DEVELOPMENT NURO-RAM: MEMORY MANAGEMENT ARCHITECTURE FOR STREAMING CNN ACCELERATORS ON EDGE

by

Adarsh Navath Sawant

A thesis submitted to the faculty of
The University of North Carolina at Charlotte
in partial fulfillment of the requirements
for the degree of Master of Science in
Electrical Engineering

Charlotte

2019

ABSTRACT

ADARSH NAVATH SAWANT. Development NURO-RAM: Memory management architecture for streaming CNN accelerators on edge. (Under the direction of DR. HAMED TABKHI)

Development of accelerators for deep learning accelerators have been gaining a lot of popularity due to sheer amount computation performed by deep learning algorithms. From the onset of Moore's law failure it has become difficult to improve the performance of the general purpose processors and hence computer architects are inclining towards more heterogeneous solutions for accelerating deep learning applications efficiently. Secondly the computation performed in the deep applications are repetitive and predictable which naturally leads to three choices *ASICSs*, *GPUs* and *FPGAs*. *FPGAs* due to its configurability, deep pipeling abilities and high performance per watt has been one of favorite devices for accelerator architecture research. As *FPGAs* are really difficult to program and thus there has been thus rise in development reusable accelerator templates which can be instantiated even by software developers.

Memory always has been the main bottle-neck even for the architecture with most efficient compute data-path. This problem is further compounded as *FPGAs* have low on-chip memory footprint (in form of BRAMs). Most of the deep learning applications have a very high model size (ex: AlexNet has model size of >100Mb). Thus to accelerate deep learning applications there is a need to develop memory systems to support these application. Conventional accelerators try to mitigate with these issue by accelerating single layer sequentially which has its own implication like bandwidth wastage, power consumption, etc. Since the presented work serves streaming accelerators, separate strategy has to developed. This work presents development of such memory management system called NURO-RAM. NURO-RAM uses minimum sized pre-fetch buffers and static weight scheduler in order to support the deep learning accelerator AWARE-DNN. This work implements three different network AlexNet, Shallow mobile net and Tiny Darknet to show the diversity of the NURO-RAM to serve streaming accelerators like AWARE. We then compared NURO-AWARE solution (implementing AWARE-DNN with support of NURO-RAM memory system) to Chai DNN an HLS based deep learning accelerator library and NVDIA Xavier mobile *GPUs*. The purposed network consumes lower power 4.5 watts (NURO-AWARE) vs 10 watts (Chai DNN). The power consumption against *GPUs* is comparable with 5.7 watts consumed by NVDIA Xavier. This work also consumes lesser BRAM for both AlexNet 75% in NURO-AWARE vs 88% in Chai DNN and for Tiny Darknet 48% in NURO-AWARE vs 88% in Chai DNN. With lesser BRAM utilization than state of the art architecture and separate utilization for three different networks shows that the NURO-AWARE architecture is resource aware as well as application aware. The lesser power consumption and resource utilization of the presented work can be attributed to the custom data path used by the AWARE DNN accelerator and the custom memory access path developed by the presented solution for each layer which reduces the off-chip memory access. The presented work beats Chai DNN which can support 10.21 FPS with fully connected layers whereas NURO-AWARE can support 30 FPS, and even Xavier *GPUs* with support of 2 FPS for performance metric. This is because the presented solution uses its architectural knobs to satisfy the real-time frame rate requirement. Overall the presented solution beats *GPUs* as well as *FPGA FPGA* based state of the art solution in performance per watt.

DEDICATION

This work is dedicated to my family, friends and colleagues.

ACKNOWLEDGEMENTS

I would like to express my gratitude and thank my advisor Dr. Hamed Tabkhi for his insights about the topic, continuous motivation, patience and enthusiasm for the work presented here. I would also like to also thank my committee members Dr. Andrew Willis and Dr. Ronald Sass for their valuable feed-back and guidance. Finally, I would like to thank my family, friends and TECSAR team for their support through good times and bad.

TABLE OF CONTENTS

LIST OF TABLES ix

LIST OF FIGURES x

LIST OF ABBREVIATIONS xii

CHAPTER 1: INTRODUCTION 1

1.1. Why *FPGAs* for AI on Edge? 1

1.2. Difficulty in *FPGA* deployment 3

1.3. Contribution 5

1.4. Thesis outline 6

CHAPTER 2: Background and Related Work 7

2.1. Deep Learning and Convolutional Neural Network 7

2.1.1. How does learning happen? 7

2.1.2. Convolutional Neural Network 9

2.2. AWARE-DNN 11

2.2.1. Functional Composition of AWARE 12

2.2.2. AWARE framework 14

2.3. Xilinx Chai DNN 14

2.4. Related work 15

CHAPTER 3: Approach 19

3.1. Motivation for use of off-chip memory 19

3.1.1. Design formalization for NURO-RAM 20

3.2. NURO-RAM Architecture Overview 22

3.2.1. System architecture goals 23

3.2.2. Off-chip memory layout 24

3.2.3. NURO-RAM Micro architecture 25

3.2.4. Weight Bank 25

3.2.5. Weight Blocks 26

3.3. System Design Components 30

3.3.1. Xilinix IPs used in the design 30

3.3.2. Arbitration and Scheduling 34

3.4. Software Design 39

3.4.1. Firmware 40

3.4.2. System Initialization 41

3.4.3. Handling the Interrupts 43

CHAPTER 4: Experimental methodology and Evaluation 45

4.1. Experimental methodology 45

4.2. Evaluation and Results 47

4.2.1. Implementation of neural networks 47

4.2.2. Evaluation of networks with other Edge solutions 49

4.2.3. AWARE-DNN Extension 51

CHAPTER 5: Conclusion And Future Work 61

5.1. Conclusion 61

5.2. Future Work 62

REFERENCES 63

LIST OF TABLES

TABLE 1.1: AlexNet-Weight Requirement 5

TABLE 3.1: Layer info 41

TABLE 4.1: Ultra-scale Device Resources 45

TABLE 4.2: Power report settings 46

TABLE 4.3: Alexnet- NURO-RAM Weight configuration 53

TABLE 4.4: Coupling for shallow mobile net 54

TABLE 4.5: coupling for Tiny Darknet 57

LIST OF FIGURES

FIGURE 1.1: CNN Model Size 4

FIGURE 1.2: BRAM Memory available on FPGA 4

FIGURE 2.1: Deep network 8

FIGURE 2.2: Neural node 8

FIGURE 2.3: Lenet Example 10

FIGURE 2.4: An abstraction of a full architecture Instance 12

FIGURE 2.5: Aware toolflow 14

FIGURE 3.1: Weight elements 21

FIGURE 3.2: Overview of NURO-RAM System 23

FIGURE 3.3: Example of off-chip memory tiling. 25

FIGURE 3.4: Weight Bank 26

FIGURE 3.5: Weight Block 27

FIGURE 3.6: Asynchronous State Machine 28

FIGURE 3.7: Memory interface generator 30

FIGURE 3.8: AXI Central Direct Memory Access 32

FIGURE 3.9: AXI interrupt controller 33

FIGURE 3.10: Static Scheduling using variable size pre-fetch buffers 36

FIGURE 3.11: Timing diagram for Scheduling 37

FIGURE 3.12: Scheduler Firmware 40

FIGURE 3.13: System initialization 42

FIGURE 3.14: Interrupt handler flow 43

FIGURE 4.1: Latency of CNN algorithms using NURO-AWARE 48

FIGURE 4.2: Implementation of CNN algorithms using NURO-AWARE 49

FIGURE 4.3: Power usage of CNN algorithms using NURO-AWARE 50

FIGURE 4.4: Comparision with Chai DNN implementation 51

FIGURE 4.5: NURO-AWARE performance comparison 51

FIGURE 4.6: NURO-AWARE power comparison 52

FIGURE 4.7: Reduction in memory footprint in AlexNet 53

FIGURE 4.8: BRAM utilization comparison 55

FIGURE 4.9: Comparison OCP vs DCP 56

FIGURE 4.10: BRAM utilization Shallow Mobile net 57

FIGURE 4.11: Total DSP utilization in shallow mobile net 58

FIGURE 4.12: BRAM utilization comparison 59

FIGURE 4.13: Comparision of reduction in ocp vs dcp design point 59

FIGURE 4.14: Total BRAM utilization Tiny Dark-net 60

FIGURE 4.15: Tiny Darknet Dsp utilization 60

LIST OF ABBREVIATIONS

3D	Three dimensions
4G	foruth generation device
5G	fifth genration device
AI	Artificial intelligence
APE	Agregation processing unit
ASIC	Application specific integrated circuit.
ASYNC	Asynchronous
AXI	Advanced eXtensible Interface
BRAM	Block random access memory
CMDA	Central direct memory access
CNN	Convolutional Neural Network
CPE	Convolutional processing unit.
DDR4	Double Data Rate 4 Synchronous Dynamic Random-Access Memory
DMA	Direct memory access
DNN	Deep Neural Network
DRAM	Direct Random access memory
Dsp	Digital signal processing block
ECE	Electrical and Computer Engineering
FIFO	First in first out
FM	Feature maps
FPGA	Field programmable gate array
GPU	Graphical processing unit

HLS	High level synthesis
Intr	interrupt
IPS	intellectual propertys
LUT	Look up table
MAC	Multiply and accumulate unit .
MIG	Memory interface generator
NURO	Neural
PE	Processing Element
RTL	Register transfer logic
SERDES	Serializer/Deserializer

CHAPTER 1: INTRODUCTION

Deep learning can be used to solve wide range of problems efficiently. Convolution Neural Networks (CNN) a special branch of deep learning, can be used to tackle vision based of problems such a image segmentation, object detection, identification and tracking with higher success rate than conventional computer vision algorithms. Even though CNNs are very compute intensive, due to the advancement in the compute architectures and the large availability of image data sets which is used to train these algorithms has made it quite popular within computer vision community.

With the advancement of the networking domain another wave of computing is becoming popular known as edge computing. With 3G networks tiny sensor network were used to log temperature, humidity for data analysis. But with advent of 4G/5G, edge computing the industry is looking towards deploying data bandwidth intensive applications like deep learning based inference and computer vision on edge. But the design of edge computing systems poses several restrictions such as

- Size: Since these devices are going to be connected to the edge, the size of devices has to be small.
- Power: Edge devices must be less power hungry as the most of the edge devices might be battery powered or have limited power source.
- Security: Since these devices are going to be connected to the network and they perform data gathering and computing, these devices need to have full proof protection against the network attacks.
- Real time constrains: As these devices work on the edge the system should work in real time latency constrains due to their use in safety critical applications such as the self driving cars.

1.1 Why *FPGAs* for AI on Edge?

Currently most of state of the art edge applications use a cloud computing paradigm where the sensor collects the data on edge and it is then sent for further processing on the cloud and the results of the computation is sent back to the edge node. Even though this model serves most of the edge computing constraints it inherently has following drawbacks:

- Security: Since the devices has to constantly send data to the cloud, it is vulnerable to security attacks
- Wastage of bandwidth: If we use this model for vision application we need to constantly stream video data to the cloud, due to which there will be a lot of bandwidth wastage.
- Power: Since the data has to be constantly streamed to the cloud server a lot of power is wasted.
- Real time constrains: As the model constantly receives the computed results from the cloud server, the system can fail due to network faults.

Due to these limitations the edge computing paradigm has been making a shift from hard cloud computing based models to soft cloud computing based models where most of the computing is done on the edge node near the sensor and only critical data is sent to the cloud server for processing. But applications like deep learning are inherently compute intensive. Thus soft cloud computing models need highly efficient edge computing nodes which can have a balance between the real time computing requirement and power, because of which this has been an active research area. Also new CNNs models are being researched actively thus there is a need for computing architectures to be configurable as to cater the needs of the new algorithms.

The current state of the art architectures used for the CNN are based off either *ASICs* or *GPUs*. ASIC can be well suited for power/watt but they lack in the flexibility which might be needed for adopting for a new algorithms. Also the time to market and the cost of developing a custom *ASIC* is high because of which this might not be the most cost effective solution. On the other hand *GPUs* can be generically used to run any CNN algorithm, but their data-path cannot be adopted in order efficiently run new algorithms. Secondly the performance/watt for the *GPUs* is really low and hence it is difficult to deploy the *GPUs* near the sensor on the edge.

Due to the reasons stated above there has been a increase in the research for deploying the deep learning application on Field programmable gate array (*FPGA*) based architectures. The *FPGA* based architectures have following advantages:

- Flexibility: *FPGAs* can be used to create a custom data paths like *ASICSs*, but these data paths can be changed numerous times depending on the algorithm's requirement.
- Performance/watt: Though *FPGAs* cannot beat the power/performance ratio of *ASICSs* but they can easily beat *GPUs*.
- Real-time deterministic performance: *FPGAs* have been traditionally used for the real time applications such as signal processing, the architectures developed on these devices can be used to efficiently tune the system for required real time performance as demanded by edge.

1.2 Difficulty in *FPGA* deployment

Even with these advantages the *FPGAs* deployment for deep learning application on edge has been hindered due to the complexity involved in developing *FPGA* based system. This is related to the fact that *FPGA* development requires hardware expertise and the deep learning community has been purely software based. Also the deep learning algorithms use weights and intermediate feature maps intrinsically for inference which requires a high memory real estate. But the memory real estate available on the *FPGA* is very low which may not been able to fit big deep learning models. To tackle these problem there has been active research conducted to developed architectural templates which are developed by using handcrafted RTL (register transfer logic) by hardware experts. These architectural templates are then integrated into deep learning software frameworks by using the architectural compilers so that the it can be used to run the deep learning algorithms by the software developer. One

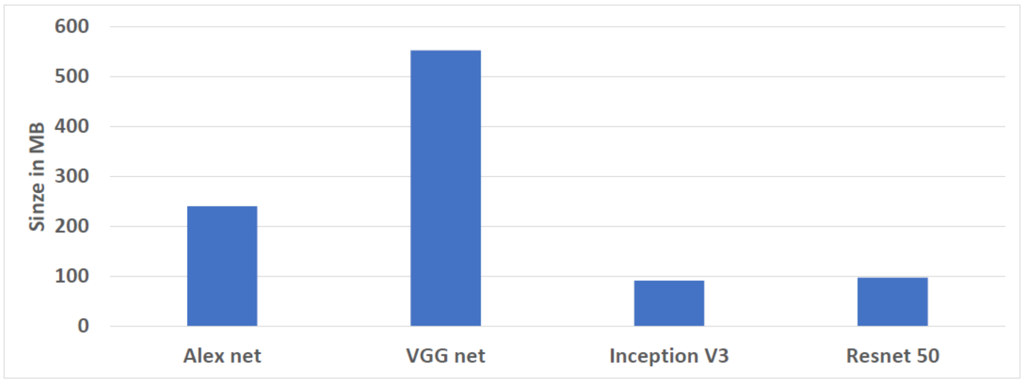

Figure 1.1: CNN Model Size

such architectural compiler under development is AWARE-DNN. AWARE-DNN uses a library of customizable hardware templates in order to generate a pipelined architecture for accelerating CNN thus easing the developer from the work of handcrafting the hardware from the scratch. More discussion about AWARE is given in section 2. But as discussed above the problem of the low memory real-estate on *FPGA* still prevails. Figure 1.2 shows the amount of on-chip memory available in form of block memory (BRAM) for the *FPGA* devices. It should be noted that even though *FPGAs* like kintex ultra-scale and virtex ultra-scale are server class *FPGAs*, the amount BRAM (block random memory access memory) available on-chip is mere 90 MB.

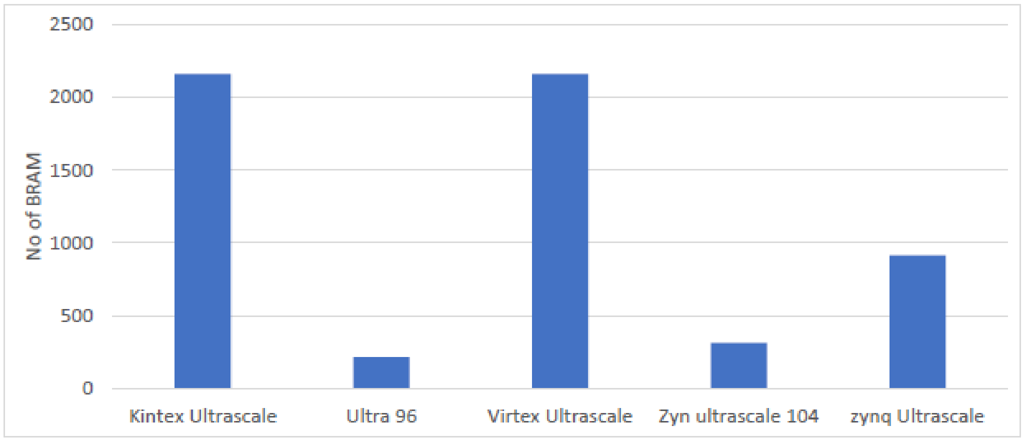

Figure 1.2: BRAM Memory available on FPGA

With large networks, such as VGG-Net or AlexNet[1], the memory footprint (amount of memory required to store the weights) of the network's weights can reach up-to 100

MB. Figure 1.1 gives overview of the model sizes of some popular CNN network. From this Figure we can deduce that the these networks will not fit easily on *FPGA* due to memory requirement. This memory footprint becomes problematic when compared to the memory real estate commonly available on *FPGAs*, around average of 30 MB. This limitation is compounded further by the need of streaming accelerators, such as AWARE-DNN, to store intermediate feature map data in on-chip memory, further reducing available resources. It should be also noted that the memory footprint on the CNN networks are heterogeneous across the layers. Table 1.1 shows the memory foot print for each layer to store weight during the inference phase for AlexNet.

Table 1.1: AlexNet-Weight Requirement

	Filter Size	Input Channel	Kernel	Total Memory
CNV 1	11X11	3	96	34KB
CNV 3	5X5	96	256	614KB
CNV 5	3X3	256	384	884KB
CNV 6	3X3	384	384	1327KB
CNV 7	3X3	384	384	884KB
FC9	6X6	384	256	37MB
FC 10	FC	256	2048	16MB
FC 11 7	FC	2048	2048	4MB

As it is apparent, storing all the weights on chip is not a viable solution. Secondly, off-chip memory access is notoriously slow. This necessitates the creation of a supporting system to interface on-chip and off-chip memory while mitigating the latency of off-chip memory access. But since the on-chip memory is less, this system has to make the efficient use of the on-chip memory by loading only the minimum required weights at particular instance of time during acceleration.

1.3 Contribution

To mitigate the memory real estate problem posed by streaming deep learning accelerators we present NURO-RAM. Statement: This work provides an approach for streaming the weights for the real-time CNN accelerator like NURO-RAM which can be effectively applied for other streaming CNN accelerators. "NURO-RAM segregates memory requirements for each of the layers in CNN by using minimum sized on-chip weight buffers , provides an interface with hardware accelerator architecture, handles arbitration for off-chip memory access, and acts as a supporting system for streaming accelerators." In a nutshell this work presents:

- Systematic approach to decouple the on-chip memory bottle neck for streaming CNN accelerators like AWARE-DNN.
- Development of memory system and its interface for supporting for streaming CNN accelerators.
- An approach towards managing the off-chip transfers.

The contribution of this work are

- Design formalization for hardware blocks for breaking the on-chip memory bottleneck.
- Architecture of the NURO-RAM system hardware and its subsystems.
- Software system for controlling the arbitration of the OFF-chip memory access from different layers.

1.4 Thesis outline

The outline of this thesis is as follows. Chapter 2 reviews the background needed for this study. It reviews deep learning and Convolutional Neural Network software execution model.It also gives a brief overview about AWARE-DNN and Chai DNN which are two seperate *FPGA* based CNN accelerators. It also briefly overviews related work in the field of *FPGA* based CNN accelerators and system development. Chapter 3 presents, the motive to use off-chip memory for NURO-RAM, design formalization, the architectural and micro-architectural details about NURO-RAM. It also describes about the system design, approach towards scheduling strategy used to schedule multiple requests for off-chip transfers. Chapter 4 represents experimental methodology, and the evaluation of presented work. Chapter 5 concludes thesis and briefs details about future work.

CHAPTER 2: Background and Related Work

2.1 Deep Learning and Convolutional Neural Network

Deep learning is a special class of machine learning algorithm which used to solved problems by mimicking how the human brain works, where each neuron is connected to other neuron and when each of these neurons are activated they also activate the neurons connected to it. Basically deep learning algorithms take inputs from real world and predicts an output particular to that problem. For example a deep learning algorithm might take input as zip code, number of rooms, and amenities available and would predict the price of the room. To learn to predict the price of the room the deep learning algorithm has to be fed with large amount of data where the output of the predication is already known, which is known as training data set. By minimizing the error in prediction from the algorithm and the prediction already present in the data set the deep learning network tries to learn the association-pattern between given inputs and outputs this in turn allows a deep learning model to generalize to real world inputs that it hasnât seen before.

