the optimization of low-precision CNNs under varying configurations and facilitate better-informed choices in designing and deploying them for different applications.

To this end, I will start by exploring binary AlexNet on the ImageNet dataset and showing the effect of having different design modes for running inference while changing the number of filters and replicas to show their effects. Later I increase the bit-width of network parameters and consequently change the logical operations of the binary inference module to see how the aforementioned criteria affect the performance of inference. Then, we will see the performance of running binary AlexNet in both modes, S and T, on other datasets. After that, I will show the performance of running binary AlexNet, VGGNet and MobileNet in both modes S and T on the PNM module. Ultimately, I will present the area and energy consumption of the PNM module for different network precisions and offer an analysis of the trade-offs between these factors in different modes and configurations of the PNM module.
Table 5.1 AlexNet Body Architecture.

<table>
<thead>
<tr>
<th>Layer</th>
<th>Filter</th>
<th>Input</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>Conv1</td>
<td>$11 \times 11$</td>
<td>$224 \times 224 \times 3$</td>
<td>$55 \times 55 \times 96$</td>
</tr>
<tr>
<td>MaxPooling1</td>
<td>$3 \times 3$</td>
<td>$55 \times 55 \times 96$</td>
<td>$27 \times 27 \times 96$</td>
</tr>
<tr>
<td>Conv2</td>
<td>$5 \times 5$</td>
<td>$27 \times 27 \times 96$</td>
<td>$27 \times 27 \times 256$</td>
</tr>
<tr>
<td>MaxPooling2</td>
<td>$3 \times 3$</td>
<td>$27 \times 27 \times 256$</td>
<td>$13 \times 13 \times 256$</td>
</tr>
<tr>
<td>Conv3</td>
<td>$3 \times 3$</td>
<td>$13 \times 13 \times 256$</td>
<td>$13 \times 13 \times 384$</td>
</tr>
<tr>
<td>Conv4</td>
<td>$3 \times 3$</td>
<td>$13 \times 13 \times 384$</td>
<td>$13 \times 13 \times 384$</td>
</tr>
<tr>
<td>Conv5</td>
<td>$3 \times 3$</td>
<td>$13 \times 13 \times 384$</td>
<td>$13 \times 13 \times 256$</td>
</tr>
<tr>
<td>MaxPooling3</td>
<td>$3 \times 3$</td>
<td>$13 \times 13 \times 256$</td>
<td>$6 \times 6 \times 256$</td>
</tr>
<tr>
<td>FC1</td>
<td>N/A</td>
<td>9216</td>
<td>4096</td>
</tr>
<tr>
<td>FC2</td>
<td>N/A</td>
<td>4096</td>
<td>4096</td>
</tr>
<tr>
<td>FC3</td>
<td>N/A</td>
<td>4096</td>
<td>1000</td>
</tr>
<tr>
<td>SoftMax</td>
<td>N/A</td>
<td>1000</td>
<td>1000</td>
</tr>
</tbody>
</table>

It is worth noticing that here I am only evaluating the performance of the inference operation and not the training process on a near-memory module. The majority of the experiments are run, considering the module is provided with an image from the ImageNet dataset as the baseline. However, an audio dataset and MNIST and CIFAR image recognition datasets are also tested. Therefore, the architecture of the simulated network will be based on the architecture of AlexNet for classifying ImageNet samples by default unless stated otherwise. This simulation is deterministic, and its goal is not to evaluate the accuracy of the network, but the performance of running the inference operation of the whole network and investigating the simulated modules’ area and energy.

Table 5.1 presents the architecture of the AlexNet network employed in my experimental analysis. Throughout this chapter, references to ‘inference time’ specifically denote the duration required for the PNM module to execute one inference process across the entire network.

### 5.1 PNM Module Evaluation Setup

I evaluate my design in commodity DRAM using Ramulator [94] with DDR4 DRAM technology and the architecture implementation as discussed earlier in Section 4.3.

Exploring the architectures of various CNNs that have been introduced at the time of writing this thesis reveals that the dimensions of the filter matrices in the convolutional layers have consistently been one of the following: $\{2, 3, 5, 7, 11\}$. The filter dimension of $11 \times 11$ from the first convolutional layer of AlexNet is the maximum size a filter has in current
CNNs; therefore, for each replica in the PNM module, the maximum size of the filter buffer is set to 128 bits. By doing so, not only can it contain all the 11 rows of the largest possible filter, but it can also facilitate running fully connected layers by buffering all bits coming in each clock cycle. Considering having a 64-bit wide data bus, 128 bits is the amount of data that the PNM module can receive in one clock cycle. When implementing this module in RTL, there would be one decoder for input buffer maps and one for filter buffers so that in each cycle the data would be sent to the corresponding buffers.

With our design choice for the sizing of filter buffers now determined, we will proceed to examine the crucial aspects of the initialisation phase. First, it is important to ensure sufficient data has been read to commence computation. The data required for initialising the input feature-map buffers for each replica is calculated as follows:

\[ \text{Filter dimension} \times \text{Input dimension} \]

For the filters, however, we need to read all bits of the filter matrices that are going to be applied on their corresponding input feature-maps. Each replica operates independently, so once the input feature-map and corresponding filters have been read into the PNM module, processing on that replica can begin immediately.

If we are computing in mode S, each popped-out bit of the input feature-map will be dropped and ignored as we will not need its data again until a whole round of operations to partially compute the final output activation nodes has been performed. However, in mode T, these popped-out bits will be enqueued back to another register buffer so that we don’t lose the input feature-map data after the convolution operation of its current corresponding filters is finished. The reason for this is that we need to keep the input feature-map data stationary in mode T of the module until all the filters have been applied on it. After the last computation is done, and we are going to fill the input feature-map buffers with new inputs’ data, we need to shift right all the filter buffers by a number equal to the filter’s row size so that the least significant bits contain the beginning bits of the new filters’ data.

Table 5.2 shows the DRAM configurations used for the simulation of the PNM module. The DRAM clock frequency is set to 1,200 MHz, which is the maximum frequency that the DRAM in our simulation operates at. The DRAM bus width is set to 64 bits. The PNM module is clocked at 1,200 MHz; this is done to ensure that the PNM module can process data as soon as it is available from DRAM. The supply voltage is 1.2V using 30nm technology. Additionally, the module is designed to function with standard DRAM, and for that, it relies on an external memory controller to interface with DRAM. Given the predominance of read operations overwrites in our simulations, prioritising reads over writes is unlikely to have
Table 5.2 The DRAM parameters used for the simulation of the PNM module.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>DRAM technology</td>
<td>DDR4</td>
</tr>
<tr>
<td>DRAM size</td>
<td>8 GB</td>
</tr>
<tr>
<td>DRAM clock frequency</td>
<td>1200 MHz</td>
</tr>
<tr>
<td>DRAM bus width</td>
<td>64 bits</td>
</tr>
<tr>
<td>DRAM burst length</td>
<td>8</td>
</tr>
<tr>
<td>DRAM row buffer size</td>
<td>2 KB</td>
</tr>
<tr>
<td>DRAM page size</td>
<td>1 KB</td>
</tr>
<tr>
<td>DRAM bank count</td>
<td>8</td>
</tr>
<tr>
<td>DRAM rank count</td>
<td>1</td>
</tr>
<tr>
<td>DRAM column count</td>
<td>1024</td>
</tr>
<tr>
<td>DRAM row count</td>
<td>32768</td>
</tr>
<tr>
<td>DRAM refresh interval</td>
<td>64 ms</td>
</tr>
</tbody>
</table>

a significant impact. As this module is designed to be predominantly used in streaming applications, (image and audio processing), having an open-row policy is favourable.

