ACCELERATING TWO DIMENSIONAL FFT AND CONVOLUTIONAL NEURAL NETWORKS ON FPGA

A THESIS

submitted by

KALA S

for the award of the degree

of

DOCTOR OF PHILOSOPHY

in the Faculty of Engineering

DIVISION OF ELECTRONICS ENGINEERING,

SCHOOL OF ENGINEERING

COCHIN UNIVERSITY OF SCIENCE AND TECHNOLOGY

January 2020

THESIS CERTIFICATE

This is to certify that the thesis titled **ACCELERATING TWO DIMENSIONAL FFT AND CONVOLUTIONAL NEURAL NETWORKS ON FPGA**, submitted by **Kala S.**, to Cochin University of Science And Technology, for the award of the degree of **Doctor of Philosophy**, is a bonafide record of research work done by her under my supervision and guidance at Division of Electronics Engineering, School of Engineering, Cochin University of Science And Technology. The contents of this thesis, in full or in parts, have not been submitted to any other Institute or University for the award of any degree or diploma.

I further certify that the corrections and modifications suggested by the audience during the pre-synopsis seminar and recommended by the Doctoral Committee of Ms. Kala S are incorporated in the thesis.

Dr. Babita Roslind Jose
Research Guide
Associate Professor
Division of Electronics Engineering,
School of Engineering
CUSAT, 682 022

Place: Kalamassery

Date: 15^{th} January 2020

To

Nalesh for his endless love and support...

DECLARATION

I hereby declare that the work presented in the thesis entitled **ACCELERATING TWO DIMENSIONAL FFT AND CONVOLUTIONAL NEURAL NETWORKS ON FPGA** is based on the original research work carried out by me under the supervision and guidance of **Dr. Babita Roslind Jose**, Associate Professor, for the award of degree of Doctor of Philosophy with Cochin University of Science and Technology. I further declare that the contents of this thesis, in full or in parts, have not been submitted to any other University or Institute for the award of any degree or diploma.

Kochi - 682 022

Date: 15/01/2020

Kala S

ACKNOWLEDGEMENTS

I am indebted to many people for their support during my life at CUSAT. First and foremost I would like to thank my supervisor Dr. Babita Roslind Jose for her guidance and support throughout the research period. I thank her for giving me the opportunity to work under her guidance in Division of Electronics. She was so approachable that her friendly attitude made it very interesting to work in the lab.

I express my deepest gratitude to Prof. Jimson Mathew, IIT Patna, for his guidance and motivation throughout my research period. He was the one who has introduced me to the world of research even before I join for MS(Engg) at IISc Bangalore. I thank him for trusting me and providing me all kind of supports for my research work. Without his encouragement and persistent help, this thesis would not have been possible. He was very instrumental in helping me while writing research papers and project proposals. Long discussions with him have given me a clear direction to go on. No words are sufficient to thank Prof. Jimson.

I am thankful to Dr. Shahana T.K., Head, Division of Electronics for providing me with all the academic and administrative support during my course of PhD. I also thank Prof. M.R.R. Panicker (former Principal, SOE), for all administrative supports that he has given to me without any hesitation during his tenure as a Principal. I am also grateful to Dr. Deepa Sankar and Dr. Rekha K. James for their valuable suggestions during the half-yearly presentations. I also thank Dr. Binu Paul and Dr. Mridula S for their timely help during my research period.

I would like to thank all research scholars in the department for creating an enthusiastic and dynamic environment. I especially thank my ex-lab mate Bijitha for being my good friend and critique. Our discussions in the lab range from computer vision to movies, food and politics. I thank Chitra and Chandny for giving me a good companionship during my life at CUSAT. I am also thankful to Debdeep Paul (Department of Electrical Engineering, IIT Patna) for helping me in my research with GPU simulations. I thank staff members of Audit section (Administration), SOE A-section (especially

Mrs. Azra) and B-section for their co-operation in the administrative matters.

This work was in part funded by Kerala State Council for Science, Technology and Environment (KSCSTE) through Back-to-Lab (BLP) research fellowship under Woman Scientists Division (WSD). I am grateful to Dr. K.R. Lekha, Head of WSD, for her support and guidance in completion of this work. I would like to thank Prof. Deepak Mishra, Indian Institute of Space Science And Technology (IIST Trivandrum), for his valuable comments and guidance in completion of the KSCSTE project. I am extremely grateful to Prof.Mishra for the arrangements that he has provided for one week, during my visit to Virtual Reality Lab, IIST.

I am thankful to organizers of 32^{nd} IEEE International Conference on VLSI Design & 18^{th} Int'l Conference on Embedded Systems (VLSID), New Delhi, 2019, for providing me with the fellowship for attending and presenting my research paper. I also thank organizers of $VLSID$ 2020, Bengaluru, for providing fellowship for presenting a poster at *Student Research Forum (Best SRF-Runner Up)*. I also thank Council of Scientific & Industrial Research (CSIR) for providing the travel grant for attending and presenting a paper in 32^{nd} IEEE International System-On-Chip Conference (SOCC) 2019, at Singapore.

I thank my parents and sisters for supporting me throughout my life. I owe everything to my father for what I have achieved so far. I also thank my sisters' kids, (Hari, Mukundan, Harshan and Darshan) for entertaining my son Gautham whenever I was busy. My son, Gautham was born during the initial stage of my research. I am deeply indebted to my mother who has taken care of my son when both his father and mother were away for research work.

Thank you Nalesh for being with me always!
His love, care and support has really helped me in getting through this journey. Without his support and help I would not have joined for PhD. Nalesh could tactfully manage Gautham during my busy schedule of paper submission deadlines and conference travels.
This doctoral thesis is dedicated to Nalesh and Gautham.

Kala S.
15^{th} January, 2020

ABSTRACT

Signal processing domain has seen tremendous growth in research and applications over the past few decades. Digital Signal Processing (DSP) algorithms are widely used in image and video processing systems. These systems find applications in the field of multimedia, digital TV, radio, biomedical imaging, weather forecasting and gaming. With the increased capacity of transistors in a chip, realization of many of these applications is possible. Digital image processing is a field of DSP where digital images are processed by means of digital computers. Most of the steps in digital image processing like image analysis, image reconstruction, image enhancement and compression can be performed using Fast Fourier Transform (FFT) algorithm. Evolution of deep learning techniques enabled implementation of most of the image processing applications with high performance and accuracy. Most popular deep learning technique used for image classification and detection task is Convolutional Neural Network (CNN), which is a variant of deep neural network. There exist several hardware and software solutions for implementing these algorithms. Hardware implementations provide better performance per watt and are more suited for real-time embedded applications compared to software implementations. This thesis focus on hardware acceleration of two dimensional FFT and Convolutional Neural Networks for image processing systems.

In the first part of thesis, a two dimensional (2D) FFT architecture for image restoration and reconstruction is presented. A 2D FFT architecture using cascade of two radix-4^3 FFTs based on a parallel unrolled radix-4 butterfly unit is proposed in this work. A 64×64 point 2D FFT architecture based on radix-4^3 algorithm using a parallel unrolled radix-4^3 FFT has been presented in this work. Radix-4^3 architecture in this work is a memory optimized parallel architecture which computes 64 point FFT, with least execution time. Row-column decomposition of two radix-4^3 blocks is used to compute a 2D FFT. Proposed architecture has been implemented in UMC 65nm 1P10M CMOS technology with a maximum clock frequency of 312.5 MHz and area of $1.22mm^2$. The architecture is also implemented in Xilinx Virtex-7 FPGA and the results are compared

with state -of-art implementations. Second part of this thesis is focused on an efficient hardware accelerator for Convolutional Neural Networks (CNNs), which are widely used in image classification tasks. Recent researches have shown the effectiveness of FPGA as a hardware accelerator for CNNs which can deliver high performance at low power budgets. Majority of computations in CNNs involve 2D convolution. Winograd minimal filtering based algorithm is the most efficient technique for calculating convolution for smaller filter sizes. CNNs also consist of fully connected layers which are computed using General Element-wise Matrix Multiplication (GEMM). In the second work, an exploration of various algorithms for computing convolution layers in CNN is performed and complexity of these algorithms are compared. A unified architecture named *UniWiG* is proposed, where both Winograd based convolution and GEMM can be accelerated using the same set of processing elements. This approach leads to efficient utilization of FPGA hardware resources while computing all layers in CNN. We have mapped popular CNN models like AlexNet, VGG-16 and ResNet-18 onto the proposed accelerator and the measured performance compares favorably with other state-of-art implementations.

KEYWORDS: Digital Signal Processing; Deep Learning; Fast Fourier Transform; FPGA; CNN; Hardware acceleration; Performance.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ACRONYMS

1D	**O**ne **D**imensional
2D	**T**wo **D**imensional
ASIC	**A**pplication **S**pecific **I**ntegrated **C**ircuit
BRAM	**B**lock **R**andom **A**ccess **M**emory
CMOS	**C**omplemenatry **M**etal **O**xide **S**emiconductor
CNN	**C**onvolutional **N**eural **N**etwork
DDR	**D**ouble **D**ata **R**ate
DFT	**D**iscrete **F**ourier **T**ransform
DIF	**D**ecimation **I**n **F**requency
DIP	**D**igital **I**mage **P**rocessing
DIT	**D**ecimation **I**n **T**ime
DNN	**D**eep **N**eural **N**etwork
DSP	**D**igital **S**ignal **P**rocessing
FC	**F**ully **C**onnected
FFT	**F**ast **F**ourier **T**ransform
FPGA	**F**ield **P**rogrammable **G**ate **A**rray
GEMM	**G**eneral **E**lement-wise **M**atrix **M**ultiplication
GFLOPS	**G**iga **F**loating **P**oint **O**perations **P**er **S**econd
GOPS	**G**iga **O**perations **P**er **S**econd
GPP	**G**eneral **P**urpose **P**rocessor
GPU	**G**raphics **P**rocessing **U**nit
IFFT	**I**nverse **F**ast **F**ourier **T**ransform
ILSVRC	**I**mageNet **L**arge **S**cale **V**isual **R**ecognition **C**hallenge
LUT	**L**ook **U**p **T**able
MAC	**M**ultiply **A**nd **A**ccumulate
MDC	**M**ulti-path **D**elay **C**ommutator
PE	**P**rocessing **E**lement
RAM	**R**andom **A**ccess **M**emory

ROM	**R**ead **O**nly **M**emory
RTL	**R**egister **T**ransfer **L**ogic
SDC	**S**ingle-path **D**elay **C**ommutator
SDF	**S**ingle-path **D**elay **F**eedback
SFG	**S**ignal **F**low **G**raph
SRAM	**S**tatic **R**andom **A**ccess **M**emory
SNR	**S**ignal to **N**oise **R**atio
TF	**T**widdle **F**actor
UniWiG	**Uni**fied **Wi**nograd **G**EMM
VLSI	**V**ery **L**arge **S**cale **I**ntegration

NOTATIONS

mm^2	Area in millimeter square
μ**s**	Time in micro seconds
MHz	Frequency in Mega Hertz
V	Votage in Volts
mW	Power in milliwatts
J	Energy in Joules

CHAPTER 1

Introduction

Rapid advancements in science and technology has paved path for researches in developing machines which have faster processing capabilities than human brain. Image processing and machine learning are currently active areas of research due to their wide range of applications like surveillance, safety, medical field, games and entertainment and so on. Digital Image Processing (DIP) refers to processing of digital images using digital computers as described in Gonzalez and Woods (2008). Fundamental steps in digital image processing is explained in Fig. 1.1 Gonzalez and Woods (2008). Some of these processes are acquisition, image enhancement, image restoration and reconstruction, color image processing, wavelets and multi-resolution processing, compression, morphological processing, segmentation, representation, object detection and recognition. An image is captured by the sensor (or a camera) in image acquisition stage. In image enhancement, an image is manipulated to get a suitable result compared to the actual image, depending on the application. Image restoration and reconstruction refers to improvements in the appearance of an image. In color processing, the color of an image is used to extract certain features. Wavelets and compression techniques are used to reduce the size of an image. In morphological processing, feature extraction for representation of shape is performed. Image segmentation is one of the difficult operations in DIP, where the objects are separated from the background of the image. Representation and description can be either of two types, namely, boundary representation or region representation. Recognition refers to labeling the image based on the descriptors. Interaction between these modules is controlled by 'Knowledge Base'. Most of these processes can be performed using Digital Signal Processing (DSP) algorithms like Discrete Fourier Transform (DFT), Discrete Cosine Transform (DCT) and machine learning algorithms.

Image reconstruction is computationally demanding operation and a standard CPU or computer is not an efficient solution for performing real-time image reconstruction. A digital image processing system consists of sensors, specialized image processing

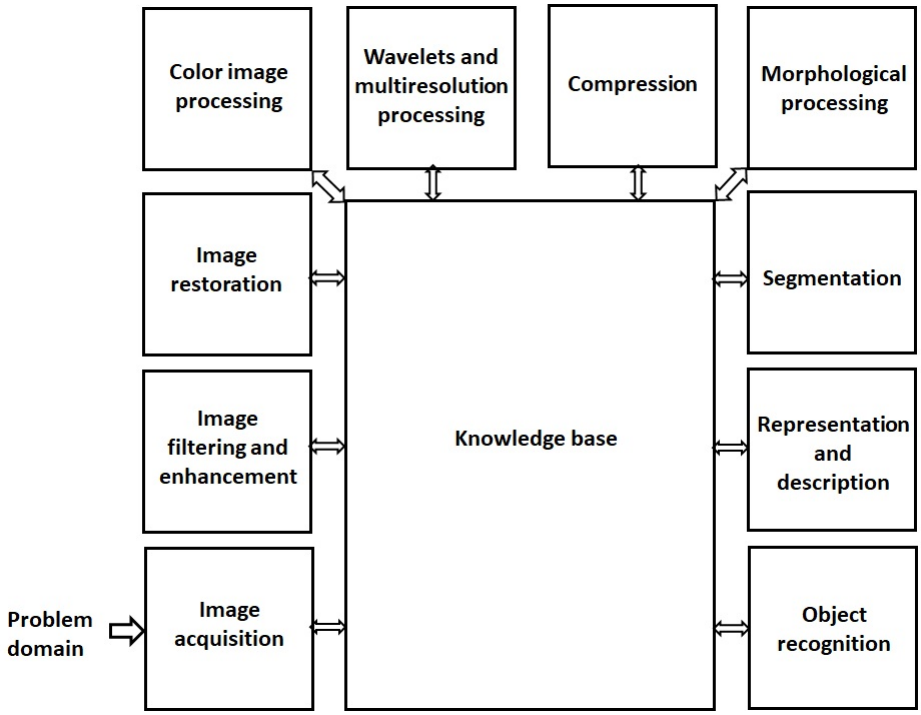

Figure 1.1: Digital Image Processing Gonzalez and Woods (2008)

hardware and software, computer, storage devices and displays. Frame rate and resolution of the sensor and reconstruction time are important factors which affect the performance of the image processing system. For performing certain signal processing operations where speed or performance is a critical parameter, hardware accelerators play key role. This thesis describes about hardware accelerators used in DIP systems for improving the performance. In this chapter, motivation for this research, along-with the objectives and scope of this thesis are presented.

1.1 Motivation, Objective and Scope

Researches in digital image processing has witnessed tremendous advances in the past few years which has become an interesting area for researchers all over the globe. Digital signal processing algorithms are the key components in majority of image processing applications. Hence efficient implementations of these algorithms have become a vital part in realizing these applications in a real-time environment. Fast Fourier Transform (FFT) and Convolutional Neural Networks (CNNs) are the two key techniques used in many phases in digital image processing systems. There exist several hardware and software solutions for FFT implementation. Hardware implementations provide better

performance per watt and are more suited for real time embedded applications. Since FFT is a computationally intensive operation, realizing an *output reordered FFT* in hardware has become a challenging task. Most of the hardware accelerators for FFT do not perform output reordering, considering that the succeeding tasks or modules will take care of the reordering issues. Otherwise, in addition to FFT module, another reordering block will be present, which increases the hardware cost. The main objective of first part of this thesis is to develop a two dimensional (2D) FFT architecture which can perform data reordering without extra hardware cost. Reduction of intermediate memory, thereby reducing the latency and improving the FFT computation time and performance is also aimed in this thesis.

In the second part of this thesis, a high-performance Convolutional Neural Network (CNN) accelerator for image classification and segmentation is presented. Computational complexity of CNNs come from convolutional layers, which account for 90% of the computations in CNN Huimin Li *et al.* (2016). Deployment of CNNs on embedded systems with lower processing power and smaller power budget is a challenging task. Recent researches have shown the effectiveness of Field Programmable Gate Arrays (FPGA) as a hardware accelerator for CNNs which can deliver high performance at low power budgets. Majority of computations in CNNs involve 2D convolution. Algorithms like conventional or direct convolution, FFT based convolution and Winograd minimal filtering are generally used for implementing convolutional layers Shen *et al.* (2018); Lavin and Gray (2016). Suitable algorithm should be selected for realizing various convolution layers of a model, based on input feature size and filter size. Winograd minimal filtering based algorithm Winograd (1980) is the most efficient technique for calculating convolution for smaller filter sizes Ahmad and Pasha (2019). CNNs also consist of fully connected layers which are computed using General Element-wise Matrix Multiplication (GEMM). Most of the researchers use dedicated processing elements for performing Winograd convolutions. The drawback here, is the need for separate processing elements to accelerate convolutional layers and fully connected layers, which result in severe under-utilization of resources.

Winograd algorithm is suited only for convolutional layers with small kernel sizes. In case of the CNN model AlexNet Krizhevsky *et al.* (2012), first convolutional layer needs direct convolution and other convolutional layers can be efficiently accelerated

using Winograd convolution. If we can perform direct convolution, Winograd convolution and fully connected layer computation using the same compute resources, we can achieve significant savings in hardware. The main objective of the second part of this thesis is to develop an efficient hardware accelerator for CNNs, which are widely used in image classification tasks. This work presents a unified architecture named *Uni-WiG*, where both Winograd based convolution and GEMM can be accelerated using the same set of processing elements. This approach leads to efficient utilization of FPGA hardware resources while computing all layers in CNN.

1.2 Choosing a Solution

Selection of a suitable platform for implementation is a challenging task. Several technology solutions are available for implementation of signal processing tasks. Digital Signal Processors (DSP) are suited for implementing signal processing tasks, but they give very less throughput. Similar is the case with General Purpose Processors (GPP). Their power consumption is also very high. Hence software solutions for DSP implementations are not very efficient. Graphics Processing Units (GPU) also consume high power and are rarely used for embedded systems. Application Specific Integrated Circuit (ASIC) solutions give better performance with limited flexibility. Field Programmable Gate Arrays (FPGA) are flexible and gives high performance, with decent power consumption. FFT processors in the first work are implemented both in ASIC and in FPGA for evaluating the performance.

CNN is the most commonly used algorithm for image classification. Although DNN and CNN algorithms have been in existence over a long period of time, practical application of these algorithms started recently only. These algorithms are massively compute intensive and training the network can take multiple days to complete. Only with the deployment of GPUs, significant improvement has been achieved in bringing down the training time so as to be used in real life applications.

Most of the operations in CNNs can be reduced to matrix multiplication operations which consist of significant amount of parallelism. GPUs provide a platform for easily parallelizing matrix multiplication operation and thus improve the performance of

CNNs. However, GPU implementation of CNNs suffer from high power consumption which prevents its adoption in embedded applications Huimin Li *et al.* (2016). A dedicated hardware accelerator for CNN is required to achieve maximum performance at relatively lower power consumption. FPGA based implementations give significant performance improvement at acceptable power dissipation.

1.3 Overview of Fast Fourier Transforms

Over the past few decades signal processing domain has seen explosive growth in research and applications. Discrete Fourier Transform (DFT) is one of the essential part in digital signal processing and communication systems. Fast Fourier Transforms (FFTs) are widely used in image/video processing, Cognitive Radio, speech processing and various biomedical applications. FFT is a faster technique to perform Discrete Fourier Transforms (DFT) Proakis and Manolakis (1996). Cooley-Tukey is the common algorithm to find FFT of a sequence, since it reduces the complexity from O(N^2) to O(Nlog_2N) as compared to DFT J.W.Cooley and J.W.Tukey. (1965). One dimensional Discrete Fourier Transform for $x(n)$, an N point sequence, can be computed as in Equation (1.1) Proakis and Manolakis (1996)

$$X(k) = \sum_{n=0}^{N-1} x(n) W_N^{nk} \quad k = 0, 1, 2, ...N-1 \tag{1.1}$$

where W_N, the Twiddle Factor, denotes the N^{th} primitive root of unity, with its exponent evaluated to $modulo\ N$ and is introduced by Equation (1.2).

$$W_N = e^{-2\pi i/N} \tag{1.2}$$

An N-point inverse DFT (IDFT) can be calculated as given in Equation 1.3.

$$x(n) = \frac{1}{N} \sum_{k=0}^{N-1} X(k) W_N^{-nk} \quad n = 0, 1, 2, ...N-1 \tag{1.3}$$

IDFT requires few more logic like division and conjugate operation apart from the logic applied in DFT. Conventional image reconstruction uses inverse Fast Fourier Transform techniques. Inverse FFT (IFFT) is directly computed from FFT, and hence separate

architecture for IFFT is not discussed in research papers.

Multidimensional Fourier Transforms find wide range of application in signal processing systems. Multidimensional FFTs frequently find applications in video and image processing like reconstruction, medical imaging, radar signal processing etc. Image reconstruction is a key component in signal processing applications like medical imaging, computer vision, face recognition etc. Reconstruction of images is an important operation in most of these applications and involves complex computations in real time. Two Dimensional (2D) FFTs which are used for reconstructing image from raw data is computationally intensive and needs efficient implementations for catering to real time applications. These applications require large memory sizes to support large size images. Therefore it is necessary to have an FFT architecture which optimizes the memory but supports various image sizes, and provides the required throughput. For hardware mapping of the architecture, a 2D FFT is split into one dimensional FFTs which are performed over rows and then columns, generally referred as row-column decomposition. Equation (1.4) gives a relation between the hardware architecture parameters and system performance Lenart (2008).

$$f \times T = 2N^2 \times f_{rate} \tag{1.4}$$
$$Area \propto T$$

where, f denotes system frequency, T gives throughput, N is the size of the FFT, f_{rate} is the frame rate and $Area$ is the complexity of the architecture.

1.4 Overview of Convolutional Neural Networks

Deep Neural Networks (DNN) have revolutionized a variety of applications in varying domains like autonomous vehicles, weather forecasting, cancer detection, surveillance, traffic management, pattern recognition, image classification and speech recognition. Convolutional Neural Network (CNN) is a variant of DNN, where computations are performed as convolutions. CNN is the state-of-art technique for many machine learning tasks in image and video processing domain. CNN is the most commonly used algorithm for image classification and recognition. Most of the layers in CNN are not

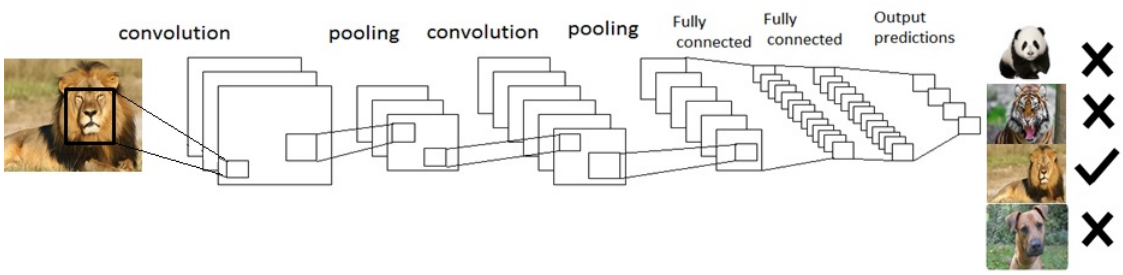

Figure 1.2: Convolutional Neural Network Model

fully connected layers as in other deep learning architectures. Instead, images are divided into tiles and convolution operation is performed to each tile with a filter (kernel) tile. The filter coefficients or weights are common for all tiles in a single image vector. Hence the number of weights for a convolution layer is considerably less than an equivalent fully connected layer. Typically CNNs consist of three main layers namely, convolutional layer (CONV) which performs feature extraction, max pooling (POOL) which is for sub sampling and fully connected (FC) layer for classification. A typical CNN is shown in Fig. 1.2.

Various CNN models were proposed from ImageNet Large-Scale Visual Recognition Challenge (ILSVRC) in the last few years, with varying number of CONV layers and filter sizes. Popular CNN models include AlexNet, which won ImageNet Challenge in 2012 Krizhevsky *et al.* (2012), Overfeat (ImageNet 2013) Sermanet *et al.* (2014), VGG Simonyan and Zisserman (2015) and GoogLeNet Szegedy *et al.* (2015) (ImageNet 2014) and ResNet (ImageNet 2015) He *et al.* (2016). AlexNet Krizhevsky *et al.* (2012), has five CONV layers and three FC layers. Overfeat Sermanet *et al.* (2014) also has five CONV and three FC layers, but with more number of filters. VGG has different models like VGG-11, VGG-13, VGG-16 and VGG-19, which has deeper layers Simonyan and Zisserman (2015). GoogleNet Szegedy *et al.* (2015) goes much more deeper, with 22 layers. ResNet He *et al.* (2016) has exceeded human level accuracy, with more than 34 layers.

Convolutions in a CNN are two dimensional (2D) operations where the input image is convolved with kernel, which is a multiply and accumulate (MAC) operation. One dimensional convolution which operates on two signals $I(x)$ and $f(x)$ is described as in (1.5).

$$f(x) * I(x) = \sum_{k=-\infty}^{+\infty} f(x)I(x-k) \tag{1.5}$$

A two dimensional (2D) convolution can be computed from one dimensional convolution as in (1.6).

$$f(x,y) * I(x,y) = \sum_{u=-\infty}^{+\infty} \sum_{v=-\infty}^{+\infty} f(u,v) I(x-u, k-v) \tag{1.6}$$

This is the generic mathematical formula for computing convolutions. Now consider an $M \times N$ input image (I) and $P \times Q$ kernel (F), which slides over the input image with a stride S to give $R \times C$ output (O). In 2D convolution, each value of the input image is multiplied with each element of the convolution kernel and the outputs are summed together. Pseudo code for two dimensional convolution is given in Algorithm 1.

Algorithm 1: Pseudo code for 2D Convolution

$M \times N$ is the input image size.
$P \times Q$ is the kernel size.
S is the stride of sliding window.
$R \times C$ is the output image size.
for $(r = 0; r < R; r++)$
for $(c = 0; c < C; c++)$
$O[r][c] += \sum_{i=0}^{P-1} \sum_{j=0}^{Q-1} F[i][j] * I[r+S+i][c+S+j]$

In CNNs, there will be multiple channels of input and kernels, and the outputs from each channel are added up together to get the final result. A summary of CONV layers of various popular CNN models is given in Table 1.1.

1.5 Contribution of the Thesis

Hardware acceleration of signal processing algorithms are on high demand for real-time embedded systems. The purpose of this work is to develop efficient hardware architecture for digital image processing systems. In the first part of thesis, an efficient two dimensional (2D) FFT architecture for image reconstruction, based on radix-4^3 algorithm has been presented. Here, a 64×64 point 2D FFT architecture based on radix-4^3 algorithm using a parallel unrolled radix-4^3 FFT as the basic block has been proposed. Proposed radix-4^3 architecture is a memory optimized parallel architecture which computes 64 point FFT, with least execution time. Here we use row-column decomposition of two radix-4^3 blocks to compute a 2D FFT. Following are the key

Table 1.1: CONV Layers of Various CNN Models

	AlexNet Krizhevsky *et al.* (2012)	Overfeat Sermanet *et al.* (2014)	VGG-16 Simonyan and Zisserman (2015)	GoogLenet Szegedy *et al.* (2015)	ResNet He *et al.* (2016)
$Feature_{in}$	224×224	231×231	224×224	224×224	224×224
CONV layers	5	5	13	57	53
Filters	11, 5, 3	11, 5, 3	3	1, 3, 5, 7	1, 3, 7
Stride	4, 1	4, 1	1	1, 2	1, 2
Parameters	2.3 M	16 M	14.7 M	6.0 M	23.5 M
MAC	666 M	2.67 G	15.3 G	14.3 G	3.86 G

contributions of first part of the thesis.

- A novel architecture for computing two dimensional FFT based on radix-4^3 algorithm
- An efficient output reordering technique using the six-bit control signal
- Memory reduction within 1D FFT and optimizing the intermediate memory between two 1D FFTs from N^2 to N.
- ASIC and FPGA implementations of the proposed architecture were performed. Comparison of proposed work with the state-of-art implementations and show significant improvement in computation time and the comparable area in terms of slice look-up tables (LUTs).

Proposed architecture has been implemented in UMC 65nm 1P10M CMOS technology with a maximum clock frequency of 312.5 MHz and area of $1.22mm^2$. The architecture is also implemented in Xilinx Virtex-7 FPGA and the results are compared with state-of-art implementations.

In the second part of this thesis, various techniques for implementation of convolution layers and fully connected layers in CNNs were discussed. Training and inference of different CNN models on various platforms were discussed. Based on the analysis of algorithms and hardware availability, a high performance CNN accelerator based on Winograd minimal filtering and general matrix multiplication (GEMM) is presented. A unified, blocked Winograd-GEMM architecture for accelerating CNNs on FPGA has been proposed, where, CONV layers and FC layers can be implemented using efficient

hardware utilization. An architecture, where both GEMM and Winograd filtering can be performed using the same processing element arrays has been presented. Block matrix multiplication scheme has been used for improving the efficiency and bandwidth. We propose a blocked version of Winograd minimal filtering algorithm for higher performance of the accelerator. Following are the key contributions of this work.

- Design space exploration of various algorithms for performing convolutions in CNN has been performed. Also, training and inference time of various layers in the feed forward network of AlexNet, VGG-16 and ResNet on platforms like GPP, GPU, server and Jetson TX2 board were performed.
- A unified architecture for performing general element-wise matrix multiplication (GEMM) as well as Winograd filtering algorithm using the same array of processing elements (PEs) has been proposed. Such a unified architecture gives the most efficient implementation for CONV layers irrespective of the filter sizes and also for FC layers. Winograd filtering is transformed to a GEMM operation as proposed in Lavin and Gray (2016) so that PEs for GEMM can be reused for Winograd algorithm.
- A novel algorithm is proposed, which transforms Winograd minimal filtering algorithm into blocked general element-wise matrix multiplication which targets optimal utilization of BRAMs and DDR bandwidth.
- An analytical model for estimating performance and BRAM usage has also been proposed, using which appropriate tile sizes for each layer can be identified, which can be used to optimize the tile size for a given convolution layer.
- The proposed architecture has been used to accelerate popular CNN models like AlexNet, VGG-16 and ResNet-18 models on Xilinx XC7VX690T FPGA. Comparison of various CNN implementations has been performed to show the efficiency of proposed accelerator.