2.1.1 How does learning happen?

Deep learning algorithm models use layers of nodes which are analogous to the neurons in the brain. Each model can consist of three or more layers of node. The first and second layers are knows as the input and output layers. The rest of the layers are the hidden layers. Each node is connected to one or more node from the previous layer. Following Figure 2.2 shows a view of a full deep neural network.

Figure 2.2 shows a view of a full deep learning node. Following are the key elements in the neural node

- Weights:The weights decides how much effect the output from previous connected node has on the current node. Basically the output from the previous nodes are multiplied by the weight and the output is fed to the activation func-

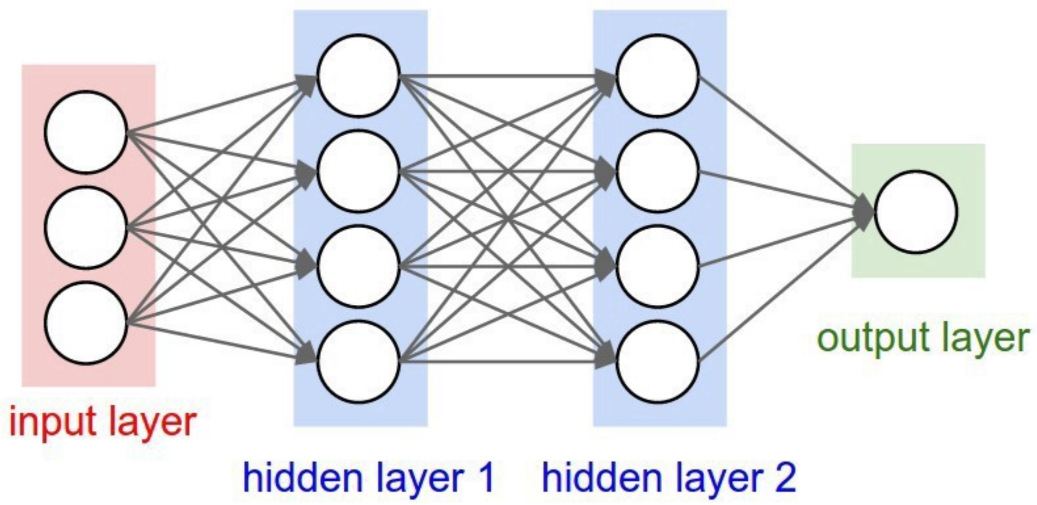

Figure 2.1: Deep network

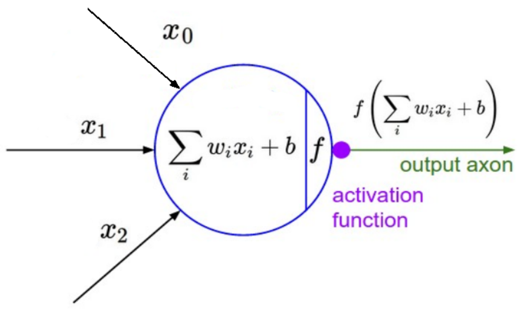

Figure 2.2: Neural node

tion. In Figure 2.2 these are indicated by $x0$, $x1$, $x2$, etc.

- Activation function: Decides whether the node output will be activated or not. In Figure 2.2 these are indicated by function f.
- Biases:These are another learnable parameter like weights in deep learning. The biases are related to particular node only rather than connection to other node like weights. In Figure 2.2 these are indicated by b.

During the learning phase all the input weights to the neural network are initialized to the random values. When the training set inputs are passed, the output predicted by the neural network is compared to the actual value from the training data set. The algorithm then uses algorithms like back propagation in order to adjust the weights so that the difference between the prediction and actual value is reduced.

2.1.1.1 Deep learning for images

It is possible to flatten out the entire image and pass it through the deep neural network for predictions but it is not a good idea because

- Scaling of Weights: Regular neural network do not scale well for images. For example In CIFAR-10 [2], images are only of size $32X2X3$ (32 wide, 32 high, 3 color channels), so a single fully-connected neuron in a first hidden layer of a regular neural network would have $32X32X3 = 3072$weights considering the size of the network it can see that it is impractical to use the regular neural network.
- locality: an image consist of lot of information in temporal and spatial locality in it, flattening the image would remove all of these localities and reduce the predication accuracy.

2.1.2 Convolutional Neural Network

Due to the problem discussed above a New class of deep learning algorithm are developed for detection and identification of objects in images called Convolutional Neural Networks (CNN). The role of CNN is to reduce the demensionality of the image so that processing of the image becomes easy while still maintaining the temporal and spatial locality in the image. In traditional image processing hand-designed filters are used to detect the objects. But this method is not robust when the scene is diverse as it is not practical to design filters for all different kind of objects. A CNN algorithm learns all these filters (also called weights) by them self during the training phase. It is similar to how the human visual system works, where each of the portion in the visual cortex respond to set of stimulus in receptive field. Similarly the filters learned by CNN algorithm respond to the set patterns in the image.

2.1.2.1 Basic architecture of CNN

The basic architecture of the CNN is shown below in Figure 2.3 The three important

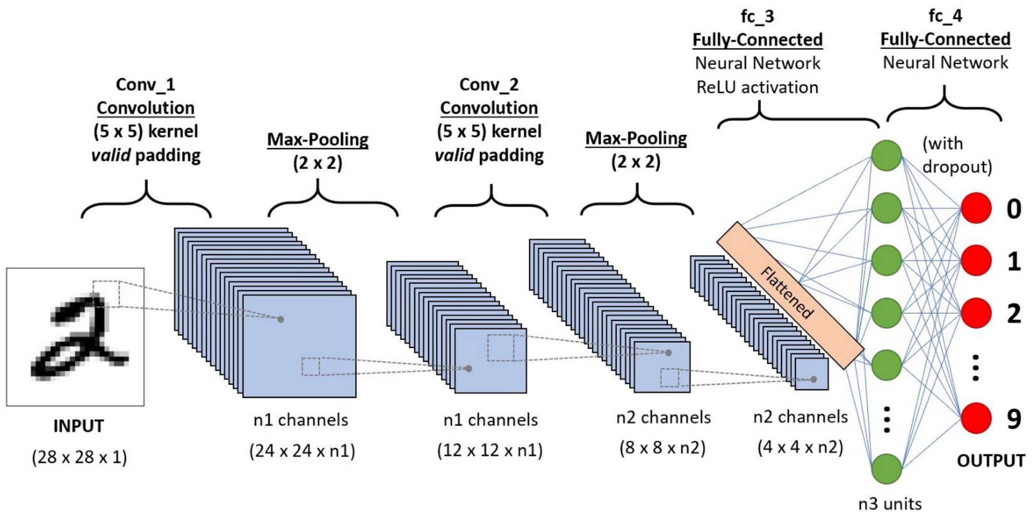

Figure 2.3: Lenet Example

building blocks of the CNN are explained as follows:

- Convolution layers: The convolutional layers receive input from the previous layers and compute the dot product between the weights to generate the 3D output feature map.
- RELU layer: These layers apply activation functions to the output from the convolutional layers.
- MAX POOL: Layer will perform a down sampling operation along the spatial dimensions (width, height) using convolution, so that only dominant part of the output volume from the previous layer would be sent forward to next convolutional layers.
- Fully connected layers: This layers perform the final identification. Fully connected layer will compute the class scores, each neuron in this layer will be connected to all the numbers in the previous volume.

Using the architecture components described above a CNN network transforms an image layer by layer from pixel values to the detection class scores. Each of the filters in the CNN are learned during the training phase by uses algorithms like back propagation.

2.2 AWARE-DNN

AWARE-DNN architecture compiler framework is used to develop efficient hardware accelerators for application by using the hand optimized architectural templates.Figure 2.4 shows an abstract view of a full architecture instance. Each layer is mapped to an individual pipeline. Each stage contains a buffer sized for a small section of the FM tile. When enough FM data has accumulated, the stage will begin computation feeding to the next stage in a producer consumer fashion. This cycle continues for each layer until the last, which will output the classification vector when complete. This is called temporal layer parallelism, or more specifically, layer pipelining. This minimizes inter-layer data movement but puts pressure on the memory hierarchy used for weight storage. Due to which special memory hierarchy might required in order to generate exploit more parallelism. The pipeline stages utilize the targeted layer's unique parallelism and data flow, and from multiple granularities of complexity, in order to guarantee efficient mapping for the domain. The available parallelism knobs which can be exploited to cater the design needs are given below:

- **Convolution parallelism** The convolution operation can be done with a single Multiply Accumulate Unit (MAC). But if the FM data source is extended, convolution becomes a sliding window operation. This creates data dependencies that must be maintained. AWARE-DNN design, however, can benefit from this data extension through buffering pipe-lining Strictly defined, convolution parallelism is temporal parallelism realized with a single MAC and spatial parallelism realized by accumulating the results of numerous multiplications.
- **Kernel parallelism:** AWARE-DNNN uses arrays of convolutional processing

elements in repeated fashion to compute all the kernels of the a particular layer simultaneously. This operation needs the weights and feature map data to be fed to the architecture simultaneously.

- **Channel parallelism:**Due to the nature of DNNs, the channel parallelism of a layer is determined by the kernel parallelism of the previous layer. The convolution operation at this granularity needs every channel to complete a full convolution. AWARE-DNN architecture takes advantage of this in the form of channel buffering.

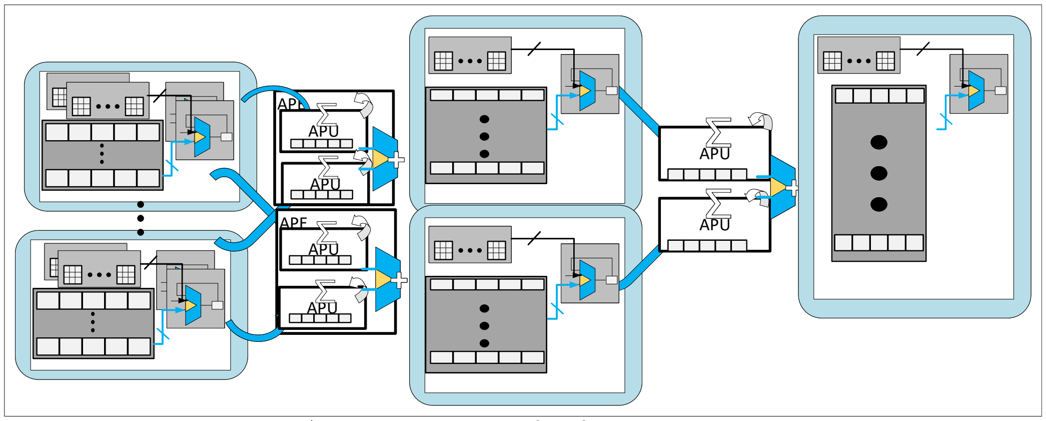

Figure 2.4: An abstraction of a full architecture Instance

2.2.1 Functional Composition of AWARE

The Functional composition of each architecture instance also consists of the following handcrafted hardware units. An overview of each of these units are explained below:

- **Convolution Processing Engine:** The essential function unit of AWARE-DNN is the Convolution Processing Engine (CPE). CPEs consist of two subcomponents. The first is the Mac-Engine, which handles the computation. It is driven by the 2D line buffer which is extension to 1D line buffer[3] and accesses weights according to the line buffer's control signals. All the control overhead of the multidimensional direct convolution operation is exposed to the 2D buffer.

Convolution parallelism requires parallel read operations, putting pressure on the weight banks and 2D line buffers. Kernel parallelism results in replication of the data path's backend. This avoids the mixing of multiple kernel contexts.

- **Aggregation units:** Handles the aggregation of multiple CPE. Since every CPE is constructed at the granularity of a single channel, the results have to be accumulated then passed to the next processing element.
- **RELUs:** Handles the non-linear activation found in DNNs. Compares its input to zero and outputs the max. This is required for the summation of every channel for an individual convolution window.
- **Pooling Processing Units:** Pooling Processing Units are the final atomic function unit. It accelerates the max pool operation on streaming feature maps.
- **Tensor buffer:** The tensor buffer is a Fused-Function Unit. The tensor buffer is an extension to the 2d line buffer [3]. It can be thought of as fused convolutional processing unit. The number of tensor buffers is dictated by the layer's degree of channel parallelism. The MAC engines of each tensor buffer require same degree of kernel parallelism, meaning the total number of MACs for every layer is dictated by kernel parallelism times channel parallelism.
- **Multi-Dimensional Aggregation Processing Engine (MDAPE):** The fusion of tensor buffers results in channel aggregation units to work at the granularity of a single row, instead of a single pixel. This ensures that the channel parallelism of the fused CPE results in the aggregation of those channels while kernel context is kept separate. The Multi-Dimensional Aggregation Processing Engine (MDAPE) performs this aggregation function. Each MDAPE consists of sub blocks called Aggregation Processing Engines (APE). The number of APEs is equal to the channel parallelism in the layer. Each APE consists of Aggregation Processing Units (APU). The number of APUs is equal to the ker-

nel parallelism of the particular layer. Each APU operates on single kernel and aggregates the convolution for each channel sequentially.

2.2.2 AWARE framework

The functional units are designed around highly parameterized chisel modules organized in a hierarchy of functional composition, as shown in Figure 2.5. The network requirement form the high level tool (Caffe) are converted into architectural template after design space exploration has been performed . Once the layers are finalized the framework generates architecture instances of each layers by using chisel templates, these templates are then converted in to verilog files for synthesis. The individual modules are then converted into vivado IPs and the architecture is realized on *FPGA* using Vivado synthesis tool.

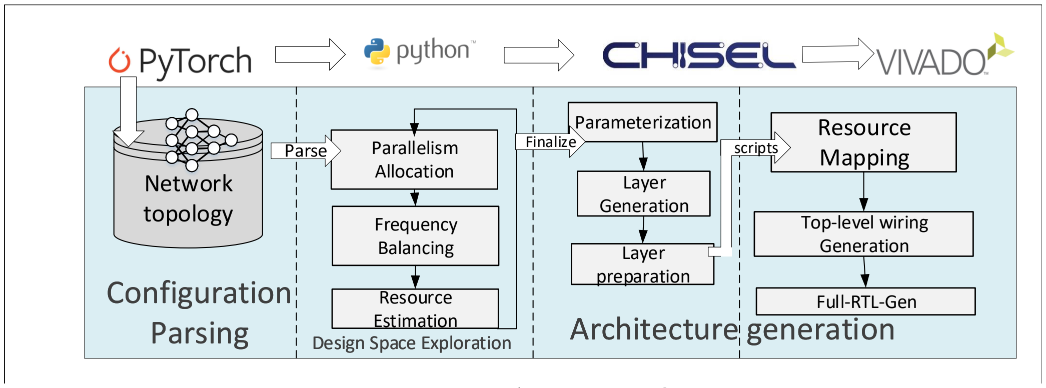

Figure 2.5: Aware toolflow

2.3 Xilinx Chai DNN

Xilinx Chai DNN is HLS based deep neural network library for acceleration of deep neural networks[4]. The architecture used in Chai DNN is systolic array based architecture where individual processing elements work independently to complete same task. Chai DNN has received a lot of appreciation from industry due to its ease of use and support for the network. It also supports two type of quantization dynamic fixed point and Xilinx quantization. Both of which help two reduce model size. The flow includes creating hardware using their pre-built overlay file, and then developing application using the Xilinx provided API calls to run the inference on the software stack for pre-built networks. Currently it supports all major CNN networks. If the user want to build the new networks which is already not supported by framework, user will have to parse the prototxt file in order to create the data flow graph. Then generate optimised weigths if required using the quantizer tool and then run the software stack. Even though the architecture is implemented on *FPGA*, Xilinx allows only certain hardware configuration which can be used to run Chai DNN, due which this work feels that its configurability is limited to several configurations.

2.4 Related work

A traditional CNN can be viewed simply as a sequence of convolutional layers and max pooling layers. In algorithmic terms, a convolution layer is simply a sliding dot product of a series of multi-channeled filters (kernels) against equally multi-channeled feature maps (FM). The generalized solutions of convolution layers typically take the form of massive systolic arrays, such as the Google TPU [5], or flexible spatial architectures [6]. These architectures define compute parallelism by the data movement of the targeted layer, which can be enormous for DNNs. This leads to large compute and memory requirements, but results in increased power efficiency by way of throughput. The power efficiency will scale with the throughput of the design, but will be limited by the extreme bandwidth requirements and resource consumption that comes with mapping the multidimensional operations of DNNs into a single large matrix multiplication [7]. This large resource requirement leads to sequential execution of the DNN layers, forcing all the layers to map onto the same architecture. This results in under-utilized resources, under-served layers, or both.

The work proposed by Shen et al. [8] takes into account the unique computational dataflow and resource requirements of differing layers. However, due to the exceptional resource expense of building specialized architectures for each individual layer, some architecture designs are reused across layers with similar requirements. Putic et al. [9] proposed re-configuring the accelerator at run time and analyzed the overhead of reconfigurability against the benefit of hyper specialization.

Systolic arrays, such as Eyeriss [6], are a specific form of spatial dataflow architecture. Spatial dataflow architectures map weight and FM reuse to a compute engine of inter-connected processing elements (PEs). Typically a complex memory hierarchy is emulated through an underlying Network-On-Chip to enable higher bandwidth for the processing elements. Efficient computation of DNNs is inherently coupled with an understanding of the algorithm's intrinsic dataflow and exploitable parallelism [10]. Reordering, pipelining, and tiling of nested loops is used to alter the dataflow of DNNs. The granularity of pipelining, spatial parallelism, and buffering with respect to the FM and weight stream are of grave importance [11].

Spatial dataflow architecture by itself is not always enough to satisfy application constraints. Since accuracy tends not to scale beyond 8-bit resolution [12, 13, 14, 15], quantization is a useful optimization, reducing resource requirements while increasing power efficiency. [16, 17] explore binary quantization for DNNs. However, our work focuses on 8-bit resolution for its minimal loss of accuracy. In addition to reducing the numerical precision, transforming the computation into the log-domain, thus reducing computation and memory requirements, has also been explored [18, 19].

The work of [20] exploits the close proximity of the edge device to the sensor for mitigating data movement costs. This approach eliminates the transfer of intermediate FMs into main memory. In addition to FM streaming, all weights are stored on-chip as well. These two joint approaches amortize the dominating energy cost of data movement [21, 22] at the cost of data storage.

Instead of just reducing the memory accesses cost, memory access can be completely eliminated through exploitation of sparsity or compression. Han et al. [23] utilized a sparse matrix multiplication accelerator in combination with pruned and compressed network topologies, bringing the power consumption down significantly. The work of [24] presents an ASIC architecture that compresses both input images and kernel data to minimize DRAM accesses. This work also intelligently partitions memory and computational resources so they can map easily to the targeted algorithm, allowing the design to maintain a level of flexibility.

Overall, the works above were centered around either eliminating data movement or mitigating computational requirements. Systolic arrays and other pure spatial dataflow architectures are generally unable to satisfy the domain of edge DNN applications. These architectures scale their power efficiency with bandwidth and data storage in order to satisfy a throughput constraint. This methodology ignores latency and is only energy efficient when scaled properly.

Network pipelined architectures are an alternative to these pure spatial dataflow architectures. These architectures utilize the idea of fused layers [25] to the extreme in order to keep all FM data on-chip. By tiling the feature map, the resources required for specializing each layer are reduced, but at the cost of streaming the weights for multiple tiles. This energy cost is amortized by chaining the specialized layers together in a producer consumer fashion. The result of this is a macro-pipeline defined for the targeted network at hand. Many works, such as [26], exploit the reconfigurability of *FPGAs* to make their architecture work with different kernel sizes and other such parameters. However, by utilizing a pipeline methodology, the works of [27, 28, 29, 30] take algorithm specialization to an extreme. Rather than supporting one flexible dataflow through a generalized architecture [6], pipelined architectures define the architecture at the granularity of the entire network to directly support the most efficient dataflow for each layer. Yang et al. [31] purported that power efficiency improvements are primarily caused by the underlying memory hierarchy rather than the data-flow. [4] uses systolic array based design in order to perform acceleration in a sequential manner where different layers are accelerated on the same engine at different time frame. As we will see in the section3.3.2.1 that this has implication of wastage of power and inefficiency in the area. DNN accelerators that utilize algorithmic optimization, such as [32, 3], are able to reduce computation and data movement for DNNs. Specifically, MobileNet [33] utilizes depth-wise separable convolution, an extreme form of group convolution, to reduce the memory requirements and compute intensity of DNNs. Tiny Darknet is the baseline for tinyYolo [34] and utilizes channel squeezing to reduce the memory requirements as well as the compute intensity. The work in [35] does scheduling by controlling the rate at which the algorithm run. It might be not possible in our case as real time edge requirement might not allow us to change rate at which the algorithm is run dynamically. The work presented in [36] describes how the presented work can generate portable memory architecture for in fabric memory access for parallel application.[37] Presents an analytic memory performance model suitable for memory hierarchies that use application managed buffers. The presented work does borrows the ideas on both from and develops an interface as required by the NURO-RAM accelerator. Even though this work is tested for supporting NURO-RAM, it can be extended to to other accelerator with minor or not modification.