5.2 Performance of the Inference Operation

In this section, we will first explore the performance of binary AlexNet in both modes S and T. We will start by showing the effect of having different numbers of filters and replicas on the performance of the network in both modes. Then, I will increase the bit-width of the network parameters and, consequently, the hardware of the inference module and see how the aforementioned criteria affect the performance of inference.

5.2.1 Increasing the Number of Filters per Replica

In this first experiment, we will show the effect of increasing the number of filters per replica on the performance of the network in both modes S and T. I will keep the number of replicas fixed to show the isolated effect of increasing the number of filters. In Figure 5.1, the number of replicas is fixed at 32, where the difference between the two modes can be seen more clearly. As you can see in this figure, the performance of the network in mode T is significantly better than in mode S as we increase the number of filters from 1 to 16; then, as we change the number of filters from 16 to 32, mode T still shows a better performance however the difference between the performance of these two modes is not as significant as before. This is because in mode S, as we keep proceeding through the space of input
5.2 Performance of the Inference Operation

Fig. 5.1 The inverse relationship between filter count and inference time in both modes S and T when operating with 32 replicas, indicating that an increase in filters leads to faster processing.

feature-maps and filters and refilling the input feature-map buffers with the data of the next channels of input, we have to reread that data over and over again from DRAM, which is an expensive operation. However, in mode T, we keep the data in the input buffers and reuse it immediately for the next set of filters. As a result, the performance of mode T is better than mode S.

As we increase the number of filters, the amount of parallel computation increases, reducing overall inference time. In mode S, we also have to read less data from DRAM since we are now applying more filters on the input feature-maps at a time; therefore, we do not read the input feature-maps as many times. This accounts for the faster increases in performance for mode S compared to mode T. However, the performance of mode T is still better than mode S due to the reusing of data in the input buffers.

In Figure 5.2, the number of replicas is fixed at one, showing that the performance of the two modes is almost identical. The underlying cause for this observation is that when we have one replica and one filter, the computational demands outweigh communication requirements. So, even though in mode T we avoid repetitive readings of the same input feature-map, this does not result in a substantial overhead. Consequently, the performance in mode T remains largely similar to that in mode S.
Next, we will proceed to examine the impact of altering the number of replicas. As we increase the number of replicas, while keeping the number of filters fixed to one, the difference between the performance of these two modes becomes more significant. In Figure 5.3, where only the number of replicas changes across the experiments, we can see that it is only after having 16 replicas that using mode T rather than mode S would be justified. If we use any fewer numbers, the performance of both modes will be similar, so dedicating additional hardware space for the inference module to operate in mode T is not justifiable. Also, the reason for the divergence in performance in this figure is that mode S reaches bus saturation to DRAM sooner than in mode T, due to the continuous and repetitive reading of input feature-maps. This bus saturation at approximately 32 replicas becomes the limiting factor for the module’s overall performance in mode S.

![Change in Inference Runtime as the Number of Filters per Replica Increases for Mode T and Mode S](image)

**Fig. 5.2** No difference seen in inference time between modes S and T for when we increase the number of replicas while keeping the number of filters per replica to one.

Having examined the impact of the number of replicas and filters individually, the next two figures aggregate all observations for various combinations of filters and replicas in modes S and T into two comprehensive diagrams. Figure 5.4 and Figure 5.5 present these two cases. You can see that in both modes, having more replicas results in faster run-time.

For the upcoming experiment, I have chosen to use 32 replicas, as this number represents a point where the performance difference between modes T and S becomes significantly evident in the data.
5.2 Performance of the Inference Operation

Fig. 5.3 With the number of filters per replica fixed at one, the figure illustrates that beyond 16 replicas, mode T consistently exhibits a decrease in inference time compared to mode S.

Fig. 5.4 Demonstrating the variation in inference time for mode S as the number of replicas increases, across different filter counts per replica.
However, these were the cases when we had binary network parameters. In the next section, we will see how increasing the bit-width of the network parameters will affect the performance of the network in both modes.

### 5.2.2 Altering Bit-width

In this section, I show how having smaller bit-width for the network parameters impacts the performance of the network in both modes, S and T, in terms of inference run-time, positively. Figures 5.7 and 5.6 illustrate that as we increase the bit-width, it takes longer for the inference phase to be done. This occurs because having data encoded in smaller bit-width results in less information being read from DRAM for computing a single output, and the computations become less complex due to the feasibility of operation simplifications with binary or lower bit-width parameters. Consequently, the performance of both modes is negatively influenced by an increase in the bit-width. Furthermore, when comparing the two modes, it becomes evident that the higher the bit-width of network parameters, the earlier the benefits of using mode T manifest in the experiments. Hence, while using fewer than 16 replicas might not justify the choice of mode T for binary networks, this changes with network parameters of 4-bit, 8-bit, and lower bit-widths.
Another point to notice here is that while in mode S of binary AlexNet, increasing the number of replicas after 32 is still beneficial; for non-binary networks, this is not necessarily the case. A 4-bit network still benefits from having more than 32 replicas but the difference in performance is not as marginal as it was for binary networks. For 8-bit networks, the performance of the network is almost the same for 32 and 64 replicas. This is because the amount of data that needs to be read from DRAM in order for the operations to have enough data to start gets larger as we increase the bit-width. Therefore, the performance of the network does not improve as much as it did when we increase the number of replicas. And then we have 32-bit networks that show a worse performance after 32 replicas because the number of replicas is not the bottleneck any more (instead, it is the amount of data that needs to be read from DRAM), and by increasing the number of replicas, we will get bus saturation that will slow down the performance of the network.

Figures 5.8 and 5.9 show the same information but with 32 filters per replica. As we increase the bit-width, the time taken to perform inference increases. In mode S, after having 16 replicas, the performance stays the same, and in mode T this happens after having 8 replicas.