To maximize the resource utilization, we have designed the architecture of the accelerator, a reusable one. Conventionally, CNNs like AlexNet, VGGNet etc follw a layer-by-layer sequential operation. A layer-by-layer computation flow of CNN has been designed in this work. Proposed architecture has been implemented on Xilinx XC7VX690T FPGA using both single precision floating point arithmetic and 16-bit fixed point arithmetic. Since various layers in a CNN are to be sequentially exectuted even in an end-to-end implementation, FPGA resources remain the same.

1.6 Organization of the Thesis

The main theme of this research work is to develop efficient architectures for high performance image processing applications, in a real-time scenario.

Chapter 2 covers various existing FFT algorithms and architectures in literature. Hardware complexity analysis of different architectures and output reordering techniques are also discussed in this chapter. Secondly, a survey of various CNN accelerators implemented using GPU, ASIC and FPGA are carried out and explored different CNN implementations. A study of CNN architectures based on different convolution algorithms has also been conducted.

Chapter 3 discusses radix-4^3 based two dimensional FFT architecture. Here, a 2D FFT architecture based on cascade of two unrolled parallel-pipelined radix-4^3 FFT has been discussed. A 64$\times$64 point 2D FFT architecture is implemented in ASIC and FPGA and comparison has been performed with various existing 2D FFT architectures to show the effectiveness of our architecture.

In Chapter 4, an exploration of various convolution algorithms are performed. Merits and demerits of various convolution schemes for CNN are discussed in this chapter. Need for a unified Winograd-GEMM architecture is also briefly explained. In this chapter, training and inference of popular network models in CPU, GPU and Jetson TX2 platforms are also presented. An FPGA based architecture for accelerating AlexNet CNN model is also discussed in this chapter.

In Chapter 5, blocked version of Winograd minimal filtering algorithm is presented. Unified Winograd-GEMM architecture for accelerating CNNs on FPGA is proposed in this chapter. Blocking techniques for improving the performance are also discussed. An analytical model for estimating performance and BRAM usage has also been proposed, using which appropriate tile sizes for each layer can be identified. Floating point implementation of AlexNet model on FPGA is described here and compared with the existing architecture.

In Chapter 6, FPGA implementation details of proposed CNN architecture in fixed point arithmetic are explained and the results are discussed. Fixed point implementation of AlexNet, VGG-16 and ResNet-18 models are performed and the results are

evaluated. Comparisons with state-of-art implementations are also done in this chapter.

Chapter 7 presents the conclusions and future directions of this research work.

CHAPTER 2

Background and Related Works

In this chapter, hardware implementations of various two dimensional (2D) FFT architectures are described. Hardware and software accelerators for convolutional neural networks, from various research groups are also discussed in this chapter.

2.1 FFT Algorithms and Architectures

Cooley-Tukey is the common algorithm to find DFT of a sequence. DFT can be computed in a faster method using FFT, since it reduces the complexity from O(N^2) to O(Nlog_2N) as compared to DFT J.W.Cooley and J.W.Tukey. (1965). One dimensional Discrete Fourier Transform for $x(n)$, an N point sequence, can be computed as Proakis and Manolakis (1996)

$$X(k) = \sum_{n=0}^{N-1} x(n) W_N^{nk} \quad k = 0, 1, 2, ...N-1 \tag{2.1}$$

where W_N, the Twiddle Factor, denotes the N^{th} primitive root of unity, with its exponent evaluated to $modulo\ N$ and is introduced by the Equation (2.2),

$$W_N = e^{-2\pi i/N} \tag{2.2}$$

Each of the coefficients in a DFT requires N complex multiplications and $N-1$ complex additions. Hence, for an N-point DFT, N^2 complex multiplications and $N(N-1)$ complex additions are required. An N-point inverse DFT (IDFT) can be calculated as given in Equation (2.3),

$$x(n) = \frac{1}{N} \sum_{k=0}^{N-1} X(k) W_N^{-nk} \quad n = 0, 1, 2, ...N-1 \tag{2.3}$$

IDFT requires few more logic like division and conjugate operation apart from the logic applied in DFT.

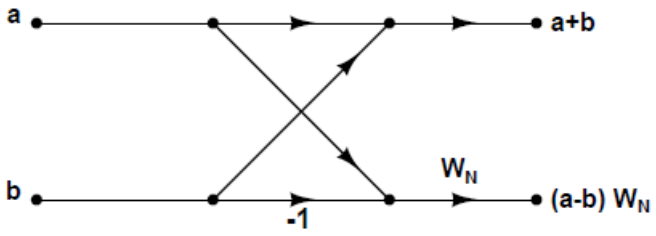

Figure 2.1: Butterfly diagram of Radix-2 DIF FFT

2.1.1 Radix-2 Cooley-Tukey FFT

Most of the applications which uses Cooley-Tukey FFT, has a block-length N, which are powers of two or four. If the block-length 2^m is factorized as $2 \times 2^{m-1}$, then it is called radix-2 FFT Blahut (2010). The decomposition size of an FFT is denoted using $radix$. Computation of FFTs can be done either in decimation-in-frequency (DIF) or decimation-in-time (DIT) technique. In DIT technique, the N input sequence is divided into two $\frac{N}{2}$ sequences, which are even and odd sequences of the input, $x(n)$, and their DFTs are computed as shown in Equation (2.4).

$$X_1(k) = \sum_{r=0}^{(N/2)-1} x(2r)W_{N/2}^{rk} \quad k = 0, 1, 2, ...\frac{N}{2} - 1 \quad r = 0, 1, 2, ...\frac{N}{2} - 1$$
$$X_2(k) = \sum_{r=0}^{(N/2)-1} x(2r+1)W_{N/2}^{rk} \quad k = 0, 1, 2, ...\frac{N}{2} - 1 \tag{2.4}$$

Here the inputs are in bit-reversed order, while outputs are in natural order. DIT follows in-place computation, where the output values are stored in the same registers where the previous input values were stored. This saves significant memory for computation.

In DIF technique, the output sequence $X(K)$ is divided into odd and even sequences as shown in Equation (2.5). Here the output is bit-reversed and input is in natural order.

$$X(2k) = \sum_{n=0}^{(N/2)-1} [x(n) + x(n + \frac{N}{2})]W_{N/2}^{nk} \quad k = 0, 1, 2, ...\frac{N}{2} - 1$$
$$X(2k+1) = \sum_{n=0}^{(N/2)-1} [x(n) - x(n + \frac{N}{2})]W_N^{nk}W_{N/2}^{nk} \quad k = 0, 1, 2, ...\frac{N}{2} - 1 \tag{2.5}$$

Butterfly diagram of radix-2 DIF technique is shown in Fig. 2.1. The computational complexity of both DIT and DIF are same and both follow in-place computation.

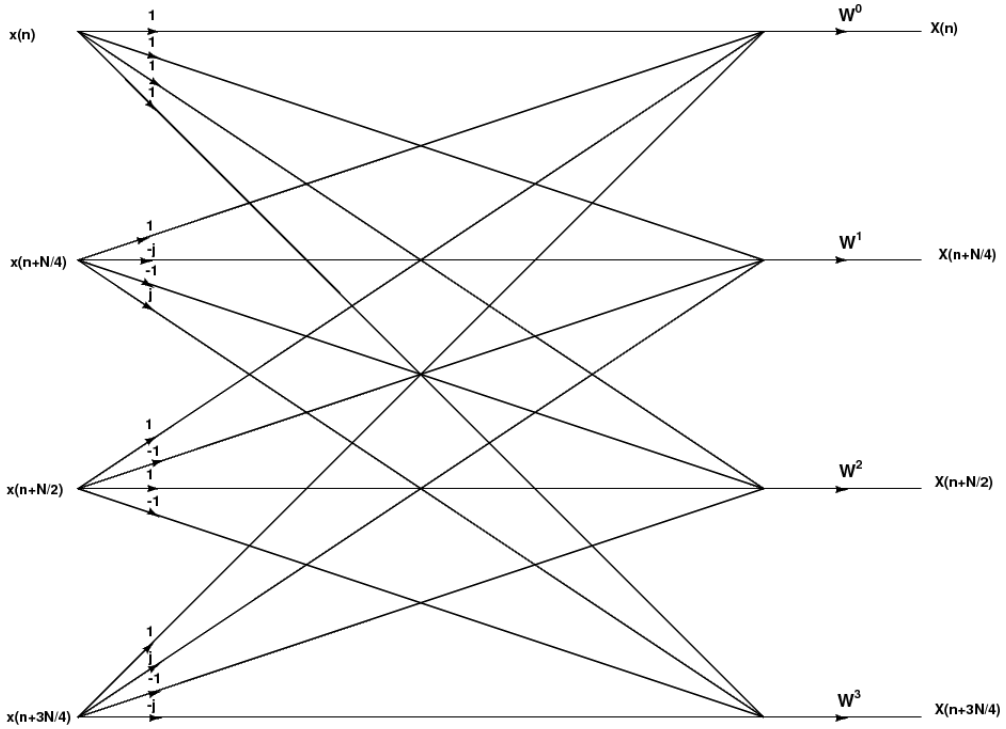

Figure 2.2: Radix-4 DIF FFT Butterfly

2.1.2 Radix-4 FFT

If the block-length 4^m is factorized as $4 \times 4^{m-1}$, then it is called radix-4 FFT Blahut (2010). Here an N-point sequence is decomposed into four smaller sequences as given in Equation (2.6).

$$
\begin{aligned}
X(4k) &= \sum_{n=0}^{(N/4)-1} [x(n) + x(n+\frac{N}{4}) + x(n+\frac{N}{2}) + x(n+3\frac{N}{4})]W_{N/4}^{nk} \\
X(4k+1) &= \sum_{n=0}^{(N/4)-1} [x(n) - jx(n+\frac{N}{4}) - x(n+\frac{N}{2}) + jx(n+3\frac{N}{4})]W_N^n W_{N/4}^{nk} \\
X(4k+2) &= \sum_{n=0}^{(N/4)-1} [x(n) - x(n+\frac{N}{4}) + x(n+\frac{N}{2}) - x(n+3\frac{N}{4})]W_N^{2n} W_{N/4}^{nk} \\
X(4k+3) &= \sum_{n=0}^{(N/4)-1} [x(n) + jx(n+\frac{N}{4}) - x(n+\frac{N}{2}) - jx(n+3\frac{N}{4})]W_N^{3n} W_{N/4}^{nk}
\end{aligned} \tag{2.6}
$$

Radix-4 butterfly diagram is given in Fig. 2.2.

2.1.3 2D FFT

A two dimensional Fourier transform can be computed from one dimensional FFT. An N×N 2D FFT can be computed from 2N one dimensional FFT computations, where N is the sequence length. A two dimensional DFT of size N×N, with inputs x(i_1, i_2) is given as,

$$\begin{aligned} y(k_1,k_2) &= \sum_{i_1=0}^{N-1}\sum_{i_2=0}^{N-1} x(i_1,i_2)W_N^{k_1 i_1 + k_2 i_2} \\ where \quad & k_1, k_2 = 0, 1, 2, ...N-1 \end{aligned} \tag{2.7}$$

Using two, one dimensional DFTs, a 2D DFT can be performed based on Row-Column (RC) decomposition algorithm, as given in Equation (2.8).

$$\begin{aligned} X(k_1,i_2) &= \sum_{i_1=0}^{N-1} x(i_1,i_2)W_N^{k_1 i_1} \\ where \quad & k_1 = 0, 1, 2, ...N-1 \\ Y(k_1,k_2) &= \sum_{i_2=0}^{N-1} X(k_1,i_2)W_N^{k_2 i_2} \\ where \quad & k_2 = 0, 1, 2, ...N-1 \end{aligned} \tag{2.8}$$

Various FFT algorithms and architectures are available in research papers. FFT architectures and algorithms are chosen based on power, area and throughput requirements Cortes *et al.* (2006) Manolopoulos *et al.* (2007). Different pipelined based and memory based architectures are proposed by several research groups over decades Li (2008)Gold and R.Rabiner (1975)Saponara *et al.* (2012). Pipelined architecture generally gives more area overhead and consumes more power. Pipeline based architectures based on linear decomposition of radix-2 are available in Shousheng He and Torkelson (1998*a*,*b*); Jeesung Lee *et al.* (2006); Cho and Lee (2013); Oh and Lim (2005); Lin *et al.* (2006). These are either Single-path Delay Feedback (SDF) or Multi-path Delay Commutator (MDC) architectures. In Shousheng He and Torkelson (1996) and Yu-Wei Lin *et al.* (2005) different schemes of SDF and MDC pipelined architectures are discussed. In Yin *et al.* (2016), a two-parallel real FFT based on radix-2 is presented. Their architecture is a pipelined architecture with latency reduction and less hardware complexity. Many more pipelined FFT architectures were proposed in past years Maharatna *et al.*

(2004); Chin-Teng Lin *et al.* (2006*a*); Chen *et al.* (2008); Chih-Peng Fan *et al.* (2006); Li *et al.* (2010); Yu *et al.* (2011); Tang *et al.* (2010); Reisis and Vlassopoulos (2008); Huggett *et al.* (2005); Chin-Teng Lin *et al.* (2006*b*); Hasan *et al.* (2003); Cheng and Parhi (2007); Lin and Yu (2007); Yu-Wei Lin *et al.* (2004); Liao *et al.* (2018). Radix-2 systolic FFT architecture is presented in Swartzlander (2007). Authors in Garrido *et al.* (2016) proposes a serial commutator Fast Fourier Transform, with proper data management, to simplify the rotator, and minimize the arithmetic complexity. Memory based designs for 1D FFT are presented in Cohen (1976); Hsiao *et al.* (2010); Tsai and Lin (2011); Babionitakis *et al.* (2010*b*); Ma *et al.* (2015). Memory based architectures consume more area and less power compared to pipelined FFT, for long size FFTs Xia *et al.* (2016). Several memory access schemes for 1D FFT are proposed in Xing *et al.* (2017); Huang and Chen (2016); Xia *et al.* (2016). In Shih *et al.* (2018), a reconfigurable FFT architecture is proposed, which supports 46 different modes for LTE applications. Various reconfigurable FFTs are also proposed in research papers. FFT of size upto 256 K using radix-4^3 algorithm is presented in Babionitakis *et al.* (2010*b*). Yang et al. propose a multiple stream MDC architecture which computes 128 - 2 K point FFT and achieve memory reduction by an efficient memory scheduling mechanism Yang *et al.* (2013*a*). FFT processor to compute 16 - 1 K point FFTs with reconfigurable address generation block is presented in Yutian Zhao *et al.* (2005). A variable length FFT processor ranging from 512-8 K points, using mixed radix SDF architecture is presented in S.-S. Wang and C.-S. Li (2008). Multi-processor ring based FFT architecture in Guichang Zhong *et al.* (2006) and CORDIC based cached memory architecture in J.-C. Kuo *et al.* (2003), have the disadvantage of taking higher number of clock cycles per FFT computation, resulting in higher execution time and lower throughput. FFT processor in Tang *et al.* (2012) is based on Multi-path Delay Feed-back architecture and supports FFT sizes from 64 - 1 K point. Although these architectures support reconfigurability, they are not scalable in terms of FFT sizes.

Different architectures for 2D FFT has been proposed by various research groups. Software solutions like FFTW Frigo and Johnson (1998), Spiral Puschel *et al.* (2005), gives high performance, but consume more power and are not suitable for embedded system applications. Various hardware solutions such as FFT processor ASICs and FPGA implementations are available in D'Alberto *et al.* (2007), T. (2001), Lenart *et al.*

(2008), Kim *et al.* (2009). A new two dimensional decomposition algorithm is implemented in Yu *et al.* (2011). Here the data is divided into several sub blocks and butterfly operation is performed between these sub blocks. 2D FFT of each of these sub block is computed, where the size of sub block is determined using available FPGA resources. In Chen and Prasanna (2014), several energy optimization techniques are used to develop an efficient and high throughput 2D FFT architecture on FPGA. In Lenart *et al.* (2008) 2D FFT for digital holographic imaging is presented. They have used mixed radices, ie., radix-2 and radix-2^2 butterfly units in the 1D FFT. A transpose unit is required after the first 1D FFT block, to compute 2D FFT.

In Akin *et al.* (2012) the issue of off-chip memory bandwidth is addressed. An ILUT based 1D FFT is used to develop a 2D FFT on FPGA in Kee *et al.* (2009). In Akin *et al.* (2012) and Kee *et al.* (2009), N^2 intermediate memory is required after the first phase of FFT. But for the architecture proposed in this thesis, the intermediate memory requirement after performing a 1D FFT reduces to N. Moreover, within a 1D FFT block, there is significant reduction in intermediate RAM.

Output reordering is a major functional block when designing FFT architecture. The purpose of reordering is to convert the non-natural FFT output to natural order. Extensive research papers are available for design and implementation of FFT architecture. But only few papers discuss about output reordering techniques in FFT and its complexity. In OFDM based applications, output reordering is taken care by the preceding block. Hence most of the research papers of FFT focus on design of FFT core alone, rather than bit reversal. An output reordering logic is presented in Chidambaram (2005). But the logic block becomes complex with large sizes of N. In memory based designs, efficient schemes for conflict free addressing is a challenging task Babionitakis *et al.* (2010*b*). Address schemes used in memory based FFT architecture will produce the output in reordered form Sorokin and Takala (2011); Hsiang-Sheng Hu *et al.* (2009). For SDF and MDC architectures, buffers are required at the output, for data reordering. Output reordering in parallel pipelined FFT architectures are highly complex and are discussed in Garrido *et al.* (2011); Yang *et al.* (2013*b*). In F. Kristensen (2003), two buffers each of size N are required to store even and odd FFT outputs for reordering. In Chakraborty and Chakrabarti (2008) single memory of size N is used for bit reversal of even and odd outputs alternatively. In MDF (Multi path Delay Feedback) pipelined

architectures 3×N memory will be used for output reordering. ie., 2×N will be required in output side and N will be used within the processing elements Huang and Chen (2012). In Garrido *et al.* (2011) a framework for performing bit reversal in output data is proposed. They use minimum number of registers for calculating bit reversal and new circuits were designed for output reordering in different radices. An algorithm for performing bit reversal is discussed in Marti-Puig and Reig Bolano (2009). But the algorithm is applicable only if transform length is a power of four. In most of these architectures, either a dedicated hardware or significant memory is required for reordering. Our architecture does not require buffers for output reordering. This is possible using the six-bit control signal used in our FFT.

2.2 Convolutional Neural Networks

Neuromorphic computing, which is based on brain-inspired computing has become a contrast architecture to von Neumann architecture. Neuromorphic architectures are highly appreciated for their parallel and fast processing capabilities, and also for their low power requirement in the context of Dennard scaling Esmaeilzadeh *et al.* (2011), as Moore's law is about to end and Dennard scaling demands for more power. Neuromorphic architecture are suitable candidates for implementation of machine learning algorithms. Researchers work on developing new network models that mimic biological brain, along-with developing new machine learning algorithms. Machine learning is a sub-field of artificial intelligence, where the computers have learning abilities without being programmed explicitly.

In brain-inspired computing, an algorithm will take some of the functionalities in a way, in which the brain functions. Computations in brain are centered on *neurons*. There are billions of neurons in human brain, which are interconnected each other. Each of the neuron is a processing unit, which receives many real inputs and produces a single real output. The inputs enter a neuron through *dendrites* and leaves through *axons* as shown in Fig.2.3. These input-output signals are named as *activations*. Axon is connected to the dendrite through a *synapse*. As seen from Fig. 2.3, the input signal x_i is scaled by the *weight* w_i, when it is passed through the synapse. Learning factor of the brain is indicated as the $weight$, and various outputs are generated depending

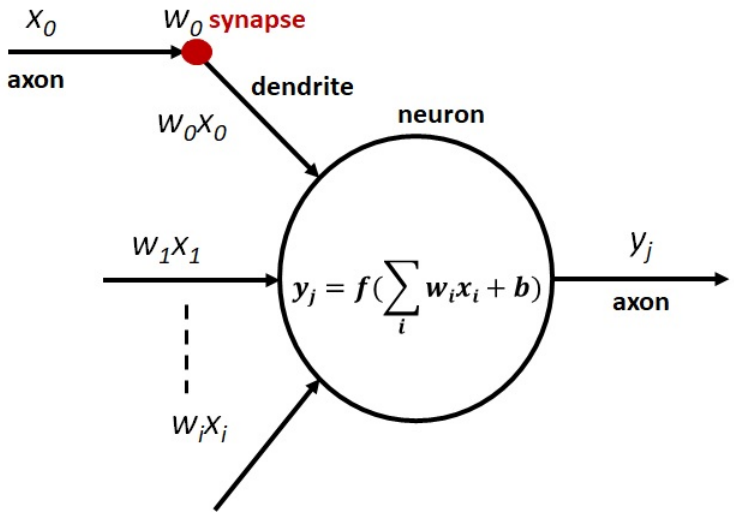

Figure 2.3: Neuron in the brain

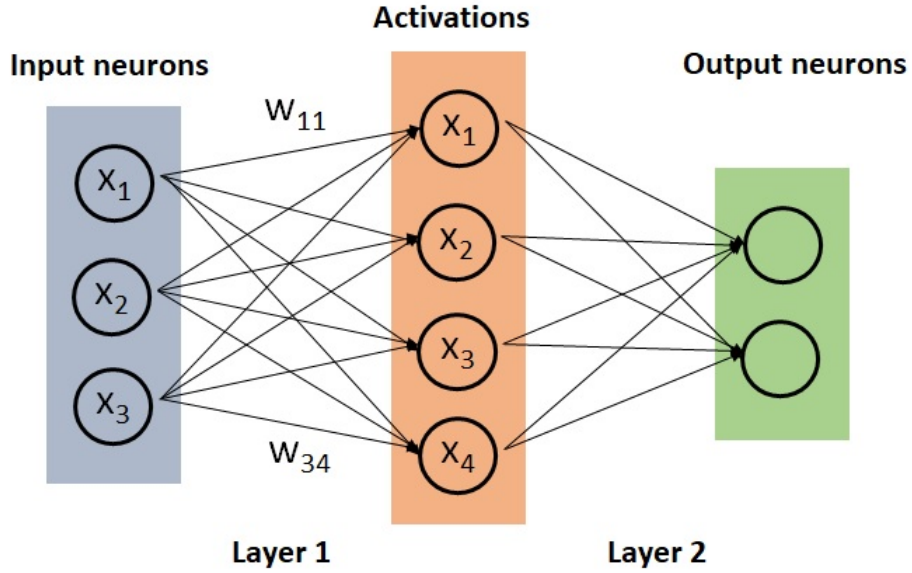

Figure 2.4: Typical Neural Network

on the variations in these weights. In general, neural networks involves computation of a neuron as the weighted sum of all the input values. This is given in Equation (2.9) where b denotes $bias$,

$$Y_j = f(\sum_i w_i x_i + b) \tag{2.9}$$

Fig. 2.4 shows a simple neural network, with three input neurons receiving three different values and propagating it to the middle layer, which is called *hidden layer*. The weighted sums from many hidden layers are given to output neurons, which are the outputs of a neural network. If the neural network has more than one hidden layer, then it is generally referred to as *deep neural network* and the area in which they are used is

deep learning.

Recent breakthrough in the advancement of deep learning has led to deployment of Deep Neural Networks (DNNs) for most of the artificial intelligent tasks. DNNs have become a dominant approach for a variety of machine learning tasks like object detection, image classification and recognition, autonomous automobiles, autonomous speech processing and so on. DNNs are inspired by the functionality and behavior of human brain, where the computations are performed by 'neurons'. *Training* of a network refers to a process where an algorithm will learn to approach a problem. Training involves selection of *weights* of the network so as to produce the desired output. Running a network with these computed weights is called as *inference*. In other words, inference is referred as execution of the feed forward path of a neural network. Training can be performed using *supervised learning* (with labeled classes), *unsupervised learning* (without labeled classes), *semi-supervised learning* (a subset being labeled) or *reinforcement learning* (where the state of environment is known, to maximize the output) techniques. Another technique used is *transfer learning* for better precision, where the network is learned from previously adjusted weights.

Typically, Artificial Neural Networks (ANN) are composed of Multi-Layer Perceptrons (MLP), which are fully connected layers, where all output neurons connected to all input neurons. Here, the weights (coefficients) to be stored is significantly high and the computation is also increased. One way to reduce the huge memory requirement is to set some of the weights to zero, thereby removing some connections, without losing the precision. Another way to reduce the memory requirement is *weight sharing*. Here same set of weights can be used to calculate the output. An efficient technique for reducing the memory and computation is to use convolutions as computation instead of fully connected layers. Here weight sharing and windowing techniques are used for reducing the complexity.

Convolutional neural networks (CNNs), a variant of DNN, is a promising solution to wide range of applications like robotics, weather forecasting, video surveillance and applications in the field of medicine like cancer detection etc. CNNs out-perform human intelligence in many image classification tasks. CNNs are composed of different layers namely, convolutional layer (referred as CONV), pooling layer (POOL), normalization, ReLU (recti-linear unit) and fully connected (referred as FC) layers. Convolution layer

extracts various features of the input image, while sub-sampling is done by POOL layers. It is followed by FC layers which performs classification of the feature map. CONV layer convolves the input feature map with filter and produces output feature map. Filter weights are calculated during the training process. Pooling layer is done either with MAX or AVERAGE operation. Each neuron in an FC layer is connected to all other neurons in the previous layer, which increases the number of weights in FC layer and makes it a memory intensive layer. But in CONV layer, computations are higher and filter coefficient storage requirement is less compared to FC layer.

Non-linearity is introduced after CONV layers or FC layers using activation functions. Sigmoid and Hyperbolic tangent functions were considered as traditional activation functions, whereas Rectified Linear Unit (ReLU), Leaky ReLU, Exponential ReLU were referred as modern activation functions. Pooling combines a set of values into a smaller set of values. Pooling is done on non-overlapping blocks of values. A stride greater than one is used to minimize the dimension of feature map. In Max pooling, maximum value in the block is selected where as in Average pooling, average value of the block is taken into account. Batch normalization is performed at the layer input activations so as to have zero mean and standard deviation as unity. This normalized value is then scaled and shifted. Drop out is also a regularizer, which is used in many CNN models. Sometimes batch normalization allows drop outs to be omitted because of the presence of noise in statistical estimations in the variables. Performance increases when the model is regularized well using drop out.

Convolutions in CNN are 2D operations, where filter weights are multiplied with each element in the feature map and are summed together. ie., it involves multiply and accumulate (MAC) operations of filter weights and input feature map. Each element in the input feature and filter is multiplied and accumulated to get the corresponding element in the output feature map. Consider M channels of $K \times K$ filter, and a total of N such filters. Also consider M channels of $H \times W$ input feature maps, for performing convolution. Each filter performs convolution on the input feature map with a stride of S, and generates the output feature map, Y. Element (u, v) in n^{th} filter, m^{th} channel is represented as $G_{n,m,u,v}$ and input element (x, y) in m^{th} channel is denoted as $D_{i,m,x,y}$.

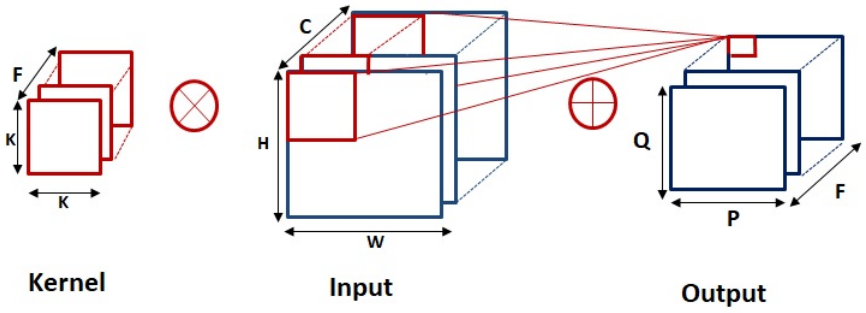

Figure 2.5: Convolution Operation

Convolution is given by the formula as in (2.10),

$$Y_{i,n,x,y} = \sum_{m=1}^{M} \sum_{u=1}^{K} \sum_{v=1}^{K} D_{i,m,x+u+S,y+v+S} \times G_{n,m,u,v} \tag{2.10}$$

Entire output of convolving the image with filter can be written as in (2.11),

$$Y_{i,n} = \sum_{m=1}^{M} D_{i,m} * G_{n,m} \tag{2.11}$$

where * denotes correlation.

Convolution operation is shown in Fig. 2.5.

Various convolution architectures were proposed by different research groups over the last few years. GPU based 2D convolver proposed in Iandola *et al.* (2013) reduces memory communication by reorganizing the convolution algorithm and working on few threads. In Ciobanu and Gaydadjiev (2013), a vectorized separable 2D convolver optimized for polymorphic register files, for high performance, is proposed and implemented in GPU. GPU implementations give high performance but they are not suitable for embedded applications due to their high power consumption. Various FPGA based convolvers are presented in Perri *et al.* (2005); Aguilar-González *et al.* (2019); Rao *et al.* (2016); Chang and Sha (2017); Russo *et al.* (2012); StrÃűm (2016); Mohammad and Agaian (2009). In Perri *et al.* (2005) a reconfigurable architecture for 2D convolution is presented, which uses 8-bit and 16-bit fixed point arithmetic. Here even though high performance is achieved, precision is degraded. CAPH language has been used in the convolver design in Aguilar-González *et al.* (2019), to reduce the design time. The design flow is a data-flow model, where tokens are interchanged through a network. Here also 16-bits are taken for representation. Reconfigurable one dimensional convolution

architecture presented in Rao *et al.* (2016) has been implemented in both ASIC and FPGA. They have used separable convolutions for implementation.

2D convolution for CNN presented in Chang and Sha (2017), uses singular value decomposition approximation method, which in turn transformed to conventional row and column technique. Russo *et al.* (2012) compare GPU and FPGA implementations, which concludes that GPU achieves high performance with high power consumption, whereas FPGA can operate in small boards with less power consumption. Convolution architecture for CNN proposed by Wang *et al.* (2016) uses parallel finite impulse response algorithm for convolution and implemented in 90nm CMOS process. StrÃűm (2016) presents an architecture with 16 parallel convolutions and implemented in FPGA. Mohammad and Agaian (2009) also uses fixed point arithmetic with less precision for implementation. The above mentioned papers follow conventional technique for computation of convolutions. This uses more hardware resources as the number of multiply and accumulate (MAC) operations are very high in conventional technique.

Several GPU accelerators for Convolutional Neural Networks are available in various research papers. In Li *et al.* (2016), performance evaluation of CNN on Tesla K40c, for various kind of implementations is presented. Training period of AlexNet models are evaluated for various frameworks using GPU in Kim *et al.* (2017). FFT based CNN implementations using Torch 7 environment were performed on GeForce GTX Titan GPU in Mathieu *et al.* (2013). GPU accelerators for CNN are also presented in Chakradhar *et al.* (2010); Krizhevsky *et al.* (2012); Szegedy *et al.* (2015); Gupta *et al.* (2015). Many open-source frameworks for deep learning like Caffe Jia *et al.* (2014), Torch Collobert *et al.* (2011) Theano Bergstra *et al.* (2010) etc. were also introduced which can use CUDA programming. Specific libraries like cuDNN Chetlur *et al.* (2014) are also available for accelerating CNNs.