CHAPTER 3: Approach

3.1 Motivation for use of off-chip memory

Most of the neural network are big in size, with the model size averaging to 50 MB. With exceptionally large networks, such as VGG-Net [38] or AlexNet, the memory footprint of the network's weights can reach upto 100 MB. This memory footprint becomes problematic when compared to the memory real estate commonly available on *FPGAs*, which is around 30 MB. This limitation is compounded further by the need of streaming accelerators, such as AWARE-DNN, which uses on-chip memory to store intermediate feature data in on-chip memory, further reducing available resources. As is apparent, storing all the weights on chip is not a viable solution for bigger network. However, off-chip memory access is notoriously slow and ill-suited for streaming accelerator use. This necessitates the creation of a supporting system to interface on-chip and off-chip memory while mitigating the latency of off-chip memory access. In addition, AWARE requires parallel access to weights so it can perform pipelined acceleration across multiple layers simultaneously. This complexity of the system is further compounded due to AWARE-DNN needing the weights to be fed to it in a specific format to exploit parallelism in the network. This may or may not be equal to granularity of the off-chip to on-chip data transfer. This calls for the development of the interface circuit which would transfer the weights required by the accelerator in the accelerator format.

On the other hand there are some smaller Convolutional Neural Network that can fit all its weights in the on-chip memory of *FPGA*. But to perform efficient acceleration on these network we need to increase the spatial parallelism of the network. This leads to rise in the number of intermediate feature map buffering which inturn gives rise to BRAM memory resource utilization. This would leave very less real-estate for weight storage on chip. In such cases external memory system would come in handy to store the weights of the layers which cannot be stored on chip, to facilitating more efficient acceleration. To solve these issues discussed above and improve the efficiency of AWARE-DNN, this work presents NURO-RAM, which provides an interface for the accelerator architecture, handles arbitration for off-chip memory access, and acts as a supporting system for AWARE-DNN.

3.1.1 Design formalization for NURO-RAM

As discussed in section2 NURO-RAM exploits three forms of parallelism kernel, channel, and convolutional, special considerations must be made in regards to the re-structure the format in which weights are supplied to the architecture. In addition, as a streaming accelerator, AWARE-DNN requires that a weight element to be available for every network layer with single cycle latency after the request. Thus the constrains required to be satisfied by the NURO-RAM in order to support AWARE accelerator are :

- Each of the pipelined layer using the off-chip memory for weights should have parallel access to weights.
- The weight elements for each of the layer should be in the format required by each layer before they are streamed to the pipelined layer.
- There should not be any memory stalls while streaming the weights to the accelerator, since this would hurt the streaming nature of the accelerator

We define the granularity at which the accelerator requires the weights to be fetched as weight element. The width of the weight element is defined in equation 3.1, where $Channel_{Par}$ is the channel parallelism for the layer and $Convolution_{Par}$ is the convolution parallelism. This is also the width of the architecture-NURO-RAM weight block interface. Figure 3.1 gives the idea how the weights are accumulated to form weight elements. Here the $Channel_{Par}$=2 and $Convolution_{Par}$ =2, thus the size of the each weight elements is 4 bytes.

$$Wghtelem\_width = Channel_{Par} \times Convolution_{Par} \tag{3.1}$$

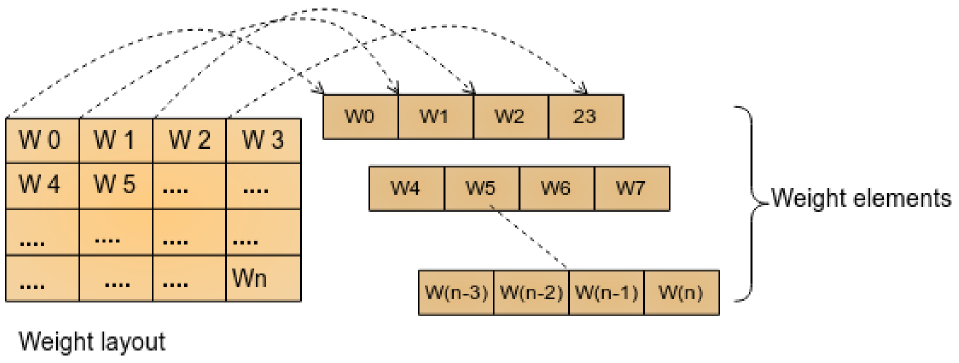

Figure 3.1: Weight elements

In this regards one cannot directly use the off-chip memory component directly as it does not support the multi-ported memory for parallel access and as off-chip memory access is very slow. This calls for a need of usage of multiple memory banks which would feed each of the pipelined layers independently. The Block Random memory access memory (BRAMs) present on the *FPGA* are best suited for this purpose. Since the BRAM memory is fast , can be tailored to form memory banks of variable size according to need of each layer and each of these memory banks can provide parallel access to the weights for the layers. The proposed NURO-RAM architecture uses BRAMs to pre-fetch the weights required by the accelerator. A custom accumulate and interface unit is developed to perform the data conversion for the accelerator from raw weights to the weight elements. These weight elements are then fed to the AWARE-DNN accelerator whenever there is request for the weight element. It should be noted that the layout in which the weights are stored on the off-chip memory needs to be altered so that the layout supports efficient fetching of the weights from the off-chip memory. This is discussed further in section 3.2.2. To arbitrate the off-chip requests from mulitple layers Xilinx Micro-Blaze soft core is being used as arbiter.

Since channel and convolution parallelism for each layer is different, the size of the weight element for each layer is also different. Thus, the rate at which weights must be fetched from the pre-fetch BRAM buffers is be different for each layer. This is called the weight element fetch rate $Wght\_ftchrate$, and the equation is given by equation 3.2, where $kern_{Par}$ is the kernel parallelism for the layer and F_{arc} is the clock frequency of the layer.

$$Wght\_ftchrate = F_{arc} \times fWghtelem \times kern_{Par}. \tag{3.2}$$

Since each layer replicates the CPEs, exploiting kernel parallelism, the weight element fetch rate can be reduced by replicating the required hardware for every CPE instance. Thus the weight element fetch rate is reduced to:

$$Wgftchrate = F_{arc} \times fWghtelem. \tag{3.3}$$

The weight element fetch rate can become much larger than what can reasonably be fetched in a single cycle. This may lead to a decoupling of the architecture and NURO-RAM clocks. With the NURO-RAM running at a higher frequency than the architecture, weights can be fetched and accumulated for use within a single F_{arc} . The methodology to calculate NURO-RAM frequency is discussed in section3.2.5.2.

3.2 NURO-RAM Architecture Overview

As shown in Figure 3.2, the NURO-RAM architecture is categorized into three main subsystems and consists of both hardware and software components. The first subsystem, shown in yellow, is denoted as the off-chip block. This subsystem uses a Xilinx DDR4 MIG Core[39] in conjunction with Xilinx AXI DMAs[40] to handle off-chip memory requests and transfers for weights into on-chip BRAM. This system will discussed in detail in section 3.2.3. The second subsystem, shown in green, is known as the architecture interface and storage block. This block consists of hierarchical memory and control blocks that store weights transferred from off-chip memory into on-chip BRAM based pre-fetch buffers and then accumulate these weights from pre-fetch buffers in form of weight elements into aync-fifo element. This would be further discussed in section 3.2.5and 3.2.4. The third subsystem, shown in orange, is known as the arbiter block. This block consists of a Xilinx MicroBlaze soft core and a Xilinx

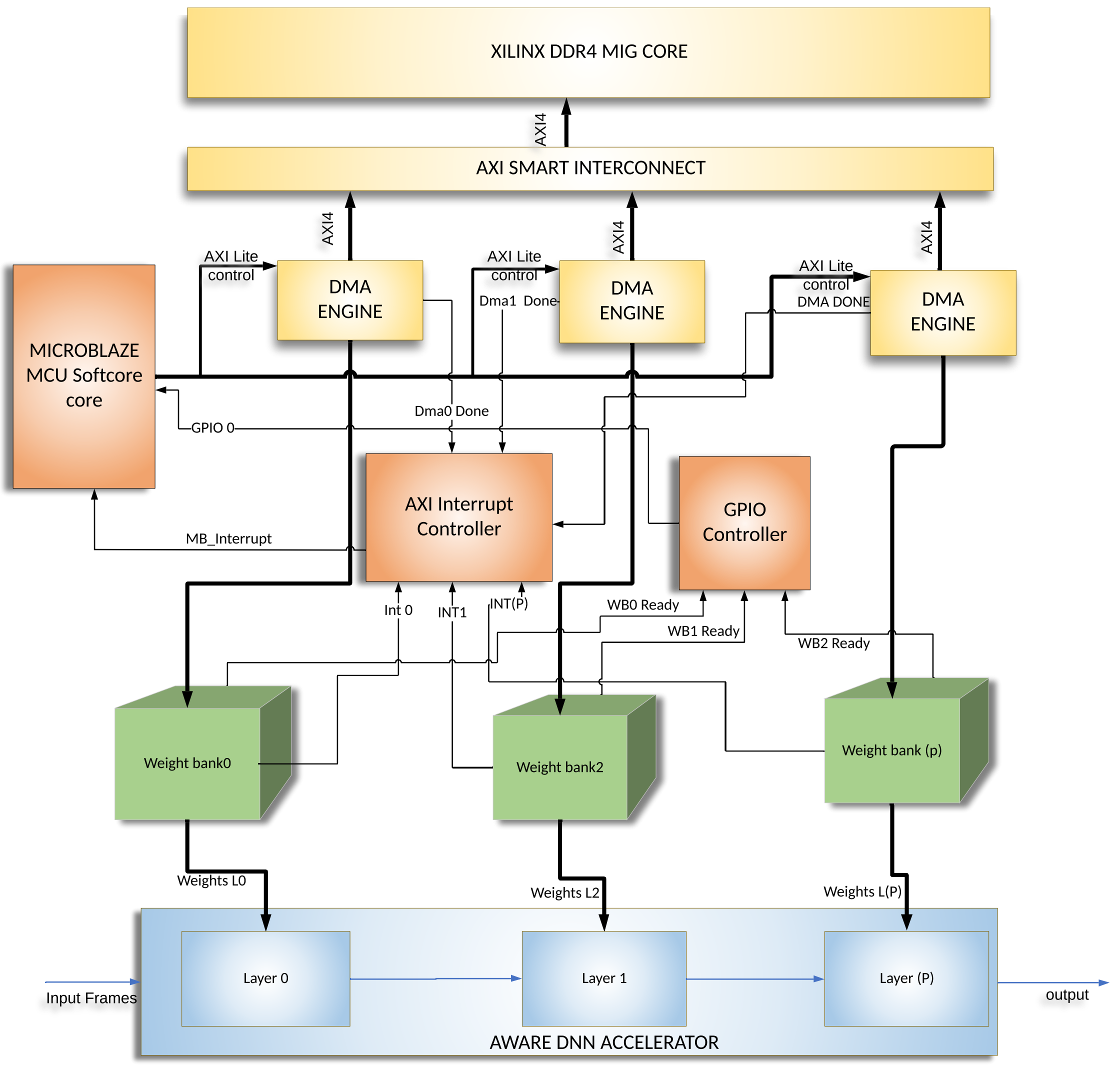

Figure 3.2: Overview of NURO-RAM System

AXI interrupt controller. This subsystem is the arbiter for managing off-chip memory requests from the different layers of the DNN which would be discussed in section3.10.

3.2.1 System architecture goals

As previously discussed, the architecture requires weights in the granularity of the weight element. Since the length of this weight element depends on the channel, convolution parallelism, and network configuration, it is possible that the length of the weight element may exceed the access width of an on-chip BRAM. In such a case, the different sections of the weight element will need to be accumulated before being exposed to the architecture data-path. The necessary ratio between the architecture clock frequency and NURO-RAM weight bank clock frequency needed to accumulate these weights within one architecture clock cycle is derived in equation 3.4.

$$aggr\_cycles = Wghtelem/BRAM\_awdt \tag{3.4}$$

Here $aggr\_cycles$ is the number of cycles required to aggregate the weight element, and $BRAM\_awdt$ is the maximum read width on the BRAM read port. In equation 3.5 FNR_min is the minimum NURO-RAM bank frequency needed to provide a full weight element for every architecture clock cycle.

$$(FNR\_min/aggr\_cycles) >= F \tag{3.5}$$

This equation can be reduced to equation 3.6 for clarity.

$$FNR\_min >= F\_arc * aggr\_cycles \tag{3.6}$$

Thus equation 3.6 is the minimum frequency at which the NURO-RAM bank should be clocked so that it supports acceleration in AWARE-DNN. There are optimization that can be further applied to reduce this frequency which would be discussed in 3.2.5.2.

3.2.2 Off-chip memory layout

For traditional architectures, weights for an entire layer are stored in a single block of memory. This enables the weights to be fetched in one memory block so they can be convolved with input of a single layer. However, since AWARE exploits kernel, channel, and convolution parallelism, this paradigm fails to provide the architecture the weights it requires in a single fetch. As such, the weights in off-chip memory need to be stored in a way that accounts for the kernel, channel, and convolutional parallelism of AWARE. Figure 3.3 show the tiling method adopted to store weights in memory while keeping parallelism constraints in mind.

K0 CH0 CEL0	K0 CH0 CEL1	K0 CH0 CEL2	K0 CH0 CEL3	K0 CH1 CEL0
K0 CH1 CEL1	K0 CH1 CEL2	K0 CH1 CEL3	K1 CH0 CEL0	K1 CH0 CEL1
K1 CH0 CEL2	K1 CH0 CEL3	K1 CH1 CEL0	...	Kn_0 CHn_1 $CELn_2$

Figure 3.3: Example of off-chip memory tiling.

3.2.3 NURO-RAM Micro architecture

Now let us dive into the micro-architectural details of NURO-RAM with respect to design formulation. This sections describes how the NURO-RAM system architecture is build by using configurable custom designed blocks. This section will be divided into for subsections, Weight Bank, Weight Blocks, asynchronous state machine, and asynchronous FIFO which are the basic building blocks of NURO-RAM architecture.

3.2.4 Weight Bank

As seen in Figure 3.4, a weight bank is made up of smaller elements called weight blocks, which will be discussed in section 3.2.5. The weight bank is responsible for transferring an interrupt request to the Micro-Blaze when new weights are needed to be loaded onto on-chip pre-fetch buffers. This interrupt request is a reduction of all the interrupt requests from each weight block in the weight bank. Each of the BRAMs in the weight block in the weight bank are connected in a cascade mode. Thus the DMA master who writes to the on-chip memory would see the entire address range of weight bank as a whole rather than individual weight blocks. Memory. This is done to reduce the number of write ports required, lowering the hardware overhead

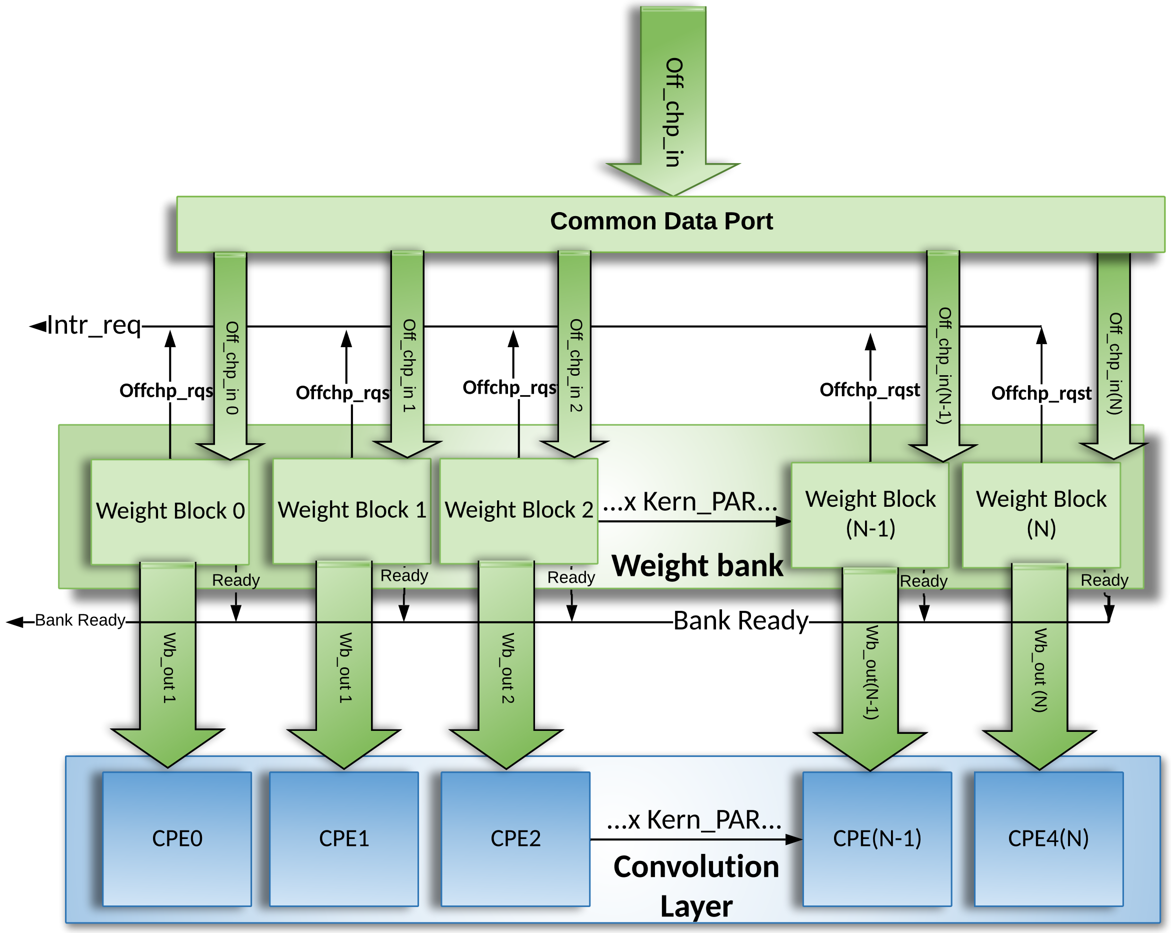

Figure 3.4: Weight Bank

for each DMA. This block is also responsible to transfer the weight Bank ready signal generated by weight blocks to Micro-Blaze arbiter. This signal is raised when the initial weight transfer is completed by the firmware thus indicate that the NURO-RAM is ready to source the weights to the accelerator.

3.2.5 Weight Blocks

The weight block is the micro-architectural interface between NURO-RAM and the architecture. A weight block is responsible for

- generation of interrupt for off-chip transfer request.
- aggregation of partial weights to create an entire weight element.