![Change in Runtime as the Bit Size Increases for Mode T](image)

Fig. 5.6 The impact of changing bit-width on the inference time in mode T, specifically when operating with a single filter per replica, showcasing how bit-width adjustments influence processing efficiency.
Fig. 5.7 the impact of changing bit-widths on the inference time in mode S, specifically when operating with a single filter per replica, showcasing how bit-width adjustments influence processing efficiency.

5.3 Comparison

In this section, I present a series of experiments that compare the performance of the PNM module in both modes S and T for different network architectures, datasets, and data layouts while examining the impact of various design factors such as the number of replicas, filters, and bit-width. I also show the area and energy consumption of the PNM module for different configurations and modes. Some of the architectures that are tested here are AlexNet, VGGNet and MobileNet on datasets of ImageNet, MNIST, CIFAR-10 and Audio. The focus of most of these experiments is on binarised networks, where both weights and activations are binary. However, I also present some results for networks with higher bit-widths to offer a broader perspective.

5.3.1 Different Datasets

In these set of experiments, I compare the inference time of running a binary AlexNet in both modes S and T on different datasets. The datasets I used are the ImageNet, Fashion-MNIST, CIFAR-10 and Audio datasets. The architecture of the network alters according to the dataset that is being used. For example, for the Audio dataset, the network is a 1D CNN, and for the
5.3 Comparison

The results are shown in Figure 5.10. These are the performance of running the inference operation on a PNM module in modes S and T with four replicas and four filters per replica. The reason for choosing these numbers is the smaller size of networks running on MNIST and CIFAR10 datasets. As can be seen, the run time of the network is better for the simpler datasets, and it is longer for the more complicated datasets. This is because the more complicated the dataset is, the more complex the network needs to be to be able to classify the samples in the dataset. Therefore, the more complex the network is, the more time it takes to run the inference operation on the network.

5.3.2 Different CNN Architectures

In this set of experiments, I compare the performance of running three different BCNN architectures (AlexNet, VGGNet16 and MobileNet.v1) using the ImageNet dataset in both modes S and T on the PNM module. The results are shown in Figure 5.11. The number of
Fig. 5.9 The impact of adjusting bit-widths on inference time within mode S, under the condition of utilizing 32 filters per replica, highlighting the relationship between bit-width variations and computational performance.

Fig. 5.10 The performance of running a binary AlexNet in both modes S and T across various datasets including MNIST, CIFAR10, ImageNet and audio.
replicas and filters per replica is set to 32 and 8, respectively. As VGGNet and MobileNet have more layers than AlexNet, the run time of inference on these networks is longer than when running AlexNet. VGGNet16 has 16 layers and MobileNet.v1 has 30 layers, while AlexNet has 10 layers. Therefore, the more layers the network has, the more time it takes to run the inference operation. However, we can see as the number of layers increases, the difference between the performance of the two modes becomes more significant. In MobileNet in which the number of layers is three times more than AlexNet, mode T shows noticeably better performance than mode S due to the fact that we can reuse the data in the input buffers in mode T, and we do not have to read the data from DRAM again.

![Bar chart showing the performance of AlexNet, VGGNet16, and MobileNet.v1 in modes S and T.]

Fig. 5.11 The performance of running a binary AlexNet, VGGNet16 and MobileNet.v1 in both modes S and T on the PNM module.

5.3.3 Different Data Layouts

As mentioned in the previous chapter, we explored three distinct data layouts: (1) sequencing one input feature-map followed by all corresponding filters for each layer; (2) arranging all input feature-maps first, then all filters for each layer; and (3) separating all inputs for all layers from all filters for all layers.

Despite testing various combinations of replicas and filters across these three data layouts, the differences in inference run-times are minimal, resulting in virtually identical diagrams.
To illustrate this, Figure 5.12 is provided, showcasing a scenario with 32 replicas and 8 filters per replica. In this specific case, the run-time difference between layouts (1) and (2) in mode S is merely 1.7%. Other scenarios exhibit similarly negligible differences. Consequently, the diagrams for these layouts appear indistinguishable, indicating that the choice of data layout among these three options has little impact on performance despite significant layout variations.

![Fig. 5.12](image)

Fig. 5.12 The effect of different data layouts on inference run-time of binary AlexNet when we run the model for both modes S and T, across 32 replicas with 8 filters per replica.

### 5.3.4 Area and Energy Analysis

The PNM modes S and T have different areas and energy usage. The generic formula for the area of the PNM module is:

\[
\text{Area}_{\text{replica}} = \text{Area}_{\text{reg}} + \text{Area}_{\text{multipliers}} + \text{Area}_{\text{adders}} + \text{Area}_{\text{misc}} \\
\text{Area}_{\text{total}} = \text{Area}_{\text{replica}} \times \text{Number}_{\text{replicas}} + \text{Area}_{\text{buffers}} \tag{5.1}
\]

Here, \(\text{Area}_{\text{reg}}\), \(\text{Area}_{\text{multipliers}}\), \(\text{Area}_{\text{adders}}\), \(\text{Area}_{\text{buffers}}\) and \(\text{Area}_{\text{misc}}\) are the area of registers, multipliers, adders, buffers and other miscellaneous components of the PNM module respec-
5.3 Comparison

The miscellaneous components include the decoders, wiring, the multiplexers and the comparators.

Similarly, the generic formula for energy consumption of the PNM module is:

\[
\text{Energy}_{\text{replica}} = \text{Energy}_{\text{reg}} + \text{Energy}_{\text{multipliers}} + \text{Energy}_{\text{adders}} + \text{Energy}_{\text{misc}} \\
\text{Energy}_{\text{total}} = \text{Energy}_{\text{replica}} \times \text{Number}_{\text{replicas}} + \text{Energy}_{\text{buffers}}
\] (5.2)

Here, \text{Energy}_{\text{reg}}, \text{Energy}_{\text{multipliers}}, \text{Energy}_{\text{adders}}, \text{Energy}_{\text{buffers}}, and \text{Energy}_{\text{misc}} are the energy consumption of registers, multipliers, adders, buffers and other miscellaneous components of the PNM module respectively.

The number of multipliers and adders is the same in both modes, whereas the other components, especially registers and buffers, differ. Therefore, the area and energy consumption of the PNM module in mode T is more than in mode S. In mode S, there are no buffers, and results are directly written to memory. However, in mode T, there are buffers for each filter, and the results are written to these buffers and then written to memory. Also, the number of registers differs in both modes. In mode S, the data and filters need to be stored in registers. The data kept is only as much as needed for a single filter operation; hence, the number of registers for data is the filter size. For the filters, the number of registers is the product of the number of filters supported by the replica and the size per filter. The equation of the size of the PNM module in mode S is:

\[
\text{Size}_{\text{registers}} = (\text{Number}_{\text{filters}} + 1) \times \text{Filter}_{\text{size}} \\
\text{Size}_{\text{buffers}} = 0 \\
\text{Size}_{\text{total}} = \text{Size}_{\text{registers}} + \text{Size}_{\text{buffers}} + \text{Size}_{\text{multipliers}} + \text{Size}_{\text{adders}} + \text{Size}_{\text{misc}}
\] (5.3)

In mode T, the partial result is stored in a shared buffer. The size of the buffer is, hence, the output size of the layer. The number of registers for the data is the product of the length of data along the direction of filter movement and the dimension of the filter in the orthogonal direction. The number of registers for the filters is the product of the number of filters supported by the replica and the size per filter.