ASIC based implementations of CNN are presented in Tu *et al.* (2017); Chen *et al.* (2014, 2016). Authors in Chen *et al.* (2014) propose an architecture with interconnected nodes which contains 64 nodes. ASIC based accelerator proposed by Chen *et al.* (2014), gives high performance but with limitations in flexibility. A reconfigurable and scalable CNN architecture is presented in Tu *et al.* (2017) with reconfigurable deep neural architecture (DNA). Fast FIR Algorithm (FFA) has been introduced in Wang *et al.* (2018*a*) to develop a reconfigurable convolution core, with high power efficiency.

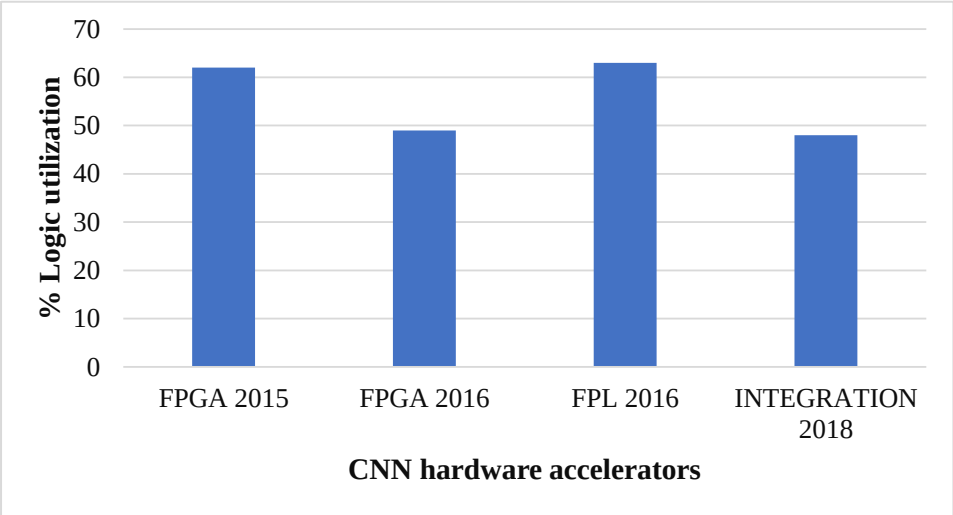

Figure 2.6: Hardware Utilization for AlexNet implementation

FPGA based implementations of CNN are available from various research groups Ma *et al.* (2018); Guo *et al.* (2018); Zeng *et al.* (2018); Motamedi *et al.* (2016); Lu *et al.* (2017); Zhang *et al.* (2015); Huimin Li *et al.* (2016); Yufei Ma *et al.* (2016); Ma *et al.* (2017*a*); Guan *et al.* (2017); Podili *et al.* (2017); Ma *et al.* (2018); Zhang *et al.* (2016); Suda *et al.* (2016); Xiao *et al.* (2017); Yu *et al.* (2017); Qiu *et al.* (2016); Peemen *et al.* (2013); Wang *et al.* (2018*b*); Aydonat *et al.* (2017); Zhang *et al.* (2016); Nguyen *et al.* (2019); Lian *et al.* (2019); Kyriakos *et al.* (2019); Yu *et al.* (2019). FPGA based implementations give significant performance improvement at acceptable power dissipation. Implementation of CNN models in hardware platforms is a challenging task, because of the extreme compute complexity of convolution layers and memory constraints raised by fully connected layers. Machine learning tasks in emerging applications demand higher accuracy levels which will lead to more complex networks. Various hardware architectures for CNN are proposed in recent years. Hardware under-utilization is a major issue in most of the implementations, which is unacceptable for applications involving embedded systems. Fig. 2.6 shows the hardware utilization for convolutional layers in some of the FPGA implementations in literature. Fig. 2.6 gives the % utilization of logic used in AlexNet CNN accelerators in research papers published in FPGA 2016 Zhang *et al.* (2015), FPL 2016 Huimin Li *et al.* (2016), FPGA 2016 Qiu *et al.* (2016) and Integration 2018 Ma *et al.* (2018).

Most of the researches in CNN implementations try to effectively exploit the abundant parallelism inherent in the matrix-matrix operations that dominate the computations in CONV and FC layers. In Huimin Li *et al.* (2016), a method for utilizing the parallelism of each layer and improving the bandwidth utilization of FC layers is presented. In Motamedi *et al.* (2016), the authors present an FPGA based accelerator for CNNs using the direct convolution algorithm. Here the architecture tries to optimize the performance by exploiting various levels of parallelism available in the convolution algorithm. A design space exploration algorithm for this purpose has also been presented. Hardware CNN accelerator in Nguyen *et al.* (2019) implements YOLO (you-only-look-once) on FPGA with high throughput and power efficiency. Authors in Lee *et al.* (2019) propose a novel method called double MAC to improve the computation throughput of CNN, by packing two MAC operations in one DSP slice of FPGA. AlexNet and VGG models were implemented using this technique, without much accuracy loss. An FPGA based CNN accelerator which uses block-floating point has been presented in Lian *et al.* (2019). Here, the design is based on shifting operations and achieved 50% reduction in memory and bandwidth requirements. Authors in Lian *et al.* (2019) have also proposed an analytical model for error propagation of CNN, based on their design. In Hailesellasie *et al.* (2018), a technique for reducing the parameters of CNN by removing all fully connected layers is presented. In Lu *et al.* (2019), architecture for sparse CNNs on FPGA has been proposed, where the non-zero weights are compressed into arrays and convolutions were performed as element-wise matrix multiplications. A hardware/software co-design approach for inference of complex CNNs is presented in Xia *et al.* (2019), which is a part of PAI, a Platform for Artificial Intelligence. A digital electronic and analog photonic CNN architecture is presented in Bangari *et al.* (2020), where computations are reduced using photonic CNNs. Here, silicon photonics devices are used to perform convolutions. The CNN architecture has been applied for digit recognition using MNIST database. Dedicated blocks for convolution, fully connected layer and pooling layer are presented in Kyriakos *et al.* (2019). An overlay domain specific unit is proposed in Yu *et al.* (2019), which is software programmable and gives faster compilation time for CNN models, which are taken from deep learning frameworks like tensorflow. FPGA implementations give better power efficiency for VGG and YOLO networks. In Wu *et al.* (2019), dedicated processing engines are implemented for point-wise convolution and depth-wise convolution, to improve the

performance. They have implemented MobileNet on their architecture for object detection. A flexible training hardware is designed in Kolala Venkataramanaiah *et al.* (2019), where an RTL compiler has been developed to generate FPGA synthesizable RTL.

Fast Fourier Transform (FFT) based convolution technique is another method for performing convolution operations in CNN. FFT based convolution gives reduced computational complexity compared to conventional method. However FFT based convolution techniques are useful only for larger filter sizes Podili *et al.* (2017). But optimizations in FFT based convolutions can result in improved performance,even for small filter sizes Zeng *et al.* (2018). FFT Overlap and Add technique method is used to implement ResNet on a many core architecture in Abtahi *et al.* (2017). Here filter size is 3×3 for all layers. Comparisons with direct convolution show significant improvement in performance for layers with smaller input sizes. This is because for small filter sizes, FFT based convolutions are inefficient if there is mismatch between the filter size and input size as reported in Podili *et al.* (2017). In Zhuge *et al.* (2018), authors have compared FFT and Winograd based convolution techniques for varying filter sizes and conclude that FFT based technique is more suited for larger filter sizes. In Zeng *et al.* (2018), authors improve the performance of FFT based convolution by employing Concatenate-and-Pad (CaP) and frequency domain loop tiling technique.

Another method of computing convolution involves Winograd minimal filtering algorithm Winograd (1980). Winograd reduces the number of multiplications in convolution significantly. Complexity of Winograd algorithm depends on the tile size chosen for implementation. Choosing a larger tile size can reduce the complexity at the cost of reduced precision. For applications which require less precision, we can use large tile sizes. Tile sizes in Winograd algorithms are typically represented as $F(m \times m, r \times r)$ which denotes that $m \times m$ outputs are computed using an $r \times r$ tap filter. In Podili *et al.* (2017); DiCecco *et al.* (2016); Lavin and Gray (2016) tile size of $F(2 \times 2, 3 \times 3)$ is chosen for implementing CONV layers. Tile size of $F(4 \times 4, 3 \times 3)$ has been used in Xiao *et al.* (2017); Ahmad and Pasha (2019); Lu *et al.* (2017); Yu *et al.* (2017); Zhuge *et al.* (2018) while Zhuge *et al.* (2018) uses $F(2 \times 2, 3 \times 3)$ also. Architecture in Podili *et al.* (2017) was implemented for accelerating VGG-16 using Winograd algorithm where as Lu *et al.* (2017) implemented both AlexNet and VGG-16. A hybrid approach is to use different methods for various CONV layers depending on the filter size.

Various research groups have proposed Winograd filtering algorithm based hardware accelerators for CNN Xiao *et al.* (2017); Podili *et al.* (2017); Lu *et al.* (2017); Yu *et al.* (2017); Zhuge *et al.* (2018); Wang *et al.* (2018*c*, 2019); Ahmad and Pasha (2019). These approaches propose dedicated processing elements (PEs) for performing Winograd based convolutions, which necessitates separate processing elements for convolution layers and fully connected layers, resulting in severe under-utilization of resources. Performing all convolution and FC layer operations on same processing elements can improve resource utilization.

In Lu *et al.* (2017), loop unrolling is performed to parallelize Winograd operations which are mapped to multiple processing elements. Each PE is designed to perform Winograd operation using Element-Wise Matrix Multiplication (EWMM). Tile sizes of $F(5\times5, 3\times3)$ and $F(4\times4, 3\times3)$ have been used. Here FC operations are transformed to EWMM and mapped to the PEs. In Wang *et al.* (2019), sparse-Winograd CNN accelerator has been presented, where the sparsity of activations and weights are exploited and computations are reduced. A dedicated Winograd convolution engine for CNN is presented in Ahmad and Pasha (2019), where multiplication complexity is reduced and power efficiency is improved. In Yang *et al.* (2019), a Winograd based dynamically Reconfigurable Architecture (WRA) for CNNs on FPGA has been presented. WRA can implement conventional convolutions, depthwise separable convolution and group convolution. Winograd algorithm is generally used for strides equal to one. In Yang *et al.* (2019), convolutions with stride greater than one are decomposed, to perform Winograd filtering. They have implemented VGG-16 and lightweight models like MobileNet using this architecture.

In this work, a novel hybrid architecture which implements Winograd algorithm and general element-wise matrix multiplication using the same set of processing elements has been proposed. This architecture consists of PEs which are optimized for GEMM and transform modules with very small resource overheads which maps Winograd algorithm to GEMM operations. This unified architecture gives the most efficient implementation for CONV layers irrespective of filter sizes and also for FC layers. A comparison of various approaches used in CNN accelerators is given in Table 2.1.

Data representation for hardware implementation can be either fixed point or floating point. Lu *et al.* (2017); Huimin Li *et al.* (2016); Yufei Ma *et al.* (2016); Qiu *et al.*

Table 2.1: Various Approaches in CNN Accelerators

Algorithmic Optimization		Hardware Generation	
GEMM	Suda *et al.* (2016)	HLS (OpenCL)	Suda *et al.* (2016)
Winograd	Aydonat *et al.* (2017) DiCecco *et al.* (2016)	HLS (Vivado)	Zhang *et al.* (2015)
FFT	Zeng *et al.* (2018)	RTL	Motamedi *et al.* (2016)
Our work is based on Winograd using GEMM and implemented using RTL			

(2016); Bai *et al.* (2018); Zeng *et al.* (2018) uses 16-bit fixed point data. Qiu *et al.* (2016) presents an FPGA implementation of AlexNet. Authors go deeper with CNN layers in Qiu *et al.* (2016), where dynamic precision technique for data quantization is presented. A 32-bit floating point based AlexNet is implemented in FPGA in Zhang *et al.* (2015). In above cited implementations, a separate processing element is dedicated for performing Winograd based convolution, in those CONV layers with smaller filter sizes, and another PE for performing conventional convolution operation of other layers in CNN. This results in reduced utilization of FPGA resources leading to lower overall performance. Also, most of the FPGA implementations are performed using fixed point arithmetic and only few are done in 32-bit floating point. There arises a trade-off between the precision and resource utilization of the design while choosing the word-length. Recently several researchers have addressed the issues related with accuracy of CNN models, using compression techniques like pruning and quantization of weights and activations Guo *et al.* (2016); Gupta *et al.* (2015); Han *et al.* (2015).

CHAPTER 3

Radix-4^3 based Two Dimensional FFT Architecture

Two dimensional FFTs are widely used for reconstruction of images from raw data. This require efficient implementations for catering to the real time scenario. Image processing applications require large memory for supporting the image size. Therefore, a suitable architecture is needed which optimizes the memory and supports various image sizes, while providing the required throughput. In this chapter, a Radix-4^3 based 2D FFT architecture with an efficient output reordering technique is discussed.

A 2D FFT is computed from 2×N one dimensional (1D) FFTs. So the performance of 1D FFT directly influences the performance of 2D FFT. An N×N 2D FFT requires N row-wise 1D FFT followed by N column-wise 1D FFT, which produces N^2 intermediate values to be stored, between the two 1D FFTs Kee *et al.* (2009).

Radix-4^3 algorithm Babionitakis *et al.* (2010*a*) based on a novel radix-4 butterfly unit is used to present the proposed 64×64 FFT architecture. Two radix-4^3 blocks are cascaded to compute 64×64 complex point FFT. A radix-4^3 block is implemented such that, the outputs generated are already reordered, which results in savings in intermediate memory and reduces latency. Our method uses parallel unrolled architecture of radix-4^3 for implementing 2D FFT, which is an extension of our previous work on 1D FFT S (2013); Kala *et al.* (2013*a*,*b*).

The contributions in this chapter are summarized as follows:

- A novel architecture for computation of 2D FFT based on radix-4^3 algorithm
- An efficient output reordering technique using six-bit control signal
- Memory reduction within 1D FFT and optimizing the intermediate memory between two 1D FFTs from N^2 to N
- ASIC and FPGA implementations of the proposed architecture
- Comparison of proposed work with the state-of-art implementations and show significant improvement in computation time and comparable area in terms of slice LUTs

3.1 Proposed 2D FFT Architecture and Data Reordering

In this section, a novel architecture for 2D FFT using radix-4^3 algorithm is presented. 2D DFT for size 64 $\times$ 64, with inputs x(i_1, i_2) can be expressed as,

$$y(k_1,k_2) = \sum_{i_2=0}^{63}[\sum_{i_1=0}^{63} x(i_1,i_2)W_{64}^{k_1 i_1}]W_{64}^{k_2 i_2} \quad (3.1)$$
$$where \quad k_1, k_2 = 0, 1, 2, ...63$$

Consider the inner block of summation in Equation (3.1),

$$\sum_{i_1=0}^{63} x(i_1,i_2)W_{64}^{k_1 i_1}$$

This is a 64 point FFT, which can be computed using a radix-4^3 algorithm. Similarly the outer summation of Equation (3.1) can also be computed using a radix-4^3 FFT. Thus a 2D FFT is computed using a cascade of two radix-4^3 blocks. Radix-4^3 architecture presented in Kala *et al.* (2013*b*) has been used for computing the 1D FFT. Each radix-4^3 has three stages of radix-4 butterfly units. The algorithm and architecture of radix-4^3 is explained in 3.1.1 and 3.1.2 respectively. Hereafter radix-4^3 will be denoted as R4^3 and radix-4 as R4.

3.1.1 R4^3 Algorithm

Radix-4^3 algorithm can be derived from the DFT formula given in Equation (3.2). Here, a four dimensional index mapping of n and k has been applied to Equation (3.2). The algorithm presented in Babionitakis *et al.* (2010*b*) has the following steps.

$$X(k) = \sum_{n=0}^{N-1} x(n)W_N^{nk} \quad k = 0, 1, 2, ...N-1 \quad (3.2)$$

$$n = n_1 + \frac{N}{64}n_2 + \frac{N}{16}n_3 + \frac{N}{4}n_4$$
$$k = 64k_1 + 16k_2 + 4k_3 + k_4 \quad (3.3)$$

Apply Equation (3.3) to the DFT formula in (3.2),

$$X(64k_1 + 16k_2 + 4k_3 + k_54) = \sum_{n_1=0}^{\frac{N}{64}-1} \sum_{n_2=0}^{3} \sum_{n_3=0}^{3} \sum_{n_4=0}^{3} x(n_1 + \frac{N}{64}n_2 + \frac{N}{16}n_3 + \frac{N}{4}n_4)W_N^{nk} \tag{3.4}$$

Decomposing the twiddle factors,

$$W_N^{nk} = W_{\frac{N}{64}}^{n_1k_1} W_N^{n_1(16k_2+4k_3+k_4)} W_{64}^{n_2(4k_3+k_4)} W_{16}^{n_3k_4} (-j)^{(n_2k_2+n_3k_3+n_4k_4)} \tag{3.5}$$

Substituting (3.5)) in (3.4) and expanding the summation with index n_4,

$$X(64k_1 + 16k_2 + 4k_3 + k_4) = \sum_{n_1=0}^{\frac{N}{64}-1} \sum_{n_2=0}^{3} \sum_{n_3=0}^{3} [B(n_1 + \frac{N}{64}n_2 + \frac{N}{16}n_3)](-j)^{(n_2k_2+n_3k_3)} W_{\frac{N}{64}}^{n_1k_1} W_N^{n_1(16k_2+4k_3+k_4)} W_{64}^{n_2(4k_3+k_4)} W_{16}^{n_3(k_4)} \tag{3.6}$$

The first butterfly unit, B is given by

$$B = x(n_1 + \frac{N}{64}n_2 + \frac{N}{16}n_3)(-j)^{k_4} x(n_1 + \frac{N}{64}n_2 + \frac{N}{16}n_3 + \frac{N}{4}) + (-1)^{k_4} x(n_1 + \frac{N}{64}n_2 + \frac{N}{16}n_3 + \frac{N}{2}) + (j)^{k_4} x(n_1 + \frac{N}{64}n_2 + \frac{N}{16}n_3 + \frac{3N}{4}) \tag{3.7}$$

Expanding the summation in (3.6) with index n_3,

$$X(64k_1 + 16k_2 + 4k_3 + k_4) = \sum_{n_1=0}^{\frac{N}{64}-1} \sum_{n_2=0}^{3} [H(n_1 + \frac{N}{64}n_2)] \times (-j)^{(n_2k_2} W_{\frac{N}{64}}^{n_1k_1} W_N^{n_1(16k_2+4k_3+k_4)} W_{64}^{n_2(4k_3+k_4)} W_{64}^{n_3(4k_4+k_5)} \tag{3.8}$$

The second butterfly unit, H is given by

$$H = B(n_1 + \frac{N}{64}n_2) + (-j)^{k_3}[B(n_1 + \frac{N}{64}n_2)]W_{16}^{k_4} + (-1)^{k_3}[B(n_1 + \frac{N}{64}n_2 + \frac{N}{8})]W_8^{k_4} + (j)^{k_3}[B(n_1 + \frac{N}{64}n_2 + \frac{3N}{16})]W_{16}^{k_4}W_8^{k_4} \tag{3.9}$$

Expanding the summation in (3.8) with index n_2,

$$X(64k_1 + 16k_2 + 4k_3 + k_4) = \sum_{n_1=0}^{\frac{N}{64}-1} [T(n_1) W_N^{n_1(16k_2+4k_3+k_4)}] W_{\frac{N}{64}}^{n_1 k_1} \tag{3.10}$$

The third butterfly unit, T is given by

$$\begin{aligned} T = H(n_1) + (-j)^{k_2}[H(n_1 + \frac{N}{64})] W_{16}^{k_3} W_{64}^{k_4} + \\ (-1)^{k_2}[H(n_1 + \frac{N}{32})] W_{16}^{k_3} W_{8}^{k_4} + \\ (j)^{k_2}[H(n_1 + \frac{3N}{64})] W_{64}^{3(4k_3+k_4)} \end{aligned} \tag{3.11}$$

Equations (3.7), (3.9) and (3.11) are the three butterfly units in radix-4^3 algorithm. Each of these butterfly operations are based on radix-4 butterfly and hence the name Radix-4^3.

3.1.2 R4^3 Architecture

Signal Flow Graph (SFG) for R4^3 algorithm is shown in Fig. 3.1. SFG follows a Decimation-In Frequency (DIF) pattern. From Fig. 3.1 it is clear that the first stage has 16 butterfly operations, second stage has four butterfly computations and the third stage has one radix-4 butterfly operation. Each node in the SFG represents one radix-4 butterfly unit. Inputs to the first R4 unit are x(0), x(16), x(32) and x(48). This produces the four outputs X(0), X(16), X(32) and X(48). Each node of R4 butterfly unit is shown in Fig. 3.2. Signals shown in SFG are explained in this section.

A fully unrolled R4^3 architecture which uses parallel R4 butterfly units Kala *et al.* (2013*a*) is the basic building block of our proposed architecture. R4 unit has four parallel inputs and gives one output based on a two-bit control input called *mode select*. The *mode select* signal decides the generation of one output out of the four. Based on the *mode select* signal, outputs can be generated in any order. But in a conventional R4 butterfly, outputs are generated in a particular order Proakis and Manolakis (1996). The

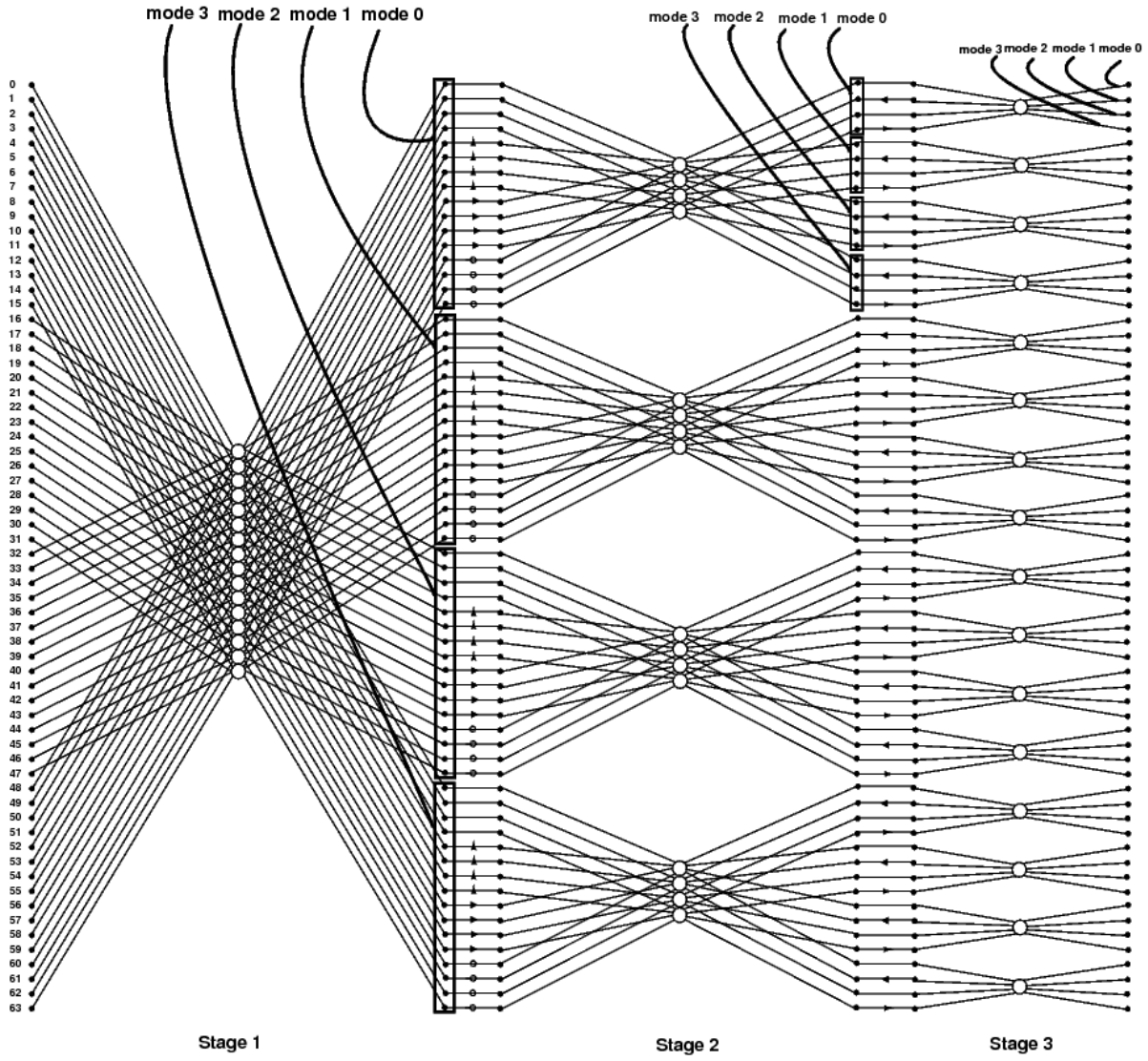

Figure 3.1: Signal Flow Graph of $R4^3$ algorithm

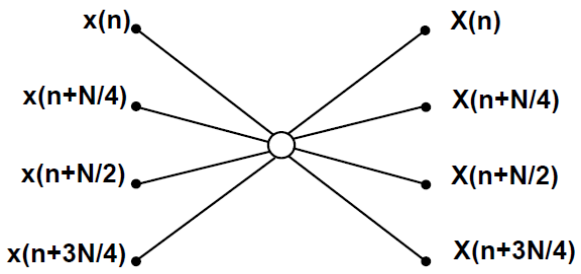

Figure 3.2: Nodal representation of radix-4 butterfly

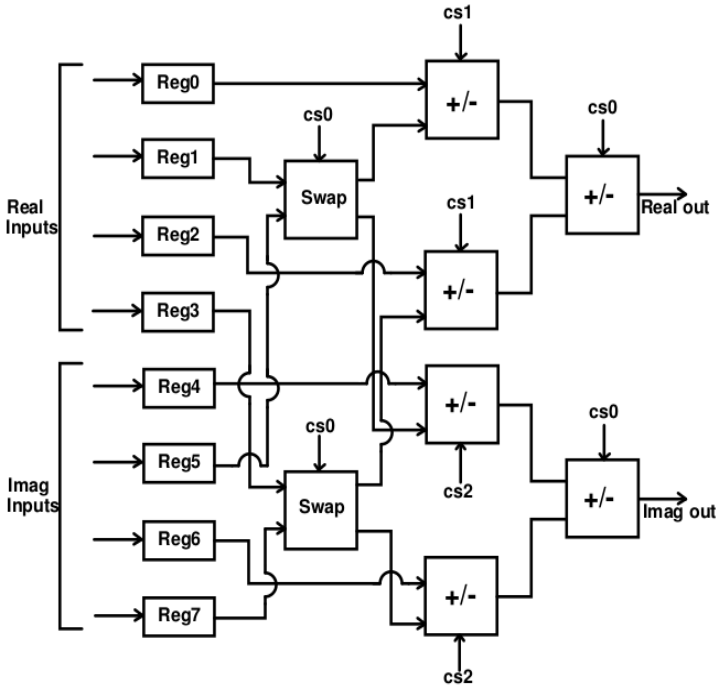

Figure 3.3: Radix-4 Butterfly Unit

equation for a radix-4 butterfly unit is given in Equation (3.12),

$$\begin{aligned} X(0) &= x(0) + x(1) + x(2) + x(3) \\ X(1) &= x(0) - jx(1) - [x(2) - jx(3)] \\ X(2) &= x(0) - x(1) + x(2) - x(3) \\ X(3) &= x(0) + jx(1) - [x(2) + jx(3)] \end{aligned} \tag{3.12}$$

Radix-4 butterfly architecture is shown in Fig. 3.3 Kala *et al.* (2013*a*).

Unlike conventional Cooley-Tukey FFTs, this architecture generates the output in required order rather than in bit reversed order. The output of R4 unit is decided by a two-bit control signal *Mode Select*. Based on this signal, one out of four outputs is produced. Control signals of R4 unit are given in Table 3.1. The signals $cs0$, $cs1$ and $cs2$ can be derived from *mode select* signal. These are control signals for performing swap operation (multiplication with $'j'$) and adder/subtracter operation.

Fig. 3.4 shows the parallel unrolled architecture of $R4^3$ Kala *et al.* (2013*b*). First stage has 16 R4 units, second stage has four R4 units and third stage has one R4 unit. All R4 blocks are identical in this architecture. The notations R4 0, R4 4, R4 8, R4 12 indicates 0^{th}, 4^{th}, 8^{th}, 12^{th} radix-4 butterfly blocks and so on. **W16** and **W64** indicate

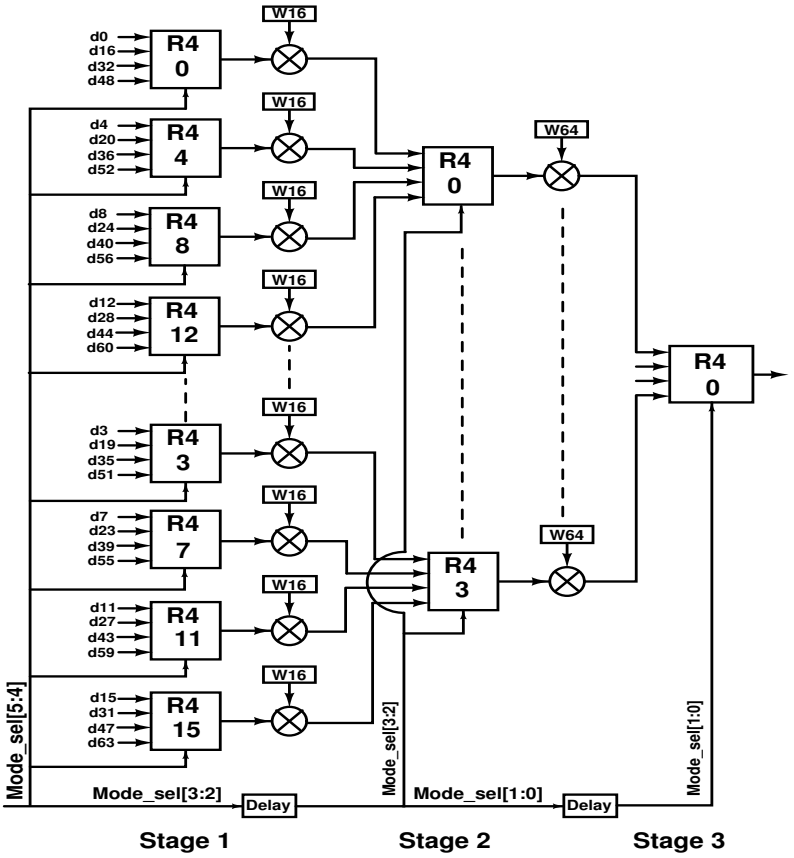

Figure 3.4: Radix-4^3 block

twiddle factors of first and second stage. There are sixteen twiddle factor ROMs in the first stage for storing **W16**. Each of these ROMs contain four twiddle factor values. Second stage consists of four R4 engines and has four ROMs for storing **W64**. Each of these ROMs in second stage consists of sixteen twiddle factor values.