- buffering the weights elements into asynchronous FIFO
- clock domain crossing

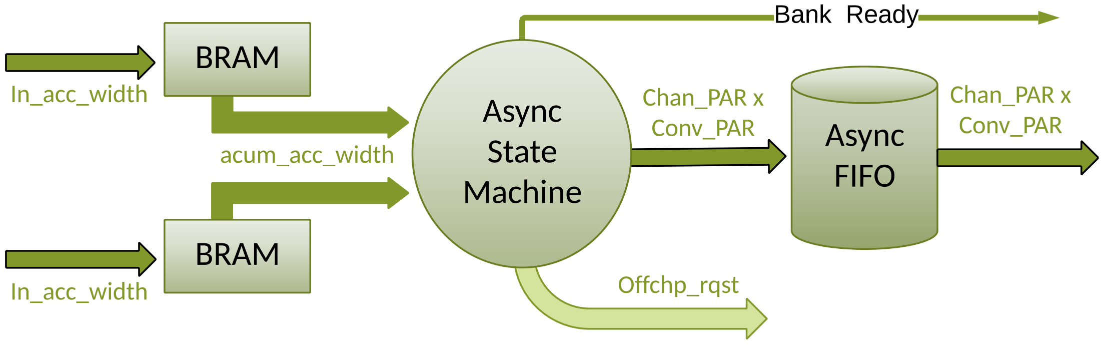

Figure 3.5: Weight Block

As shown in Figure 3.5, weight blocks consist of two identical blocks of BRAM used as as pre-fetch buffers, a weight block asynchronous state machine, and an asynchronous FIFO buffer. The two blocks of pre-fetch buffers alternate between active and transfer roles in ping pong manner. In the active role, the buffer supplies weights to the asynchronous state machine for accumulation of weight element and then transfer into the asynchronous-FIFO. In the transfer role, the buffer is written with the next batch of the weights required by the accelerator by the DMA controller via BRAM controller. As the design is already aware of the memory layout the weights elements requested by the AWARE-DNN accelerator layer is always in the sync with the off-chip memory layers. The asynchronous machine requests the off-chip memory transfer when the active BRAM has emptied itself to half of its maximum depth, giving it sufficient of time to accumulate weights before the active becomes empty. Once the active BRAM becomes empty, the two buffers switch roles. This is done to eliminate the impact of off-chip memory access latency on the architecture. This is further discussed in section 3.10

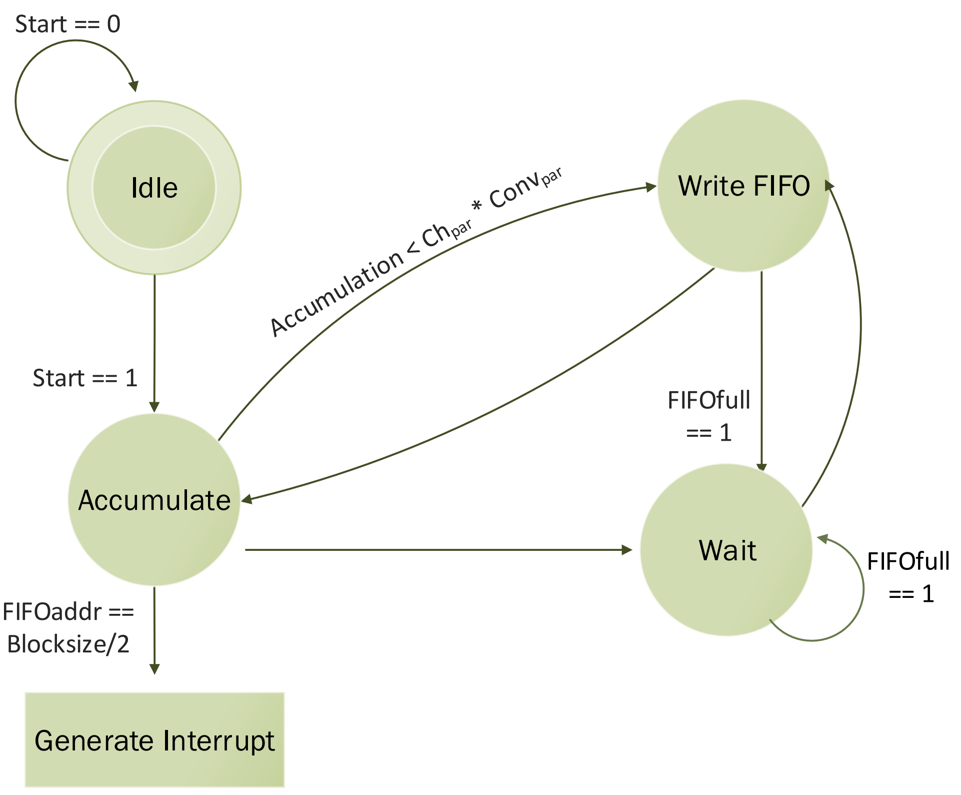

Figure 3.6: Asynchronous State Machine

3.2.5.1 Asynchronous state machine

The asynchronous state machine acts as the control unit of the weight block. It is responsible for fetching weights from the active BRAM and accumulating them in the asynchronous FIFO buffer. That is when accumulation is equal to $Ch_{par} \times Conv_{par}$. A flow diagram demonstrating how this control logic is handled can be seen in Figure 3.6. This state machine control actually generates the interrupt for the weight block. The interrupt request is raised when the active address from which the state machine is reading becomes equal to half of the size of the depth of the pre-fetch buffers. This block also generates the layer Bank ready signal once the firmware transfers initial weights to the weights banks and the asynchronous fifoS are filled. As the system design is tightly coupled with fifo depth and read write frequency of asynchronous

FIFO this signal remains high for entire period of acceleration. The state machine is also responsible for switching the roles of the BRAM, as discussed in section 3.2.5.

3.2.5.2 Asynchronous FIFO

The asynchronous FIFO buffer handles the exchange between the decoupled clock frequencies of the NURO-RAM and the architecture. It also buffers the weight elements, enabling the NURO-RAM to provide the architecture with entire weights elements at every clock cycle. The size of the FIFO buffer for each layer can be calculated using equation 3.7 and equation 3.9, where f_ratio is the ratio between accelerator architecture and NURO-RAM frequencies, and $FIFOdepth$ is the size of the FIFO buffer.

$$f\_ratio = f\_arc/FNR\_min; \tag{3.7}$$

$$FIFO\_depth = (f\_ratio * aggregation\_cycles) + 1 \tag{3.8}$$

$$FIFO\_depth = (f\_ratio * (Wghtelem/BRAM\_awdt)) + 1 \tag{3.9}$$

As we can see, there is a system architectural knob in $FIFO\_depth$. Lowering $FIFO\_depth$ reduces memory utilization and increases frequency. Conversely, increasing $FIFO\_depth$ will increase memory utilization while lowering frequency, which can be used to lower power consumption. It should be noted that we can change the size for the fifo buffers in-order to restrict the frequency at which all the weight banks in NURO-RAM work. This can be used to avoid creation of multiple clock island and thus avoiding the violation of clock domain. We can increase the size of BRAM access width and lower the aggregation cycle, if the $FPGA$ has sufficient resources, by lowering the aggregation cycles we can run the accelerator and the NURO-RAM interface at the same clock speed, thereby saving the power. Another point to note here is that the parameters for $aggregation_{cycles}$ are actually dictated by the parallelism defined by AWARE architecture compiler knobs like channel and convolution parallelism. These are inturn dependent on the application requirement and frames per second used. Thus we can see from here that the NURO-RAM architecture is loosely coupled with the application requirements.

3.3 System Design Components

To built the system for the accelerator we have used pre-built IP provided by Xilinx. The important IPS are discussed in the sections ahead.

3.3.1 Xilinix IPs used in the design

3.3.1.1 MIG core

Xilinx MIG (Memory interface generator) core [39] is to interface the Xilinx based *FPGAs/SOCs* to the external DDR4 component. These core can support full memory controller or PHY (physical layer only) support for the system. With the 'physical only support', consuming lower amount of resources. The proposed design uses entire memory controller support. The block diagram of the Xilinx DDR 4 MIG core is shown below: The entire controller can be divided into three blocks

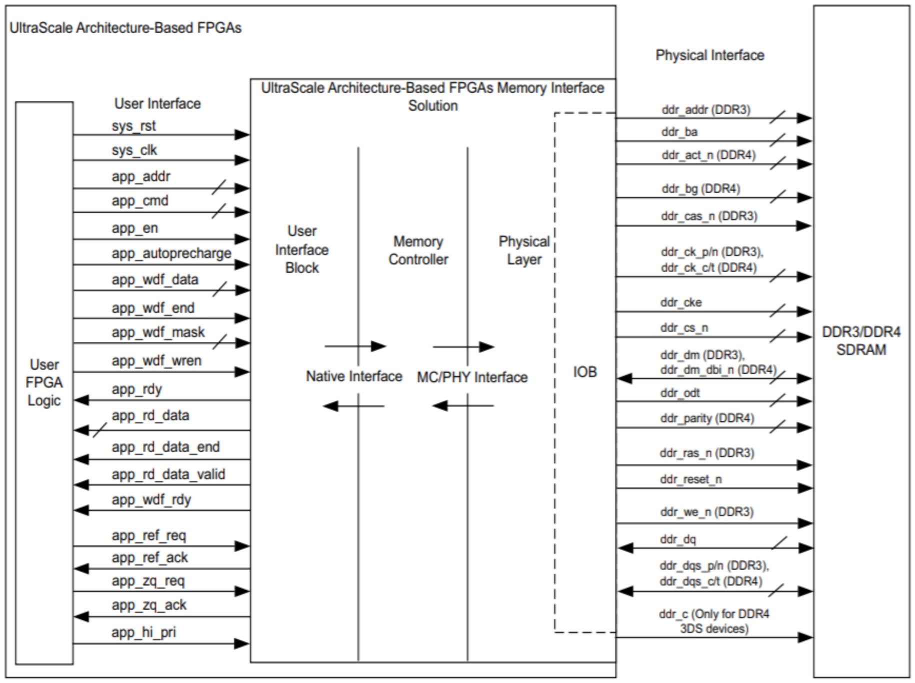

Figure 3.7: Memory interface generator

- User/Application interface: The block provides a fifo interface to the application for reading and writing to the DDR4. Data is buffered to present the read data in requested order.
- Controller block : This block will handle the burst transactions to and from the user interface block to the SDARAM. It will also perform read write transaction coalescing , handle the refresh and reorder the command to the physical layer to improve the utilization of the data bus to the SDRAM.
- physical interface:This block would provide high speed interface to the SDRAM component. These block include the SERDES block, High speed clock generation block memory initialization block, calibration block etc. The system design uses an AXI based version of the MIG core. In this version the user interface block described above is connected to AXI4 bus. Thus each of the transaction sent to the DDR4 are AXI packets. Due to this multiple masters connecting to the DDR4 can be easily handled by the AXI based smart interconnect systems. Presented work use component present on the ultrascale board has interface speed of 1200MHz. The PHY to clock frequency ratio supported is 4:1, the MIG core runs on 300MHZ in current configuration.

3.3.1.2 CDMA Engine

This work have used AXI CDMA (central direct memory access) IPs [40] to transfer data to and from memory mapped slaves. Hardware built for this IP supports variable data width from 32 upto 1024. Burst size for transfers are supported from 2 to 256. AXI CDMA IP supports an additional AXI lite interface to dynamically configure setting of the DMA by master processor. The IP supports hardware built of single cycle transfer, burst transfer modes and scatter gather mode, out of which the rpoposed design have used to burst transfer mode. On the software side it supports the polling mode and interrupt mode, where the interrupt mode in order to free-up the processor.

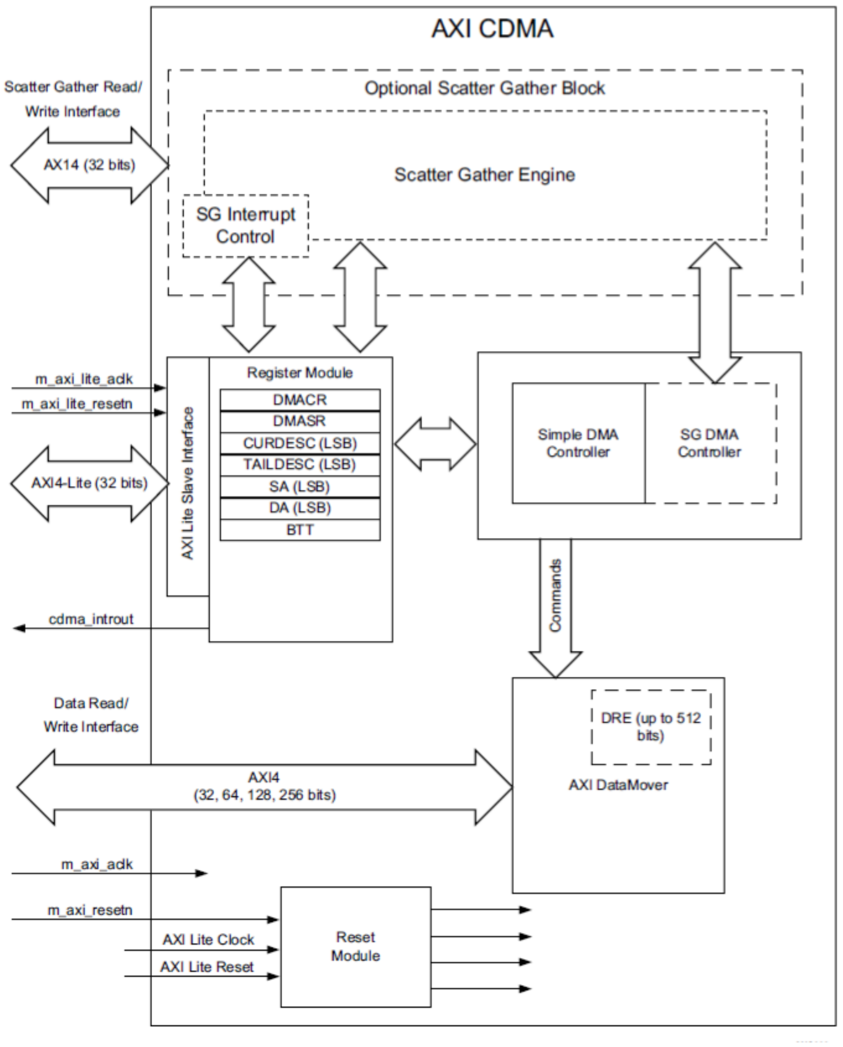

Figure 3.8: AXI Central Direct Memory Access

3.3.1.3 Interrupt controller

Xilinx LogiCORE IP INTC core INTCreceives interrupt from the multiple sources and generates one interrupt to the system processor. In case of NURO-RAM this is the microblaze processor. The registers used on the software stack to enable makes and acknowledge an interrupt are accessed via an axi lite interface. The block diagram of this IP is shown in the Figure below: The major blocks of the IP are:

- Register:This blocks consist the control and status registers all of these are accessed via and AXI lite the slave interface.

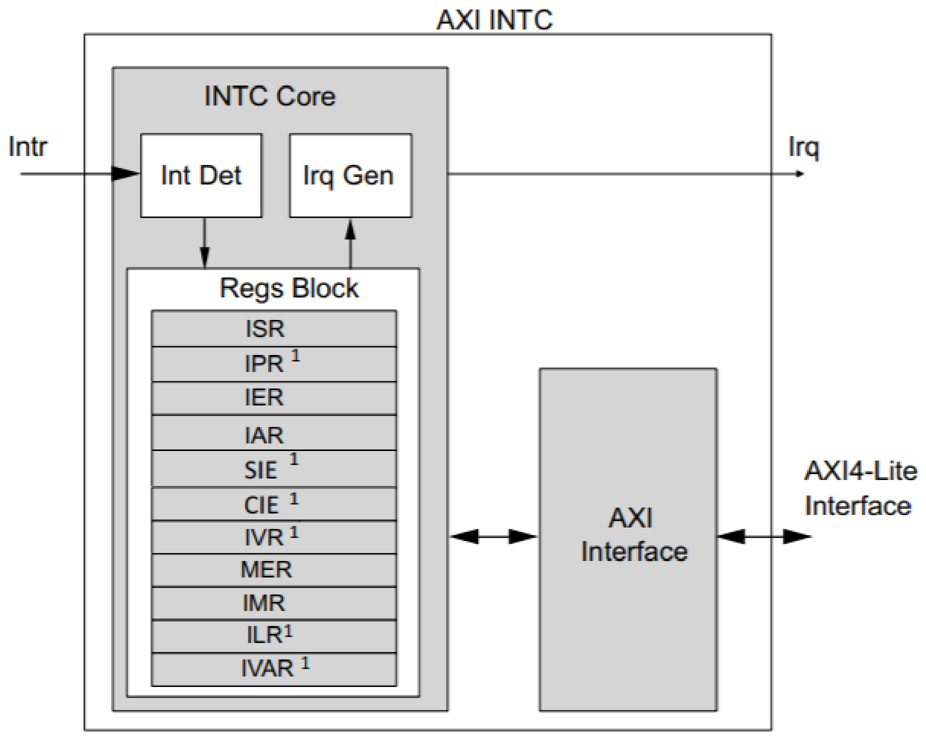

Figure 3.9: AXI interrupt controller

- Interrupt detection:This block will detect the interrupt on the interrupt line depending upon the configuration, falling edge or rising edge.
- Interrupt generation:This block generates the final output interrupt depending upon the configuration parameters like enable conditions. This block also handles resetting the interrupt after acknowledgement.

3.3.1.4 Micro blaze core

The microblaze processors is 32 bit a highly configurable embedded soft processor core from Xilinx [41]. The detailed explanation of all the configuration of the core is out of the scope of theses thesis. The main point to know abou the core are:

- 32 Bit RISC Harvard architecture.
- supports the AXI interface, current configuration uses the AXI interface.
- Big/Little endian support, design configuration uses the little endian mode.
- Instruction side cache and data side cache support.

3.3.2 Arbitration and Scheduling

The Micro-Blaze soft core is used as the arbiter module, handling off-chip memory transfer requests for all layers. The requests are received in the form of the interrupts mentioned in section 3.2.3. The Micro-Blaze stores all information about the individual layers in a data structure called Network Topology. This data structure holds array of another data structure called layer info. The MicroBlaze uses a set of centralized DMA engines and and interrupt engine to handle the transfers. These are configured by the Mirco-Blaze during initialization. The following section gives more detailed view of how presented work performs scheduling of weights. In subsequent section we will dive deeper into the firmware design.

3.3.2.1 Scheduling the off chip weights

CNN algorithms are highly predictable with respect to the weight access as each process in the layer is repetitive for each frame passed to the layer. This is highly exploited in the AWARE architecture. By modifying the weight layout in external memory as discussed in section 3.3, we can pre-fetch the weights in on-chip BRAMs. As previously discussed the weight blocks are the basic building block of the NURO-RAM. Each layer needing the support of off-chip memory is provided with single weight bank. The system uses a single off chip memory element and a single memory controller to feed the on-chip BRAMs present on *FPGA*. This causes contention for the memory controller and the off-chip memory which calls for a need of efficient scheduling algorithm which will schedule the off-chip -on-chip memory transfers. In terms of scheduling nomenclature it can consider that the weight transfer process is an IO bound process as it uses off-chip DDR4 memory. Scheduling algorithms are based on theirs metrics such as fairness, turnaround time and response. We can relate these metrics to the NURO-RAM metrics as follows:

- Fairness: Since we are accelerating multiple layers, we need the algorithm to serve all the layers when it asks for the service. There should no starvation in any case.
- Response time: As AWARE is streaming accelerator the turn around time for each service should be minimum, else the layer would miss the frame and might produce faulty results due to wrong weights. Here service is an off-chip on chip weight transfer initialization.
- Turn around time: The time taken to weight transfer should be minimum, as other layers would also request for weight transfer due to the streaming nature of the accelerator.