The equation of the size of the PNM module in mode T is:
\begin{align}
\text{Size}_{\text{registers}} &= \text{Number}_{\text{filters}} \times \text{Filter}_{\text{size}} + \text{Size}_{\text{data}[0]} \times \text{Filter}_{\text{size}[1]} \\
\text{Size}_{\text{buffers}} &= \text{Number}_{\text{filters}} \times \text{Output}_{\text{size}} \\
\text{Size}_{\text{total}} &= \text{Size}_{\text{registers}} + \text{Size}_{\text{buffers}} + \text{Size}_{\text{multipliers}} + \text{Size}_{\text{adders}} + \text{Size}_{\text{misc}}
\end{align}
(5.4)

In the case of Binary PNM mode, the operations are performed in a bit-wise manner. Therefore, the number of multipliers and adders is replaced with the number of XNOR and popcount operations. The number of registers and buffers is also replaced with the number of registers and buffers needed for the binary network. The area and energy calculations are considered using the same equations as above but with the number of multipliers and adders replaced with the number of XNOR and popcount operations. The number of registers and buffers is also replaced with the number of registers and buffers needed for the binary network.

**Area Comparison**

To show how the area of the PNM module changes as we increase the number of filters and replicas in both modes S and T, I have implemented equations 5.3 and 5.4 in Accelergy tool [95]. For the network, the components per layer are calculated based on the number of filters and the size of each filter. To get the number of components needed, the minimum number of components needed to support the network is calculated. This is the maximum of the number of components needed for each layer. This is then extended to the number of replicas and filters, and these statistics are then used by the Accelergy tool to get the area of the PNM module. Based on the calculation results of the PNM module size for different numbers of filters and replicas in modes S and T, Figures 5.13 and 5.14 are created. As you can see, the area of the PNM module in mode T is significantly more than in mode S.

In previous discussions, I highlighted that the area difference between Modes S and T primarily stems from the size of the shared buffer, along with a marginally higher number of registers in Mode T. Specifically, Mode S utilizes no shared buffer, as the limited immediate partial sums generated during inference are stored in registers. Conversely, Mode T incorporates a substantial storage buffer, sized to accommodate the entirety of the output feature-maps being computed.

To make this distinction clearer, let’s break down the PNM module’s area for both modes into its integral parts: Multipliers, Adders, Registers, and Buffers. A detailed comparative analysis reveals that in Mode T when operating with a single replica and filter, the global buffers constitute an overwhelming 99.9% of the module’s total area. This disproportionate
5.3 Comparison

Fig. 5.13 Displaying the PNM module’s area in mode S against the number of filters with separate lines representing different numbers of replicas. Each line illustrates how variations in filter count impact the module’s area for a specific replica configuration.

share persists until an upper bound is reached – at 64 replicas and 64 filters – where the shared buffer represents 50% of the total area.

In contrast, Mode S features no shared buffer. Consequently, as we increase the number of replicas and filters, it becomes easier to see the impact of each component without any single element overshadowing the others. Excluding the shared buffer, Modes S and T share remarkable similarities in their logical components, differing only slightly in their register counts.

Therefore, I will focus on presenting an area breakdown for Mode S, particularly examining how the addition of replicas and filters influences the proportionate contributions of Multipliers, Adders, and Registers to the module’s total area. Later, I will assess how binarising network parameters can lead to significant area savings in these three components.

The results for a 32-bit case are also shown in Figure 5.16. Figure 5.15a shows the case in which we have only one replica and one filter per replica, while Figure 5.15b shows the case in which we have eight replicas and eight filters per replica.

As you can see, the share of multipliers and registers in the total area of the PNM module is more than the share of adders when we only have one replica and filter; however, multipliers become more dominant in taking the chip space when we change the number of replicas
Fig. 5.14 Displaying the PNM module’s area in mode T, plotting the number of filters against the module’s area with separate lines representing different numbers of replicas. Each line illustrates how variations in filter count impact the module’s area for a specific replica configuration.

Fig. 5.15 The area breakdown for 32-bit PNM module for mode S.
and filters to 8. In the binary case, multipliers and a share of adders would be replaced with XNOR and popcount operations to replace the MAC operations with their bitwise version. As a result, not only the total area of the PNM module but also the correlation between these components was reduced significantly. Figures 5.16a and 5.16b show the area breakdown of the PNM module in mode S for the same configurations as Figures 5.15a and 5.15b but for binary networks. As you can see, the share of multipliers and adders in the total area of the PNM module is almost the same, and the share of registers is significantly less than the other two components.

![Area Breakdown](image)

(a) Number of Replica:1
Number of Filter:1

(b) Number of Replica:8
Number of Filter:8

Fig. 5.16 The area breakdown for binary PNM module for mode S.

**Energy Comparison**

We estimated energy using two methods: one is for the PNM module, and the other is for the DRAM module. The energy consumption of the PNM module is calculated using the Accelergy tool [95]. For each architecture, the number of operations and the number of accesses to DRAM are calculated. This is done at each layer level, and the results are aggregated to give the total number of operations and accesses to DRAM for the whole network. The equations in Section 4.4 are used to calculate the number of operations of the network.

On the other hand, the energy estimates for the DRAM are calculated using statistics from the Ramulator [96] simulator trace. The network was run on the ImageNet dataset, and the
trace of the memory requests was generated. The trace was fed into the DRAMPower [97] tool to generate the energy consumption of the DRAM module.

Fig. 5.17 The energy consumption of the PNM module in mode T for different numbers of filters and replicas.

Fig. 5.18 The energy consumption of the PNM module in mode S for different numbers of filters and replicas.

Figures 5.17 and 5.18 show the energy consumption of the PNM module in modes T and S, respectively for different numbers of filters and replicas. It is important to note that the energy is for a single inference task. As you can see, the energy consumption of the PNM
module in mode T is more than in mode S. This is because the energy cost of the shared buffer is significantly more than the energy cost of the other components. The reduction in time for the computation as the number of replicas and filters are increased causes the total energy cost per inference to reduce.

In mode S, there is another factor that affects the energy consumption of the PNM module. As the number of replicas and filters increases, the time for completion reduces, and the energy consumption of the PNM module reduces. However, once the number of filters and replicas increases beyond a certain point, the energy consumption of the PNM module increases. The reason is the performance of mode S plateaus after a certain point.