Out of the N multipliers in each stage, first $N/4$ multipliers of all stages have unity twiddle factor. So while implementing, these multipliers are removed. Thus first stage has 12 complex multipliers instead of 16 and second stage has 3 multipliers instead of 4.

Table 3.1: Control Signals in Radix-4 Unit

Mode	Output	cs0	cs1	cs2
0 0	X(0)	0	0	0
0 1	X(1)	1	0	1
1 0	X(2)	0	1	1
1 1	X(3)	1	1	0

Table 3.2: Mode Selection in Different Stages of $R4^3$ Block

Stage 1	Stage 2	Stage 3	Output generated
0 0	0 0	00	0
		01	16
		10	32
		11	48
0 0	0 1	00	1
		01	17
		10	33
		11	49
:	:	:	:
:	:	:	:
1 1	1 1	00	15
		01	31
		10	47
		11	63

Mode select signals in $R4^3$ architecture is given in Table 3.2. *Mode select* is a six-bit control signal. Each stage is assigned with two bits of *mode*. Based on the *mode* of each stage, one out of 64 outputs is generated. Thus the outputs of $R4^3$ block is obtained in reordered form in this architecture. Hence memory and logic for reordering can be saved. Initially the *mode select* of all radix-4 engines are configured as *mode 0*. The outputs produced from first stage are multiplied with the corresponding twiddle factors. Similar operation is performed in the second stage. Here, four radix-4 engines are required to process the 16 outputs that are obtained from first stage. Again, the first four outputs are generated by configuring the *mode select* of all four radix-4 engines as *mode 0*, keeping the first stage radix-4 engines in *mode 0* itself. Now using this four outputs, the output required in the third stage can be generated, with another mode selection in the last stage.

Table 3.2 shows that when the six-bit signal is $'000000'$ *mode* is 0, $'000001'$ gives *mode* as 16 and so on. That is, *mode select* of the first and second stage remains $'00'$ while that of the third stage is changing. This control signal can be generated using

an up-counter. *Mode select* signals can be applied to radix-2 architectures also. In that case, single bit will be sufficient for a radix-2 basic block.

3.1.3 Data Scheduling in Proposed Architecture

Consider the 64 $\times$ 64 input data which is stored in RAM given by matrix **A**, where $A_{i,j}$ is the element in i^{th} row and j^{th} column.

Data scheduling and reordering in First R4^3 Block is shown in Fig. 3.5. We start with the *mode select* signal set as 0 and inputs as $A_{1,1}, A_{2,1}, A_{3,1}, ..., A_{64,1}$ in cycle 1. FFT of these 64 inputs is performed producing the first output of first row in **B** matrix, say $B_{1,1}$ in first cycle. In cycle 2, the inputs $A_{1,2}, A_{2,2}, A_{3,2}, ..., A_{64,2}$ are given, keeping the *mode select* as 0 itself. This produces the output $B_{1,2}$. In a similar fashion, in cycle 64, $A_{1,64}, A_{2,64}, ..., A_{64,64}$ with *mode select = 0*, the output $B_{1,64}$ is produced. Thus all the elements in first row are computed.

In the 65^{th} cycle, all the computed elements in first row of **B** matrix i.e, $B_{1,1}, B_{1,2}$, ... ,$B_{1,64}$ are fed in parallel to the second R4^3 block. Here, as soon as one row of first R4^3 block is computed, second R4^3 begins to perform the next phase of computation. Intermediate registers between the two R4^3 blocks store output from first block to be used by the second block. In the second R4^3 block, *mode select* signal changes from 0 to 63 in each cycle, producing 64 outputs, that is, $C_{1,1}, C_{1,2}, ..., C_{1,64}$. This is shown in Fig. 3.6.

Since the output from first R4^3 block is required to be reordered before applying it to second block, sequentially incrementing the *mode select* signals is not sufficient. For the first block outputs to be reordered, *mode select* for first R4^3 block has to be given in the order 0, 16, 32, 48, 1, 17, 33, 49,...,63.

In 65^{th} cycle, the *mode select* for first R4^3 is changed to 16, and inputs $A_{1,1}, A_{2,1}$, $A_{3,1}$, ..., $A_{64,1}$ are provided. FFT of these 64 inputs is performed producing the first output of second row in **B** matrix, $B_{2,1}$ in 65^{th} cycle. In the 66^{th}cycle, second output of second row of **B** matrix, that is, $B_{2,2}$ is computed and so on. In the 128^{th} cycle, $B_{2,64}$ is calculated. Thus all the elements in second row is computed.

During 129^{th} cycle, second block starts computation using the second row of **B**

matrix, by changing the *mode select* from 0, 1, 2, 3,..., to 63 in 64 cycles, resulting in the second row elements of **C** matrix. This procedure is repeated by changing *mode select* till 63 to get all the elements in 64×64 **B** matrix, which is the output of first $R4^3$ block. The corresponding **B** matrix outputs are generated in each cycle. This completes the column-wise FFT computation.

The elements in **C** matrix gives the final 2D FFT output. One output is generated per clock cycle from second block. Thus 4096 cycles are required to produce all 4096 outputs of the 2D FFT.

3.1.4 2D FFT Architecture

In architectures which follow conventional Cooley-Tukey algorithm, only one R4 butterfly unit will be present in each stage. Hence the outputs from each stage should be reordered before applying to the next stage. If two $R4^3$ blocks are cascaded, intermediate outputs are to be reordered. Since subsequent blocks will require the inputs to be reordered before processing, intermediate RAMs are required between every cascaded stage in such implementations. This reordering needs intermediate memory and memory size grows as FFT length increases. For example for a 64 × 64 2D FFT, a 4096 (64 × 64) deep memory is required between the two blocks. That is, for N × N 2D FFT, N^2 intermediate memory is required between each phase of $R4^3$.

As already mentioned, a parallel unrolled implementation for $R4^3$ block has been used in this work. The order of output is controlled using a few control bits. With this architecture, for a given set of inputs, at each stage, only those outputs required for the next stage can be computed without losing any performance, so that most of the intermediate buffers can be avoided. The resulting implementation is thus fully optimized in terms of memory and latency.

A one dimensional FFT using $R4^3$ which is shown in Fig. 3.4 is performed, and its output is fed to a second $R4^3$ block to perform row wise FFT, to get the two dimensional FFT. The first processor performs sixty four 64-point FFT operations which gives 4K intermediate values. The second FFT processor performs sixty four 64-point FFT operation on these outputs and gives the final 64 × 64 point FFT. The two $R4^3$ blocks are identical. Block diagram of the proposed architecture is shown in Fig. 3.7.

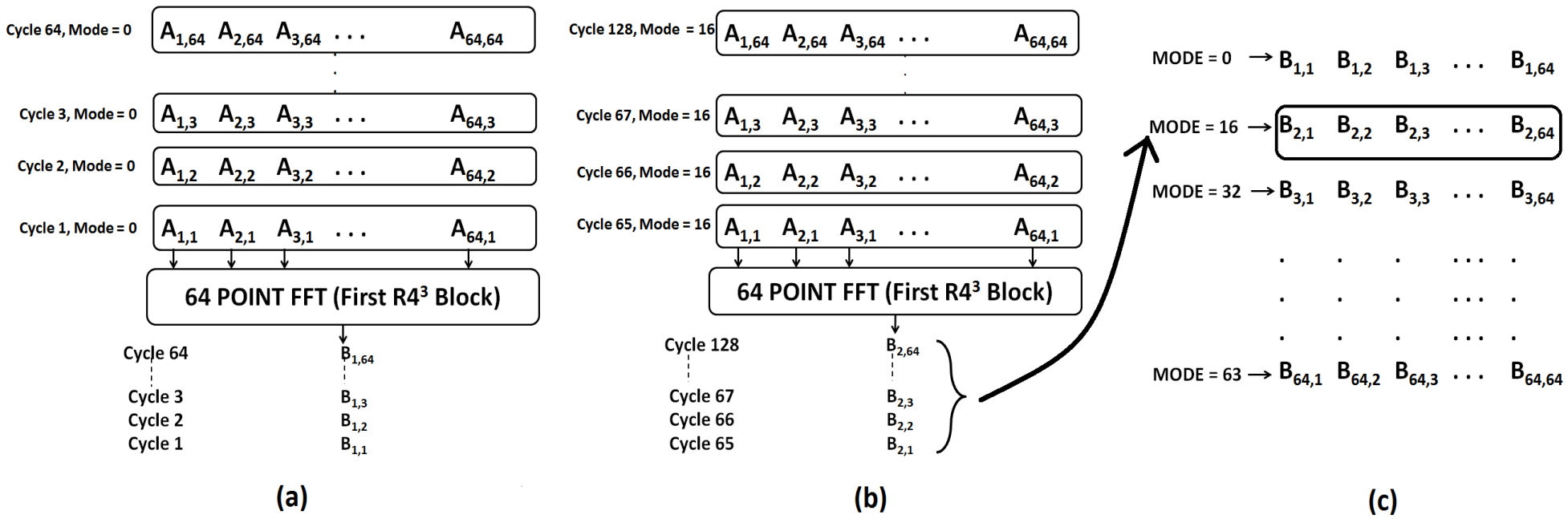

Figure 3.5: Data Scheduling in First Radix-4^3 Block (**a**) Output when $Mode = 0$ (**b**) Output when $Mode = 16$ (**c**) Output for $Mode = 0$ to $Mode = 63$

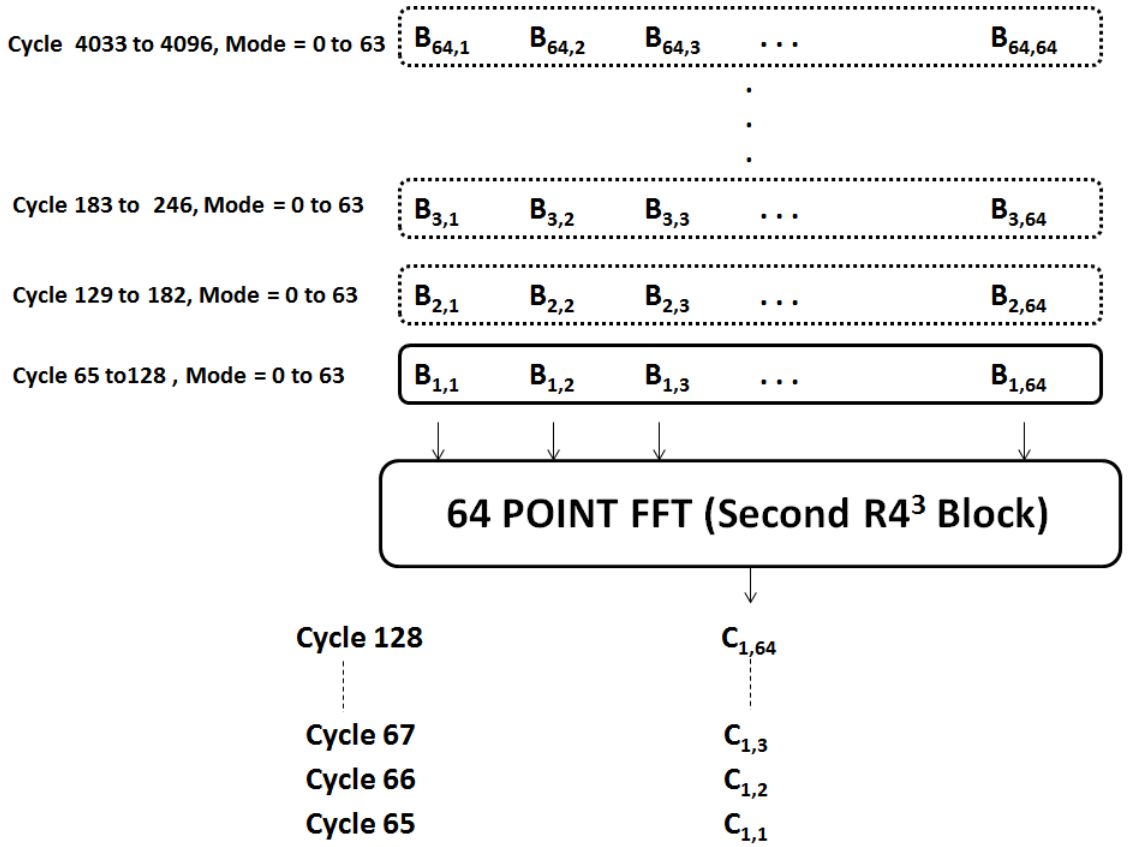

Figure 3.6: Data Scheduling in Second Radix-4^3 Block

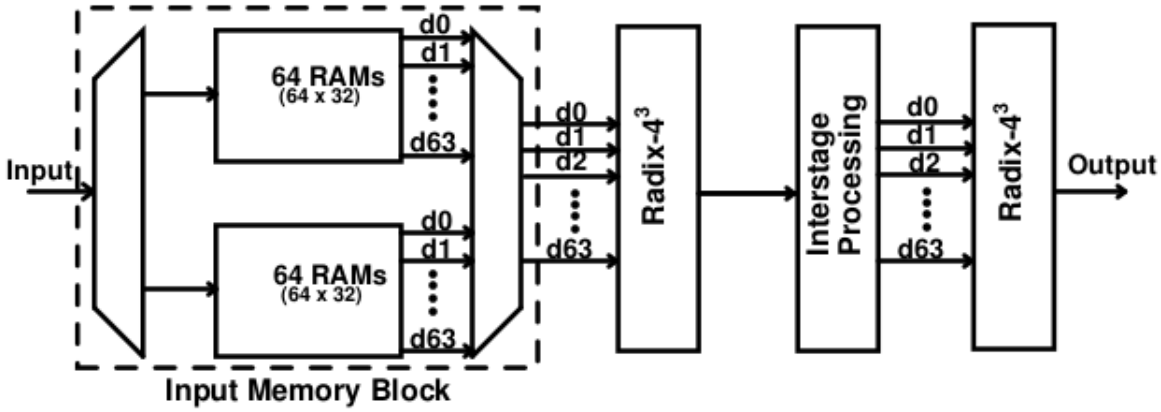

Figure 3.7: Proposed 2D FFT Architecture using R4^3 blocks

Input Memory

The input memory block consists of two banks of 64 RAMs which operate in ping-pong fashion. One set of input memory reads in the data, while the other set will read out the data. The detailed layout of one of the banks is shown in Fig. 3.8. Inputs are written into consecutive locations of RAM. ie., $RAM0$ takes in the first 64 inputs, $RAM1$ takes 65^{th} input onwards and so on. During the read operation, one input is supplied from

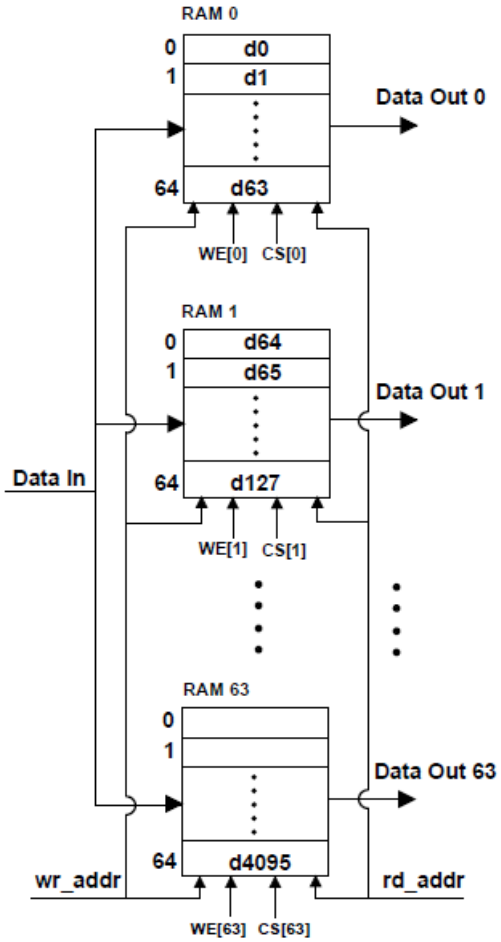

Figure 3.8: Input Memory

each RAM, so that 64 inputs are available to the $R4^3$ block. Read operation is done in parallel. Here, read address is same for all RAMs.

Interstage Processing

Interstage processing consists of two banks of registers with 64 registers in each bank. Registers in the first bank are arranged as a chain of 64 shift registers. Outputs from first $R4^3$ block enter into the first bank in serial. Registers in the first bank are advanced in every clock cycle. Once in every 64 cycle, outputs from all the 64 registers in the first bank are loaded in parallel to the second bank. Registers in the second bank act as the input for the second $R4^3$ block.

Control Circuitry

Control circuitry consists of an 12-bit up counter. Input RAMs in the input side requires six-bits for addressing. There are two such memory banks in our architecture. The read address, write address and chip select signals are generated from this counter. Also, mode select signals for both R4^3 blocks are produced from this counter. While writing, $chipselect$ signals are separately generated for each of the RAMs. While reading, same locations from all the RAMs are accessed in parallel and control signals are generated accordingly. All these signals are synchronized with respect to the previous stage delay.

Continuous Flow FFT

Proposed FFT uses a novel data scheduling mechanism which supports continuous-flow of data. Here, the butterfly units are constantly performing computations on the streaming data. The FFT processor accepts one input sample in every clock cycle. Here a conflict-free addressing scheme as discussed in Section 3.1.3 has been used.

Scalability

Data-path and control logic of the proposed architecture are scalable to support various FFTs. Radix-4^3 block uses three stages of radix-4 FFT and if only two stages are used, with appropriate twiddle factors, we get a radix-4^2 block. Cascading two radix-4^2 blocks will result in a 16$\times$16 FFT. Similarly cascading two radix-4^4 blocks will result in 256$\times$ 256 FFT. This requires four stages of radix-4 FFT in a radix-4^4 block.

3.2 Implementation Results and Analysis

3.2.1 Matlab Simulation

The proposed two dimensional FFT architecture using R4^3 algorithm has been simulated in matlab and the outputs are compared with matlab inbuilt function for 2D FFT, for validation. Floating point implementation is not efficient for hardware implementations owing to its complexity and high power. So fixed point implementation is pre-

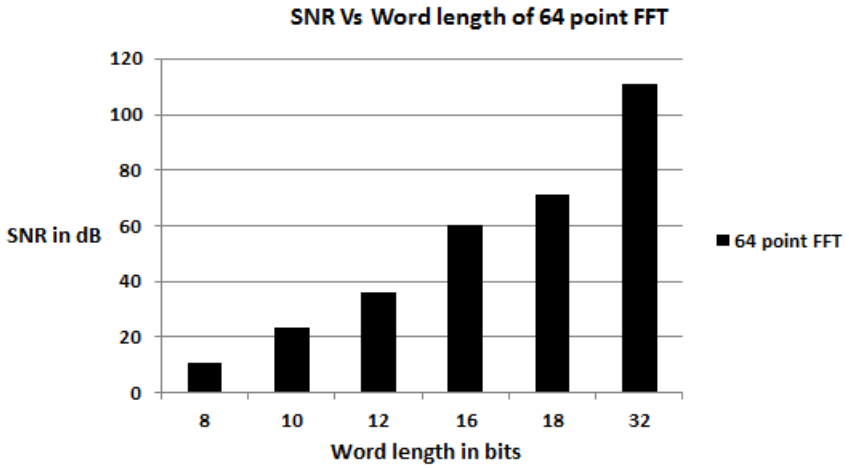

Figure 3.9: SNR Vs Word length for 64 point FFT using Radix-4^3 Algorithm

ferred. Selection of word length is a design issue for fixed point implementations. For choosing an adequate word length, the Signal to Noise Ratio (SNR) for various word length is analyzed. Based on the size of FFT and SNR, selection of word length also varies. We have calculated the SNR of 64-point FFT for various word lengths as shown in Fig. 3.9. Based on this analysis, 16 bits were selected as word length. 16 bit word length gives SNR of 59.9 dB, which is sufficient for image reconstruction.

For fixed point implementation, in order to avoid overflow at all stages, inputs at each butterfly node are scaled down by 0.25. Hence floating point FFT outputs also have to be scaled down by the FFT size before performing the comparison. For real and imaginary parts of data words, out of 16 bits, 1 bit is used for sign and 15 bits for fraction, giving a range of -1 to +1. For representing twiddle factors, a 16 bit word with 1 bit for integer, 1 bit for sign and 14 fractional bits is used.

3.2.2 ASIC Implementation

The proposed architecture of 2D FFT has been implemented in RTL using Verilog HDL. The implementation uses fixed point arithmetic with 16 bits each for representing real and imaginary part of the data. The SRAMs used are generated by Faraday's Memory Compiler.

RTL simulation and verification has been done using Modelsim Simulation and Verification tool. RTL has been synthesized with Cadence RTL Compiler using Faraday 40 nm standard cell library, tailored for UMC's 40nm,1.2V logic process. Activity in-

formation generated using simulation are input to RTL compiler so as to estimate the power dissipation. The FFT processor has an operating frequency of 500 MHz with a throughput of 500 MSPS (Mega Samples Per Second). It has a total silicon area of $0.841mm^2$ with $0.409mm^2$ logic area and $0.432mm^2$ memory area. Power estimations are done with a clock frequency of 500 MHz.

At 500 MHz, power consumption is 358 mW and execution time is 8.19μs. Energy of the proposed 2D FFT is calculated as *Power* $\times$ *Execution time*, which is 2.9μJ. Energy efficiency is referred as number of FFT points computed per Joule as given in (3.13) Yu-Wei Lin *et al.* (2004).

$$\frac{FFTs}{Energy} = \frac{1}{Power \times ExecutionTime \times 10^3} \tag{3.13}$$

Energy efficiency of the proposed architecture is 341 points/Joule. For the computation of a single point FFT, 2.9mJ is consumed.

The synthesized netlist from RTL Compiler is simulated and verified with input test vectors generated by Matlab functional model. Verification structure is shown in Fig. 3.10.

Latency Calculation

Input memory block requires 4096 cycles. After inputs are written to RAM, one extra cycle is required before it can be read out. Two $R4^3$ block require 3 cycle each. Intermediate registers between two $R4^3$ block require 65 clock cycles. Total latency required for the 64$\times$64 point FFT is 4169 clock cycles. Effective computation time is 4096 cycles per 64$\times$64 FFT. Operating frequency of proposed architecture is 500 MHz.

ASIC synthesis results are given in Table 3.3. Computation time for various 2D FFT architectures are compared in Table 3.4. For a fair comparison, we have considered only smaller size 2D FFTs in literature. Table 3.4 shows that our architecture takes less number of clock cycles for execution and gives 47.5% reduction in computation time for 64 $\times$ 64 2D FFT compared to the best existing implementation. Since the implementations in references shown in Table 3.4 are of different technology,for evaluating the execution speed, we have considered cycle count. Proposed architecture takes less clock cycles

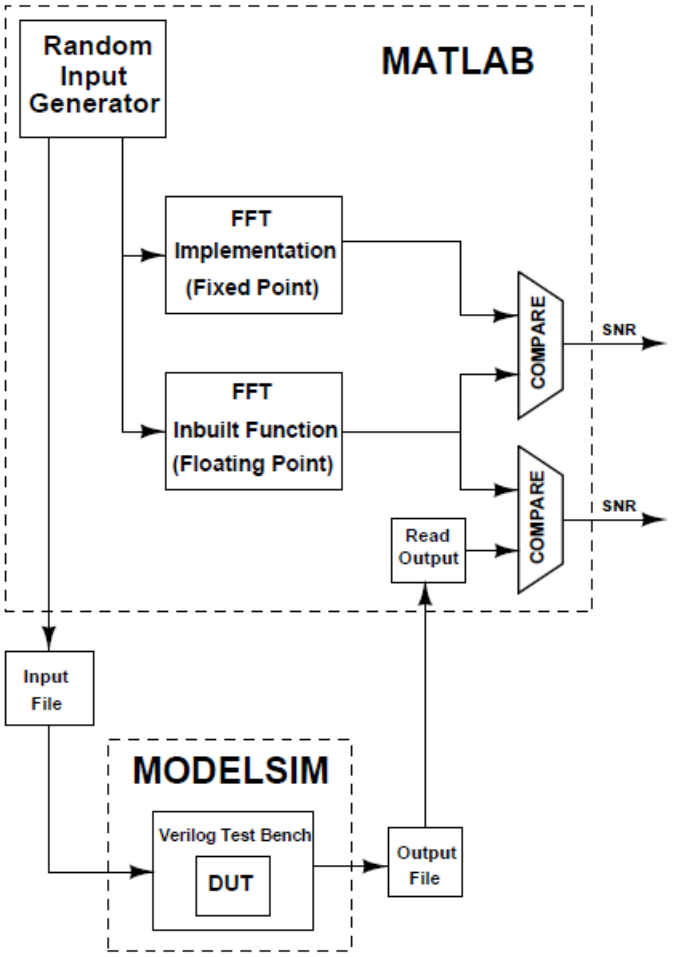

Figure 3.10: Verification

Table 3.3: ASIC Synthesis Results of 64 × 64 FFT

Process	40 nm
Word length	16 bits
Max. Frequency	500 MHz
Area	0.841 mm^2 (0.409 mm^2 logic + 0.432 mm^2 memory)
Voltage	1.2V
Power @ 500 MHz	358 mW
Computation time	8.19 μs
Energy	2.9 μJ
Energy per FFT	2.9 mJ
Throughput	500 MSPS

Table 3.4: Comparison of Computation Time

	FFT Size	Frequency (MHz)	Clock cycles	Computation time
Proposed	64 × 64	500	4096	8.19μs
Rodrguez-Ramos *et al.* (2008)	64 × 64	100	4516	45.16μs
Wang *et al.* (2010)	64 × 64	263	4103	15.6μs
Yu *et al.* (2010)	128 × 128	100	179000	1.79ms
Deng *et al.* (2008)	128 × 128	100	33350	0.33ms
Yu *et al.* (2010)	256 × 256	100	605000	6.05ms
Deng *et al.* (2008)	512 × 512	100	295494	2.955ms**
Mahmood *et al.* (2015)	512 × 512	-	-	32.4ms**
Wang *et al.* (2010)	512 × 512	266	524288	0.985 ms**
Chen and Prasanna (2014)	1024 × 1024	-	-	2.63ms*
Uzun *et al.* (2005)	1024 × 1024	-	-	62.5ms
Shirazi *et al.* (1995)	512 × 512	10	-	0.417s

**Estimated computation time for 512 × 512 FFT based on our architecture is 0.131ms
*Estimated from Chen and Prasanna (2014). Estimated computation time for 1024 × 1024 FFT based on our architecture is 2.09ms

for FFT computation.

3.2.3 FPGA Implementation

FPGA implementations are targeted to Xilinx Virtex 7, XC7V2000T FPGA device. Table 3.5 gives the FPGA implementation results of 64 $\times$ 64 FFT. At 156.25 MHz, total power consumption is 1422 mW, out of which 912 mW is dynamic power and 510 mW is static power. Table 3.6 shows the resource utilization of proposed architecture.

From Table 3.4 it can be seen that our architecture gives the least cycle count among all other implementations. For ASIC implementation with a frequency of 500 MHz we get an execution time of 8.19μs, which is the best among other implementations. However in the case of FPGA, our operating frequency is 156.25 MHz. So even though we have lower cycle count, this result in higher execution time.

We implement non pipelined adders and multipliers using DSP48 in FPGA. If we use pipelined adders and multipliers, we should be able to improve the operating frequency. 2D FFT comparison of resources is given in Table 3.7. Various FFT architectures are compared with proposed work. Comparison of area in terms of slice LUTs shows that we have comparable area with other implementations.

In this work, an efficient data scheduling technique has been implemented, so that

Table 3.5: FPGA Implementation Results of 64 $\times$ 64 FFT

Word length	16 bits
Frequency	156.25 MHz
Static Power	510 mW
Dynamic Power	912 mW
Total Power	1422 mW

Table 3.6: Resource Utilization of 64 $\times$ 64 FFT

Resource	Used/Available	Percentage
Slices	5692 / 305400	1.86
Flip flops	14103 / 2443200	0.57
Block RAM (RAMB18/FIFO18)	64 / 2584	2.47
DSP48	124 / 2160	5.74

2D FFT output is in normal order. SDF and MDC architectures require output memory for bit reversal Huang and Chen (2012). Our architecture does not require any additional hardware blocks for performing bit reversal. Output of proposed FFT architecture is already in reordered form. Therefore extra buffers are not required here.

3.2.4 Hardware Complexity Analysis

We analyze the hardware complexity in terms of complex adders, complex multipliers, memory requirement and control circuitry of various 1D FFT architectures in Table 3.8. From Table 3.8, in R4^3 systolic architecture, the number of adders and multipliers are comparable to other architectures.

In this work, fully parallel unrolled architecture of R4^3 FFT has been used. Our R4^3 FFT uses 63 adders, 15 complex multipliers and 64 words of memory. Even though adders and multipliers are more, total area remains less as seen from ASIC results. This is because memory consumes more area compared to logic. This architecture reduces intermediate memory significantly, at the cost of adders and multipliers.

3.3 Summary of the Chapter

In this chapter, a novel 2D FFT architecture with efficient data reordering technique, using radix-4^3 algorithm is presented. The architecture uses two parallel unrolled radix-4^3 blocks in cascade to develop a 64 $\times$ 64 2D FFT architecture. We have used six-bit *mode select* as the control signal in radix-4^3 architecture for performing data reordering. Radix-4^3 architecture gives significant reduction in intermediate memory within a 1D FFT and reduces the latency. Proposed architecture of 2D FFT reduces the intermediate memory between two 1D FFTs from N^2 to N. Fixed point arithmetic simulation of the proposed architecture is done in matlab. SNR for various word lengths is analyzed and 16 bit is chosen as the required word length for image processing applications. The architecture has been implemented in RTL using Verilog HDL and simulated using Modelsim. RTL has been synthesized with Cadence RTL Compiler using Faraday 40nm standard cell library, tailored for UMC's 40nm process. ASIC synthesis results give a clock frequency of 500 MHz and core area of 0.841mm^2. At 500 MHz the power

consumption is 358mW. Energy efficiency in terms of number of FFTs per Energy is 341 points/Joule. 64×64 FFT takes 4096 cycles for computation and the execution time is 8.19μs. Comparison with existing implementations shows 47.5% reduction in computation time for 64×64 FFT. Proposed architecture has been also implemented in Virtex-7 FPGA with an operating frequency of 156.25 MHz. FPGA implementation results show comparable area in terms of slice LUTs.