As it is a known fact that its very difficult to perform best in all of the metrics as stated above by a single scheduling algorithm. Thus we use a static scheduling system which uses the knowledge of the network architecture and takes help from the hardware design so that system arbiter is not overwhelmed with off-chip transfer requests. Since turn around time for the system would be time taken to do a off-chip memory transfer which depends upon the size of the transfer performed by the DMA, as other factors such as the DRAM latency would be constant in this case. This is related to DMA transfer latency which is a function of system frequency. Since we are working towards reducing the power of taken entire system, there is a very little margin to reduce this latency. NURO-RAM system uses the interrupts for requesting the off-chip transfer service via DMA controllers, thus we can consider the interrupt status register as the request que for the static scheduler. Figure 3.3.2.1 gives an idea how the scheduling happens over time. Consider three weight banks for three layers. As the time passes each of the weights from the on-chip BRAM (which was previously filled by a external memory transfer) are read by the asynchronous state machine for further accumulation and transfer into the asynchronous fifo. When this process reaches half of the depth of the depth of the weight block's pre-fetch buffer size, the weight bank would generate interrupt request for the off-chip memory transfer and

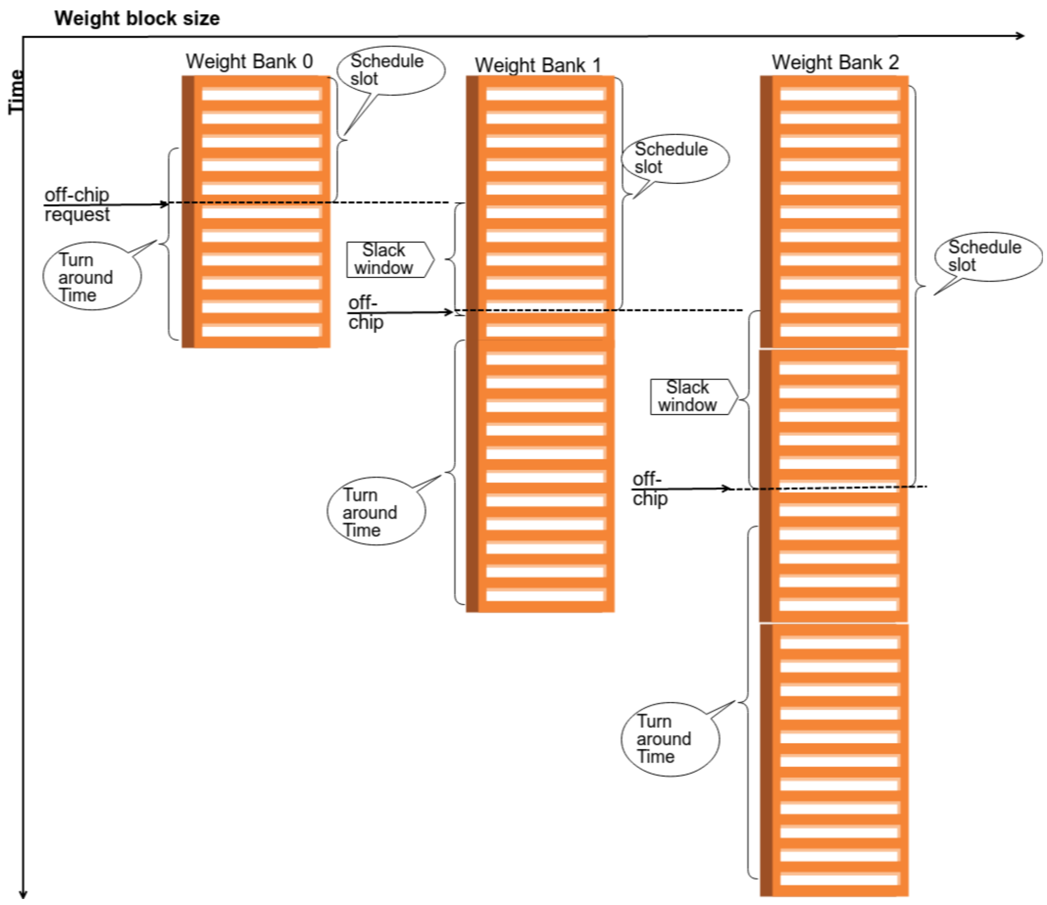

Figure 3.10: Static Scheduling using variable size pre-fetch buffers

then proceeds with further accumulation. This time from the start of weight block pre-fetch buffer address till the time it reaches half-way of BRAM depth is the time where requests for other layers can occur is called schedule slot. After the weight bank generates the interrupt request it moves on to reading the next available address till the time it reaches the depth of the pre-fetch buffers and performs the buffer switch. This time is called turn around time for the layer. Before the end of this time the DMA engine should have filled at least the initial addresses of the pre-fetch buffer. It is clear from the above discussion that if we size the pre-fetch buffers we can increase the response time window and can mitigate already constrained turn around time.

timing diagram.png

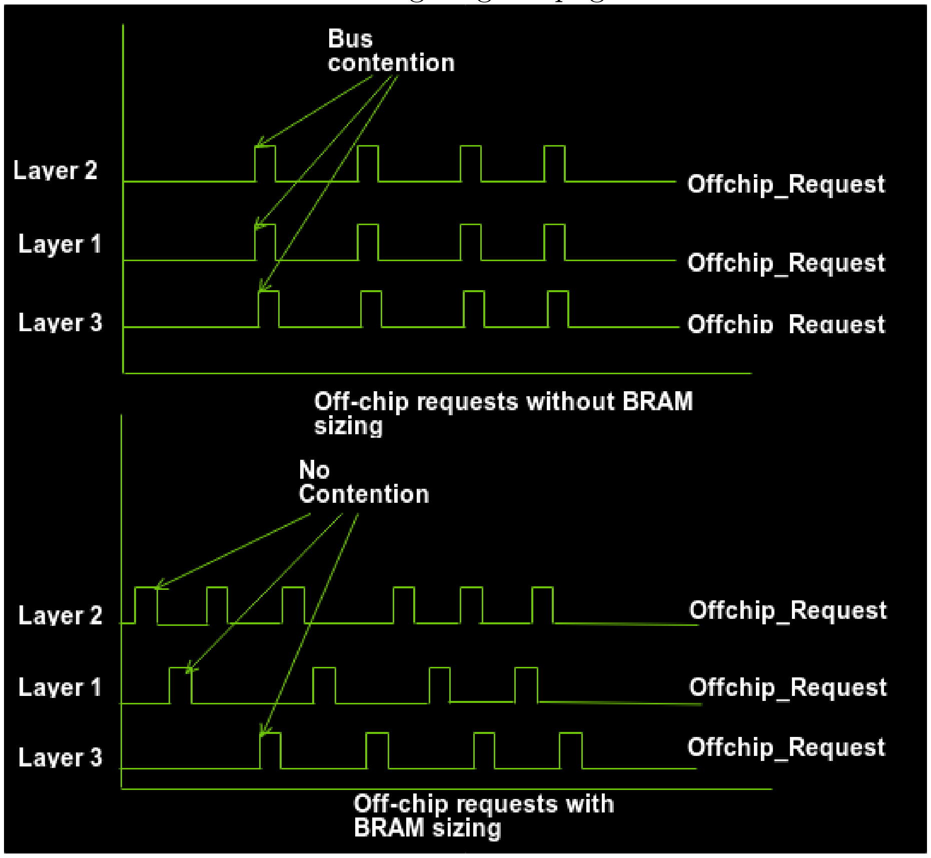

Figure 3.11: Timing diagram for Scheduling

Sizing BRAMs for pre-fetch buffers

We will now move our attention as to how we can size the pre-fetch buffers. Here we consider the unit block RAM size as 4KB because the targeted Ultrascale *FPGAs* consist of RAMB36 and RAMB18 as the unit block RAM element and the size of this element is 4KB and 2KB respectively. Now since each layer has its own channel and kernel parallelism, denoted by Ch_{par} and $Convolution_{par}$ the number of weight element present in unit block of memory is given by

$$Num\_Wghtelem = \frac{4096}{Channel_{par} \times Convolution_{par}} \tag{3.10}$$

If the transfer latency for the DMA engine is given by $DMA\_latency$. The rate at which the weights are fetched are given by $Layer\_reqfr$. For successful DMA transfer for the same layer the requests should comes after the DMA engine has finished previous transfer, essentially $DMA\_latency$. Thus the minimum number of the weight elements that should be present in the pre-fetch buffer is given by $Weight\_min$ given by equation3.11

$$Weight\_min = \frac{DMA\_latency}{Layer\_reqfr} \tag{3.11}$$

Thus by increasing the weight more than $Weight\_min$ we can vary the time the weight block takes to generate an interrupt. Equation3.10 allows us to find the number of weights present in the unit memory element. Thus if we characterize the $DMA\_latency$ for the unit memory transfer (4KB) we can accurately find what is the size of the pre-fetch buffer to store $Weight\_min$. Using this we can size the pre-fetch buffer more accurately. If two buffer sizes turn out two be same then we use a simple multiplication factor for the layer that has lower kernel parallelism in-order not to stagger the request in time between two buffers. If we relate the increasing of number of weights in weight block to scheduling terminology we could imply that it would increase time in the schedule slot thus easing the task of scheduling. Secondly it also provides high turn around time. The timing diagram 3.11 shows the off-chip request with fixed sized pref-etch buffers vs the variable size buffers. Let us now move our focus to the fairness. We can see from above discussion that if we size our BRAM such that each weight bank has differently sized pre-fetch buffers the interrupt request for each layer would occur separately for most of the time. There might be cases where two layers can have weight block sizes are multiples of each other and the request are generated at the same time. But this will be taken care by the interrupt controller module as the un-serviced interrupt would be still be served as the interrupt handler registers its status even when the processor is in process of initiating other transfers, and we are assured to have fairness to each of the layers. The following Figure explains the differences between using a constant size BRAMs in weight blocks vs variable size BRAMs.

Coupling optimization

Note that there can be configuration in smaller CNN algorithms where the Ch_{par} and $Convolution_{par}$ are very low due the size of the filters they use or due to the inherent nature of algorithm. Since the lowest size of the pre-fetch buffer used in the design is 4KB, this causes be wastage of the on-chip memory. Consider the example of layer that has $Kernel_{par}$ of 64 $Channel_{par}$of 1 and $Convolution_{par}$ of 1. One such design point can be found in Shallow mobile net. If we size the pre-fetch buffer of this layer to be 1KB then rest of 3KB is a wastage of memory real estate. This problem is aggrandized by $Kernel_{par}$ of 64. Thus total 192KB of memory can get wasted. For this kind of layers we used an optimization called coupling where the $Kernel_{par}$ is broken into smaller value and these values are coupled into channel parallelism or convolution parallelism. Due to this the size of the Weight_element increases and will increase the aggregation cycles. But the rise in the aggregation cycles can be neutralized by increasing the BRAM access width. Since we have reduced the $kernel_{par}$ the memory wastage is reduce. This change is easily supported by the accelerator interface as due to the flexibility of wiring in the interface

3.4 Software Design

The firmware for scheduling the off-chip , on chip transfers is completely handled by the Micro-Blaze processor [41]. The firmware performs system initialization (discussed more section 3.13) after which it spends most of its time in servicing the interrupt and housekeeping activities. The housekeeping activities involve updating the addresses for next transfer and updating the transfer statuses for each layer. There are two different interrupt sources for this system. First the off-chip transfer request from the layers them-self. Second the DMA transfer completion interrupt from the DMA engines. We have used interrupt based DMA transfers as using easier methods like polling will hurt the system's strict latency requirement.

3.4.1 Firmware

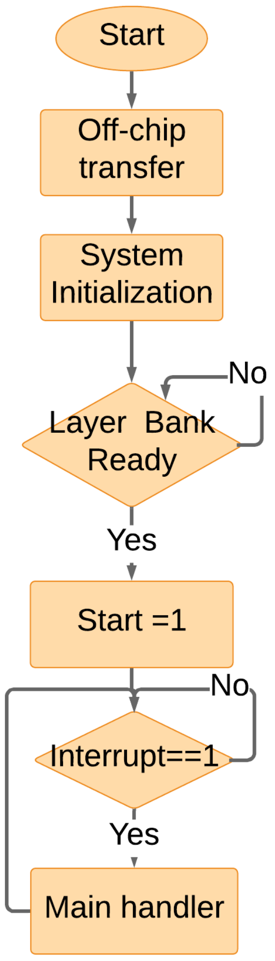

Figure 3.12: Scheduler Firmware

Figure 3.12 shows high level view of software events used for arbitration of off-chip memory requests. Each layer is assigned a data structure as Layer info. Each of the Layer info instance consist of information about the layer as given in table 3.1. During the system initialization the weights initial batch of weights are first transferred to the off-chip memory. Once this initial transfer is completed the system performs initialization of the other peripheral engines.

Once the entire initialization is complete the system would wait for the layer Bank ready signal to high.After the Bank ready signal goes high, at this point the system would start the AWARE-DNN accelerator. From here the architecture would receive

Table 3.1: Layer info

Name	Usage
CDMA	The instance of the DMA Enigne the layer connected to
interrupt Handler	The interrupt handler for the layer
Status	Status of the DMA transfer
Base off-chip	Base address in DRAM from which layer weights start
High off-chip	High address in DRAM from which layer weights start
Active off-chip	Active address which will be used for next transfer
Base on-chip	Base address of BRAM controller for on-chip
High on-chip	High address of BRAM controller for on chip
Active on-chip	Active address for the next transfer

a weight element for every layer on every architecture cycle, and the firmware loop is repeated until acceleration is completed.

3.4.2 System Initialization

Figure 3.13 gives us a brief idea of the initialization performed by the system at the start. The numbers given at the connection in Figure indicate the step at which the particular initialization is performed. Initial steps include the GPIO initialization and the interrupt controller initialization. Since the system consist of single interrupt controller in the design all the layers share the same interrupt controller driver, this controller driver is initialized by $Init\_int\_cont()$ function. The interrupt controller has first level of interrupt handler which will be called when any of the interrupt is generated to this controller. This handler would inturn call the second level handler. This is determined by which second level interrupt handler is connected to the source peripheral by using $INTC\_Connect$ function. Interrupt controller initialization is followed by the layer initialization function. Each layer would have two interrupt sources one from the layer itself (for the off-chip memory transfer) and second from the DMA engine, to indicate the status of the transfer. The $Layer\_Init()$ function initializes both of these sources. The first interrupt source (off-chip transfer) is initialized and connected to the interrupt controller by the function $Layer\_INTR\_Init()$

where it calls the function $INTC\_Connect()$ to connect the appropriate second level interrupt handler when off-chip request is generated. The second level interrupt handler, which in this case is $Init\_DMA\_Transfer$ which would perform initialization of DMA transfer and also connect a callback function that would be called once the transfer is completed. Each of the layers have their own DMA engine and

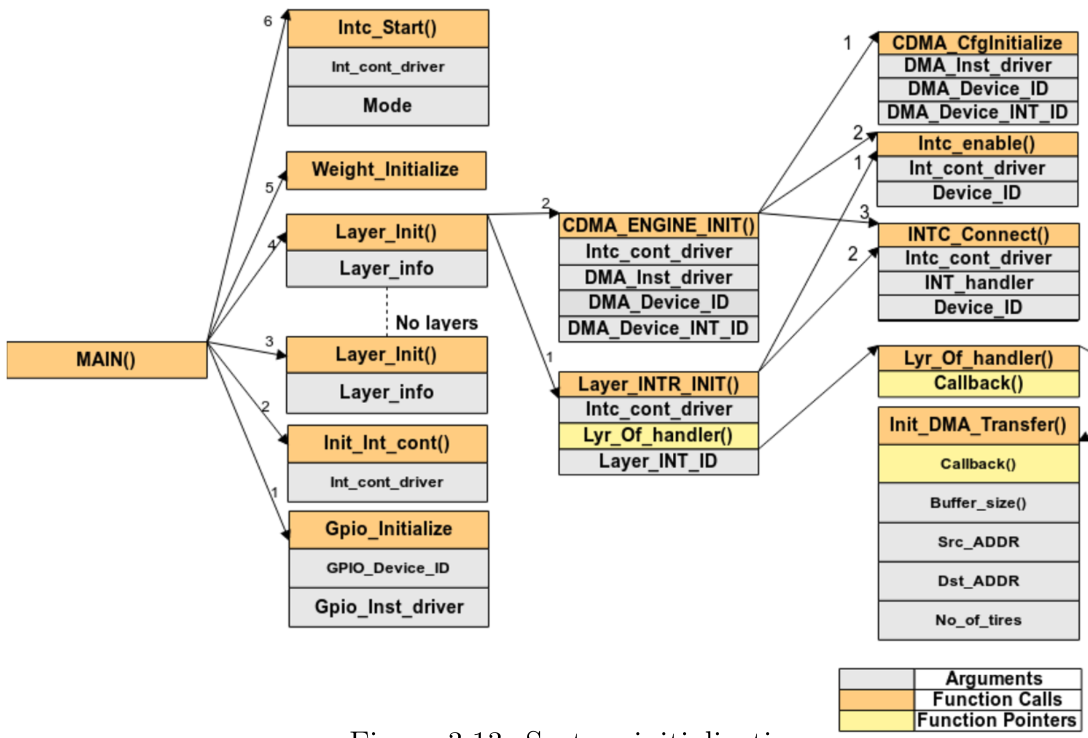

Figure 3.13: System initialization

thus the structure layer info as in table 3.1 would consist of separate driver for the DMA engine it is connected to. The driver for the DMA handler is initialized by $CDMA\_ENGINE\_INIT()$ function. The this function is is provided with the other arguments so that the driver instance can be connected with the interrupt controller using $INTC_{c}onnect()$ function. This same function$INTC_connect()$ is also used to connect the driver instance with the appropriate second level interrupt handlers. Function $CDMA\_ENGINE\_INIT()$ is will also enable the interrupts for the given CDMA engine by calling the function $Intc\_start()$. The function calls to $INTC\_connect()$ and $Intc\_start()$ are common between $Layer\_INTR\_Init()$ and

$CDMA\_ENGINE\_INIT()$ initialization flow. After the layer initialization completes the function $Intc\_enable$ function is used to enable the peripheral interrupt source so that it can generate interrupt on the interrupt controller. In the end we perform initial weight transfers to on-chip memory and then wait for the layer Bank ready signal.

3.4.3 Handling the Interrupts

As previously discussed the firmware spends its majority of time in servicing the interrupt requests. Let us now dive deeper as in how these request are handled by the firmware. The interrupt controller has numerous layers of interrupt handlers which are called on by one when the interrupts are fired. The process is shown in Figure 3.14. Let us first discuss about the off chip transfer interrupt. As previously discussed

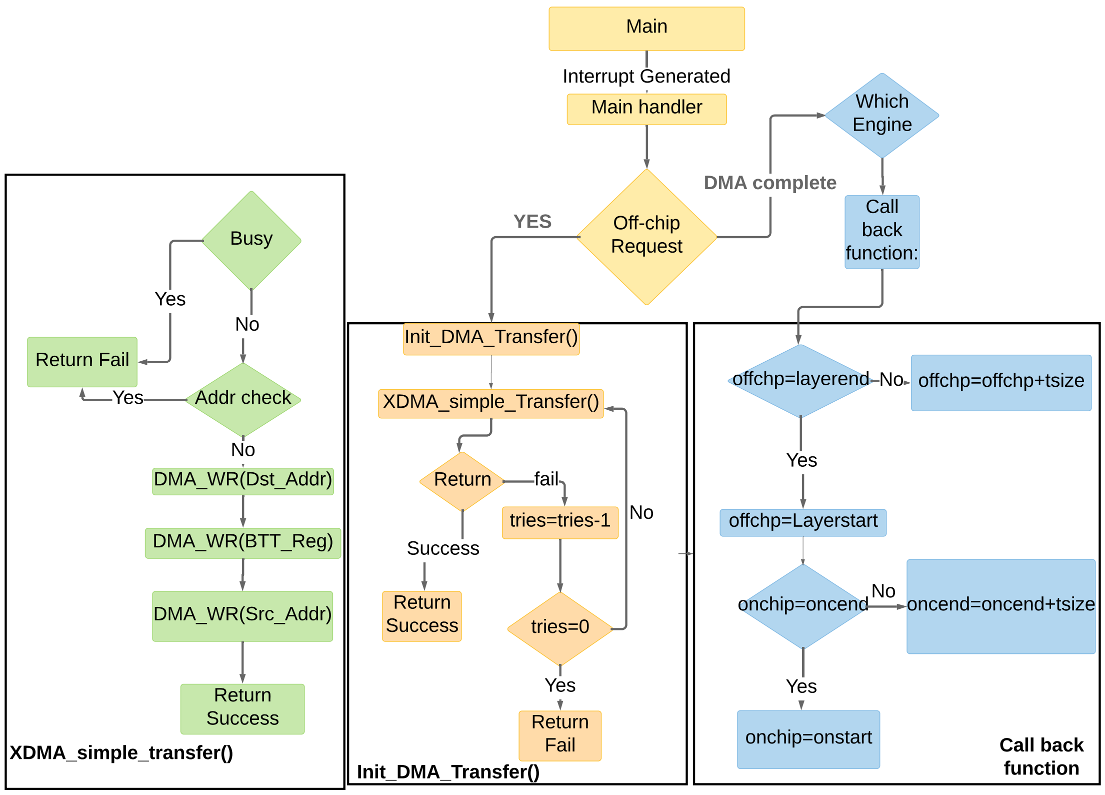

Figure 3.14: Interrupt handler flow

in section 3.4.2 the first level interrupt handler (Main handler) is called every time an interrupt is fired at the interrupt controller. This handler would check the peripheral source of the interrupt generated in the interrupt controller and would call appropriate second level interrupt handler. If the source is the the layer off-chip request then the handler would call $Init\_DMA\_Transfer()$ function to initiate a DMA transfer. This function would then call the previous $XDMA\_simple\_transfer()$ function to perform the actual transfer. This function would also connect the call back function to the transfer. If the transfer initialization fails the $Init\_DMA\_Transfer()$ would try to perform the transfer again. Since the off-chip transfers are scheduled using the scheduling policy discussed in section 3.10 $XDMA_Simple_transfer()$ barely fails. If the source of the interrupt is DMA complete interrupt from one of the DMA engine then the handler would call the DMA handler function to perform the house keeping activities related to the DMA engine. For the house keeping it would then call the Call back function which was connect with the transfer when the $XDMA\_simple\_transfer()$ function was called. The call back function would then update the the addresses for the on-chip memory as well as the off-chip memory.