Table 5.3 The energy consumption of the DRAM and PNM module for different modes and replica configurations.

<table>
<thead>
<tr>
<th>Mode</th>
<th>Replicas</th>
<th>Filters</th>
<th>DRAM Energy (pJ)</th>
<th>PNM Energy (pJ)</th>
</tr>
</thead>
<tbody>
<tr>
<td>T</td>
<td>1</td>
<td>1</td>
<td>$1.75 \times 10^9$</td>
<td>$1.03 \times 10^9$</td>
</tr>
<tr>
<td>S</td>
<td>1</td>
<td>1</td>
<td>$1.94 \times 10^9$</td>
<td>$1.18 \times 10^7$</td>
</tr>
<tr>
<td>T</td>
<td>8</td>
<td>8</td>
<td>$3.39 \times 10^8$</td>
<td>$3.72 \times 10^7$</td>
</tr>
<tr>
<td>S</td>
<td>8</td>
<td>8</td>
<td>$4.80 \times 10^8$</td>
<td>$1.12 \times 10^7$</td>
</tr>
<tr>
<td>T</td>
<td>32</td>
<td>32</td>
<td>$1.87 \times 10^8$</td>
<td>$2.26 \times 10^7$</td>
</tr>
<tr>
<td>S</td>
<td>32</td>
<td>32</td>
<td>$3.61 \times 10^8$</td>
<td>$1.155 \times 10^7$</td>
</tr>
</tbody>
</table>

Table 5.4 DRAM statistics for different modes and numbers of replicas and filters per replica.

<table>
<thead>
<tr>
<th>Config</th>
<th>Row hits</th>
<th>Row misses</th>
<th>Row conflicts</th>
<th>Reads</th>
<th>Writes</th>
</tr>
</thead>
<tbody>
<tr>
<td>T, 1 filter, 1 replica</td>
<td>477339</td>
<td>11642</td>
<td>41</td>
<td>485565</td>
<td>3503</td>
</tr>
<tr>
<td>S, 1 filter, 1 replica</td>
<td>977557</td>
<td>40351</td>
<td>14356</td>
<td>1029205</td>
<td>3503</td>
</tr>
<tr>
<td>T, 8 filters, 8 replicas</td>
<td>164838</td>
<td>1350</td>
<td>10</td>
<td>162753</td>
<td>3503</td>
</tr>
<tr>
<td>S, 8 filters, 8 replicas</td>
<td>228378</td>
<td>3265</td>
<td>811</td>
<td>229049</td>
<td>3503</td>
</tr>
<tr>
<td>T, 32 filters, 32 replicas</td>
<td>130559</td>
<td>1057</td>
<td>6</td>
<td>128166</td>
<td>3503</td>
</tr>
<tr>
<td>S, 32 filters, 32 replicas</td>
<td>145235</td>
<td>1314</td>
<td>201</td>
<td>143318</td>
<td>3503</td>
</tr>
</tbody>
</table>

To balance the discussion on energy, a comparison of the energy consumption of the DRAM module for different numbers of replicas and filters is shown in Table 5.3. Table 5.4 shows the related statistics for the DRAM module. Mode S has a higher DRAM energy consumption than mode T. This is because the number of reads from DRAM is higher in mode S than in mode T, as the data is read multiple times from the DRAM in mode S. The results are for a single inference; as such the energy consumption depends on the total time taken for the inference.
5.3.5 Expected vs. Actual Performance of Binary AlexNet

In the previous chapter, I discussed how we can estimate the expected performance of running a binary network in modes S and T, the different factors that affect the network’s performance, and how we can predict the expected performance of the network based on these factors. In this section, I compare the expected performance of the network with the actual performance of the network.

The expected performance of AlexNet is calculated using the framework that was discussed in the previous chapter. The actual performance of the network is calculated by running the network in both modes S and T on the ImageNet dataset and measuring the amount of time it takes for the network to finish the inference operation. The results are shown in Figure 5.19. These diagrams are scatter plots. While each dot represents an observation and the position of each dot on the x-axis shows the actual performance of the network, the position of each dot on the y-axis shows the expected performance of the network. As you can see, the expected performance of the network is very close to the actual performance of the network. This shows that the framework that was discussed in the previous chapter is a good estimation of the expected performance of the network.

The average accuracy of the prediction is 90% across all the cases. One major observation is that computing heavily dominates communication in almost all cases; hence, the overlap between the two is insignificant. This is why the prediction is so accurate. The reason for this is that the PNM module is designed to be compute-heavy, and the communication overhead is not significant enough to affect the performance of the network. Additionally, it is notable that the communication overhead for mode S is higher than for mode T. This is especially true when there are a large number of replicas and filters. This is because in mode S, the data is read from DRAM multiple times, and hence the communication overhead is

![Fig. 5.19 Predicted vs. Real Inference Run-time for binary AlexNet on ImageNet.](image-url)

(a) Mode T  
(b) Mode S
5.3 Comparison

higher. Additionally, since there are multiple replicas, the memory controller has to schedule the memory requests for each replica, and this adds to the communication overhead. The modules are also faster in completing the computation; hence, the time for waiting for data is much higher than the time for computation. It is important to note that the saturation point is at the extreme of the configuration and there are multiple design points which have a better performance than the saturation point.

![Predicted compute time vs communication time, mode T, pim_replicas 1](image)

Fig. 5.20 Comparison of computation and communication times versus actual time when there is one replica in Mode T

In our predictions, for mode T, with 64 filters and replicas, the predicted time for computation was only 16% of the total time, and the predicted time for communication was 92% of the total time. With an 8% correction due to the overlap between the two. On the other extreme, for a single filter and replica, the compute time was 90% of the total time. These results are shown in Figures 5.20 and 5.21. The total time taken clearly follows the dominant component, which is computed for the single filter case and communication for the 64 filter case.

This is also evident in mode S, where a large number of replicas and filters lead to a higher communication overhead. These results are shown in Figures 5.22 and 5.23.

An important observation is that the overlap between computation and communication is significant. The PNM module is able to compute while the data is being read from DRAM.
5.3.6 Non-determinism

In our simulations, one rank of the DRAM has been dedicated to the PNM module, and in the simulations, DRAM does not receive any other queries from the memory controller for any other application running in the system. Also, Ramulator is used as the DRAM simulator, which itself is a deterministic simulator. Therefore, the experiments are purely
5.4 Comparison of Area and Performance Trade-offs

Fig. 5.23 Comparison of computation and communication times versus actual time when there are 64 replicas in Mode S

deterministic. However, in a real system, the PNM module will have to share the DRAM with other modules, even though its data is separated into a dedicated rank, but it still may have to wait for some higher priority tasks every now and then to be served. To take this factor into account, I have added perturbation to the DRAM’s response time by adding random delays. The results are shown in Figure 5.24. As you can see, the performance of the PNM module in mode S, which is more data-intensive as it keeps reading the same data in intervals from DRAM, is not affected by the perturbation in the DRAM’s response time. This is because the amount of time that the PNM module spends waiting for the DRAM to respond is very small compared to the amount of time it spends on computation. Therefore, the perturbation in the DRAM’s response time does not affect the performance on a scale that the mean and average response time of several experiments would change. Mode T shows a similar behaviour; therefore, I have not included the results for mode T in this section.