Table 3.7: Comparison of FFT Architectures

	FFT Size	FPGA used (Technology)	Frequency (MHz)	Area (Slice LUTs)	Clock cycles	Throughput (MSPS)
Ours	64×64	XC7V2000T	156.254	5692	4096	156.254
Rodrguez-Ramos *et al.* (2008)	64×64	XC4VSX35	100	6472	4516	90.6
Rodrguez-Ramos *et al.* (2008)	128×128	XC4VSX35	100	11326	17084	95.9
Deng *et al.* (2008)	128×128	XC2VP -100	100	5933	33350	103
Kim and Lee (2009)	512×512	XC4VLX60	157	11733	-	-
Shirazi *et al.* (1995)	512×512	2 × XC4000E	10	13600	-	0.55
Wang *et al.* (2010)	512×512	XC5VLX330	266	15604	524288	266.1
Uzun *et al.* (2005)	512×512	XCV2000E	-	8480	-	-
T. (2001)	2K×2K	2 × XC2V6000	125	67584	-	-
Chen and Prasanna (2014)	1K×1K	ARTIX-7	100	-	-	398

Table 3.8: Hardware Complexity of various 1D FFT Architectures

Architecture	Adders	Multipliers	Memory	Control
R2SDF Wang and Li (2016)	4log_4N	2log_4N-2	N-1	Simple
R4SDF Wang and Li (2016)	8log_4N	log_4N-1	N-1	Medium
R2^{2}SDF Shousheng He and Torkelson (1998*a*)	4log_4N	log_4N-1	N-1	Simple
R2MDC Shousheng He and Torkelson (1998*a*)	4log_4N	2log_4N-2	3$N/2$ -2	Simple
R4MDC Wang and Li (2016)	8log_4N	3log_4N-3	5$N/2$ -4	Complex
R4SDC Wang and Li (2016)	3log_4N	log_4N-1	2N -2	Complex
R4^3 Systolic Babionitakis *et al.* (2010*a*)	3log_4N	log_4N-1	$N/3$ log_4N	Simple

CHAPTER 4

Convolution Algorithms for CNN Implementation

In this chapter, various convolution algorithms for accelerating CNNs on hardware are discussed. Direct (or conventional) convolution and fast algorithms like FFT based convolution and Winograd minimal filtering are described in this chapter. Training and inference of popular CNN models in various platforms are also discussed in this chapter. Hardware architecture for accelerating AlexNet model is proposed in this chapter and has been implemented in Virtex-7 FPGA.

4.1 Direct Convolution

In 2D convolution, input data is convolved with kernel, which is a multiply and accumulate (MAC) operation. There will be multiple channels of input and kernels, and the outputs from each channel are added up together to get the final result. As the image size increases, number of MAC operations also increases. This is a slow process since each pixel in the image is multiplied with each of the weight in the kernel. Real time CNN architecture comprises of several channels of input features and many convolutional layers. Typical 2D convolution operation of 5×5 input data and 3×3 kernel is shown in Fig. 4.1. Here a stride of one is used for convolving the data. For efficient computation, strides may vary in different CNN models. Direct convolution can be performed using general matrix multiplication (GEMM) algorithm.

We perform complexity analysis of 2D convolutions, in terms of adders and multipliers, which is shown in Fig. 4.2. For an N×N kernel, number of adders required is $N^2 - 1$ and multipliers required is N^2.

Fig. 4.3 shows the architecture of a one dimensional convolution which uses conventional method. Stride of one is used in this architecture, with a filter size of three. Streaming inputs are denoted as a0, a1, a2, etc. and the filter coefficients are b0, b1

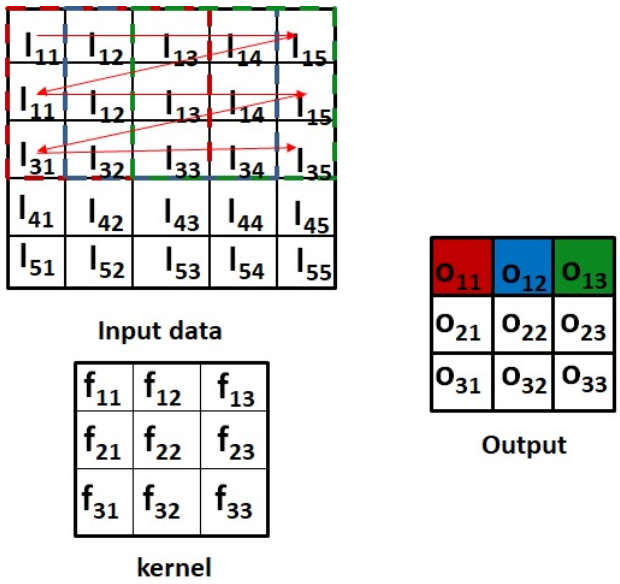

Figure 4.1: Two dimensional convolution

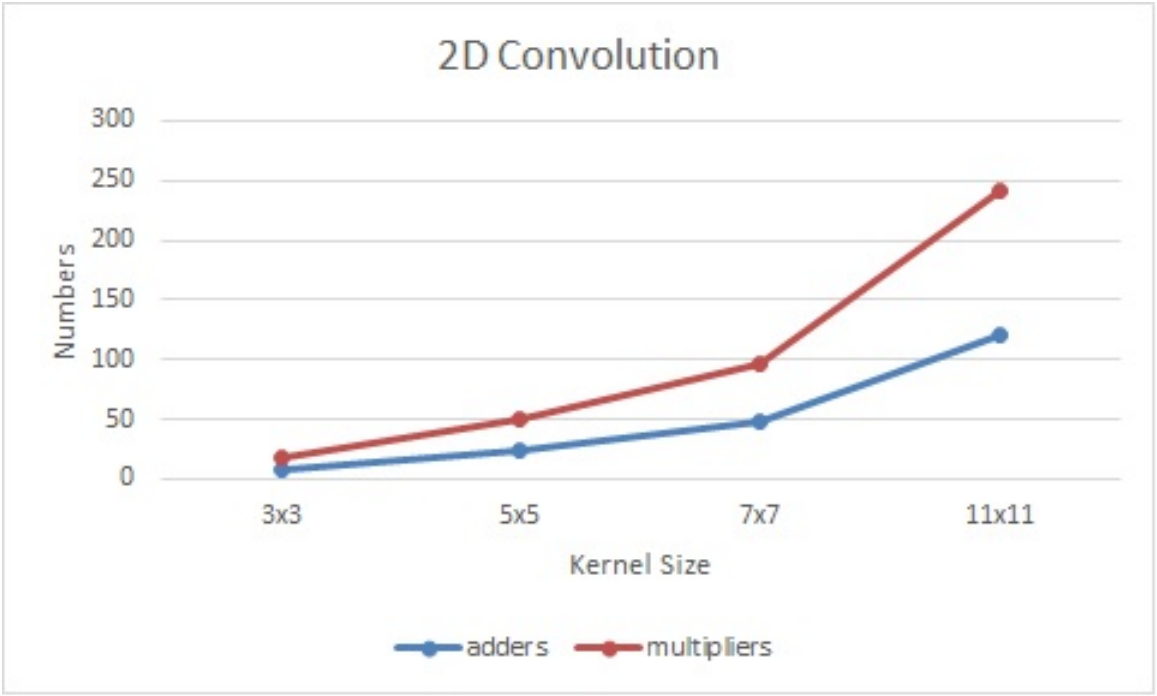

Figure 4.2: Complexity analysis of 2D convolution

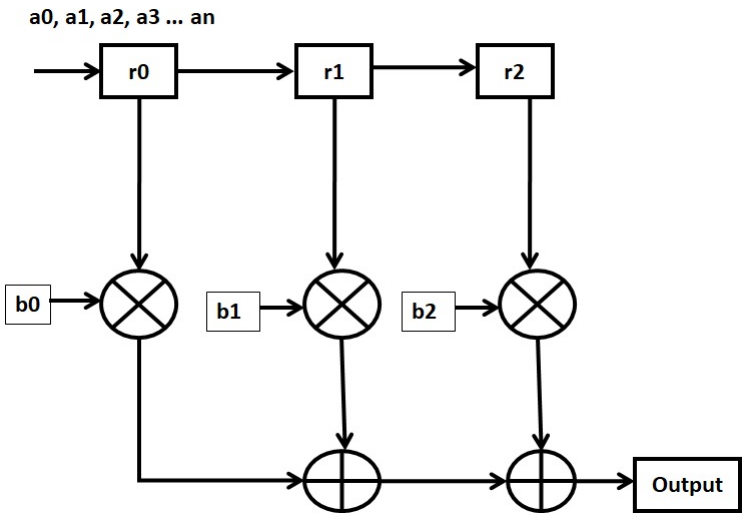

Figure 4.3: Architecture of 1D Convolution

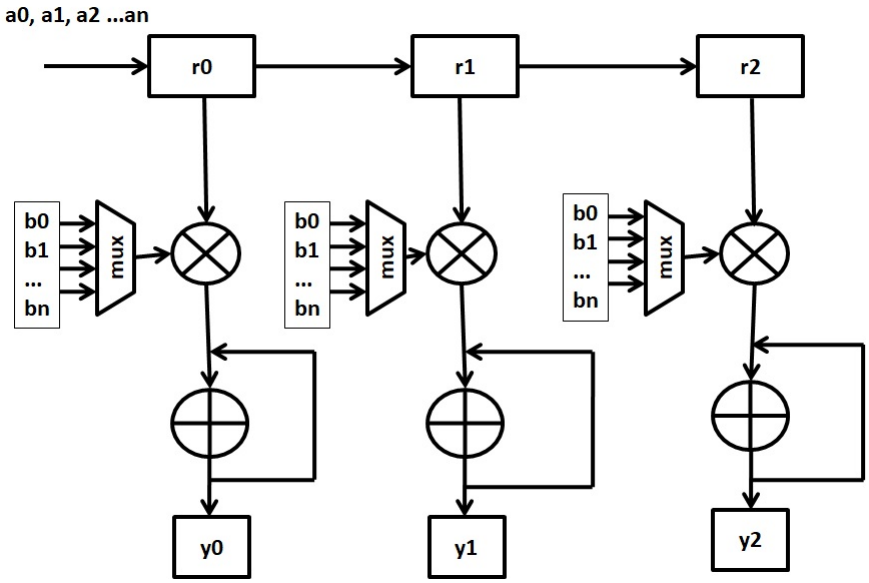

Figure 4.4: Architecture of 2D Convolution

and b2. Filter coefficients are the inputs to a constant multiplier. Registers used in the architecture are denoted as r0, r1 and r2.

Fig. 4.4 shows the architecture of a two dimensional convolution which uses conventional method. Stride of one is used in this architecture, with a filter size of three. Streaming inputs are denoted as a0, a1, a2, etc. and the filter coefficients are b0, b1, b2 etc., which can be selected using the multiplexer. Registers used to shift and store the input data are denoted as r0, r1 and r2.

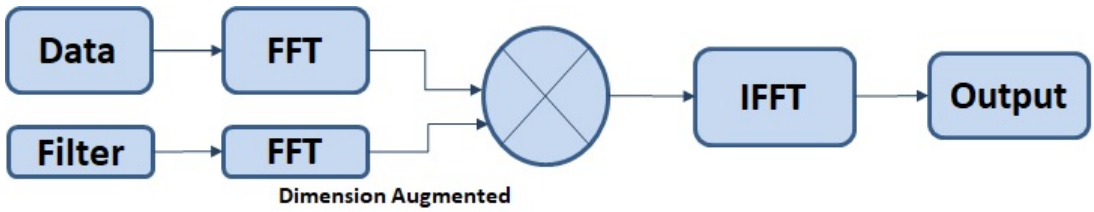

Figure 4.5: FFT based convolution

4.2 FFT Convolution

FFT based convolution is a fast algorithm for computing convolutions. Convolution can be efficiently performed in frequency domain. Equation (4.1) gives mathematical representation of FFT based convolution.

$$x(n) * h(n) = IFFT\{FFT(x(n)) \times FFT(h(n))\} \tag{4.1}$$

Here, Fourier transform of the input image and kernel are taken thereby transforming them to frequency domain. Both FFTs should be of same length for element-wise multiplication. Perform inverse FFT to get the results in time domain. Fig. 4.5 shows FFT based convolution scheme. FFTs of the input feature and the kernel are taken and are multiplied together. To perform matrix multiplication, augmentation is done. Inverse FFT is computed for the result. FFT based convolution gives high performance when kernel size is large. Computational complexity of FFT convolution is of the order $O(n^2 log(n))$ where n denotes FFT size.

4.3 Winograd minimal filtering

Winograd minimal filtering is a fast algorithm for computing convolution based on Chinese Remainder Theorem and involves polynomial multiplication. This algorithm can be used for computing convolutions when filter sizes are small. As filter size increases, transformation overhead also increases quadratically Zhuge *et al.* (2018). One dimensional Winograd algorithm for computing m outputs using an r tap filter will require $m + r - 1$ multiplications Lavin and Gray (2016). Consider $F(m, r)$ =$F(2, 3)$ where 2 outputs of a 3-tap filter are computed. The algorithm is given below, where d denotes input data and g denotes filter coefficients.

Table 4.1: Hardware Complexity of Winograd Algorithm

Operation	**Arithmetic complexity**
Winograd operation	4 Multiply
Input transform	4 Add
Filter transform	4 Add, 2 Division (Shift)
Output transform	4 Add
Total	4 Multiply, 12 Add, 2 Shift
Assuming offline filter transform	4 Multiply, 8 Add

$$\text{F(2,3)}=\begin{pmatrix} d_0 & d_1 & d_2 \\ d_1 & d_2 & d_3 \end{pmatrix} \begin{pmatrix} g_0 \\ g_1 \\ g_2 \end{pmatrix} = \begin{pmatrix} m_0+m_1+m_2 \\ m_1-m_2-m_3 \end{pmatrix}$$

where m_0, m_1, m_2 and m_3 are calculated as in Equation (4.2),

$$\begin{aligned} m_0 &= (d_0-d_2)g_0 \qquad & m_1 &= (d_1+d_2)\frac{g_0+g_1+g_2}{2} \\ m_3 &= (d_1-d_3)g_2 \qquad & m_2 &= (d_2-d_1)\frac{g_0-g_1+g_2}{2} \end{aligned} \tag{4.2}$$

Here, the input data is a tile of size $m+r-1$, ie., in the case of $F(2,3)$, it is 4. For computing m_0, m_1 , m_2 and m_3, only 4 multiplications are required. The algorithm require 4 additions involving data, 3 filter additions and 2 constant multiplications involving the filter. Finally the multiplied outputs are added together using 4 additions. Hardware complexity of Winograd algorithm is shown in Table 4.1. In matrix form, output Y is written as,

$$Y = A^T[(Gg) \odot (B^T d)] \tag{4.3}$$

where $\odot$ denotes element-wise multiplication.

Two dimensional (2D) Winograd algorithm can be implemented from nested one dimensional (1D) Winograd algorithm. 2D minimal algorithm for computing $m \times m$ output tile using $r \times r$ filter requires $(m+r-1) \times (m+r-1)$ multiplications, where $(m+r-1) \times (m+r-1)$ inputs are required. This can be denoted as shown in Equation (4.4).

$$\mu(\ F(m \times m, r \times r)) = \mu(\ F(m \times m)) \times \mu(\ F(r \times r)) \tag{4.4}$$

That is, input requirement and number of multiplications are same in minimal algorithm. This nesting can be used for non-squared filters and outputs also. Output Y can be written as,

$$Y = A^T[(GgG^T) \odot (B^T dB)]A \tag{4.5}$$

The transform matrices A, B and G can be precomputed, once the value of m and r are known. Conventional convolution for a one dimensional function $F(2,3)$ has 2×3 = 6 multiplications, whereas in minimal filtering algorithm, only $2 + 3 - 1 = 4$ multiplications are required. In the case of 2D convolution, $F(2 \times 2, 3 \times 3)$ involves $2^2 \times 3^2 = 36$ multiplications, whereas Winograd takes $4^2 = 16$ multiplications. There is a complexity reduction in terms of multiplication, by a factor $\frac{36}{16} = 2.25$. In general, the ratio of hardware complexity in terms of multiplication for conventional and Winograd convolution is given by Equation (4.6),

$$Ratio\ of\ Multiplications = \frac{m^2 \times r^2}{(m + r - 1)^2} \tag{4.6}$$

For computing convolution operation in CNN, each image channel is divided into tiles of size $(m + r - 1) \times (m + r - 1)$. Neighboring tiles have an overlap of $r - 1$ elements. For each of these channels $F(m \times m, r \times r)$ is computed and the results are summed up over all the channels.

Total number of multiplications per CONV layer = No. of tiles $\times (m + r - 1)^2$

For an $H \times W$ input feature with C channels and K number of kernels,

$$Total\ number\ of\ multiplications = H \times W \times C \times K \times \frac{(m + r - 1)^2}{m^2} \tag{4.7}$$

As the tile size increases, number of additions and constant multiplications also increase. The size of transform matrix also increases with tile size. The transform matrices contain coefficients which needs approximation. Filtering require less numeric precision as discussed in Lavin and Gray (2016). Acceleration rate will be higher if Winograd tile is large. Transform matrices for various tiles are given in Apendix. $F(6 \times 6, 3 \times 3)$ involves $(6 + 3 - 1)^2 = 84$ multiplications, whereas conventional convolution uses $6 \times 6 \times 3 \times 3 = 324$ multiplications. Winograd algorithm gives complexity reduction by a factor of $5.06\times$.

Table 4.2: Design parameters for convolution

Tile	input	output	kernel size
$F(3,2)$	4	3	2
$F(2,3)$	4	2	3
$F(3,3)$	5	3	3
$F(4,3)$	6	4	3
$F(6,3)$	8	6	3

4.4 Depthwise Separable Convolution

Novel CNNs like MobileNet, ShuffleNet etc uses depthwise separable convolution approach which reduces the computational complexity in terms of operations and number of parameters. Depthwise separable convolution is a factorized form of conventional convolution, where convolutions are split into depthwise and pointwise convolutions Wu *et al.* (2019). In depthwise convolution, kernel is slided over each of the input channel individually to obtain the output channel. Number of input and output channels are same here. In pointwise convolution, conventional convolution takes place with 1×1 kernel. This factorization will reduce the complexity of computation and number of parameters.

4.5 Analysis of Convolution Schemes

FFT and Winograd filtering are fast algorithms for performing convolutions. However these algorithms are optimum at certain input feature size and kernel size. For large kernel sizes either direct convolution or FFT based convolution is suited, while Winograd algorithm gives better performance for small kernel sizes.

We have conducted experiments with various kernel sizes and input feature maps for conventional, Winograd minimal filtering and also FFT convolution. The kernel sizes used and the input feature map considered are shown in Table 4.2. We have chosen radix-2 FFT for convolution computations. Zero padding has been done to get the powers of two. We have chosen $F(2,3)$, $F(3,3)$, $F(4,3)$, $F(3,2)$ and $F(6,3)$ Winograd tile sizes for experiments. Fig. 4.6 shows the execution time of convolution operations with varying input size and filter size for different convolution methods.

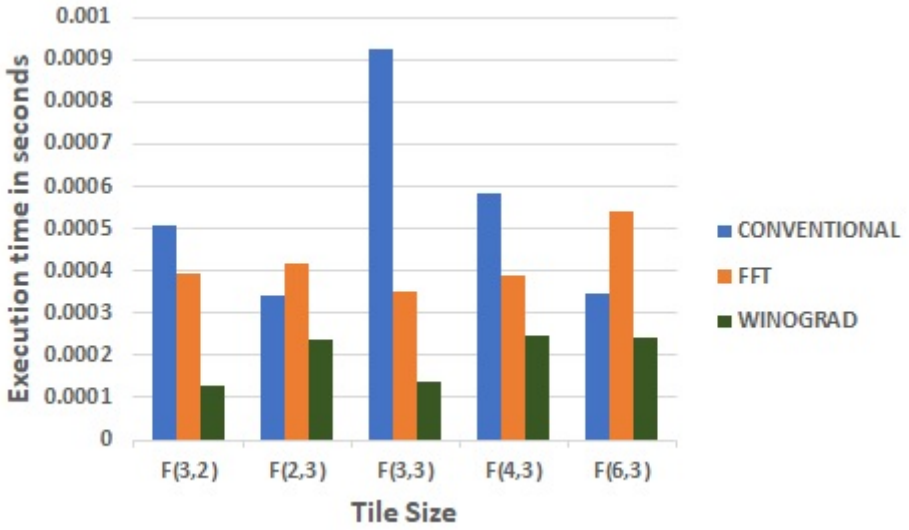

Figure 4.6: Execution time for various convolution schemes

4.6 Training and Inference of CNN Models

Training of a network refers to a process where an algorithm will learn to approach a problem. Training involves selection of weights of the network so as to produce the desired output. Running a network with these computed weights is called as inference. In other words, inference is referred as execution of the feed forward path of a neural network. Graphic Processing Units (GPUs) are efficient for training and are widely used for accelerating CNNs. Training and testing of deep learning models became easier with the introduction of several deep learning frameworks like Caffe Jia *et al.* (2014), Tensorflow and Torch Collobert *et al.* (2011) which can use CUDA programming. Many companies have launched deep learning accelerators for CNN inference and training. Google's second generation tensor processing unit (TPU), Nvidia's deep learning accelerator (NVDLA) and Intel's Nervana neural network processor (NNP) are some of them. This Section briefs about the experiments conducted for training and inference of some of the popular CNN models like AlexNet, VGGNet and ResNet on platforms like CPU, GPU and Nvidia Jetson TX2 board. Training period and feed forward network simulation time are compared for all these platforms.

Table 4.3: AlexNet Layer Configuration

Layers	$Feature_{in}$	Kernel	#Kernels	#Channels	Stride
CONV1	227×227	11×11	96	3	4
POOL1	55×55	3×3		96	2
NORM1	27×27			96	
CONV2	27×27	5×5	256	96	1
POOL2	27×27	3×3		256	2
NORM2	13×13		256	96	
CONV3	13×13	3×3	384	256	1
CONV4	13×13	3×3	384	192	1
CONV5	13×13	3×3	256	192	1
POOL3	13×13	3×3		256	2
FC6	4096 neurons				
FC7	4096 neurons				
FC8	1000 neurons				

4.6.1 CNN Models for Evaluation

For evaluation purpose, we have considered three popular CNN models which are discussed in the following subsections.

AlexNet Model

AlexNet was the first CNN model to win the ImageNet challenge in 2012 Krizhevsky *et al.* (2012). AlexNet CNN model is used for image classification tasks which can classify thousand categories. AlexNet consists of five convolution (CONV) layers and three FC layers. Kernels used in AlexNet are of sizes 11, 5 and 3. Configurations of AlexNet model are given in Table 4.3. Number of multiply and accumulate (MAC) operations in convolutional and fully connected layers are shown in Fig. 4.7.

VGGNet Model

VGG model got second place in ILSVRC challenge in 2014, with top-5 accuracy of 92.6%. VGG network model is available in three different versions, namely VGG-11, VGG-16 and VGG-19, based on the number of layers. Number of convolution stages

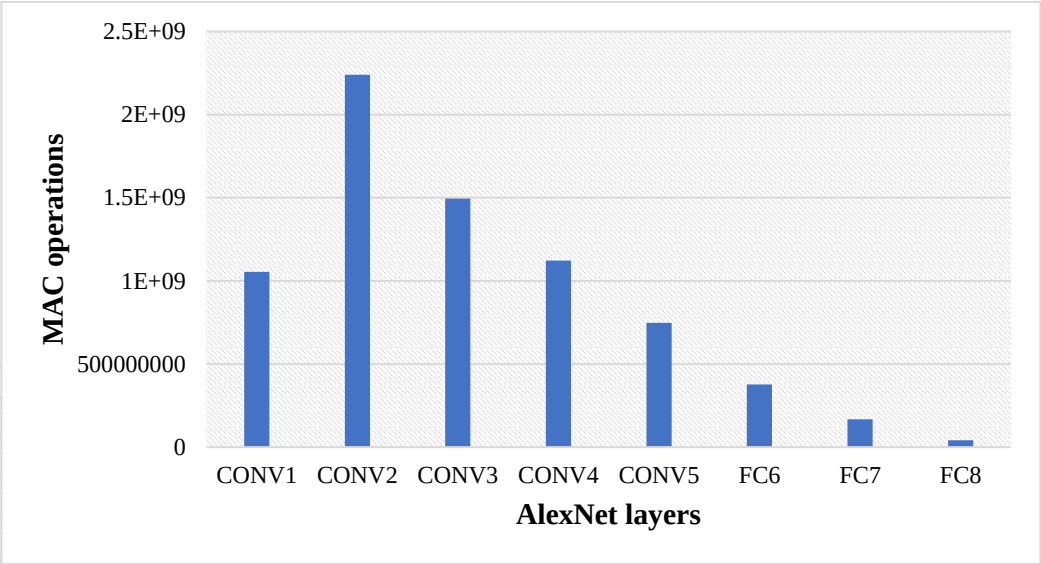

Figure 4.7: MAC Operations in AlexNet

Table 4.4: VGG network models

VGG Model	CONV stage 1	CONV stage 2	CONV stage 3	CONV stage 4	CONV stage 5	FC	Total
VGG-11	1	1	2	2	2	3	11
VGG-16	2	2	3	3	3	3	16
VGG-19	2	2	4	4	4	4	19

in VGG network model is given is Table 4.4.

We analyze VGG models namely, VGG-11, VGG-16 and VGG-19 with top-5 error rates 10.4%, 8.8% and 9.0% respectively. VGG models have five stages of convolution layers. VGG-11 has one CONV layer in first two stages and two CONV layers, each in the remaining stages. VGG-16 has two CONV layers in first and second stages, and three CONV layers in remaining each of the stages. VGG-19 also has two CONV layers in first two stages, whereas four CONV layers are present in each of the other stages. Max pooling layer is present after each stages in both the networks and a soft max operation is performed after the final FC layer. Here convolutions use a stride of one and the 2×2 max pooling uses a stride of two. There are three FC layers in all VGG models. The top-5 error rate of VGG-19 is greater than VGG-16, since the network started converging and hence addition of further layers will not improve accuracy. Number of MAC operations in different layers of VGG-16 are shown in Fig. 4.8.

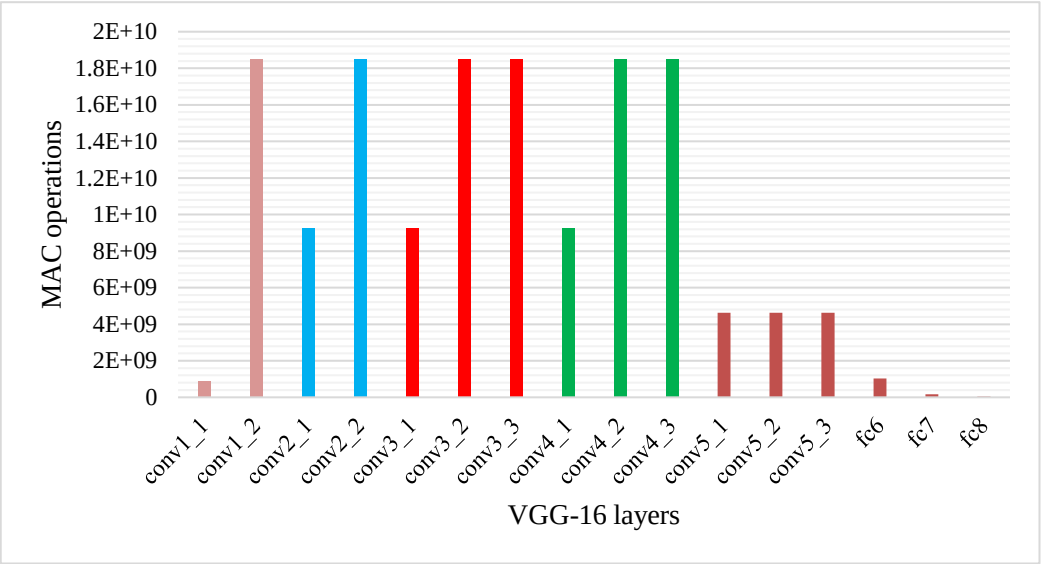

Figure 4.8: MAC Operations in VGG-16

ResNet Model

ResNet model won ISLVRC challenge in 2015 with top-5 error of 3.57% He *et al.* (2016). ResNet uses residual connections to avoid performance degradation and form highly deep networks. For ImageNet challenge, a depth of 152 layers were used in ResNet which has only one fully connected layer as the last layer. Each of the residual block has two 3×3 convolutional layers. ResNet uses depth of 34, 50, 101 or 152 layers for realization.

4.6.2 Implementation and Results

We have performed the analysis in three different platforms. GPU simulations are performed on Nvidia GeForce GTX 1080 Ti GPU with 3584 CUDA Cores, 1582 MHz, 11GB GDDR5X. We have used CUDA Version 8.0.61 and CuDNN library with Version 6.0.21. General purpose processor platform used for the simulation is Intel Core i7 7^{th} generation CPU, with frequency 3.60 GHz and 8GB RAM. For inference, no other workloads were distributed over CPU and GPU. We have also performed the implementation of AlexNet, VGG models (VGG-11, VGG-13, VGG-16) and ResNet-50 on Nvidia Jetson TX2 platform. Nvidia Jetson has been introduced as an artificial intelligence (AI) platform for autonomous machines. Jetson provides high efficiency with less

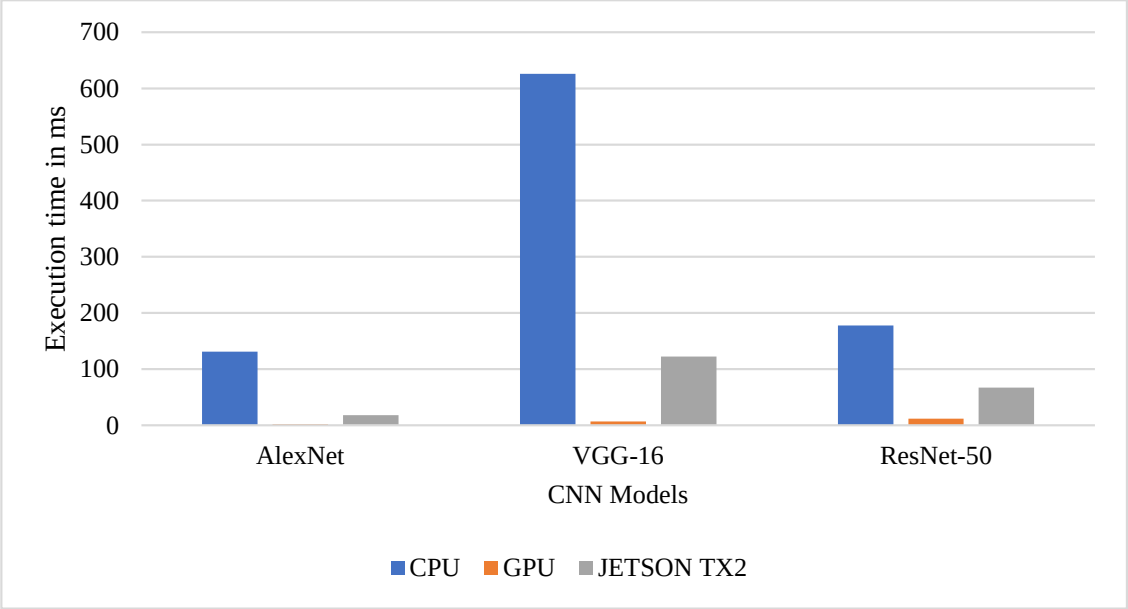

Figure 4.9: Comparison of feed forward simulations

power consumption and is widely used for low power scenarios like camera drones. Jetson TX2 is a fast and power-efficient embedded device for applications involving deep learning techniques. We have performed training and inference of CNN on Jetson TX2 which has 256 Nvidia CUDA cores. We have used ImageNet Russakovsky *et al.* (2015) dataset for performance evaluation, with input size 224×224.