CHAPTER 4: Experimental methodology and Evaluation

4.1 Experimental methodology

For evaluation AWARE-DNN / NURO -RAM system we have used Xilinx ultrascale *FPGA* (ZCU 102 board). The resources availability for this chip is given in the table 4.1.

Table 4.1: Ultra-scale Device Resources

Resource	Availability
CLB LUTS	27408
CLB Registers	548160
BRAM	34 Mb
DSPs	2520

For the development of the NURO-RAM hardware IP we used Vivado 2018.2 suite and was based off verilog HDL. Model-sim simulator was used for IP verification for both functional and timing analysis. For hardware verification, post-implementation net-list was used for functional and timing analysis. For the firmware, development was carried out using Xilinx SDK 2018.2 [42] and the drivers for the firmware was developed using the C. For verification of system level implementation we have implemented the system on Xilinx's Ultra-scale *FPGA* board and ILA cores were used in order to verify the communication of Microblaze -NURO-RAM interface. A debug UART port was also implemented in the system for further verification on the software side. The power and utilization numbers reported in this work are reported are gathered from post implementation results from Vivado synthesis and implementation tool. It should be noted that the power analysis report generated by Vivado considers 50% activity on all the lines. Thus the power results presented are in the ballpark range. Secondly Vivado tool is unable to capture the off-chip power used by the MIG core, and thus special considerations have to made to report the *SOC* power as whole. Thus to get the *SOC* power to greater accuracy this work incorporates Xilinx power estimator tool to generate off-chip power result. Then the results from the Xilinx power estimator and Vivado are added together to report the total power. Table4.2 configurations we used to generate the power report from Vivado implementation.

Table 4.2: Power report settings

Constraint	Value
Junction Temp.	25 degree
Ambient Temp.	25 degree
Airflow	250 LFM
Borad layers	12 to 15

For creating the layers each of the chisel generated layers files we integrated with the weight block memory module developed in verilog in Vivado and IP were generated for each of these layers. The DMA transfer width and burst size effects does change the timing of system and indirectly affects the BRAM utilization. This is because the Xilinx RAMB 36 is 32 bit wide BRAM element and to support higher sizes the BRAM elements are cascade together because of which there is some wastage of memory resources. The current DMA transfer width is 32 bits and burst size is 8. The accelerator frequency is kept at 50 Mhz for all the design points reported and thus clock frequency at which NURO-RAM weight banks are run is regularised by setting the fifo depth to 8. Thus weight banks in entire NURO-RAM uses 250 MHz. This was done to avoid the timing issues which might rise due to multiple clock island. All the sizing of the pre-fetch buffers are made according to the configuration chosen above and calculations from3.3.2.1.

Further more to test the NURO-RAM we evaluate three separate CNN algorithms AlexNet, Tiny Darknet and Shallow mobile net [33]. We choose to implement this network showcase the flexibility of the NURO-AWARE system to bigger network. Second network this work implements is Shallow mobile net. This network is very compute efficient because of which there is lot of pressure on memory sub-system. The third network this work implements is the Tiny Darknet which is memory efficient network but it is not compute efficient and was chosen to evaluate the flexibility of the system towards diverse set of networks. Each of these network algorithms are run at 30 FPS as this suits well for the real time requirement. To compare the our results this work have implemented the Xilinx Chai DNN for Tiny Darknet using 1024 DSP configuration with Xilinx quantization support for inference. The report for the AlexNet for FPS were taken from the Xilinx performance reports The reports for power for Chai DNN for tiny Darknet and AlexNet are obtained by running hardware from DSA file into Vivado tool synthesis and implementation. For *GPU* result we ran AlexNet using the py-torch model on NAVDIA Xavier *GPU*.

4.2 Evaluation and Results

There are two separate evaluations which are to be considered while evaluating NURO-RAM. First is to compare the AWARE-DNN and NURO-RAM (NURO-AWARE)as a system to previous *FPGA/*, *GPU* based implementations for accelerating CNN on edge. In the second section we discuss about NURO-RAM's ability to augment AWARE-DNN's ability support larger network and to improve its performance. The pipeline latency and the number of layers for all the all three network at 30fps is shown in Figure in 4.1. Figure 4.2 shows implementation of three separate algorithms using NURO-AWARE system.

4.2.1 Implementation of neural networks

In this section we compare implementation of three CNN networks on NURO-AWARE system. NURO-AWARE system consist of the AWARE-DNN hardware accelerator in conjunction with NURO-RAM system. This section also compares the presented work with *FPGA* based as well as *GPU* based Edge CNN accelerator solutions.

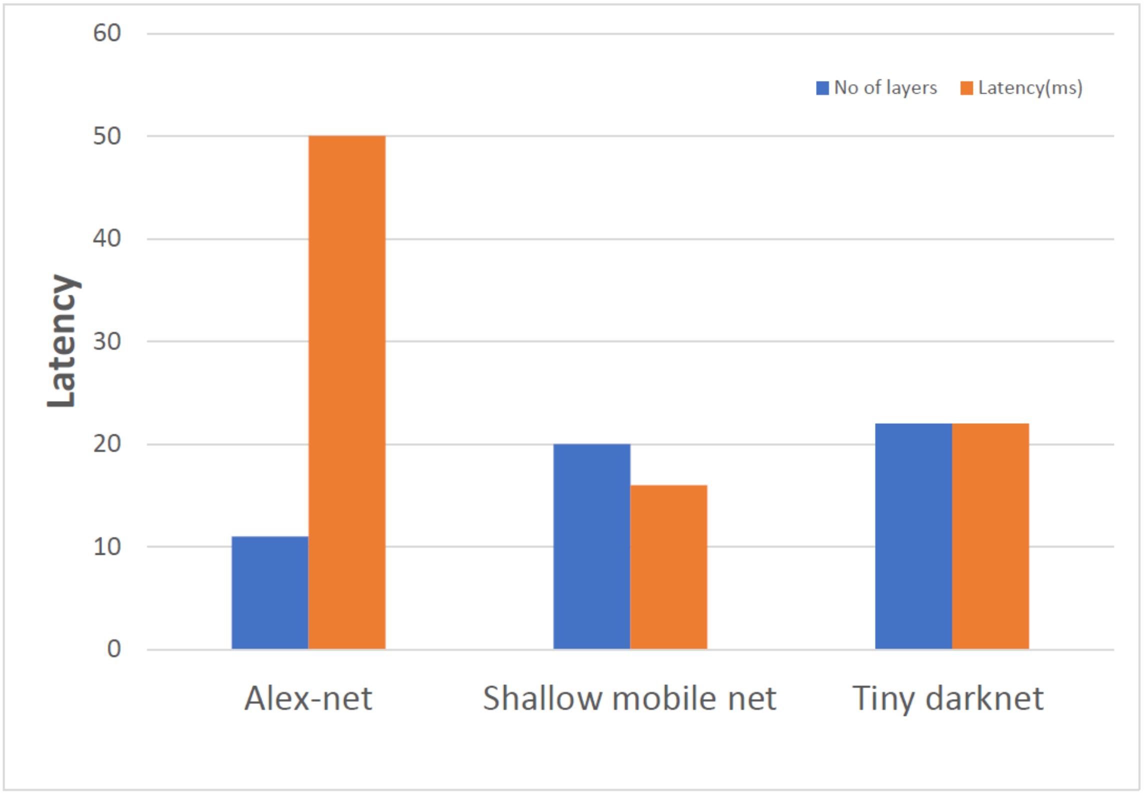

Figure 4.1: Latency of CNN algorithms using NURO-AWARE

4.2.1.1 Evaluation of implemented networks using NURO-AWARE

Figure 4.2 shows implementation of three separate algorithms using NURO-AWARE system. Note that results depicted in Figure 4.2 for shallow mobile net and Tiny Darknet have some of the optimizations implemented to get best system performance. This optimisations will be discussed in further section 4.2.3. The resource utilization shown in the Figure 4.2 shows the *FPGA* resource utilization for implementing each of the network. AlexNet consumes highest amount of resources in each resource section. This is because of the bigger size model AlexNet. The LUT utilization shows the expected graph and is proportional to the size of the network. In the case of BRAM memory, the results are as expected, where the utilization is proportional to the size of the network. It should be also noted that the Tiny Darknet utilizes the least amount of memory as it is has small model size and this network is memory efficient. The

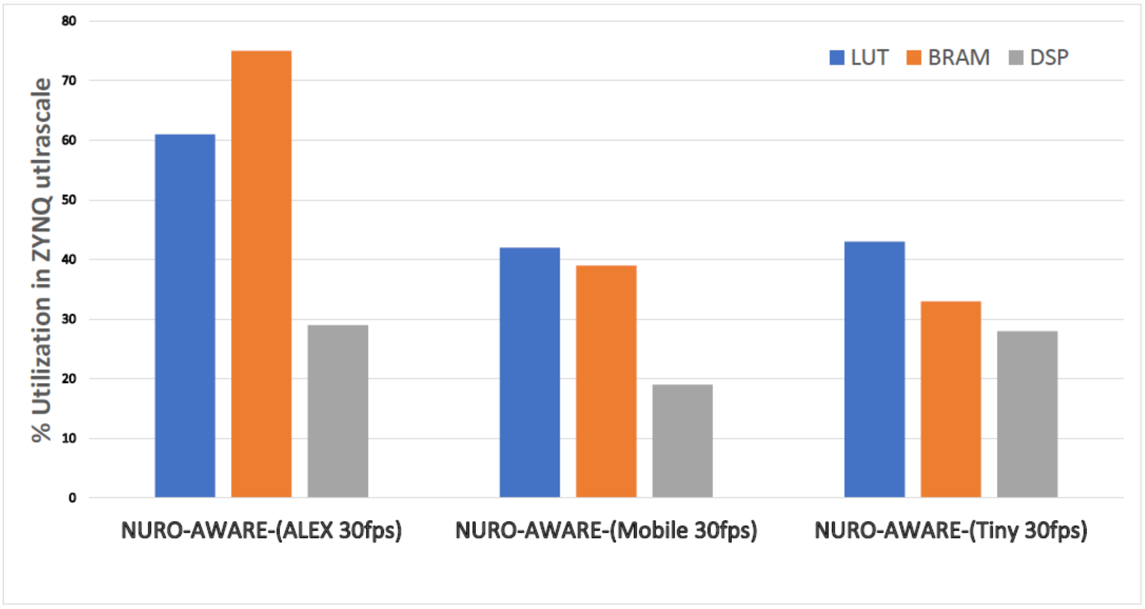

Figure 4.2: Implementation of CNN algorithms using NURO-AWARE

power utilization of these network are shown in 4.3 also shows the expected results, which is proportional to the size of the network and thus AlexNet consumes highest power of 4.5 watts.

4.2.2 Evaluation of networks with other Edge solutions

Next we compare the implementation of AlexNet [1]and Tiny Darknet [34] using NURO-AWARE to implementation of these network using Xilinx's Chai DNN. Chai DNN is available in different overlay versions. We chose version with 1024 DSPs version as this was closet to the FPS supported by NURO-AWARE for AlexNet. Lower DSP version of Chai DNN might not be able to hit the same frame rate (with fully connected layers). The results for this comparison are shown in the Figure 4.4. It can be noted that Chai DNN resource utilization is same for Tiny Darknet and AlexNet due to the overlay setting we have to use for Chai DNN. We can see that the NURO-AWARE system does beat Chai DNN in terms of utilization in both AlexNet implementation and Tiny Darknet implementation. The reason for this is because Chai DNN framework uses systolic arrays and performs sequential acceleration for

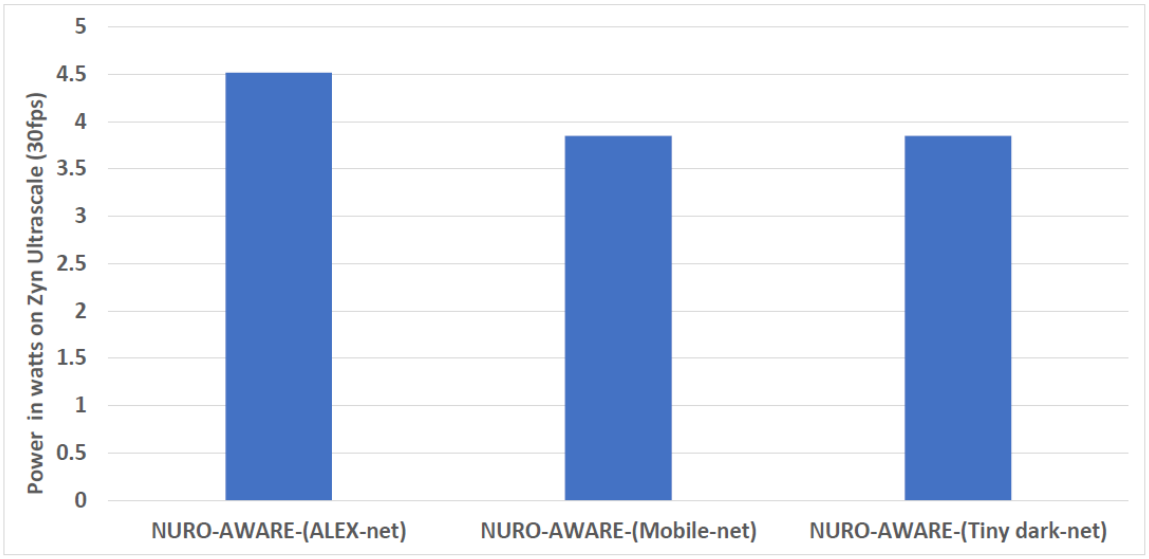

Figure 4.3: Power usage of CNN algorithms using NURO-AWARE

each of the layer. This causes regularization of hardware for each of the layer and thus there is wastage of resources. On contrary NURO-AWARE customizes the data path as well as the memory access path for each of the layers and hence the wastage of the resources is reduced. Figure 3.2.3 shows the performance evaluation for NURO-AWARE system versus Chai DNN AlexNet implementation and NVDIA Xavier which is AI super compute module. This evaluation is performed only on AlexNet implementation to perform worst case evaluation. Evaluation shows that NURO-AWARE system does beat NVDIA Xavier as well as Chai DNN both with respect to power. The low power utilization of NURO-AWRE can be attributed to the custom data path and memory access path catered towards each of the layers. Whearase the high power utilization of Chai DNN, is due to the sequential nature of its acceleration because of which of has to perform large feature map transfers from off-chip to on chip after each layer acceleration is completed. NURO-AWARE solution also beats the performance of NVDIA Xavier as well as Chai DNN. This is because we pipeline the data path efficiently so that we can perform maximum data reuse across the feature maps using the optimized compute engine for each of layers. Where as *GPUs* as well as systolic arrays use sequential processing. Figure 4.6 shows the power utilized by

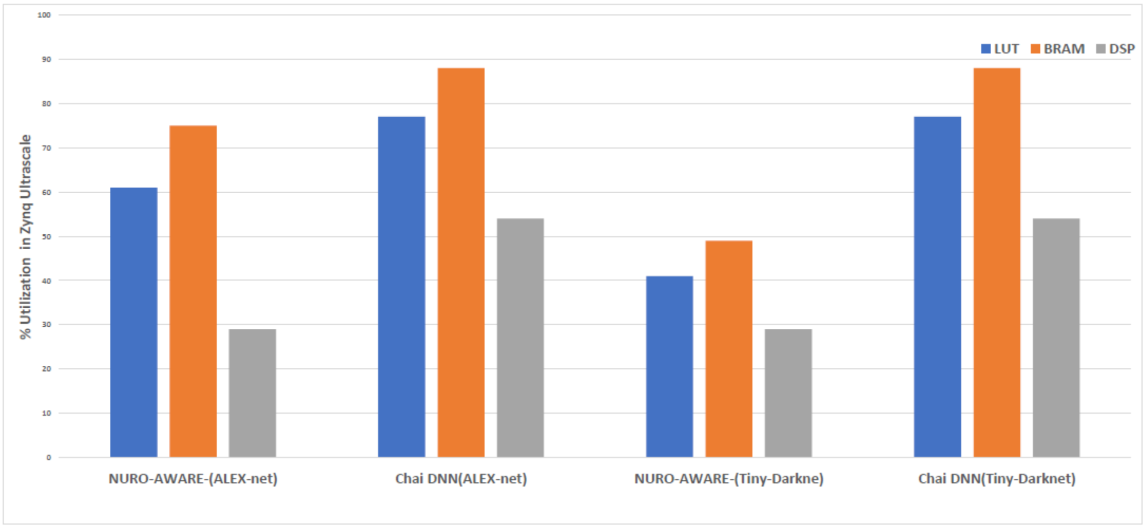

Figure 4.4: Comparision with Chai DNN implementation

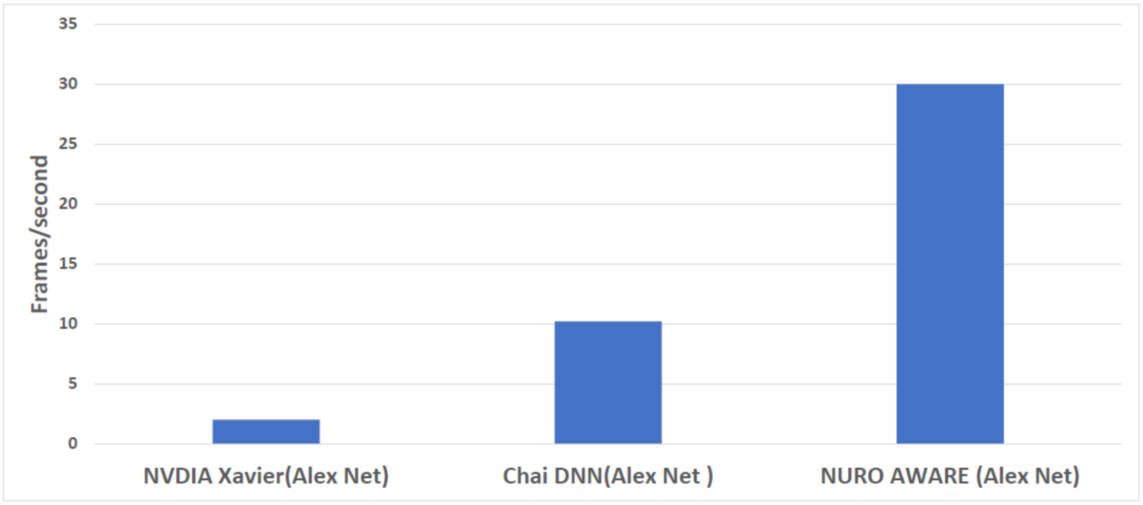

Figure 4.5: NURO-AWARE performance comparison

Chai DNN , NVDIA Xavier and nuro-ram running AlexNet. Nuro-AWARE is using the least power of 4.5 watts then the others solutions which can be attributed to the custom data path and the memory path used by NURO-RAM.

4.2.3 AWARE-DNN Extension

In this section we compare implementation of CNN algorithms AWARE-DNN architecture with off-chip memory support provided using the NURO-RAM system architecture. We first evaluate the implication the NURO-RAM has on the entire

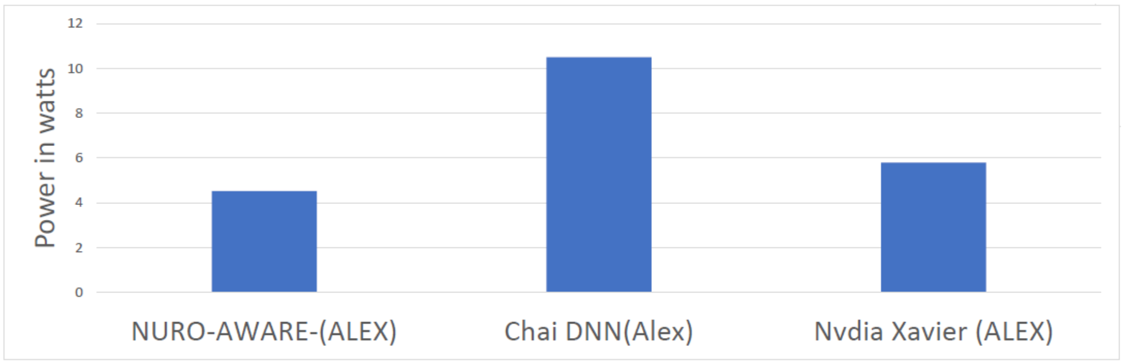

Figure 4.6: NURO-AWARE power comparison

system when supporting bigger network which cannot be fitted given the memory constraints on *FPGA*. Latter we will evaluate the effect of NURO-RAM architecture and its effect on the increasing the performance of smaller architecture such as Shallow mobile-net. In the end we evaluate increasing the efficiency of memory efficient network Tiny Darknet.