5.4 Comparison of Area and Performance Trade-offs

To find the best configuration of the PNM module when comparing both area and time of completion together, we need to find the Pareto-optimal points of the design space. The Pareto-optimal points are those where we cannot improve one metric without degrading the other metric. In our case, we cannot improve the area without degrading the performance, and we cannot improve the performance without degrading the area. The Pareto-optimal points are, hence, the points where the area and performance trade-off is the best. To do
Fig. 5.24 The performance of the PNM module in a non-deterministic environment.
such an analysis, the area and performance of the PNM module in both modes S and T are compared.

We use the area and performance of the configuration with one replica and one filter as the baseline. Here, the area is minimal, but performance is the worst.

Fig. 5.25 The Area-Performance product for Mode T

Fig. 5.26 The Area-Performance product for Mode S

The results are shown in Figures 5.25 and 5.26. In mode T, the area of the shared buffer is a significant factor in the area of the PNM module. Hence, as we increase the number of filters, the area does not increase at the same rate at which the performance increases.
Hence, in mode T, having a larger number of filters and replicas is better. At 32 filters, the area-performance product is similar for all the configurations. In mode S, the contrast is the opposite. The area of the PNM module is small to begin with, but it increases with the number of replicas and filters. Hence, in mode S, there is a trade-off between the area and performance.

As depicted in Figure 5.26, across all numbers of replicas, using eight filters achieves a balanced trade-off between area and performance. Beyond this, for configurations with eight or fewer replicas, increasing the filter count to 16 enhances the product, but any further increase begins to degrade it. For configurations with 16 or more replicas, the decline in area-performance product occurs after exceeding eight filters. This trend is confirmed by earlier findings in Section 5.2.1, which noted that the performance of mode S stabilizes after reaching 16 filters. However, as more filters are added, the required computational and storage capacities expand, requiring additional hardware. This increase in area contributes to a rise in the overall product due to the higher resource demands.

A more stricter comparison is to also add energy to the product. Figures 5.27 and 5.28 show the area-performance-energy product for modes T and S, respectively.

![Fig. 5.27 The Area-Performance-Energy product for Mode T](image)

Here, it is clearly visible that mode S provides a more optimal trade-off between the area, performance and energy. Mode T, on the other hand, has a higher energy consumption, and hence the product is higher. The area and energy overhead of the buffer is the main reason for this. Figure 5.28 illustrates a pattern consistent with that observed in Figure 5.26. For a module with eight replicas or fewer, the benefit of increasing the number of filters per replica beyond 16 starts to diminish. Similarly, for a module with 16 replicas or more, this threshold
5.4 Comparison of Area and Performance Trade-offs

5.4.1 Comparison with CPU and BRein Near-Memory Accelerator

In this section, first, I examine the impact of near-memory computation on the performance of a network during the inference phase without binarising the network’s parameters. Then, I proceed to show the impact of binarising the network on its inference performance both on CPU and near-memory modes of running operations. Then, I compare my module in its four different configurations to the BRein near-memory accelerator [88], which also works with a commodity DRAM and discuss their differences.

For the near-memory implementation of a full-precision network, I employed the PNM module in modes S and T. Specifically, the configuration for mode S includes 32 replicas with eight filters per replica, while mode T features 32 replicas with 32 filters per replica. I have chosen these configurations based on the observations in section 5.2.2 on running a 32-bit AlexNet on the PNM module. Table 5.5 presents the inference times for both the CPU and PNM modules. The data clearly indicates that the PNM module completes the inference significantly faster than the CPU, demonstrating the efficiency of near-memory computation in enhancing network performance. Additionally, it’s noteworthy that the PNM

![Change in Area-Time-Energy Product as the filters increases for Mode S](image)

Fig. 5.28 The Area-Performance-Energy product for Mode S

occurs after eight filters per replica. Beyond these points, adding more filters negatively impacts the product of area, time, and energy. This trend is due to the execution time remaining relatively constant despite the addition of more filters, while the costs associated with area and energy consumption continue to rise.
module operates at a lower clock frequency than the CPU. Therefore, aligning the PNM’s clock frequency with that of the CPU could further improve run-time performance.

It should be emphasized that the run-time figures presented in the following tables represent the duration of a single inference task.

Table 5.5 Comparing the inference run-time of full-precision AlexNet on CPU vs near-memory

<table>
<thead>
<tr>
<th>Clock Freq. [GHz]</th>
<th>Exec. Time [ms]</th>
<th>Network Arch.</th>
<th>Dataset</th>
</tr>
</thead>
<tbody>
<tr>
<td>32-bit on CPU</td>
<td>2.8</td>
<td>AlexNet</td>
<td>ImageNet</td>
</tr>
<tr>
<td>32-bit on PNM S</td>
<td>1.2</td>
<td>AlexNet</td>
<td>ImageNet</td>
</tr>
<tr>
<td>32-bit on PNM T</td>
<td>1.2</td>
<td>AlexNet</td>
<td>ImageNet</td>
</tr>
</tbody>
</table>

Table 5.6 illustrates the impact of binarising the network parameters on its performance, as well as the distinctions between executing a binarised network on a CPU versus the PNM module. For the PNM module, we present two configurations for each mode: one optimised for a balance between performance, area, and energy, and the other designed for peak performance. In the CPU experiments, our results are benchmarked against XNOR-NET and BNN, where both networks utilise binary weights and activations, aligning with our assumption of all network parameters being binary. In all binarised scenarios on the CPU, run-time performance improved by factors of 58.45 and 3.95, respectively, compared to full-precision CPU execution. For binary networks on the PNM, these improvements are 25.51 times in mode S and 22.78 times in mode T for peak performances. Details on the execution times of these binarised networks on both CPU and PNM are provided in Table 5.6.

Furthermore, when comparing the performance of these various binarised networks, the PNM outperforms BNN on the CPU in all four configurations, despite BNN being tested on the simpler CIFAR-10 dataset and running at a 2.8 GHz clock frequency, approximately 2.3 times faster than the PNM’s clock rate. Comparing PNM’s run-time in different configurations with XNOR-Net on a full-scale CPU at 2.8 GHz, the PNM shows superior run-time in all cases except for mode S with the optimal configuration design.