Fig. 4.9 shows the execution time for feed forward path of AlexNet, VGG-16 and ResNet models in CPU, GPU and Nvidia Jetson TX2 platforms. Execution time for inference is much faster in GPU compared to general purpose processor and Jetson TX2 platform. Fig. 4.10 shows the training time for AlexNet, VGG-16 and ResNet-50 models in CPU, GPU and Nvidia Jetson TX2 platforms. As seen from Fig. 4.10, training period is much shorter in GPU compared to CPU and Jetson hardware simulations. Fig. 4.11 shows the feed forward network execution time and Fig. 4.12 shows the training period for VGG-11, VGG-16 and VGG-19 in CPU, GPU and Jetson TX2. Inference time and training time are large in case of CPU and minimum for GPU implementation for all the models. Even-though Jetson TX2 takes more time than GPU, Jetson boards are used widely for embedded devices, where a GPU is not advisable.

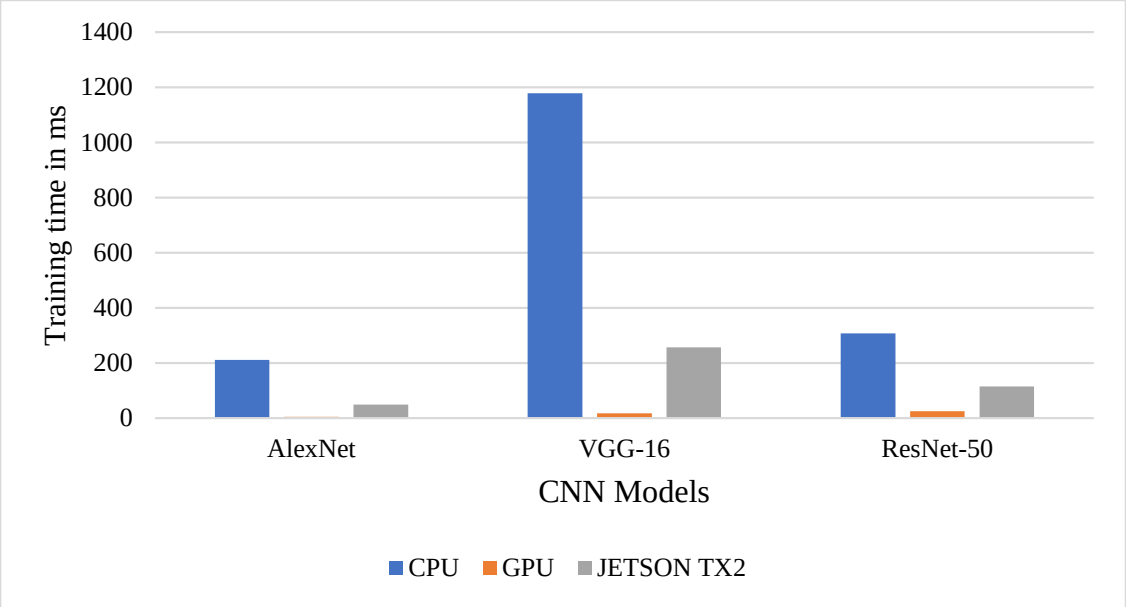

Figure 4.10: Comparison of training periods

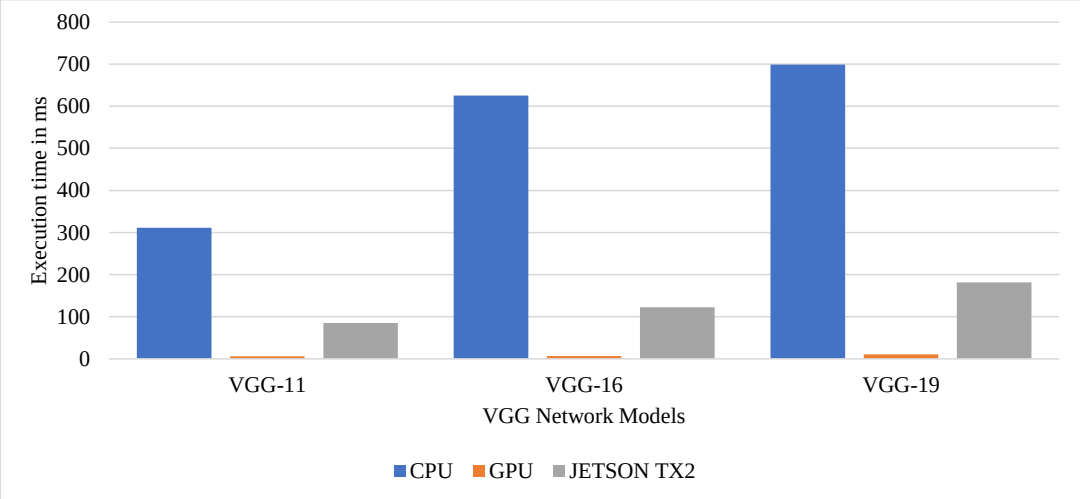

Figure 4.11: Comparison of inference time of VGG networks

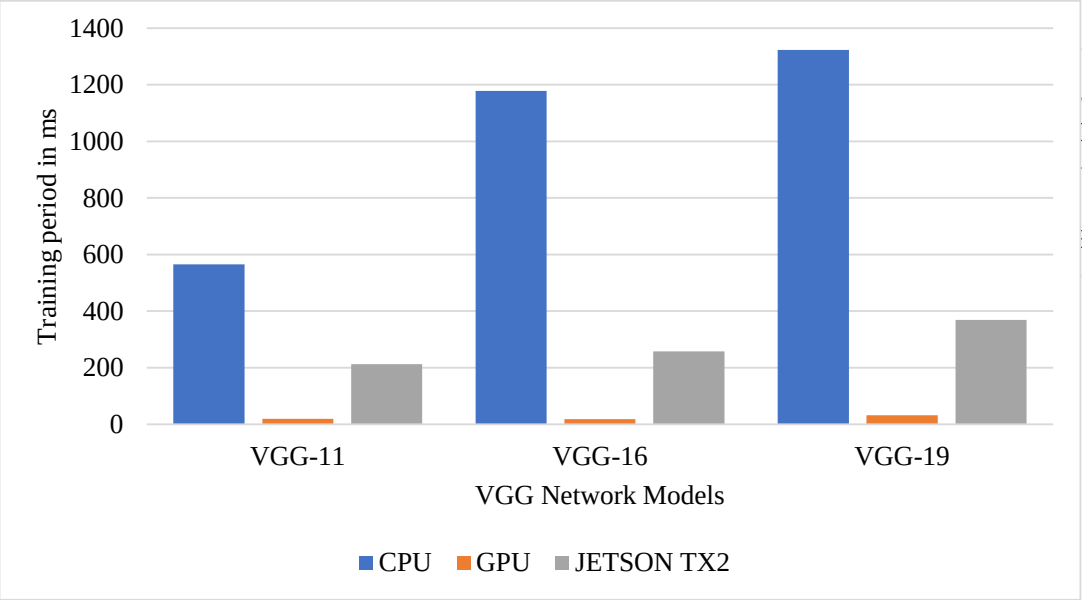

Figure 4.12: Comparison of training periods of VGG networks

Digit Recognition - Case study

We have performed the end-to-end simulation for digit recognition using LeNet CNN model, as a case study. The study includes prediction of a digit (0,1, ... 9) included in a handwritten figure. We have used MNIST dataset which consists of 60K handwritten digit (gray scale) images with dimensions of 28×28 pixels. The platform used was Jupyter Notebook and implementation has been done using Python. The python implementation uses Keras, OpenCV (CV2), Numpy libraries and tensorflow framework. Simulations were done on Intel Core i5-7200 CPU @ 2.50 GHz. The first convolution layer of CNN model uses 32 filters of size 5×5 and the activation used is ReLU. Second convolution layer uses 64 filters of size 3×3 size and ReLU activations. Max Pooling layer uses 2×2 filters.

4.7 Hardware Architecture for AlexNet Model

A hardware architecture for accelerating the first convolutional layer of AlexNet CNN model has also been implemented. First convolution layer in AlexNet model uses 11×11 filter with an input feature of size 227×227. Here, 96 filters and three input channels are used. First layer can be implemented using conventional convolution technique. It

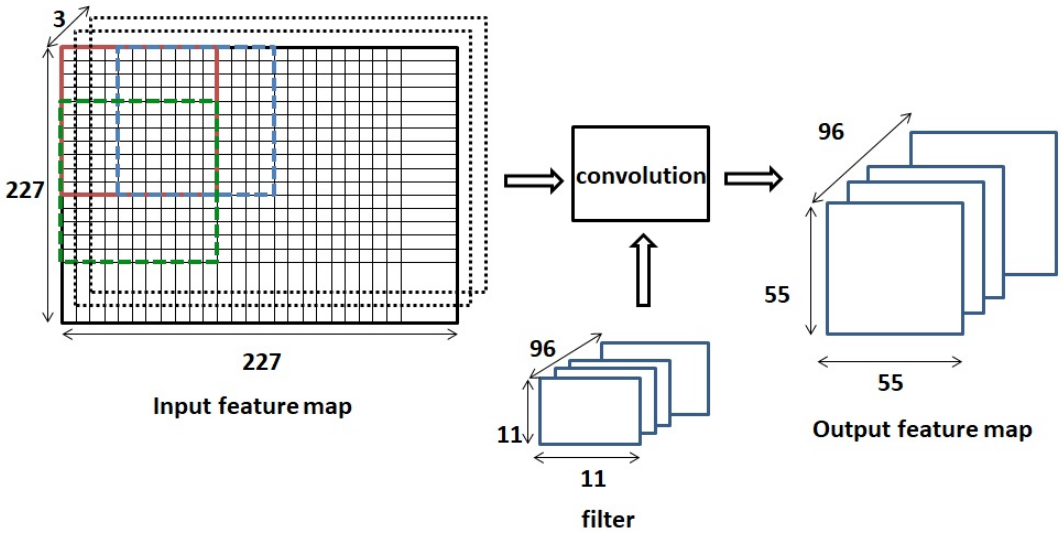

Figure 4.13: CONV1 layer of AlexNet

uses general matrix-matrix multiplication algorithm for realization. Representation of first CONV layer is shown in Fig. 4.13.

Even-though the algorithm follows general matrix multiplication, first layer can be realized in different ways. Possible ways of realizations are as follows:

- Parallel computation of all the output channels, with 96 multipliers, one for each output channel
- Sequential computation of the output channels, with 11×11 multipliers
- Implementing 11×96 multipliers, to perform row-computations in parallel

For better performance and comparable hardware resources, we implement proposed architecture using the third method. The architecture uses a series of MAC units in parallel for improving the throughput. Depth of parallelism is based on the available hardware resources and bandwidth. Proposed architecture for implementation of the first convolutional layer of AlexNet is shown in Fig. 4.14. The architecture performs convolution of an $M \times M$ input feature map with a $K \times K$ kernel, with C input channels and F number of kernels. Input features and kernel coefficients are accessed from DDR and the output feature is also written back to DDR. Single channel of $M \times M$ input feature is considered first, which has to be convolved with F kernels of size $K \times K$ each. Initially f data are fed in serial to the processing engine. We use x MAC units for performing a row convolution $(x = f)$. We have n such row convolution units which perform convolution operation in parallel, denoted as K_1 to K_n. Outputs of all the channels are added together, reusing the accumulators in MAC unit resulting in n number of

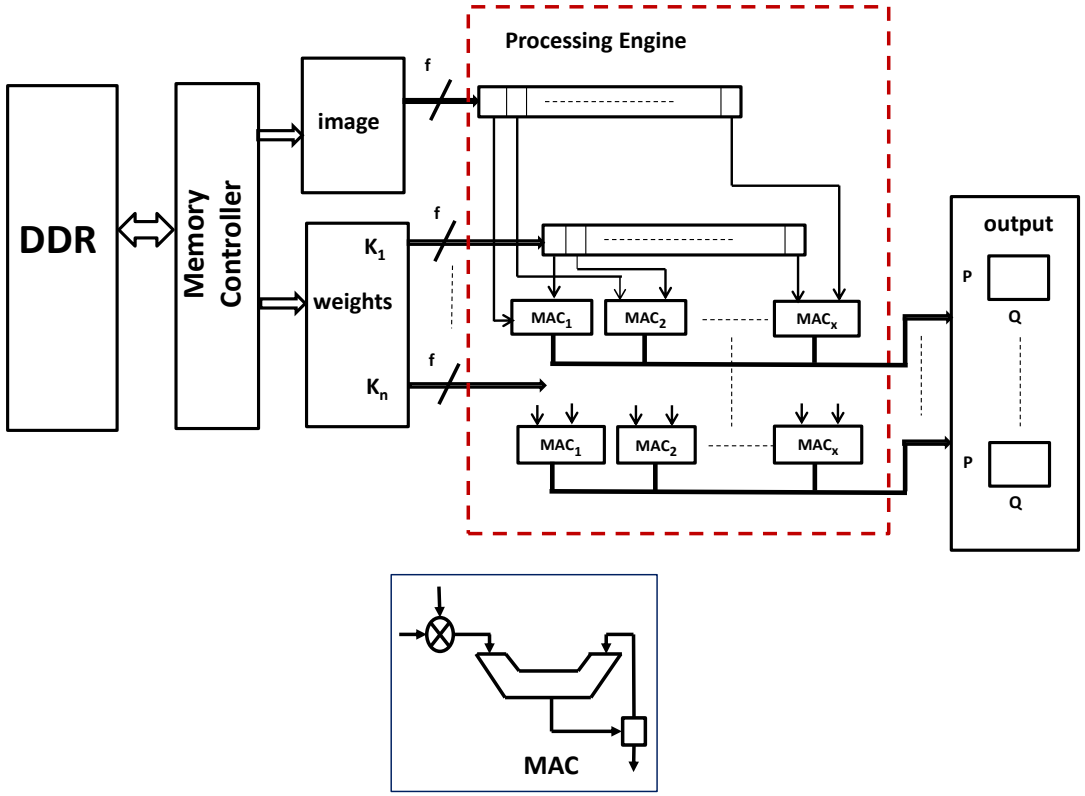

Figure 4.14: Proposed Architecture

$P \times Q$ outputs. In our architecture we use 11 MAC units per filter for performing a row convolution. We have 96 such row convolution units which perform convolution operation in parallel. Similarly three channels of input feature map in CONV1 are considered for computation.

4.7.1 FPGA Implementation Results

Convolution architecture for first layer of AlexNet model has been implemented on Xilinx XC7V2000 FPGA. We have used 32-bits floating point arithmetic for implementation. The architecture has been verified using Matlab simulation environment. Floating point multipliers in MAC unit are five stage pipelined. Proposed architecture has an operating frequency of 200 MHz. Resource utilization of the proposed architecture is given in Table 4.5. Since the architecture is a parallel convolution architecture, with 96 channels in parallel, the implementation gives a throughput of 96 samples per clock cycle. Our architecture gives a performance of 422 GFLOPs (Giga Floating point Operations Per second) for the first layer of convolution in AlexNet CNN model.

Table 4.5: Resource Utilization

Resource	Used	Available	Percentage utilization
LUTs	519312	1221600	42.51
DSP48E	2112	2160	97.78
Flip Flops	136513	2443200	5.59

Comparison of performance in terms of GFLOPs for CONV1 (first) layer of AlexNet CNN model as reported in various implementations are given in Table 4.6.

Table 4.6: Performance Comparison

Reference	Precision	Frequency	Performance (GFLOPs)
Proposed Work	32-bit floating point	200 MHz	422
Han *et al.* (2016)	32-bit floating point	189 MHz	27.01
Motamedi *et al.* (2016)	32-bit floating point	100 MHz	139
Shen *et al.* (2018)	32-bit floating point	200 MHz	59.7
Zhang *et al.* (2015)	32-bit floating point	100 MHz	27.5

But the architecture is specific for AlexNet model and is not efficient for other CNN models. So further research was carried out to come up with an efficient architecture for accelerating various popular CNN models and is discussed in the next chapter.

4.8 Summary of the Chapter

In this chapter, various convolution schemes like conventional, FFT based and Winograd minimal filtering techniques are discussed. Execution time for these schemes, with varying tile sizes are also presented in this chapter. Conventional convolutions are best suited for large kernel sizes and FFT based convolutions are suited when input and kernel sizes are matching. Winograd gives high performance when kernel size is small. Training and execution time of various CNN models on different platforms like CPU, GPU and Nvidia Jetson TX2 were also performed in this chapter. GPU takes much less time compared to other simulations. Hardware architecture for accelerating convolutional layers in AlexNet has been proposed and first layer is implemented on FPGA. Since this architecture does not give efficient acceleration of other CNN models, another high performance CNN accelerator is proposed in the next chapter.

CHAPTER 5

UniWiG: Unified Winograd-GEMM based CNN Architecture

In this chapter, a Unified Winograd-GEMM based CNN architecture, called as **UniWiG** is presented. The motivation for proposed architecture, design and implementation on FPGA and evaluation of results are discussed in this chapter.

5.1 Motivation

Various algorithms and architectures have been proposed for optimizing the computations in convolution layer. In the conventional approach, convolutions are reduced to general element-wise matrix multiplications (GEMM) Lavin and Gray (2016). Fast Fourier Transform (FFT) based convolution gives reduced computational complexity compared to conventional method Mathieu *et al.* (2013). However it is efficient only for large filter sizes or if both filter and data are of same size. Winograd minimal filtering algorithm gives efficient implementation of convolution layers when filter size is small. However the transformation is efficient only if input stride is equal to one. Performance can be improved by using a hybrid approach, in which each convolution layer is computed using the most appropriate algorithm as demanded by its filter size. Most of the convolution layers in typical CNNs use small filter sizes. These layers can be computed using Winograd algorithm. Layers with large filter sizes, layers with stride greater than one and fully connected layers can be implemented using general matrix multiplication.

Various research groups have proposed Winograd filtering algorithm based hardware accelerators for CNN Xiao *et al.* (2017); Podili *et al.* (2017); Yu *et al.* (2017); Zhuge *et al.* (2018); Lu *et al.* (2017). These approaches propose dedicated processing elements for performing Winograd based convolutions, which necessitates separate processing elements for convolution layers and fully connected layers, resulting in severe

under-utilization of resources. Performing all convolution and FC layer operations on same processing elements (PEs) can improve resource utilization. In Lu *et al.* (2017), PEs specialized for Winograd based convolution are also used to compute FC layers. However, FC layers have to fit the designed Winograd based PEs without any scope for optimizations. Also the PEs once implemented is tightly coupled with the Winograd filter sizes and hence use the same filter sizes for all layers.

In this chapter, a novel hybrid architecture which incorporates Winograd filtering algorithm on a GEMM accelerator is presented. This architecture consists of PEs which are optimized for GEMM and transform modules with very small resource overheads which map Winograd algorithm to GEMM operations. This unified architecture gives the most efficient implementation for CONV layers irrespective of filter sizes and also for FC layers. The proposed technique can be incorporated to any efficient GEMM accelerator. For demonstrating the technique we have used the FPGA based GEMM architecture in Shen *et al.* (2018).

The following are the major contributions in this chapter:

- We propose a unified architecture for performing general element-wise matrix multiplication (GEMM) as well as Winograd filtering algorithm using the same array of processing elements (PEs).
- Secondly, we propose a novel algorithm which transforms Winograd minimal filtering algorithm into blocked general element-wise matrix multiplication which targets optimal utilization of BRAMs and DDR bandwidth.
- An analytical model for estimating performance and BRAM usage has also been proposed, using which appropriate tile sizes for each layer can be identified.
- The proposed architecture has been used to accelerate AlexNet CNN model on FPGA and compared with existing architecture.

5.1.1 General Matrix Multiplication (GEMM)

Generally, matrix multiplication of two matrices **A** and **B**, to produce a matrix **C** is given by,

$$C_{i,j} = \sum_{k=0}^{N-1} A_{i,k} \times B_{k,j}, (0 <= i < M, 0 <= j < R) \tag{5.1}$$

Where **A** is an $M \times N$ matrix, **B** is a $N \times R$ matrix and **C** is a $M \times R$ matrix.

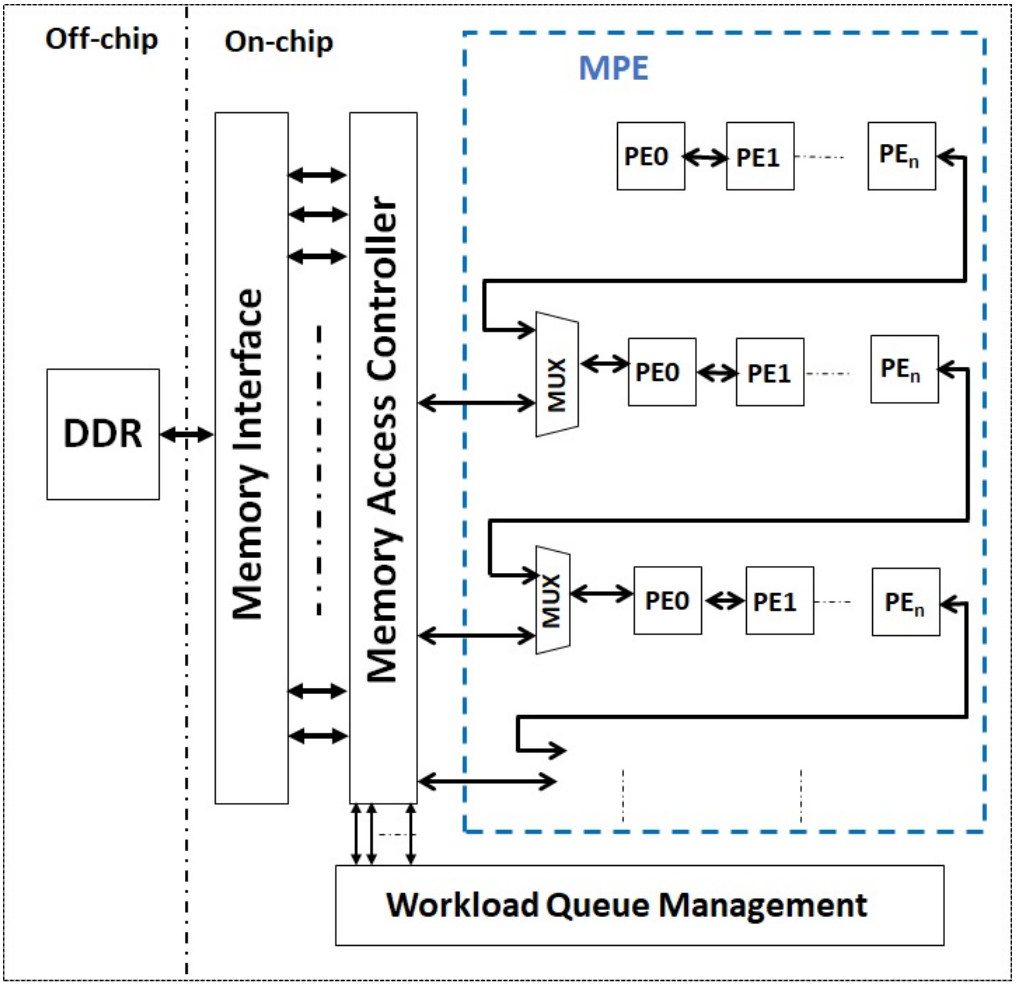

Figure 5.1: GEMM Accelerator in Shen *et al.* (2018)

This equation can be coded using three nested loops. Computational complexity of this algorithm is $2 \times M \times N \times R$, ie., $O(n^3)$. Bandwidth and memory bottleneck of matrix multiplication algorithms can be reduced by reusing the data Dou *et al.* (2005). This is performed using a technique called block multiplication as discussed in 5.1.2.

In Shen *et al.* (2018); Huang *et al.* (2018), a multi-array architecture for accelerating floating point matrix multiplication is presented. Here single array of processing engines are extended to multiple arrays so as to take the advantage of both throughput improvement and bandwidth budget. An overview of the GEMM accelerator in Shen *et al.* (2018) is shown in Fig. 5.1. Architecture in Huang *et al.* (2018); Shen *et al.* (2018) is composed of Memory Access Controller (MAC), Workload Queue Management (WQM) and Matrices Processing Engine (MPE). The architecture block diagram in detail is given in Huang *et al.* (2018). Input and output data are stored in external memory (DDR). Data transfer between DDR and processing engine is handled by MAC.

MPE consists of a number of linear PE arrays which can work in parallel. Workload queues for all the arrays are managed and load balancing is done by WQM.

5.1.2 Parallel Block Multiplication Scheme

FPGA implementations using direct convolution techniques have also been explored by various research groups. Convolution here is performed using general matrix multiplication algorithm (GEMM). In addition to CONV layers, FC layers can also be accelerated using the same hardware. In Dou *et al.* (2005) a matrix multiplication kernel using 64-bit floating point arithmetic is presented. Here a linear array of PEs is used to implement block matrix multiplication for any arbitrary matrix size. The key here is to efficiently exploit data re-usability. Matrices are partitioned into smaller size sub-matrices and computations are done within these sub-matrices. Fig. 5.2 gives the parallel block matrix multiplication scheme used in Dou *et al.* (2005).

Consider two 4×4 matrices A and B whose product is matrix C as shown in Fig. 5.2. Matrix C is partitioned into sub matrices of size 2×2. For simplicity, we consider an array consisting of only two processing elements (PE). For obtaining one sub block of C, elements of two rows and two columns of A and B respectively, are required. Sub block is initialized with data c^0_{xx}. Either first or last data pair from A i.e, $(a11, a21)$ or $(a14, a24)$ and from B i.e, $(b11, b12)$ or $(b41, b42)$ are loaded to PE0 and PE1 respectively. These are added to the data in c^0_{xx}, which is a MAC (Multiply And Accumulate) operation and results in sub matrix c^1_{xx}. The operation is continued with other elements in A and B to get the remaining intermediate results c^2_{xx} and c^3_{xx}. After 4 iterations, the final sub matrix result c_{11} to c_{22} is obtained. PE0 and PE1 will repeat these computations until all the elements of C are computed.

5.1.3 Transforming Winograd Minimal Filtering Algorithm to GEMM

Consider a CONV layer with a bank of F filters and C channels of size $R \times R$ convolving with an image of size $H \times W$. Filter elements are denoted by $G_{f,c,u,v}$ and image elements by $D_{c,x,y}$. Note that we are looking at a single image separately and not trying to batch together a set of images as normally done. Using Winograd filtering algorithm

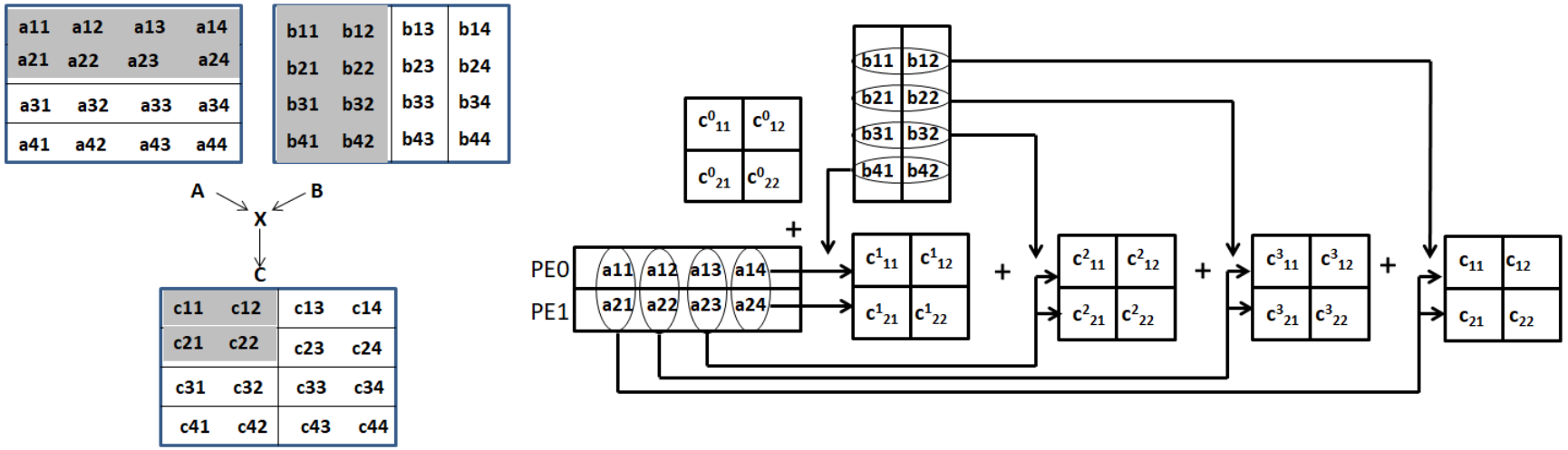

Figure 5.2: Parallel Block Matrix Multiplication Scheme

$F(m, r)$, the output Y is given by (5.2). That is

$$Y = A^T[(GgG^T) \odot (B^T dB)]A \tag{5.2}$$

where B, G and A are the data, filter and output transform matrices. Here, g is an $r \times r$ filter and d is an $(m + r - 1) \times (m + r - 1)$ data tile. There will be total $T = \lceil (H/m) \rceil \times \lceil W/m \rceil$ data tiles per channel. (5.2) can be rewritten as, Substituting $U = GgG^T$ and $V = B^T dB$, and labeling the tile co-ordinates as $(\tilde{x}, \tilde{y})$, Equation (5.2) can be transformed as

$$Y_{f,\tilde{x},\tilde{y}} = A^T \left[\sum_{c=1}^{C} U_{f,c} \odot V_{c,\tilde{x},\tilde{y}} \right] A \tag{5.3}$$

Note that each tile $(\tilde{x}, \tilde{y})$ is of size $(m + r - 1) \times (m + r - 1)$. By collapsing $(\tilde{x}, \tilde{y})$ to a single dimension t, Equation (5.3) becomes

$$Y_{f,t} = A^T \left[\sum_{c=1}^{C} U_{f,c} \odot V_{c,t} \right] A \tag{5.4}$$

Labeling each component in the tile as (ξ, ν) in the element-wise product term,

$$Y_{f,t} = A^T \left[\sum_{c=1}^{C} U_{f,c}^{(\xi,\nu)} \odot V_{c,t}^{(\xi,\nu)} \right] A \tag{5.5}$$

where $U = GgG^T$ and $V = B^T dB$.