4.2.3.1 Implementing AlexNet on AWARE-DNN

While AlexNet is not a very efficient CNN topology but it serves well as a worst case test of NURO-RAM's ability to augment the AWARE-DNN framework's domain coverage. The table 1.1 gives a brief idea of the memory requirement for an implementation of the AlexNet's convolution layers. From the table 1.1 it is clear that if we use on-chip memory for this implementation it would fail. As we can see from the table 4.1 the size of the on-chip memory is 30 Mb. In this case we utilize NURO-RAM to extend our memory hierarchy and support running AlexNet inference. The configuration for each weight bank block is given in the table 4.3

We size the BRAM by calculating the buffer size needed to support the necessary bandwidth in addition to insuring the Microblaze has sufficient time to service the interrupts. This is done by the approach discussed in section 3.3.2.1. The comparison between the raw memory usage and the bank memory for each layer is given in table 4.3. Layer 1 and layer 2 are not implemented with the NURO-RAM architecture

Table 4.3: Alexnet- NURO-RAM Weight configuration

	CNV5	CNV6	CNV7	FC9	FC10	FC11
Convolution Parallelism	3	3	1	6	1	1
kernel Parallelism	8	2	16	16	4	8
Channel Parallelism	1	8	2	1	16	4
RAW Memory (in MB)	0.8	1.3	0.8	37	16	4
BankMemory (in Kb)	0.13	0.32	0.54	1.3	0.31	0.73

interface as we can infer form the Figure 1.1 that the size layer is really small vs the latter layers in the network. The memory reduction obtained by using NURO-RAM

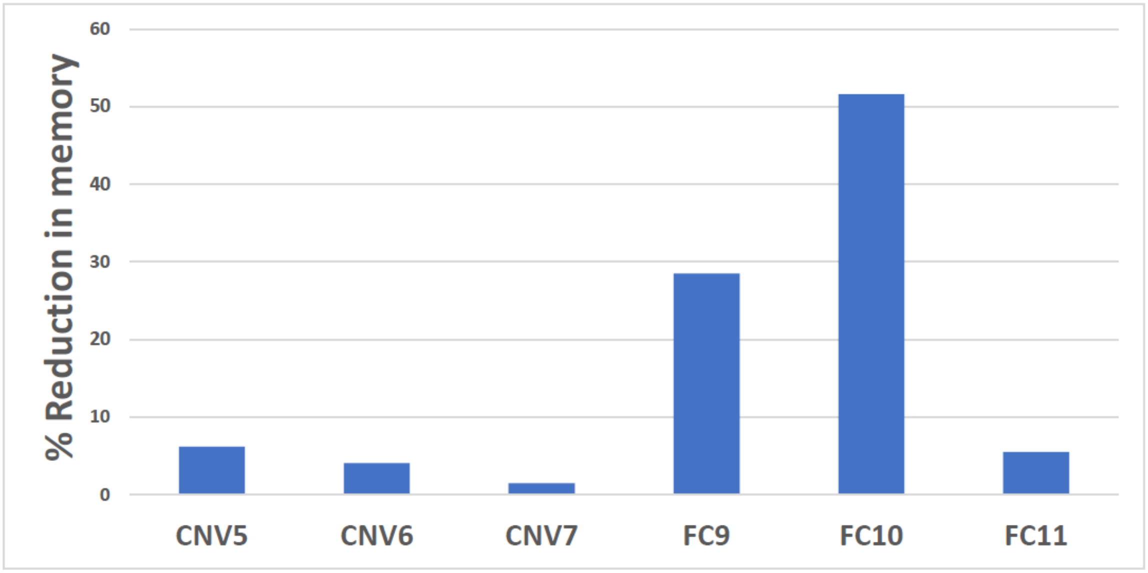

Figure 4.7: Reduction in memory footprint in AlexNet

over the raw memory is shown by the graph4.7. As we can see that the layers with a very high raw memory requirement have the highest reduction. This is related to the fact that the NURO-RAM pre-fetch buffers are sized to accumulate the weights to support minimum weight requirement for the accelerator at any point in time. Where the minimum weight requirement is related to DMA transfer latency as explained in section 3.3.2.1.

4.2.3.2 Implementation of Shallow mobile net

Shallow mobile net is considered compute efficient network due to depth wise convolution. The depth wise convolution reduce the compute intensity by limiting the receptive field to single channel. This compute efficiency is offset by the memory requirements of point-wise layers, needed to insure accuracy.

Table 4.4: Coupling for shallow mobile net

	Kern_par	Ch_par	Conv_par
CONV25	2	8	4
CONV27	2	8	64
CONV30	2	5	2

If we do-not increase the parallelism in the design, the weights can easily fit on the *FPGA*'s on chip memory for this netowork. In general to achieve better latency, spatial parallelism is need when the frequency is bottleneck. But increase in spatial parallelism increases the memory resources. This because the spatial parallelism increases the intermediate feature maps buffers which are used to store feature maps between two layers. But storing the feature maps to external memory has its implication with respect to latency and power wastage. The second reason in increase in the BRAM memory is the increase in the DSPs for spatial parallelism. Thus rather then storing the feature maps off-chip we choose to store the the weights of the layers off-chip. Thus we use the NURO-RAM in order to free-up memory resources for partial feature map buffering and for increase in DSPs. We choose layer CON25, cONV27, cONV30 as off-chip layers due to the size of BRAM memory they (512 KB,1024KB,100KB) require for storing the weights on-chip,rest other layers are not required to be used off-chip the on-chip weight buffer size is small. Figure 4.8 shows three design points depicting the amount of BRAMs used in each design point .

The first memory footprint point Ocp_design is the utilization of BRAMs for

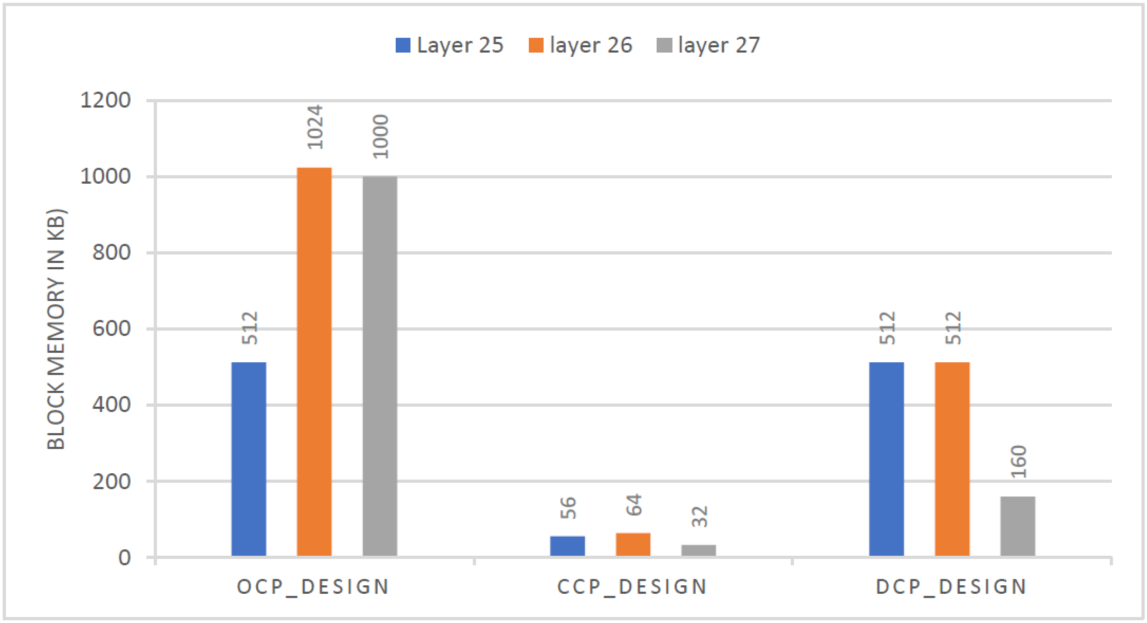

Figure 4.8: BRAMs utilization comparison in off-chip layers in shallow mobile net

off-chip layers if implemented only using on-chip weight bank solution which has pipeline latency of 24ms at 50Mhz. Second design point is an off-chip solution called as Dcp_design point (decoupled design point), where the sizing of the of the pre-fetch buffer is performed in the same way as discussed in section 3.3.2.1. This design point has increase kernel parallelism applied to it and hence can support has decreased latency of 16ms at 50Mhz. The Third design point Ccp_design point in which we applied coupling optimization as discussed in section3.3.2.1 to couple the $kernel_{par}$ to $Channel_{par}$ and $Convolution_{par}$. The parallelism used for coupling design point is given below in table 4.4

Figure 4.8 shows the overheads which are incurred while using the NURO-RAM architecture for small design without any optimization. Even though there is reduction of the number of BRAMs in Layer 26 and Layer 27, the BRAM usage in layer 25 is still the same.

While DCP_Design point has helped the off-chip layers to reduce the number of BRAMs, this utilization difference is less when we compare to the entire network

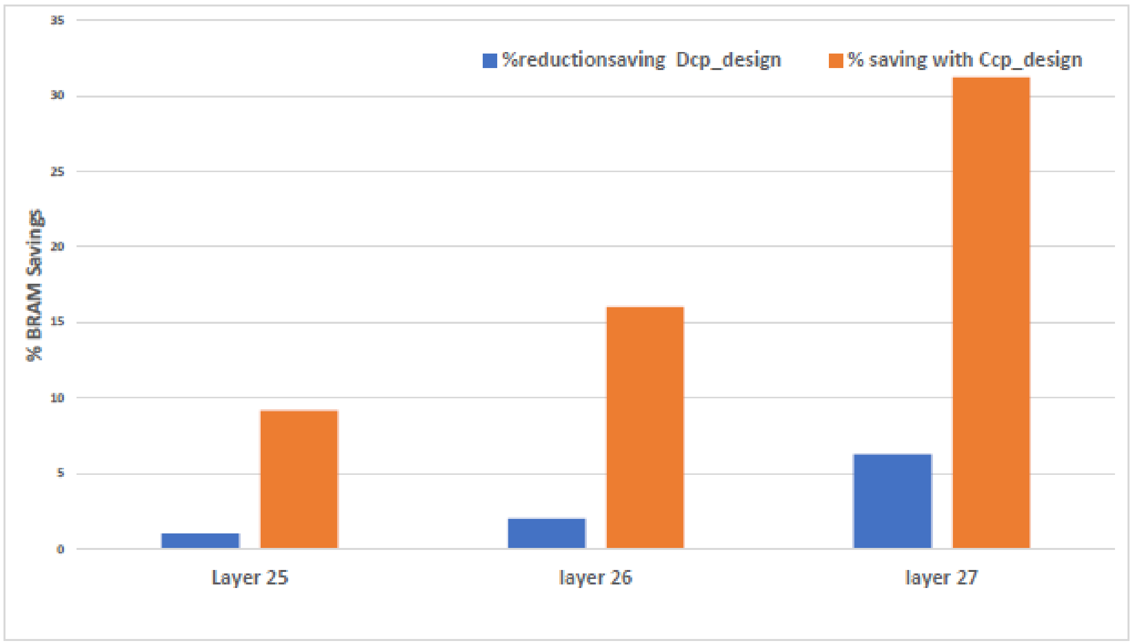

Figure 4.9: Comparison in reduction of memory for Dcp_design vs Ocp_design in shallow mobile net

BRAM utilization as can be seen it Figure 4.10. This is because to improve the performance of the smaller networks we need to increase the the kernel parallelism and since these layers are having smaller weight elements, the kernel parallelism doesn't scale up well for these layers. But it is evident that the coupling optimization design point have been able to effectively reduce the size of the BRAMs used in the implementation. This is because of the reduction of the kernel parallelism due to coupling. Figure 4.9 shows the effective the reduction comparison for CCP_ Design as well as DCP_Design. Here the CCP_Design performs extremely well while serving a smaller network. Thus by increasing the spatial parallelism we can easily boost up the performance of the network with the help of NURO-RAM.

4.2.3.3 Implementation of Tiny Darknet

Tiny Darknet is considered as one of the most memory efficient network in CNN. The total size of the Tiny weight model is mere 1.9 MB. This small size of the network is caused by its squeeze and expand layers [34]. But this network is not as compute

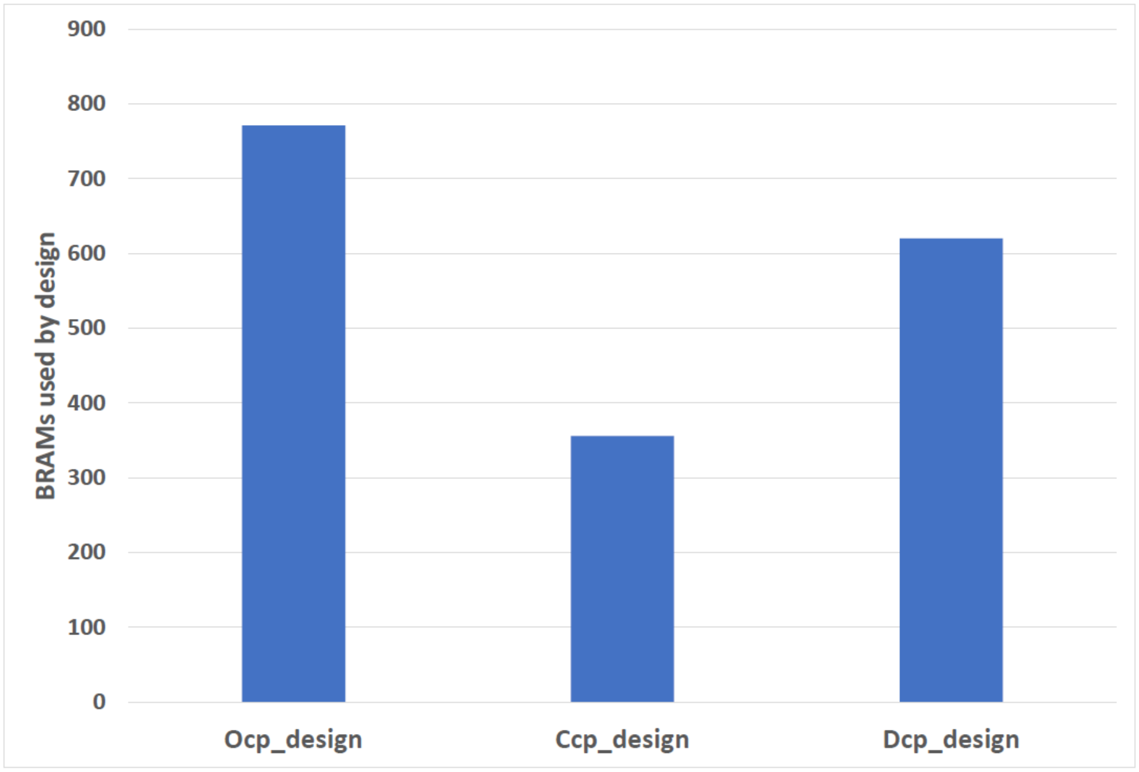

Figure 4.10: Total BRAM utilization in Shallow mobile net across entire network

efficient as shallow mobile net. Thus in order to improve on its compute efficiency we increase the spatial parallelism. This again puts pressure on the resource utilization due to the need of more Dsps and buffering more number of intermediate feature map vectors.

Table 4.5: coupling for Tiny Darknet

	Kern_par	Ch_par	Conv_par
CONV15	2	8	1
CONV17	4	8	3
CONV20	2	25	1

Thus we introduce the NURO-RAM system to release the pressure of resource utilization with respect to memory. Using the similar reasoning as for mobile net we choose the layer CNV15, cNV17, cNV19 to be off-chip layers due to its large BRAM

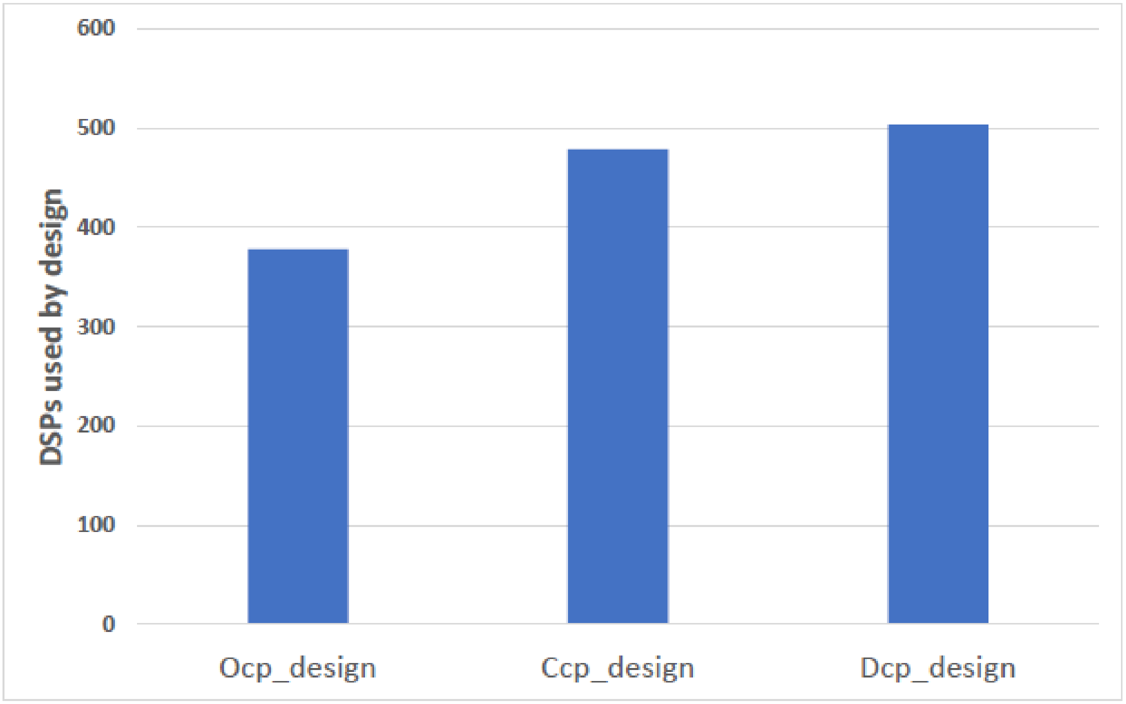

Figure 4.11: Total DSP utilization in Shallow mobile net across entire network

memory size (130 KB,290KB ,290KB) weight size as compared to others. We also use the coupling optimisation to check the diversity of this optimization to support smaller network. Here we use CCP_Design as well as DCP_Design in this section. Figure 4.13 shows the BRAM utilization of Tiny Darknet. It is clear that NURO-RAM did relax the memory resource constraint for the off-chip layers. The coupling optimization used for this configuration is given in table 4.5.

We can see from the Figure 4.13 that the inclusion of the NURO-RAM in the system did help the off-chip layers to effectively reduce the memory resources.

But if compare to the overall BRAM resource utilization from Figure 4.14 we can see that without the use of coupling the NURO-RAM adds additional memory usage to the system. This caused because of the kernel parallelism not scaling up well for smaller network. But with the coupling used with NURO-RAM we can see the system design does improve and we see the drop in total memory utilization as shown in Figure 4.14. But the saving ratio between the Dcp_design point savings and

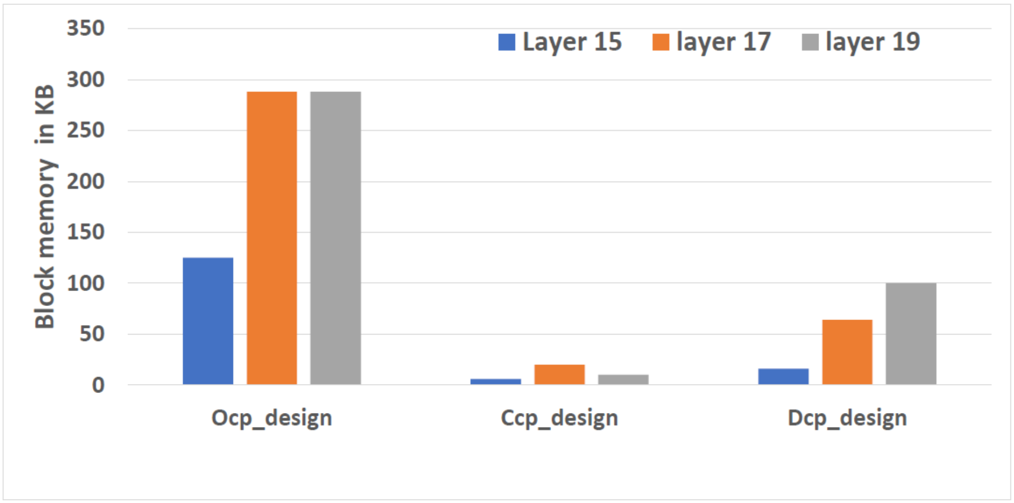

Figure 4.12: BRAMs utilization comparison in off-chip layers in Tiny Darknet

Ccp_design point savings are lesser as compared to the shallow mobile net Figure 4.9 because Tiny Darknet is inherently memory optimized network than mobile net. Thus with reduction of BRAMs in Tiny Darknet can help us to fit Tiny Darknet on smaller devices like ultra96 which 312 Bram of RAMB36 as shown in Figure 1.2.