Table 5.7 presents a comparative analysis between the PNM module and a near-memory accelerator, BRein [88]. The data for the BRein accelerator is derived from its respective publication and investigates the execution of a 13-layer fully connected network on a dataset that is not MNIST-compatible, but they have generated it from the MNIST data set by resizing the input images to $22 \times 22$. Notably, BRein’s architecture is not specifically tailored for CNNs. To execute a CNN network such as AlexNet on BRein, it is necessary to first convert the convolution layers into fully connected layers before proceeding with inference.
### 5.4 Comparison of Area and Performance Trade-offs

Table 5.6 Comparing the average inference run-time of binarised AlexNet on CPU vs. different PNM configurations

<table>
<thead>
<tr>
<th>Clock Freq. [GHz]</th>
<th>Exec. Time [ms]</th>
<th>Network Arch.</th>
<th>Dataset</th>
</tr>
</thead>
<tbody>
<tr>
<td>XNOR-net on CPU</td>
<td>2.8</td>
<td>1.06</td>
<td>AlexNet</td>
</tr>
<tr>
<td>BNN on CPU</td>
<td>2.8</td>
<td>15.68</td>
<td>Alexnet</td>
</tr>
<tr>
<td>1-bit PNM S (optimal)</td>
<td>1.2</td>
<td>5.61</td>
<td>AlexNet</td>
</tr>
<tr>
<td>1-bit PNM T (optimal)</td>
<td>1.2</td>
<td>0.87</td>
<td>AlexNet</td>
</tr>
<tr>
<td>1-bit PNM S (peak)</td>
<td>1.2</td>
<td>0.80</td>
<td>AlexNet</td>
</tr>
<tr>
<td>1-bit PNM T (peak)</td>
<td>1.2</td>
<td>0.77</td>
<td>AlexNet</td>
</tr>
</tbody>
</table>

Additionally, BRein operates at a clock frequency three times lower than our framework and uses a larger CMOS technology.

Comparing the area and energy usage directly between these systems is challenging due to their differences. The energy consumption reported for the BRein model uses less than one million transactions using a binary DNN for handwritten digit recognition. BRein’s architecture includes six systolic arrays, each conducting pipelined inferences; the model weights are stored internally, with data circulating through the system. In contrast, our module processes one inference at a time with its replicas. However, our design is scalable and compatible with larger models such as those used in GraphCore (another systolic system), which BRein’s setup cannot support due to its size limitations. Additionally, BRein’s system can handle up to 900K multiplications, significantly less than the approximately 105 million calculations required for a single layer in AlexNet. Therefore a fair comparison between these two models is not possible.

Table 5.7 Comparison of PNM module and BRein.

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>BRein [88]</td>
<td>400</td>
<td>13-layer Dense MNIST</td>
<td>65</td>
<td>3.9</td>
<td>7.3×10¹¹</td>
</tr>
<tr>
<td>1-bit PNM S (optimal)</td>
<td>1200</td>
<td>AlexNet</td>
<td>ImageNet</td>
<td>45</td>
<td>3.8×10⁸</td>
</tr>
<tr>
<td>1-bit PNM T (optimal)</td>
<td>1200</td>
<td>AlexNet</td>
<td>ImageNet</td>
<td>45</td>
<td>6.0×10⁹</td>
</tr>
<tr>
<td>1-bit PNM S (peak)</td>
<td>1200</td>
<td>AlexNet</td>
<td>ImageNet</td>
<td>45</td>
<td>1.3×10⁷</td>
</tr>
<tr>
<td>1-bit PNM T (peak)</td>
<td>1200</td>
<td>AlexNet</td>
<td>ImageNet</td>
<td>45</td>
<td>1.2×10⁷</td>
</tr>
</tbody>
</table>
5.5 Summary

In this chapter, I have shown the effect of having different design modes for running inference while changing the number of filters and replicas to show their effects. Later, I increased the bit-width of network parameters and, consequently, the hardware of the inference module and see how the aforementioned criteria affect the performance of inference. The area, power, and performance of the inference module are evaluated using the Ramulator DRAM simulator. Furthermore, I delve into the analysis of my module’s performance, taking into account two distinct modes of inference execution. Throughout this chapter, I test the relation between the number of filters and replicas, showing their collective impact on performance measured in terms of DRAM cycles and time. It was found that the area overhead of mode T was restrictive, and the performance of mode S was lower. A Pareto-optimal analysis was performed to find the best configuration of the PNM module. The results showed that Mode S provides a better trade-off between the area, performance and energy. The trade-offs were the best, especially when the filter count and replica count were 16 and one. To conclude, I demonstrated the impacts of binarising the network as well as running the inference phase near-memory isolatedly. Then, I compared the performance of running two binary networks on the CPU with the performance of PNM modules in modes S and T for optimal design and peak performance configurations. And at the end, I provided a comparison between PNM and a near-memory accelerator. In summary, the area and energy overhead of the module in mode S is small and provides a good design limitations trade-off between the area, performance and energy. Moreover, PNM shows promising efficiency in run-time, area, and energy consumption in comparison to CPU baselines and BRein’s near-memory accelerator.
Chapter 6

Conclusion

Deep learning is becoming more mainstream in various application domains, especially in applications that are widely deployed on edge devices with limited computation and storage resources. Running DNNs on such devices and still being able to maintain a reasonable degree of accuracy introduces many challenges to be addressed in this field. Reducing the precision of the networks’ parameters and the computations has shown to be a promising solution. Moreover, as DNNs have very data-intensive computations, bringing the computations closer to the memory where the data is stored can improve its performance and energy efficiency. However, it is crucial to notice that when designing hardware modules to be embedded into these systems, there are always a series of physical limitations to be considered. Constraints such as the area of the design, its storage requirements, power consumption, thermal limit and cost are the factors that limit the complexity of the design.

These restrictions often make designers go for specialised and sometimes overspecialised designs. Although overspecialised hardware would provide marginally better performance for their target applications and models, two issues are important to be considered: 1) the architecture of DNNs is evolving fast, and so the solution design provided for running them needs to be as flexible as possible, otherwise, that hardware would only be practical for a short time; 2) industry greatly prefers to go for a design that introduces the least amount of changes to the already existing systems. To this end, I have introduced a PNM module designed to be placed near commodity DDR4 DRAM. This module is specifically developed for executing inference operations for low-precision convolutional neural networks, with a particular emphasis on binary CNNs.