The term $\sum_{c=1}^{C} U_{f,c}^{(\xi,\nu)} \odot V_{c,t}^{(\xi,\nu)}$ is matrix multiplication and can be represented as $U_{f,c}^{(\xi,\nu)} V_{c,t}^{(\xi,nu)}$.
Thus element-wise product has been transformed to $(m + r - 1) \times (m + r - 1)$ matrix multiplications with the two matrices being of size $F \times C$ and $C \times T$ respectively.

The above transformation was proposed by authors in Lavin and Gray (2016). It maps Winograd based convolution into GEMM operations between transformed input feature tiles and filter tiles. Each element of input tile and filter tile, after transformation is scattered to a different matrix. Transformed input matrix consists of corresponding elements from consecutive input tiles for a single channel along the column and one column for each channel. Transformed filter matrix is arranged as channels along the

rows and filters along the columns. To apply the above transformation and perform Winograd algorithm using GEMM, input data after tiling and data transform is represented as $m+r-1$ $C \times T$ matrices, where row t in i^{th} matrix represent i^{th} element in the transformed data tile t for all channels. Similarly the transformed filter has to be represented as $m + r - 1$ $F \times C$ matrices with each column representing filter coefficients for all channels for one filter.

The complete algorithm has the following steps. First, transform the image and filter tiles and scatter the elements to different matrices. Next, matrix multiplication is applied to corresponding matrix pairs and elements are gathered back and output transform is performed. GEMM based Winograd convolutions have been implemented in GPUs in Lavin and Gray (2016). Implementing this algorithm directly onto a hardware accelerator like FPGA with limited on-chip memory is not very efficient. Multiplying large matrices on such platforms is typically implemented using blocked matrix multiplication algorithms. Here both input matrices are divided into sub-matrices of smaller sizes and multiplication is performed between these smaller matrices. In this work we propose a *blocked GEMM based Winograd convolution algorithm* targeted towards platforms with limited on-chip storage.

5.2 Proposed Blocked Winograd Minimal Filtering Based Convolution Algorithm

Consider two input matrices **A** and **B** whose product is matrix **C**. A commonly used blocking scheme is to partition **A** along the rows and **B** along the columns. Size of sub-matrices (number of partitions) is fixed according to available on-chip memory. Typically one partition from **A** is fetched and stored locally and multiplied with multiple sub-matrices from **B**. This optimizes the required off-chip memory bandwidth and reduces data transfer from off-chip memory. We adopt this blocking scheme for our proposed blocked Winograd convolution algorithm.

Basic GEMM based Winograd filtering algorithm has been presented in Lavin and Gray (2016). We have modified this algorithm to create the blocked variant which is listed in Algorithm 2.

Algorithm 2: Blocked Winograd Filtering Based Convolution Algorithm for $F(m \times m, r \times r)$

$T = \lceil H/m \rceil \lceil W/m \rceil$ is the number of image tiles.
$q = m + r - 1$ is the input tile size.
Each input tile is $q \times q$ and tile stride is m .
$d_{c,t} \in R^{q \times q}$ is input tile t in channel c.
$g_{f,c} \in R^{r \times r}$ is filter f in channel c.
G, B^T and A^T are filter, data and output transforms.
S_T and S_F are the row and column block size.
for $\tau = 0$ to $\lceil T/S_T \rceil$ **do**
  for $\phi = 0$ to $\lceil F/S_F \rceil$ **do**
    for $f = \phi * S_F$ to $(\phi + 1) * S_F$ **do**
      for $c = 0$ to C **do**
        $u = G g_{f,c} G^T \in R^{q \times q}$
        Scatter u to matrices $U : U_{f,c}^{\xi,\nu} = u_{\xi,\nu}$
      end for
    end for
    for $t = \tau * S_T$ to $(\tau + 1) * S_T$ **do**
      for $c = 0$ to C **do**
        $v = B^T d_{c,t} B \in R^{q \times q}$
        Scatter v to matrices $V : V_{c,t}^{\xi,\nu} = v_{\xi,\nu}$
      end for
    end for
    for $\xi = 0$ to q **do**
      for $\nu = 0$ to q **do**
        $Z^{\xi,\nu} = U^{\xi,\nu} V^{\xi,\nu}$
      end for
    end for
    for $f = 0$ to S_F **do**
      for $t = 0$ to S_T **do**
        Gather z from matrices $Z : z_{\xi,\nu} = Z_{f,t}^{\xi,\nu}$
        $Y_{f,t} = A^T z A$
      end for
    end for
  end for
end for

The algorithm is pictorially shown in Fig. 5.3. Initially, an input matrix of size $H \times W$ and C channels get transformed to C channels of T tiles each of size $q \times q$. Note that $T = \lceil (H/m) \rceil \times \lceil W/m \rceil$ here. $Q_t(i.e, q \times q)$ elements of a tile are scattered to Q_t different matrices. Corresponding element from all tiles are arranged along columns and each column corresponds to one channel. Thus there will be Q_t matrices of size $T \times C$. These matrices are divided into sub-blocks along the rows with S_T being the number of rows in a sub-block. Similarly, there are Q_t transformed filter matrices each of size $C \times F$. These matrices are blocked along the columns with each sub-block consisting of S_F columns. Matrix multiplication between an input and filter sub-block generates output sub-matrices of size $S_T \times S_F$. This is repeated for all pairs of sub-blocks in all the Q_t input and filter matrices. Each input and filter matrix will give $\lceil \frac{T}{S_T} \rceil \times \lceil \frac{F}{S_F} \rceil$ sub-matrices. Data from identical locations from the set of Q_t sub-matrices are gathered together to reform a tile. Output transform is performed on each of these tile to give output tiles of size $mS_T \times mS_F$.

Basic algorithm in Lavin and Gray (2016) transforms the Winograd operation into q×q matrix multiplications where $q = m + r - 1$ is the input tile size. Each multiplication is between matrices of size T×C and C×F where T, C and F are number of input tiles, channels and kernels respectively. In Algorithm 2, first matrix is divided into sub-matrices of size $S_T \times C$ and second matrix into sub-matrices of size $C \times S_F$. Input and kernel transforms are performed at sub-matrix level, GEMM is performed on the transformed sub-matrices, and resultant product sub-matrix is transformed to get S_T output tiles (each of size $m \times m$) for S_F kernels. This process is repeated for all tiles and filters. Hereafter we denote F$(m \times m, r \times r)$ as F(m, r).

By transforming Winograd filtering to GEMM, a unified architecture can be used for accelerating all layers of CNN. This architecture will consist of processing elements for performing GEMM and additional modules for performing input, filter and output transforms. We call this unified architecture as *UniWiG*.

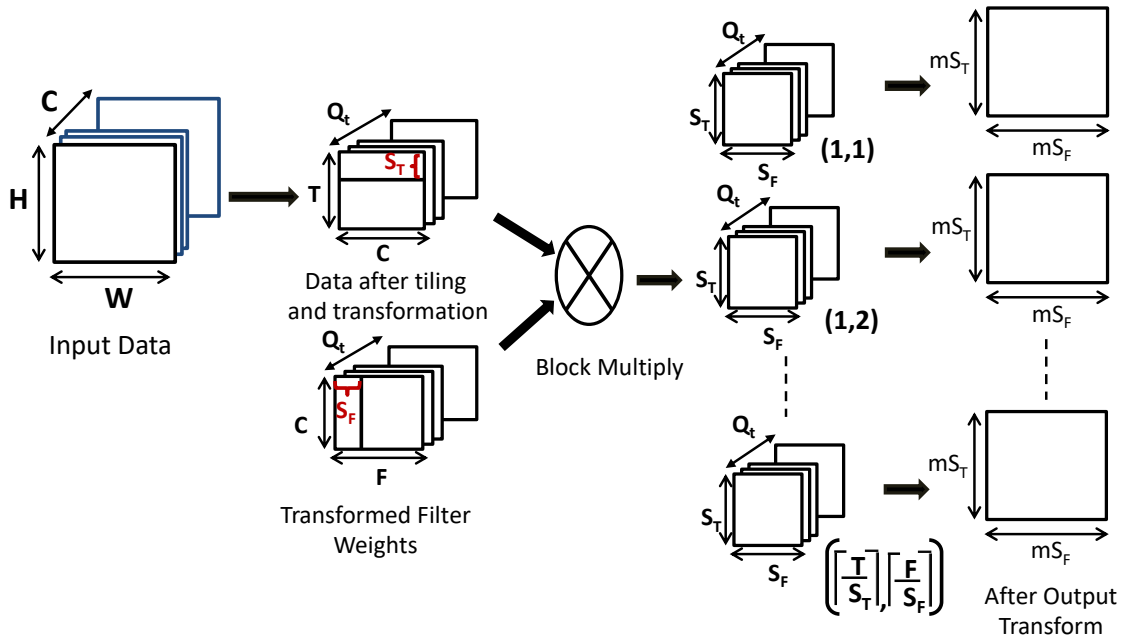

Figure 5.3: Blocked Winograd Minimal Filtering Algorithm

5.3 Proposed Unified Winograd-GEMM Accelerator Architecture

Implementing GEMM on FPGAs involve trade-off between performance, external memory bandwidth and Block RAM (BRAM) utilization. Blocked Winograd based convolution algorithm presented in Section 5.2 is targeted at memory constrained scenarios and can be mapped to any FPGA based GEMM accelerator. In this work, we use the state-of-art systolic multi-array architecture presented in Shen *et al.* (2018) for accelerating GEMM. Hereafter we refer Shen *et al.* (2018) as the baseline architecture.

We have modified the accelerator by including additional modules for performing input and output transforms. Input tiles d have to be transformed using $B^T dB$ and filter tiles g have to be transformed using GgG^T before passing to the GEMM accelerator. Similarly products from GEMM z has to be transformed using $A^T zA$. Filter coefficients are constant for a given convolution network model and hence the filter transform are applied off-line and transformed filter coefficients are stored in DDR. Fig. 5.4 gives the

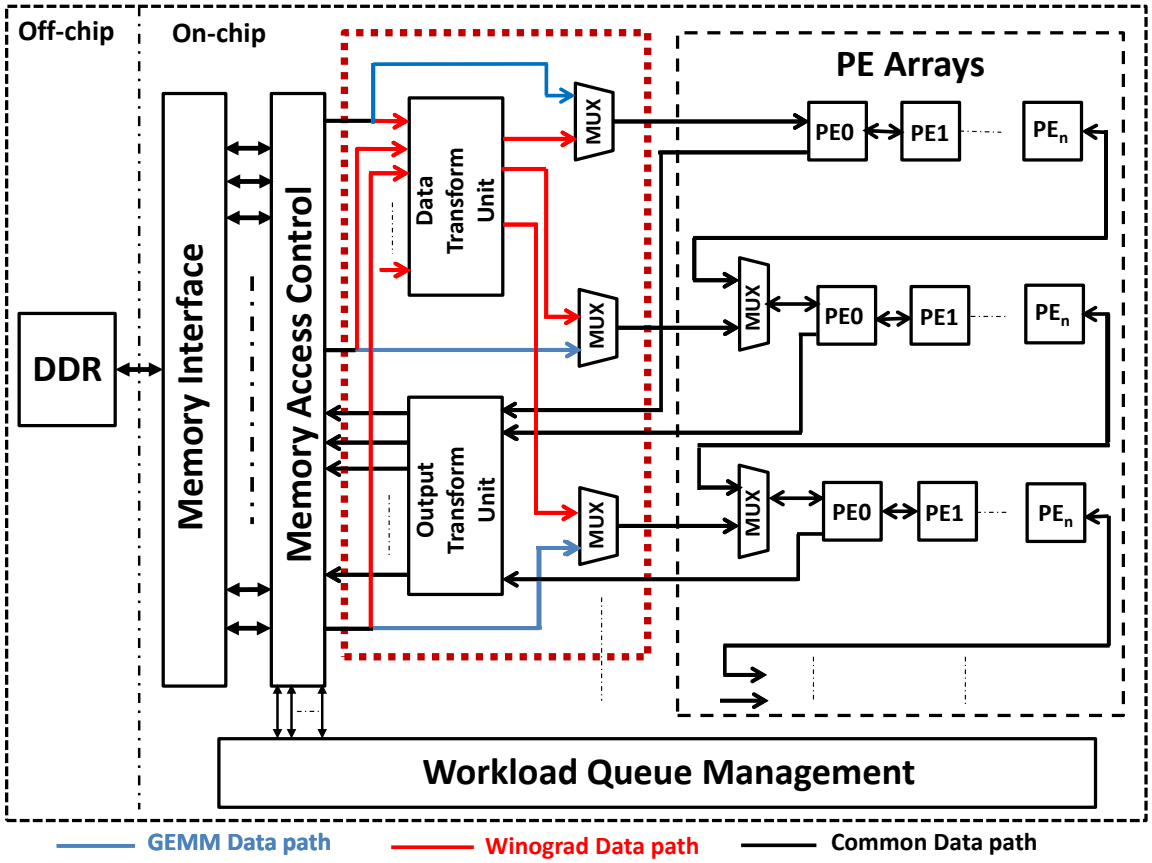

Figure 5.4: Proposed Unified Winograd-GEMM Architecture

high level block diagram for proposed unified Winograd-GEMM architecture. Two additional blocks namely *Data Transform Unit (DTU)* and *Output Transform Unit (OTU)* are introduced between the *Memory Access Control* and the PE arrays.

As explained in Section 5.2, we divide the input feature matrices and filter matrices into sub-blocks of size S_T and S_F respectively. The first step is to tile the input feature sub-block and apply the transformation $(B^T dB)$. *Data Transform Unit* in Fig. 5.4 performs this transformation. Outputs from *DTU* are sent to PE array. We do not compute the transform $(G^T gG)$ on-line. Instead these operations are computed off-line and transformed filter matrices are used directly. Filter coefficient sub-blocks are fetched directly to PE array. PEs perform sub-block multiplication and send the output to *Output Transform Unit*. This unit computes the operation $(A^T(z)A)$. Additional multiplexers are introduced so that *DTU* and *OTU* can be bypassed to perform normal GEMM computations.

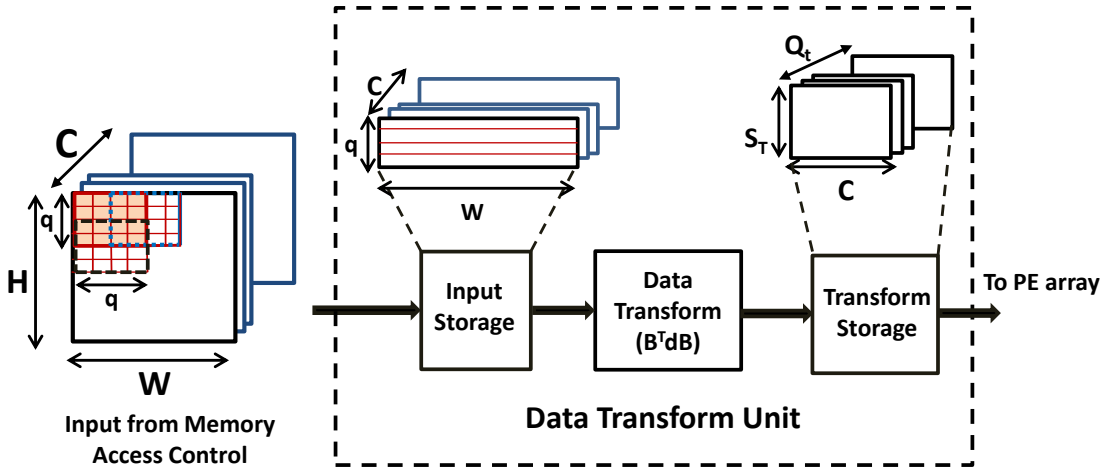

Figure 5.5: Data Transform Unit

Data Transform Unit (DTU)

Fig. 5.5 shows the block diagram for *Data Transform Unit*. Input data tiles are assembled here and the transform $(B^T dB)$ is applied. Blocked Winograd algorithm requires tiles to be created as sub-blocks of size S_T. *DTU* transforms tiles in a batch of $S_T \times C$, where C is the number of channels. Input tile size is $q \times q$ and elements from q rows are required to create one tile. One complete row is fetched together to optimize DDR bandwidth. q such rows from each channel are fetched and stored in *Input Storage*. Initially q rows from first channel is fetched and stored. $Q_t = q \times q$ elements corresponding to a tile are read out from *Input Storage* and sent to *Data Transform* stage. This is repeated for all tiles in the first q rows. Rows for subsequent channels are fetched and stored while previous fetched channels are transformed. These steps are repeated for all channels. Once all tiles from first q rows are transformed, next m rows are fetched from DDR and tiles with a stride of m are created and transformed.

Transform matrix B used in *Data Transform* stage is a constant for a given Winograd tile size and the operation can be reduced to additions and constant multiplications. Elements of transformed tiles are stored as elements of Q_t different matrices in *Transform Storage*. Elements from subsequent tiles for same channel are stored in subsequent rows of the same column. Each column stores elements corresponding to a single channel.

Input data matrix is transformed as sub-blocks of S_T rows. Once one sub-block of

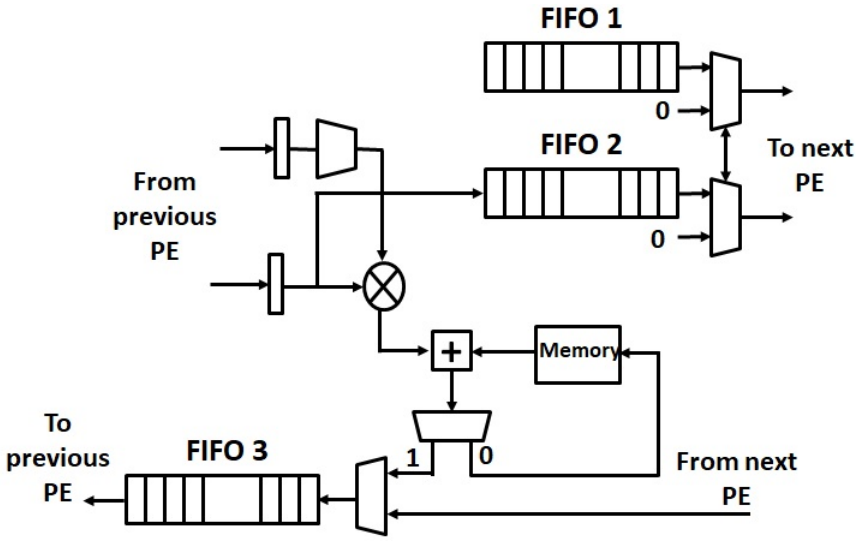

Figure 5.6: Processing Element (PE) Shen *et al.* (2018)

transformed matrix for all channels are computed, these can be sent to the PE array to be multiplied with filter sub-blocks.

Processing Element Arrays

Processing element (PE) arrays form the computational core of the accelerator. PEs are designed for performing one MAC operation every cycle. Architecture of a PE, adapted from Shen *et al.* (2018) is shown in Fig. 5.6. Multiple such PEs are connected together to operate as one systolic processing unit. Consider multiplication of two sub-blocks S_1 and S_2 of size $M \times K$ and $K \times N$ respectively. Assume there are M PEs in one array. M elements from first column of S_1 is distributed among the PEs. Each PE then takes N elements from first row of S_2 and multiply these elements with stored element from S_1 and results are stored in a local buffer. This operation is repeated with second column of S_1 and second row of S_2 and products are added with previously stored intermediate results. These steps are repeated with all K rows of S_1 and columns of S_2. The whole sub-block multiplication takes $N \times K$ cycles if we ignore cycles for fetching data and pipeline latencies in the MAC. In Shen *et al.* (2018), authors have provided detailed micro-architecture for the PE and exact analysis for computation time.

PE arrays can operate in *Independent* mode and *Cooperation* mode. In *Independent* mode each array works separately on different workloads without communicating with each other. In *Cooperation* mode, adjacent PEs are connected together so that they can

work on a single workload. *Cooperation* mode is useful for larger block sizes and need lower off-chip memory bandwidth. In our implementation, for Winograd convolution, we operate the PE arrays in *Independent* mode. For GEMM based Winograd convolution, number of rows in the first matrix T (refer Fig. 5.3), for most layers is less than 32 which is the number of PEs in one array. As a result, having arrays of large size will lead to underutilization of PEs. In case of fully connected layers and direct convolution operation, arrays are operated in *Independent* mode or *Cooperation* mode as demanded by the workload.

PE array computes product of two sub-blocks namely, input sub-block of size $S_T \times C$ and the filter sub-block of size $C \times S_F$ respectively. Filter sub-block is directly fed to PE array from *Memory Access Control* unit. During prefetch stage, one row of the filter sub-block consisting of S_F elements are fetched. Each element is sent to a different PE element in the array. S_T elements corresponding to a column of input sub-block is then fetched from *DTU*. First element from the column is sent to all PEs and multiplied with the saved filter element. This is repeated for all S_T elements and results are stored in the PE memory. Next column from input sub-block is fetched and multiplied with the same filter elements. Products are added appropriately with previously stored products in PE memory. This procedure is identical to the scheme used in Shen *et al.* (2018) with one key difference. In Shen *et al.* (2018) columns of first matrix are distributed and saved to the PEs, while here rows from the second matrix are distributed. Also here the input sub-block is fetched from *DTU* instead of *Memory Access Control*.

Q_t input and filter sub-blocks are distributed among N_p PE arrays and products are sent to *OTU*. This ensures that, products for all elements of each input data tile are computed upon which the output transform can be applied. Next, a new filter sub-block is fetched and multiplied with the same Q_t input sub-blocks. Reuse of input sub-block ensures that input data transform need to be performed only once for each input tile.

Memory Access Control unit manages data transfer between DDR and accelerator. Workloads are organized using *buffer descriptors* similar to the scheme used in Shen *et al.* (2018). *Workload Queue Management* block will assign workloads for various PE arrays. It uses a work stealing approach to distribute workloads uniformly among multiple PE arrays.

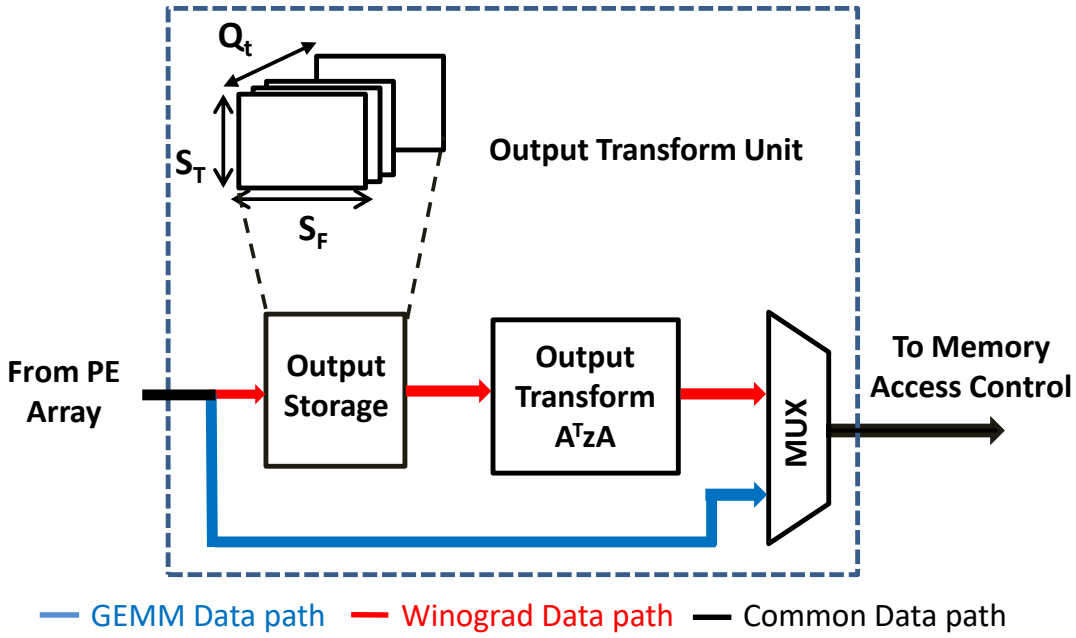

Figure 5.7: Output Transform Unit

Output Transform Unit (OTU)

Fig. 5.7 gives the block diagram for *Output Transform Unit*. Q_t sub-block products are stored in the *Output Storage*. Corresponding elements from all Q_t sub-blocks are collected to reform the tile and output transform is applied by multiplying with A^T and A. Similar to the transform in *DTU*, this transform can also be converted to constant multiplications and additions. Each tile gives outputs of size $m \times m$ after the transform. These outputs are sent to *Memory Access Control* to be written back to DDR.

DTU and *OTU* are designed such that multiple Winograd tile sizes are supported. Depending on the chosen size, transform units can be dynamically reconfigured.

5.3.1 Block RAM Memory Requirement

We estimate the extra BRAM required for *DTU* and *OTU* based on input and filter parameters. *Input Storage* needs to store $2m$ rows of image data for each class C. *Transform Storage* stores Q_t matrices of size $S_T \times C$ while *Output Storage* stores Q_t matrices of size $S_T \times S_F$. We use single precision floating point data so that total additional BRAM locations required is $4 \times (2mWC + Q_t S_T C + Q_t S_T S_F)$ bytes. For single precision floating point w is 4 and for 16-bit fixed point, w is 2.

5.3.2 Performance Model

In Shen *et al.* (2018), authors have developed an analytical performance model for GEMM on multi-array architecture. This model can be modified to evaluate the performance of Winograd algorithm on our proposed modified multi-array architecture. Let N_p be the number of PE arrays working in parallel. Average number of sub-block multiplications for Winograd performed in one array is given by

$$N_{work} = \lceil \frac{Q_t}{N_p} \times \lceil \frac{T}{S_T} \rceil \times \lceil \frac{F}{S_F} \rceil \rceil \tag{5.6}$$

Given BW, the external memory bandwidth in bytes and w, the data word size in bytes, average time taken in seconds to load a pair of sub-blocks is given by

$$T_{work} = \frac{w(mWC\frac{S_T}{\lceil \frac{W}{m} \rceil} \times \frac{1}{\lceil \frac{F}{S_F} \rceil} \times \frac{1}{Q_t} + S_F C + S_T S_F)}{BW} \tag{5.7}$$

First term in the numerator in (5.7) gives the cycles to transfer extra rows of image data for generating S_T transformed tiles. Q_t such matrices are computed from a single input tile and transformed image sub-blocks are reused for $\frac{F}{S_F}$ filter sub-blocks. Second term gives the cycles for transferring the filter sub-blocks and third term gives the cycles for output write-back. Total time for data transfer is

$$T_{trans} = N_{work} \times T_{work} \tag{5.8}$$

Computation time in seconds for a PE array is given by

$$T_{comp} = \frac{N_{work}(S_F + Max(S_T, S_F)C + Stage_{fmac})}{F_{acc}} \tag{5.9}$$

Here $Stage_{fmac}$ gives the pipeline stages in PE and F_{acc} is the clock frequency of the accelerator.

Bounds for execution time is given as

$$Max(T_{comp}, T_{trans}) < T_{total} < (T_{trans} + T_{comp}) \tag{5.10}$$

If data transfer and compute are completely overlapping, then lower bound of execution time can be achieved. This performance model in conjunction with the BRAM model can be used to fix the Winograd tile size m for a target performance and available BRAM resources. Note that the basic model given in Shen *et al.* (2018) has to be used for estimating performance estimates for regular GEMM operations.

5.3.3 Other Layers of CNN

Apart from convolutional and fully connected layers, CNN contains other layers like pooling, Rectified Linear Unit (ReLU) and normalization. Pooling layers reduce the size of feature map by replacing the subregions within a window (eg., 2×2 kernel) with their maximum value, which is referred as Max Pooling. ReLU layers will give an output equal to zero, for any input value which is less than zero. These two layers are implemented using comparators after the output transform unit. Normalization unit performs normalization operation across the feature map, and it consists of squaring operation, addition and multiplication and can be implemented as given in Ma *et al.* (2018). Outputs from OTU are given to ReLU for applying non linearity, and these are sent to normalization unit. Outputs after normalization are sent to pooling unit. Each row of the PE array will have one ReLU unit, normalization unit and one pooling unit each. Since each row consists of 32 PEs, additional hardware complexity due to the above mentioned units is significantly less when compared to PE arrays.

5.4 Implementation and Results

5.4.1 Comparison with Baseline Architecture

Proposed accelerator (*UniWiG*) is targeted at accelerating both convolution and fully connected layers. We evaluate the effectiveness of this approach by comparing performance, hardware resource utilization and off-chip memory bandwidth of *UniWiG* with the baseline GEMM accelerator presented in Shen *et al.* (2018). Performance is generally measured in terms of Giga (Floating point) Operations Per Second (G(FL)OPS). As mentioned in Section 4.3, Winograd algorithm reduces computational complexity

of convolution operation. When we compare two architectures running two different convolution algorithms with different computational complexity, GOPS may not show the real difference in performance. This is especially true for large Winograd tile sizes like $F(6,3)$ where theoretical reduction in computations is as high as $5.06\times$. In such scenarios, throughput in terms of number of convolutions per second (CONV/s) can be used to compare performance.

In Shen *et al.* (2018), authors use Xilinx XC7VX690T FPGA for implementing their architecture. Authors have evaluated the performance of this accelerator for the popular AlexNet CNN model. Single precision floating point is used for data representation. We have used the same FPGA platform with the same CNN model and data representation for *UniWiG*. Both implementations have 256 PEs. However in *UniWiG*, we have used 8 arrays of 32 PEs whereas in Shen *et al.* (2018), authors have used 4 arrays with 64 PEs. Both implementations run at a clock frequency of 200 MHz.

AlexNet comprises of five CONV layers and three FC layers. First CONV layer uses filter matrices of dimension 11×11. Winograd algorithm is not targeted at convolutions involving such large filter sizes. So this layer and the FC layers are computed using normal GEMM algorithm.

Winograd tile size is a key parameter affecting the performance. Using the performance model and BRAM model presented in Sections 5.3.2 and 5.3.1, we estimate execution time and BRAM requirement for each layer. Tile sizes which give minimum execution time with less than 15% additional BRAM units were chosen. Each layer was mapped to the architecture with selected tile size, and performance was measured.