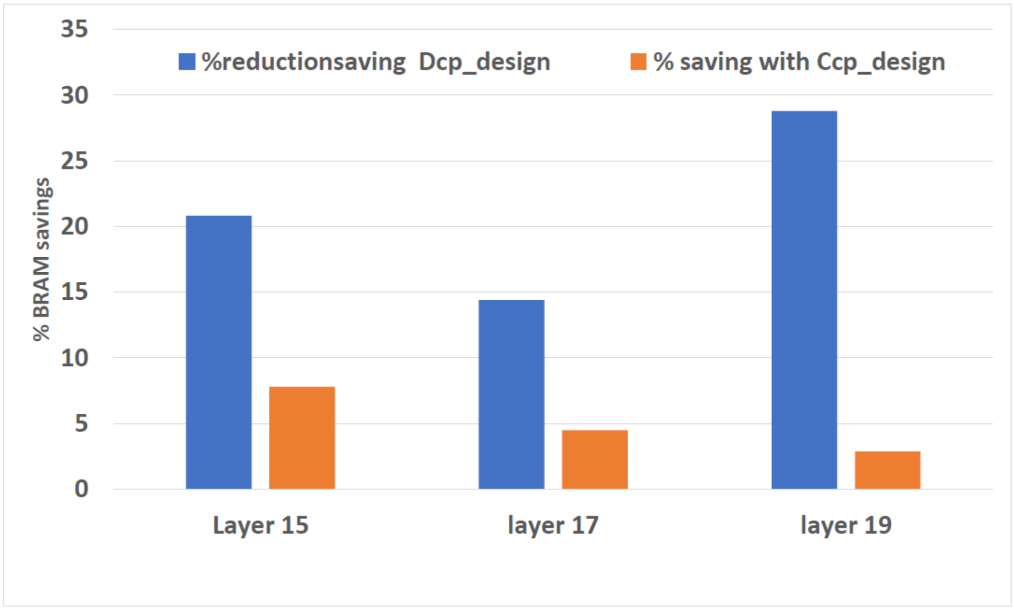

Figure 4.13: Comparison in reduction of memory for Dcp_design vs Ocp_design in Tiny Darknet

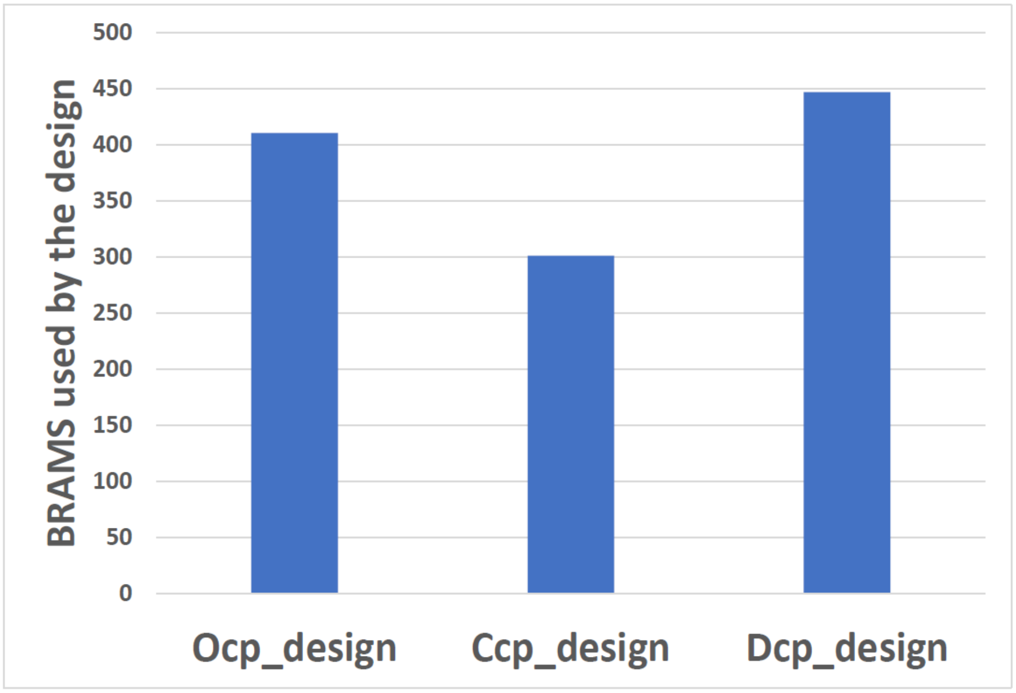

Figure 4.14: Total BRAM utilization in Tiny dark net net across entire network

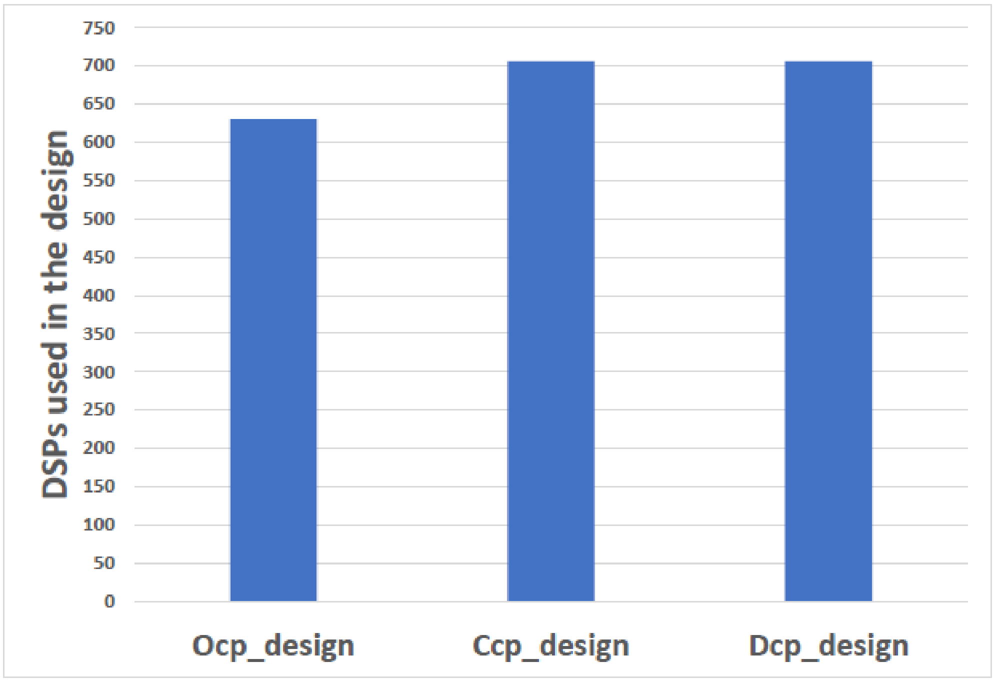

Figure 4.15: Tiny Darknet Dsp utilization

CHAPTER 5: Conclusion And Future Work

5.1 Conclusion

An efficient weight streaming system to stream the weights to the real-time CNN accelerator was developed. This research explores how we can leverage fast and efficient one chip memory present on *FPGA* to serve the parallel streaming of weights across multiple layers. As the accelerator this work was focusing on used parallelism to accelerate the CNN network and had real time time constraint this work had to develop an efficient architecture for customizable memory hierarchy and interface unit. This led to development of NURO-RAM. During the development various aspects of the architecture were taken into consideration to prevent the wastage of the memory, power and other *FPGA* resources. The architecture thus developed used some of the prebuilt Xilinx IPS as well as custom design IP. Since on-chip memory is constrained on *FPGA* we had resort to scheduling mechanism in order to schedule the off-chip transfer to any layer, so that minimum weight size needed by the accelerator was pre-fetched and used. This was followed by development of static scheduling algorithm which was responsible for scheduling the weight. The firmware for this scheduler was implemented on Xilinx Micro-Blaze soft-core and experimental results were carried out using Xilinx zcu102 board.

Comparing to the previous edge based solutions such as the Chai DNN and NDV-DIA Xavier it was found that the NURO-AWARE architecture performed pretty well in terms of power usage due to its highly crafted architecture for accelerating layers in pipelined fashion. It was also evaluated that NURO-AWARE architecture did perform better than Chai DNN and NVDIA Xavier in performance. NURO-AWARE architecture has also proved to be resource efficient as it beats the Xilinx Chai DNN for two network implementation Tiny dark net and AlexNet. For evaluation of the diversity of support of network for NURO-RAM, three separate networks were implemented on AWARE accelerator using NURO-RAM interface (NURO-AWARE). It was found that for supporting the large networks like AlexNet, the basic NURO-RAM architecture proved highly efficient more than 50x saving for one of the fully connected layer. The Nuro-RAM architecture was also evaluated to support highly paralleled smaller networks for performance. In this case it was found the basic NURO-RAM basic architecture interface has proved to be not well efficient due to increase in high kernel parallelism. In order to tackle this problem a new optimisation to the NURO-AWARE architecture interface was developed which indeed helped the architecture to accelerate these network by providing memory savings to the network.

5.2 Future Work

Currently NURO-Aware system has implemented three speerate networks . In future work we will try to add more network support. Future work also includes implementing the NURO-AWARE architecrure from ALTERA or even smaller devices from Xilinx. Since the NURO-RAM has so many system tunable knobs which can be used to suite the needs of application. One such knob is mapping different design point with respect separate frequency and asynchronous FIFO. We will also try to characterize the DMA transfer using the scatter gather engine which runs autonomously thus saving on complexity scheduler.

REFERENCES

[1] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *Advances in neural information processing systems*, pp. 1097–1105, 2012.

[2] Xilinx, "Cifar10," *https://en.wikipedia.org/wiki/CIFAR-10(visited on 06/20/2019)*.

[3] J. Sanchez, N. Soltani, R. Chamarthi, A. Sawant, and H. Tabkhi, "A novel 1d-convolution accelerator for low-power real-time cnn processing on the edge," in *2018 IEEE High Performance extreme Computing Conference (HPEC)*, pp. 1–8, IEEE, 2018.

[4] "Project brainwave for real-time ai," 2017.

[5] N. P. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, A. Borchers, *et al.*, "In-datacenter performance analysis of a tensor processing unit," in *Proceedings of the 44th Annual International Symposium on Computer Architecture*, pp. 1–12, ACM, 2017.

[6] Y.-H. Chen, J. Emer, and V. Sze, "Eyeriss v2: A flexible and high-performance accelerator for emerging deep neural networks," *arXiv preprint arXiv:1807.07928*, 2018.

[7] A. Savich and S. Areibi, "A low-power scalable stream compute accelerator for general matrix multiply (gemm)," *VLSI Design*, vol. 2014, p. 2, 2014.

[8] Y. Shen, M. Ferdman, and P. Milder, "Maximizing cnn accelerator efficiency through resource partitioning," *SIGARCH Comput. Archit. News*, vol. 45, pp. 535–547, June 2017.

[9] M. Putic, A. Buyuktosunoglu, S. Venkataramani, P. Bose, S. Eldridge, and M. Stan, "Dyhard-dnn: even more dnn acceleration with dynamic hardware reconfiguration," in *2018 55th ACM/ESDA/IEEE Design Automation Conference (DAC)*, pp. 1–6, IEEE, 2018.

[10] W. Qadeer, R. Hameed, O. Shacham, P. Venkatesan, C. Kozyrakis, and M. Horowitz, "Convolution engine: Balancing efficiency and flexibility in specialized computing," *Commun. ACM*, vol. 58, pp. 85–93, Mar. 2015.

[11] W. Qadeer, R. Hameed, O. Shacham, P. Venkatesan, C. Kozyrakis, and M. Horowitz, "Convolution engine: Balancing efficiency and flexibility in specialized computing," *Communications of the ACM*, vol. 58, no. 4, pp. 85–93, 2015.

[12] M. Rastegari, V. Ordonez, J. Redmon, and A. Farhadi, "Xnor-net: Imagenet classification using binary convolutional neural networks," in *European Conference on Computer Vision*, pp. 525–542, Springer, 2016.

[13] H. Kim, J. Sim, Y. Choi, and L.-S. Kim, "A kernel decomposition architecture for binary-weight convolutional neural networks," in *Proceedings of the 54th Annual Design Automation Conference 2017*, p. 60, ACM, 2017.

[14] P. Colangelo, N. Nasiri, E. Nurvitadhi, A. Mishra, M. Margala, and K. Nealis, "Exploration of low numeric precision deep learning inference using intelÂ® fpgas," in *2018 IEEE 26th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pp. 73–80, April 2018.

[15] M. Ghasemzadeh, M. Samragh, and F. Koushanfar, "Rebnet: Residual binarized neural network," in *2018 IEEE 26th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pp. 57–64, April 2018.

[16] P. Guo, H. Ma, R. Chen, P. Li, S. Xie, and D. Wang, "Fbna: A fully binarized neural network accelerator," in *2018 28th International Conference on Field Programmable Logic and Applications (FPL)*, pp. 51–513, IEEE, 2018.

[17] M. Shimoda, S. Sato, and H. Nakahara, "Demonstration of object detection for event-driven cameras on fpgas and gpus," in *2018 28th International Conference on Field Programmable Logic and Applications (FPL)*, pp. 461–4611, IEEE, 2018.

[18] S. Vogel, M. Liang, A. Guntoro, W. Stechele, and G. Ascheid, "Efficient hardware acceleration of cnns using logarithmic data representation with arbitrary log-base," in *Proceedings of the International Conference on Computer-Aided Design*, p. 9, ACM, 2018.

[19] D. Miyashita, E. H. Lee, and B. Murmann, "Convolutional neural networks using logarithmic data representation," *arXiv preprint arXiv:1603.01025*, 2016.

[20] Z. Du, R. Fasthuber, T. Chen, P. Ienne, L. Li, T. Luo, X. Feng, Y. Chen, and O. Temam, "Shidiannao: Shifting vision processing closer to the sensor," in *ACM SIGARCH Computer Architecture News*, vol. 43, pp. 92–104, ACM, 2015.

[21] H. Sharma, J. Park, D. Mahajan, E. Amaro, J. K. Kim, C. Shao, A. Mishra, and H. Esmaeilzadeh, "From high-level deep neural models to fpgas," in *Microarchitecture (MICRO), 2016 49th Annual IEEE/ACM International Symposium on*, pp. 1–12, IEEE, 2016.

[22] Y. Shen, M. Ferdman, and P. Milder, "Escher: A cnn accelerator with flexible buffering to minimize off-chip transfer," in *Field-Programmable Custom Computing Machines (FCCM), 2017 IEEE 25th Annual International Symposium on*, pp. 93–100, IEEE, 2017.

[23] S. Han, X. Liu, H. Mao, J. Pu, A. Pedram, M. A. Horowitz, and W. J. Dally, "Eie: efficient inference engine on compressed deep neural network," in *Computer Architecture (ISCA), 2016 ACM/IEEE 43rd Annual International Symposium on*, pp. 243–254, IEEE, 2016.

[24] S. M. A. H. Jafri, A. Hemani, K. Paul, and N. Abbas, "Mocha: Morphable locality and compression aware architecture for convolutional neural networks," in *Parallel and Distributed Processing Symposium (IPDPS), 2017 IEEE International*, pp. 276–286, IEEE, 2017.

[25] M. Alwani, H. Chen, M. Ferdman, and P. Milder, "Fused-layer cnn accelerators," in *Microarchitecture (MICRO), 2016 49th Annual IEEE/ACM International Symposium on*, pp. 1–12, IEEE, 2016.

[26] A. Aimar, H. Mostafa, E. Calabrese, A. Rios-Navarro, R. Tapiador-Morales, I.-A. Lungu, M. B. Milde, F. Corradi, A. Linares-Barranco, S.-C. Liu, *et al.*, "Nullhop: A flexible convolutional neural network accelerator based on sparse representations of feature maps," *arXiv preprint arXiv:1706.01406*, 2017.

[27] L. Gong, C. Wang, X. Li, H. Chen, and X. Zhou, "Maloc: A fully pipelined fpga accelerator for convolutional neural networks with all layers mapped on chip," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 37, no. 11, pp. 2601–2612, 2018.

[28] M. Blott, T. B. Preußer, N. J. Fraser, G. Gambardella, K. Oâbrien, Y. Umuroglu, M. Leeser, and K. Vissers, "Finn-r: An end-to-end deep-learning framework for fast exploration of quantized neural networks," *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, vol. 11, no. 3, p. 16, 2018.

[29] X. Zhang, J. Wang, C. Zhu, Y. Lin, J. Xiong, W.-m. Hwu, and D. Chen, "Dnnbuilder: an automated tool for building high-performance dnn hardware accelerators for fpgas," in *Proceedings of the International Conference on Computer-Aided Design*, p. 56, ACM, 2018.

[30] S. I. Venieris and C.-S. Bouganis, "f-cnnx: A toolflow for mapping multiple convolutional neural networks on fpgas," in *2018 28th International Conference on Field Programmable Logic and Applications (FPL)*, pp. 381–3817, IEEE, 2018.

[31] X. Yang, M. Gao, J. Pu, A. Nayak, Q. Liu, S. E. Bell, J. O. Setter, K. Cao, H. Ha, C. Kozyrakis, *et al.*, "Dnn dataflow choice is overrated," *arXiv preprint arXiv:1809.04070*, 2018.

[32] Y. Yang, Q. Huang, B. Wu, T. Zhang, L. Ma, G. Gambardella, M. Blott, L. Lavagno, K. Vissers, J. Wawrzynek, *et al.*, "Synetgy: Algorithm-hardware co-design for convnet accelerators on embedded fpgas," *arXiv preprint arXiv:1811.08634*, 2018.

[33] A. G. Howard, M. Zhu, B. Chen, D. Kalenichenko, W. Wang, T. Weyand, M. Andreetto, and H. Adam, "Mobilenets: Efficient convolutional neural networks for mobile vision applications," *arXiv preprint arXiv:1704.04861*, 2017.

[34] J. Redmon, "Tiny darknet," *URL: https://pjreddie. com/darknet/tiny-darknet/(visited on 08/23/2018).*

[35] W. Zhao, H. Fu, W. Luk, T. Yu, S. Wang, B. Feng, Y. Ma, and G. Yang, "F-cnn: An fpga-based framework for training convolutional neural networks," in *2016 IEEE 27th International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, pp. 107–114, IEEE, 2016.

[36] E. S. Chung, J. C. Hoe, and K. Mai, "Coram: an in-fabric memory architecture for fpga-based computing," in *Proceedings of the 19th ACM/SIGDA international symposium on Field programmable gate arrays*, pp. 97–106, ACM, 2011.

[37] A. Stoutchinin, F. Conti, and L. Benini, "Optimally scheduling cnn convolutions for efficient memory access," *arXiv preprint arXiv:1902.01492*, 2019.

[38] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," *arXiv preprint arXiv:1409.1556*.

[39] Xilinx, "Memory interface generator," *https://www.xilinx.com/support/documentation/ip*$_d$*ocum*
ultrascale $-$ *memory* $-$ *ip.pdf*(*visitedon*08/23/2018).

[40] Xilinx, "Central direct memory access," *https://www.xilinx.com/support/documentation/ip*$_d$*ocu*
axi $-$ *cdma.pdf*(*visitedon*06/20/2019).

[41] Xilinx, "Micro-blaze soft core," *https://www.xilinx.com/products/design-tools/microblaze.html(visited on 06/20/2019).*

[42] Xilinx, "Vivado sysnthesis and implementation tool," *https://www.xilinx.com/support/download/index.html/content/xilinx/en/downloadNav/vivado-*
design-tools/archive.html(visited on 06/20/2019).