One common way of computing convolution operations is by flattening the input feature-map and filters’ matrices to gain their equivalent Toeplitz matrix and then turning the matrix-vector process into matrix-matrix multiplication. This approach leads to significant data redundancy and requires extra hardware for rearranging the data before and after convolution.
However, my PNM module is designed so that the input feature-map and filters’ data get streamed into their corresponding buffers row by row, precisely as they are received from DRAM. All the matrices are unrolled and stored in DRAM contiguously. All the filters that need to be applied to a specific input feature-map are placed between two consecutive input feature-maps. Therefore, no data redundancy and extra hardware for rearranging data are required. In the case of binarised CNNs, because all the parameters are binary, all convolution operations take place through their equivalent bit-wise operations. Moreover, besides the buffers and registers needed for each hardware replica, minor extra simple hardware has been added: n-bit shifter, popcount, 32-bit adder, multiplexer, de-multiplexer and comparator. The module is located near DRAM and the memory controller, and there is no need to change anything in the architecture of these two components to enable the use of the PNM module.

Mode S of the design was initially suggested with the intention of processing the convolution while minimizing the area overhead of the design. In this mode, the order in which operations are performed is to prioritize completing the computation of one or more output feature-maps (depending on the number of filters per replica) before starting the processing of a new one. In comparison to mode T, this strategy helps design the module with fewer buffers to store the intermediate results as processing proceeds through the space of input feature-maps and filters by calculating all values necessary for one output feature-map. The downside of this method is for cases in which the number of filters per replica is less than the total number of filters, which results in reading every single input feature-map multiple times from the DRAM. As fetching this data from DRAM is costly, mode T attempts to reduce this by applying all filters for one input feature-map to it consecutively, then moving on to the next input feature-map. This way, each input feature-map is only read once, however, we need to have storage space for all the output feature-maps as the computation of each output feature-map would not be complete until the last input feature-map is read.

The evaluation results for a BCNN based on the AlexNet network show that for both modes, S and T, increasing the number of replicas and increasing the number of filters effectively reduces the network inference runtime. However, the benefit of adding more replicas diminishes notably beyond 16 replicas in mode S, and 32 in mode T. Mode S makes the bus saturated earlier than mode T because of more repetitive readings from DRAM. Regarding the addition of filters, a significant improvement in run-time is observed in both modes up to 32 filters, beyond which point the bus becomes saturated.

In both scenarios, mode T consistently outperforms mode S in terms of runtime efficiency and demonstrates a reduced frequency of DRAM accesses, approximately half that of mode S on average. However, its significant area and energy overhead makes it less suitable for many resource constrained use cases.
Overall, bringing three important design factors of run-time, area and energy into account put us in a better position to make configuration recommendations for each design. Given that DRAM energy consumption varies based on the technology, size and type of DRAM used, specific mode and configuration recommendations depend on the particular use case. As a standalone analysis, we state a configuration of one replica with 16 filters is a strong starting point. Alternatively, for scenarios where higher energy and area costs are acceptable, mode T with 16 replicas and 32 filters per replica becomes a viable option. Conclusively, these configurations show that mode S achieves substantial area and energy savings by factors of 3.56 and 15.99, respectively, while mode T offers a performance improvement of 6.36 times over mode S.

6.1 Future Research

There are several different avenues for continuing this research into the near-memory processing of neural networks.

6.1.1 Supporting Complex DAGs

The current version of the PNM module is limited to performing operations in a sequential manner and does not support more intricate operations that can be represented as complex Directed-Acyclic Graphs (DAGs). The reason behind this limitation is the fact that PNM is focused on CNN architectures, especially the ones reported in BCNN literature, which do not have the complexities of such advanced operations as in DAGs.

DAGs can be theoretically implemented by sequentially processing layers, where each layer reads from the output of any previously completed layer. Residual layers are good examples of that. In chapter 4, we have discussed the potential for incorporating complex DAGs into the architecture. Additionally, exploring multi-modal DAGs for application within the PNM module presents an intriguing area of further study. This involves processing disparate input types, such as images and audio, each through their unique series of layers. The outputs from these distinct layer paths are then converged in a single layer, culminating in a unified output. Enabling support for such complex DAGs would also facilitate the incorporation of functionalities like the attention operation in transformers, as referenced in [10].
6.1.2 Low Precision Integer-Only Inference

Although having binary values for the neural network parameters dramatically reduces the memory footprint and hence the energy consumption [20], it reduces the accuracy of these classification algorithms. Quantised neural networks can be applied to a broader range of more complex tasks, such as natural language processing, face attribute extraction and object detection while aiming for a better trade-off between accuracy, energy efficiency and performance. Many existing quantisation algorithms do not fully utilise the benefits of having quantised parameters. These algorithms use simulated quantisation, where the parameters are stored with quantisation but are cast to floating point numbers for inference. As such, all or part of the inference computations (e.g. convolution, matrix operations, batch norm layers) still need to be done in floating-point precision, limiting the performance and energy efficiency. However, there are existing integer-only quantisation methods [98], enforcing all the network’s quantised parameters to be in dyadic scaling. Having the parameters in dyadic numbers helps quantise any intermediate result using bit shifting instead of integer division (which typically has an order of magnitude higher latency than multiplication). Furthermore, having a similar data layout and buffer alignment as the binary PNM module presented in this thesis, by efficiently simplifying the convolution operations, we can have a more general and accurate PNM module for running the inference phase of QNNs.

6.1.3 Graph Neural Networks

My proposed PNM architecture suits the inference phase of BCNNs, however, application of QNNs and especially quantised convolutional neural networks (QCNNs) can be seen in neural network architectures such as graph neural networks (GNN) and transformers. Due to increasing sizes of models, especially in transformers, where models have billions of parameters [99], there is a move towards lower precision representation and usage of 16-bit floating point representations that are not IEEE-754 compliant [100].

Graph neural networks have architectures that are based on convolutional neural networks. Zhang et. al [101] present a review of the various graph convolutional networks. The basic convolutional operator has been shown to be amenable to quantisation and close to memory operation, as in this thesis. Extending that to graph convolutional networks with additional work towards understanding access patterns for small graphs and optimising them, means that viable methods and systems can be developed for graph neural networks. It is to be noted that there is a drift in the neural network community towards quantised neural networks across different architectural motifs. Future work will look at extending to various architectures beyond CNNs, and develop operators to do so in a near-memory setting.
6.1.4 Network Architecture Search

The PNM module is designed to be flexible and can be used with different network architectures. Additionally, it is important to note that if we vary the optimisation function, the optimal network architecture can change. Network architecture search (NAS) is a field that aims to automate the design of neural network architectures. Hence, it is important to consider the PNM module in the context of NAS. Different NAS algorithms can be used to find the optimal network architecture for the PNM module, driven by a different optimisation function.

6.1.5 Publication plans

The results of this work can be published in a peer-reviewed conference or journal in the near future. We intend to do this to share the results of this work with the wider research community in a shorter format as compared to a long format of a thesis. This will also assist in creating a community supported effort to improve the PNM module and its applications.

6.2 Summary

In this thesis, I have presented a PNM module that is designed to be placed near commodity DDR4 DRAM. This chapter has summarised the results of the evaluation and provides future research directions.
Bibliography