Table 5.1 compares the performance between the two architectures for all layers. For layers using Winograd based convolution, we compare both GFLOPS and CONV/s. Chosen tile sizes are also indicated. In terms of CONV/s, our approach gives $1.36\times$ to $4.03\times$ improvement in performance. Maximum improvement is for CONV2 which uses a larger tile size $F(6,3)$. Note that for all layers, increase in performance is less than the theoretical reduction in complexity for the selected tile size. For example, theoretically, savings in complexity for $F(6,3)$ is $5.06\times$. With reduced computations, off-chip data transfers are not completely hidden by the computations and is a major reason for this reduction in performance. Comparing GFLOPS, *UniWiG* is superior to the baseline for

Table 5.1: Performance comparison with baseline architecture in Shen *et al.* (2018)

Layer	Proposed Work			Shen *et al.* (2018)		Ratio of CONV/s
	Tech-nique	GFLOPS	CONV/s	GFLOPS	CONV/s	
CONV1	Direct	57.22	-	59.7	-	-
CONV2	Wino (6,3)	61.53	1581.21	87.8	392.22	4.033
CONV3	Wino (3,3)	72.6	592.2	64.9	217.07	2.72
CONV4	Wino (3,3)	72.25	786.05	64.1	571.77	1.36
CONV5	Wino (3,3)	72.84	1188.60	62.9	841.60	1.41
FC6	Direct	98.1	-	100.9	-	-
FC7	Direct	97.3	-	99.3	-	-
FC8	Direct	94.6	-	96.9	-	-

all layers except CONV2. For CONV2, there is a reduction in GFLOPS which indicates under utilization of PEs for larger tile sizes and needs further investigation.

For CONV1 layer and FC layers (which are computed using GEMM), we compare the performance in terms of GFLOPS in Table 5.1. Measured performance for these layers show slight degradation when compared to the baseline, even though both approaches use identical architectures for GEMM computations. This can be attributed to implementation differences most likely in *Memory Access Control* and *Workload Queue Management* blocks.

Table 5.2 compares FPGA resources for both implementations. Very less additional FPGA resources are used for introducing Winograd algorithm to the GEMM accelerator. Only 8.9% of total LUT resources and 13.4% of total BRAMs are required. Maximum off-chip bandwidth is required for FC layers and hence it remains unchanged between the two architectures. Note that percentages in Table 5.2 indicate the difference as a fraction of total available resources for the selected FPGA platform. This comparison shows that with the unified architecture, we are able to achieve significant performance improvement with low overhead in hardware resources and DDR bandwidth, which shows the effectiveness of our approach.

Table 5.2: Comparison of FPGA resources with baseline architecture in Shen *et al.* (2018)

Resource	Proposed Work	Shen *et al.* (2018)	% Increase
BRAM	757.5	560.50	13.4
DSP48E	1123	1032	2.5
Flip Flops	383K	292K	10.51
LUTs	252K	192K	8.9
DDR Bandwidth	1.6 GB/s	1.6GB/s	0

5.5 Summary of the Chapter

In this chapter, an efficient CNN accelerator based on unified Winograd-GEMM architecture has been presented. Blocked Winograd minimal filtering algorithm has been proposed to improve the performance of accelerator. Performance model of proposed architecture with estimations for BRAM requirement and compute time has been discussed. Proposed *UniWiG* architecture has been implemented on Xilinx XC7VX690T FPGA. The same FPGA platform as that used in Shen *et al.* (2018) has been selected for easier performance comparison. All the layers of AlexNet has been accelerated using single precision floating point arithmetic.

CHAPTER 6

Performance Analysis of CNN Models on UniWiG

Finding optimum tile size for Winograd algorithm is a critical step to maximize performance. In this chapter, selection of optimum tile size is explained in detail in the context of AlexNet, VGG-16 and ResNet-18 models. Estimated Performance and BRAM requirement for all layers are presented for all the models. Also, fixed point variant of the architecture with additional support for batch processing is discussed in detail. This is compared with existing state-of-art implementations.

6.1 Fixed point Implementation of UniWiG

Floating point implementations are expensive in terms of hardware resources. For most applications, required accuracy can be achieved with 16-bit fixed point precision Guo *et al.* (2018). We have also implemented a 16-bit fixed point variant of *UniWiG* on Xilinx XC7VX690T FPGA. In this section, we map AlexNet, VGG-16 and ResNet-18 CNN models on this implementation and analyze the performance and hardware utilization.

6.1.1 Batch Processing

Fixed point implementation with 256 PEs occupy only around 15% of FPGA resources. Hence we increased the number of PEs to 1280. They are separated into five groups of 256 PEs having eight 32 PE arrays. *Cooperation* mode is only allowed for arrays within a group. *DTU* and *OTU* modules are replicated for all groups, while a common DDR interface is used. Typical CNN implementations perform batch processing, with same convolution operations applied on multiple images in parallel. Fetched filter coefficients are used for all images in a batch. We configure the five groups of PE arrays to operate on five different input images. FC layer computations involve product of a

Table 6.1: AlexNet CONV parameters for various tile sizes

Parameter		CONV1	CONV2	CONV3	CONV4	CONV5
Input (H×W)		227×227	27×27	13×13	13×13	13×13
Filter (r×r)		11×11	5×5	3×3	3×3	3×3
Filters (F)		96	256	384	384	256
Channels (C)		3	48	256	192	192
Tiles (T)	$F(2,3)$	-	196	49	49	49
	$F(3,3)$	-	81	25	25	25
	$F(4,3)$	-	49	16	16	16
	$F(6,3)$	-	25	9	9	9

vector (inputs) and a matrix (filter). Batch processing is used to convert this to GEMM. Here each PE array group works on a batch of 128 inputs. Thus for CONV layers a batch of five inputs and for FC layers a batch of 640 images are processed in parallel. With this scheme, we maximize FPGA resource utilization and improve performance.

6.1.2 Fixing the Winograd Tile Size

AlexNet Model

Table 6.1 gives various parameters like input size, filter size and number of filters and feature inputs for each CONV layer in AlexNet. We have used Winograd algorithm for CONV layers with stride one only and other CONV layers were implemented using direct convolution. Thus CONV1 with 11×11 filter size is evaluated using direct convolution. For the remaining layers, number of tiles for various Winograd tile sizes are also shown. Larger filters and smaller inputs have less number of tiles. Note that, for CONV2, we have used the technique proposed in Guo *et al.* (2018) to convert 5×5 filter to four 3×3 filters. We apply these parameters for various tile sizes to the performance and BRAM model, and use the results to fix the optimum tile size. Fig. 6.1 shows estimated compute time for four convolution layers with different tile sizes. $F(6,3)$ gives lowest compute time for CONV2 and $F(4,3)$ for other layers. Table 6.2 gives the total number of BRAM instances required for each tile size. There are 1470 BRAMs in XC7VX690T FPGA. For each layer, we choose the tile size which gives minimum compute time and use less than 1470 BRAMs. For example, for CONV3, $F(4,3)$ which gives optimum compute time needs more than 1470 BRAMs and hence $F(3,3)$ is

Table 6.2: Estimated BRAM for various tiles in AlexNet

CONV Layer	F(2,3)	F(3,3)	F(4,3)	F(6,3)
CONV2	835	885	945	**1085**
CONV3	1050	**1225**	1670	2245
CONV4	980	1115	**1460**	1900
CONV5	980	1115	**1460**	1900

chosen. The selected tile sizes are $F(6,3)$ for CONV2, $F(3,3)$ for CONV3 and $F(4,3)$ for other layers. Corresponding BRAM numbers are marked as bold.

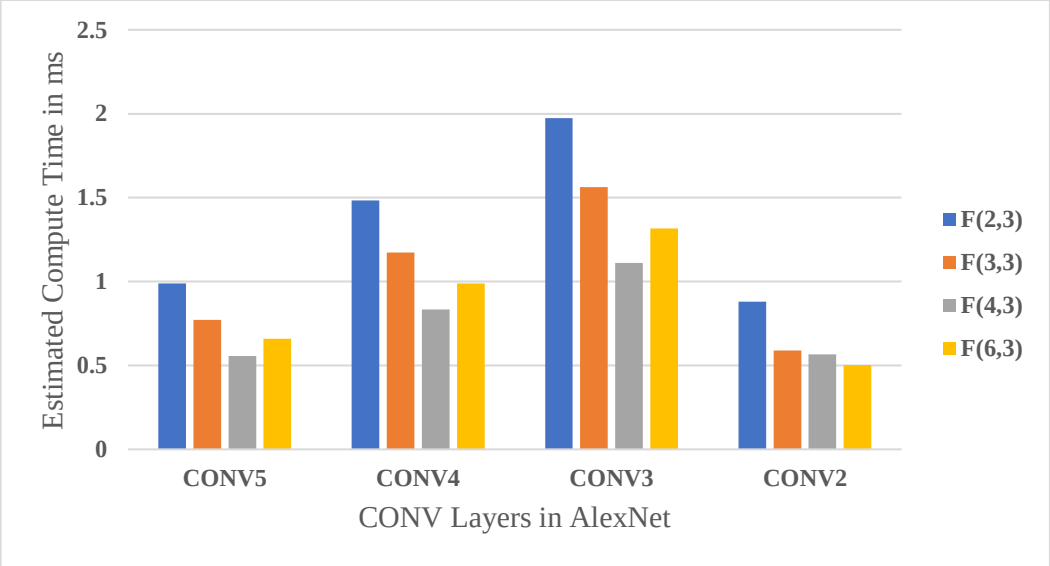

Figure 6.1: Estimated compute time for CONV Layers in AlexNet

VGG-16 Model

We repeated the above analysis for VGG-16 CNN model and fixed the optimum tile sizes. Here there are five stages of convolution layers with a total of 13 different layers and three FC layers. All convolution layers use 3×3 filter size. Compute time for different tile sizes are shown in Fig. 6.2 and corresponding BRAM requirement is given in Table 6.3. Layers with identical parameters are clubbed together. BRAM for selected tile sizes are marked as bold in Table 6.3.

ResNet-18 Model

Table 6.4 gives various parameters of ResNet-18 for different Winograd tile sizes. CONV1 layer is computed using direct convolution since filter size is 7×7. Table 6.4

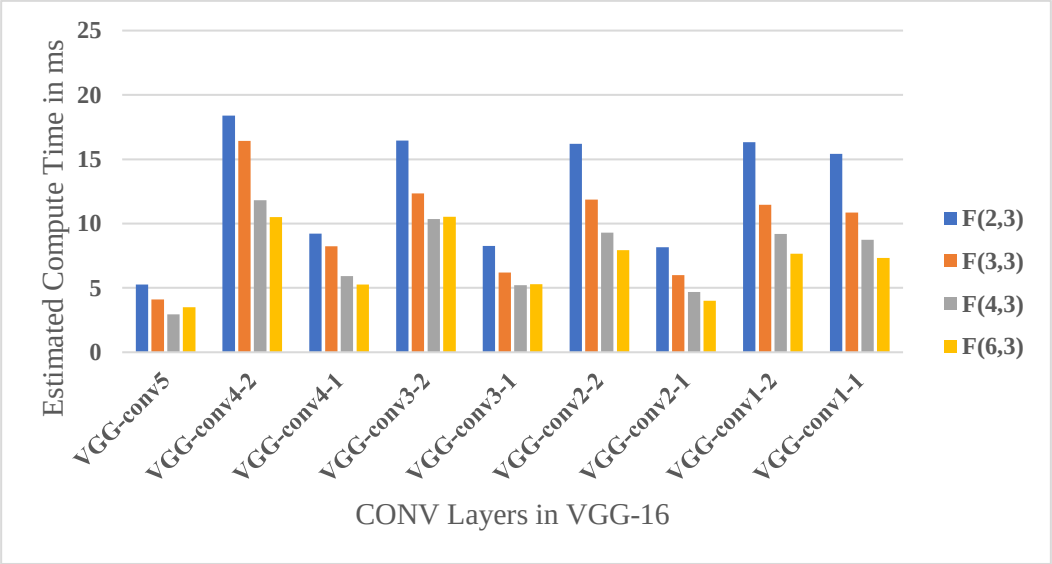

Figure 6.2: Estimated compute time for CONV Layers in VGG-16

Table 6.3: Estimated BRAM for Various Tiles in VGG-16

CONV Layer	F(2,3)	F(3,3)	F(4,3)	F(6,3)
CONV1-1	785	815	845	**925**
CONV1-2	990	1130	**1280**	1630
CONV2-1	920	1025	1140	**1405**
CONV2-2	1070	1255	**1460**	1925
CONV3-1	1000	1145	**1265**	1565
CONV3-2, 3-3	**1230**	1490	1720	2275
CONV4-1	1115	**1280**	1580	2065
CONV4-2, 4-3	**1465**	1775	2210	3275
CONV5-1, 5-2, 5-3	**1335**	1670	2540	3655

Table 6.4: ResNet-18 parameters for various tiles

Parameter		CONV Stage				
		1	2_x	3_x	4_x	5_x
Output		112×112	56×56	28×28	14×14	7×7
Filter		7×7	3×3	3×3	3×3	3×3
# Filters		64	64	128	256	512
# Channels		64	64	128	256	512
Tiles (T)	$F(2,3)$	-	3136	784	196	49
	$F(3,3)$	-	1444	361	100	25
	$F(4,3)$	-	784	196	49	16
	$F(6,3)$	-	361	100	25	9

Table 6.5: Convolution Schemes for various layers in ResNet-18

CONV Layer	Stride	Convolution
CONV1	2	Direct
CONV2_x	1	Winograd
CONV3_1	2	Direct
CONV3_2,3,4	1	Winograd
CONV4_1	2	Direct
CONV4_2,3,4	1	Winograd
CONV5_1	2	Direct
CONV5_2,3,4	1	Winograd

shows the number of tiles required in each CONV stage, for various tile sizes. Also, we have used Winograd algorithm for CONV layers with stride one in each stage and layers with stride two in each stage were implemented using direct convolution. Table 6.5 gives the convolution scheme used in each CONV layer of various stages and the strides.

For fixing Winograd tile size for each CONV stage, we have performed an estimation of computation time and BRAM requirement for each CONV stage for various tile sizes. Fig. 6.3 shows the computation time required for CONV layers in different stages, for various tile sizes and corresponding BRAM requirement is given in Table 6.6.

XC7VX690T FPGA consists of 1470 BRAMs. For CONV layers in each stage, we choose the tile with minimum computation time and BRAM within 1470. For example, for CONV5_x stage, $F(3,3)$, $F(4,3)$ and $F(6,3)$ gives minimum compute time, but uses more than 1470 BRAMs. So we have selected $F(2,3)$ for CONV layers in this

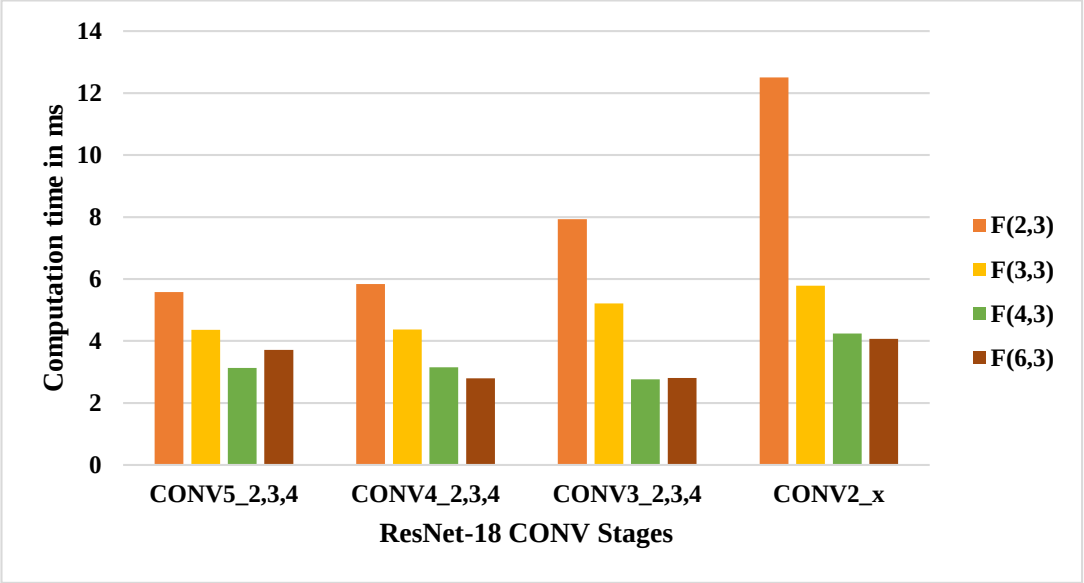

Figure 6.3: Compute time for ResNet-18 CONV layers

Table 6.6: BRAM for various tiles in ResNet-18

CONV Layer	F(2,3)	F(3,3)	F(4,3)	F(6,3)
CONV2_x	920	950	1110	**1225**
CONV3_2,3,4	1000	1025	**1265**	1565
CONV4_2,3,4	1115	**1280**	1580	2065
CONV5_2,3,4	**1335**	1670	2540	3655

Table 6.7: Hardware Resource Utilization

Resource	Used	Available	% Utilization
Flipflops	649K	866K	74.9
LUTs	468K	693K	67.6
BRAMS	1465	1470	99.6
DSP48E	1436	3600	39.8
DDR BW	2.55 GBps	6.4 GBps	39.9

stage. Similarly tile sizes were chosen for various stages. Selected BRAMs are highlighted in Table 6.6. CONV4_x uses $F(3,3)$, CONV3_x uses $F(4,3)$ and CONV2_x uses $F(6,3)$ tiles. Transform units are implemented to support all these tile sizes.

6.1.3 Hardware Resource Utilization

Based on the BRAM analysis, we decide the maximum BRAMs required for implementing all layers of AlexNet, VGG-16 and ResNet-18 models. *DTU* and *OTU* are implemented to support all selected tile sizes and configuration parameters for these tile sizes. Table 6.7 shows the hardware resource utilization for the implementation on XC7VX690T FPGA. Proposed accelerator uses 67.6% LUTs, 39.8% DSP48E blocks and 99.6% BRAMs. Maximum operating clock frequency is 200 MHz. Peak DDR bandwidth required is only 39.9% when compared to the theoretically available maximum bandwidth of 6.4 GBps.

6.1.4 Performance Model Accuracy

Here we evaluate the proposed analytical model for performance with the actual measured results. Table 6.8 gives the estimated performance and actual performance for different CONV layers in AlexNet. Chosen tile sizes are also shown. Actual measured performance is very close to the estimated performance which shows the efficacy of our model. Estimation error is around 7% on average. The slight decrease in actual performance can be attributed to additional delays introduced due to bank conflicts at the external memory.

Table 6.8: Performance Model Accuracy for CONV layers in AlexNet

CONV Layer	Selected Tile Size	Estimated GOPS	Actual GOPS	Estimation error (%)
CONV2	F(6,3)	387.2	358.1	7.5
CONV3	F(3,3)	392.3	368.1	6.2
CONV4	F(4,3)	507.8	476.4	6.2
CONV5	F(4,3)	507.8	474.9	6.5

6.2 Comparison with Existing Accelerators

In Table 6.9 and 6.10, we compare our results with state-of-art accelerators from literature for AlexNet and VGG-16 respectively. Giga operations per second (GOPS) is the most commonly used metric for comparing accelerator performance. However as discussed in Section 5.4.1, two different algorithms with varying computation complexity cannot be compared using GOPS. Since total number of operations in Winograd based convolution depend on the filter size and is significantly lower, GOPS tend to underestimate its performance when compared to direct convolution. Hence higher GOPS for direct convolution need not indicate higher convolution performance. In Lu *et al.* (2017) the authors use effective GOPS for each layer which is the total MAC operations for convolution divided by total compute time. However, since this is not a commonly used metric, we have decided to present the comparisons in terms of actual GOPS.

From Table 6.9 and 6.10, it can be seen that even in terms of GOPS, only Zeng *et al.* (2018), Aydonat *et al.* (2017) and Huimin Li *et al.* (2016) for AlexNet and Zeng *et al.* (2018) for VGG-16 show higher performance. Huimin Li *et al.* (2016) is an end to end accelerator targeted at AlexNet model using direct convolution techniques. Performance in terms of GOPS is $1.33\times$ that of *UniWiG*. In Huimin Li *et al.* (2016), authors have reported 2.56 ms processing time per image which gives a throughput of 390 images per second. For AlexNet model on *UniWiG*, we get a throughput of 942 images per second which is around three times that of Huimin Li *et al.* (2016). Throughput of 1020 images per second is reported in Aydonat *et al.* (2017), which is achieved with $1.5\times$ clock frequency and more FPGA resources. In Zeng *et al.* (2018), authors use FFT based convolutions together with Concatenate-and-Pad (CaP) and frequency domain loop tiling techniques which significantly improves the throughput. Performance in GOPS is close to twice that of *UniWiG*. Authors have not presented the processing

Table 6.9: Comparison of AlexNet CNN Implementations

	Our Work	INT'18 Ma *et al.* (2018)	FPGA'18 Zeng *et al.* (2018)	FPL'16 Huimin Li *et al.* (2016)	FPGA'15 Zhang *et al.* (2015)	FPGA'16 Suda *et al.* (2016)	FPGA'17 Aydonat *et al.* (2017)
FPGA	Virtex-7 VX690T	Stratix-V GXA7	Stratix-V GXA7	Virtex-7 VC709	VX485T	Stratix-V GSD8	Arria 10
Precision	16-fixed	16-fixed	16-fixed	16-fixed	32-float	8-16 fixed	16-float
F(MHz)	200	100	200	156	100	120	303
Logic Used	468K (67%)	121K (48%)	107K (46%)	274K (63%)	186K (61.3%)	114K (49%)	246K (58%)
DSP Used	1436 (39.8%)	256 (100%)	256 (100%)	2144 (60%)	2240 (80%)	256 (100%)	1476 (97%)
BRAM	1465 (99%)	1552 (61%)	1377 (73%)	956 (65%)	1024 (50%)	1893 (74%)	2487 (92%)
Performance	433.63 GOPS	114.5 GOPS	780.6 GOPS	565.94 GOPS	61.62 GFLOPS	117.8 GOPS	1382 GFLOPS

time in Zeng *et al.* (2018), absence of which prevents us from comparing performance in terms of throughput. Table 6.9 and 6.10 shows that logic cells and DSP blocks used are maximum for *UniWiG* which indicates that there is further scope for micro-architectural optimizations.

As mentioned earlier, Lu *et al.* (2017) reports performance in terms of effective GOPS and has not been included in the comparison tables. For AlexNet, average effective GOPS for all layers is 854.6. The corresponding figure for AlexNet on *UniWiG* is 794.38. However *UniWiG* uses half the number of DSPs and less logic blocks.

The estimated power consumption of AlexNet model is 17.3 Watts. This gives an energy efficiency (GOP/s/W) of 25.06, which is comparable to other state-of-art implementations Ma *et al.* (2018); Lu *et al.* (2017); Suda *et al.* (2016); Huimin Li *et al.* (2016).

We compare our accelerator with existing ResNet implementations and is given in Table 6.11. Note that in Baskin *et al.* (2017), authors have presented performance in terms of total run time for ResNet-18. For the purpose of comparison, we have

Table 6.10: Comparison of VGG CNN Implementations

	Our Work	FPGA'18 Zeng *et al.* (2018)	TCAD'18 Guo *et al.* (2018)	TVLSI'18 Ma *et al.* (2018)	FCCM'17 Guan *et al.* (2017)	FPL'17 Ma *et al.* (2017*a*)	ASAP'17 Podili *et al.* (2017)
Network	VGG-16	VGG-16	VGG-16	VGG-16	VGG-19	VGG-16	VGG-16
FPGA	Virtex-7 VX690T	Stratix V GXA7	Zynq XC7Z045	Stratix-V GXA7	Stratix-V GSMD5	Virtex-7 VC707	Stratix-V
Precision	16-bit fixed	16-bit fixed	16-bit fixed	16-bit fixed	16-bit fixed	16-bit fixed	32-bit fixed
F(MHz)	200	200	150	150	150	150	200
Logic	468K (67%)	107K (46%)	183K (84%)	218K (93%)	45.7K (27%)	212.1K (90%)	197K (83%)
DSP	1436 (39.8%)	256 (100%)	780 (87%)	256 (100%)	1044 (64%)	256 (100%)	256 (100%)
BRAM	1465 (99%)	1377 (73%)	486 (87%)	2210 (86%)	959 (48%)	2202 (86%)	451 (17.6%)
Perf. (GOPS)	407.23	669.1	137	348.8	364.36	352.84	229.2

Table 6.11: Comparison of ResNet models

	Our work	Baskin *et al.* (2017)	Ma *et al.* (2017*a*)	Ma *et al.* (2017*b*)
Network	*R-18	R-18	R-50	R-50
FPGA	Virtex 7 XC7VX690T	Stratix V 5SGSD8	Stratix V GXA7	Arria-10 GX1150
Precision	16-fixed	32-float	16-fixed	16-fixed
Freq (MHz)	200	105	150	150
DSP Blocks	1436 (39.8%)	-	256 (100%)	1046 (69%)
Logic used	468K (67.6%)	596K	173.5K (74%)	128K (30%)
BRAM used	1465 (99.6%)	1543 (60.5%)	1946 (76%)	2167 (80%)
Performance	383 GOPS	207.58 GFLOPS	250 GOPS	285.1 GOPS

*R denotes ResNet

estimated performance in GFLOPS based on the runtime. Table 6.11 shows that our implementation gives the maximum performance while consuming the minimum off-chip BRAM blocks which shows the effectiveness of our approach.

6.3 Summary of the Chapter

In this chapter, fixed point implementation of *UniWiG* architecture using 16-bits has been performed. Performance of AlexNet, VGG-16 and ResNet-18 models in *UniWiG* architecture were analyzed. BRAM requirement and computation time were estimated for these models and appropriate Winograd tile sizes were selected for implementation. Comparisons with state-of-art accelerators show that our implementation is effective in terms of performance.

CHAPTER 7

Conclusions and Future Work

In this chapter, summary and further directions for the research work in this thesis are presented. This thesis presents efficient hardware accelerators for various real-time image processing tasks like image reconstruction and image classification, which are key components in digital image processing. FFTs are widely used in image reconstruction and requires efficient implementation for real time applications. An efficient 2D FFT architecture for image reconstruction is presented in the first part of the thesis. Image classification can be performed efficiently with novel machine learning algorithms. CNNs are popular for performing image classification tasks. Efficient CNN accelerator for image classification has been presented in the second part of the thesis. A high throughput CNN accelerator on FPGA, using unified Winograd-GEMM architecture is proposed here. Comparisons with state-of-art accelerators show the effectiveness of proposed architecture.

7.1 Summary of the Thesis

Digital image processing is a field of DSP where digital images are processed by means of digital computers. Most of the steps in digital image processing like image analysis, image reconstruction, image enhancement and compression can be performed using Fast Fourier Transform (FFT) algorithm. Evolution of deep learning techniques enabled implementation of most of the image processing applications with high performance and accuracy. Most popular deep learning technique used for image classification and detection task is Convolutional Neural Network (CNN), which is a variant of deep neural network.

In the first part of the thesis, a novel 2D FFT architecture with efficient data reordering technique, using the radix-4^3 algorithm has been presented. The architecture uses two parallel unrolled radix-4^3 blocks in cascade to develop a 64×64 2D FFT architecture. We have used six-bit $modeselect$ as the control signal in radix-4^3 architecture for

performing data reordering. Radix-4^3 architecture gives a significant reduction in intermediate memory within a 1D FFT and reduces the latency. Proposed 2D architecture reduces the intermediate memory between two 1D FFTs from N^2 to N. SNR for various word lengths is analyzed and 16-bit is chosen as the required word length for image processing applications. The architecture has been implemented in RTL using Verilog HDL and simulated using Modelsim. RTL has been synthesized with a Cadence RTL Compiler using Faraday 40 nm standard cell library, tailored for UMC's 40 nm process. ASIC synthesis results give a clock frequency of 500 MHz and core area of 0.841 mm^2. 64×64 FFT takes 4096 cycles for computation and the execution time is 8.19 μs. A comparison with existing implementations shows 47.5% reduction in computation time for 64×64 FFT. The proposed architecture has also been implemented in Virtex-7 FPGA with an operating frequency of 156.25 MHz. FPGA implementation results show the comparable area in terms of slice LUTs.

Future directions on FFT architecture

- Extending this architecture to realize a large size 2D FFT to evaluate and validate the optimization in memory and latency. Cascading two radix-4^4 units can result in a 256×256 point FFT.
- Radix-4 engine can be used as a building block for reconfigurable 2D FFTs of various sizes. This requires further studies on existing reconfigurable and programmable 2D FFT architectures.
- Future work also focuses on a three dimensional FFT architecture for video processing applications.

In the second work, a novel unified architecture called *UniWiG* for implementing both general matrix multiplication (GEMM) algorithm and Winograd minimal filtering algorithm on the same processing elements has been presented. Both convolution and fully connected layers of CNN can be accelerated using this architecture. Such a unified architecture gives the most efficient implementation for CONV layers irrespective of the filter sizes and also for FC layers. Winograd filtering is transformed to a GEMM operation so that PEs for GEMM can be reused for Winograd algorithm. A novel algorithm is proposed, which transforms Winograd minimal filtering algorithm into blocked general element-wise matrix multiplication which targets optimal utilization of BRAMs and DDR bandwidth. Proposed architecture consists of multiple systolic arrays optimized

for GEMM and has been implemented on FPGA. An analytical model for estimating performance and BRAM usage has also been proposed, using which appropriate tile sizes for each layer can be identified, which can be used to optimize the tile size for a given convolution layer. Comparisons with baseline implementation show that *UniWiG* gives 1.4× to 4.02× performance improvements with less than 15% additional FPGA resources. Popular CNN models namely AlexNet, ResNet-18 and VGG-16 have been accelerated using *UniWiG* and results are comparable with other state-of-art accelerators. A variety of challenges were encountered during the design and implementation of the CNN accelerator. How to parallelize the multiple processing elements in the general matrix multiplier and how many PEs in an array is required for the accelerator was a critical design element.

Future work on CNN accelerator

Proposed architecture is used to accelerate AlexNet, ResNet-18 and VGG-16 models. Further work is needed to accelerate other CNN models like MobileNet, SqueezeNet etc. using this architecture. Proposed architecture supports Winograd algorithm based convolutions for stride of one, for all layers in a model. For strides other than one, general matrix multiplication algorithm is used. Architectural changes are required for performing computations using Winograd method, if different strides are used. Deep compression techniques can significantly reduce the memory and bandwidth requirement for the execution of deep neural networks. For sparse and redundant network connections, we can introduce pruning techniques to the network. Also, future work targets on minimum computation for maximum accuracy. That is, to find out the optimum bit-width for an accurate result.

Future works also include researches on security aspects of proposed architecture. Side-channel leakages pose a major threat to the security of hardware accelerators. In the future work, we analyze the vulnerability of Unified Winograd-GEMM architecture to reverse engineering attacks.

APPENDIX A

WINOGRAD MINIMAL FILTERING COEFFICIENTS

Winograd minimal filtering is a fast algorithm for computing convolution based on Chinese Remainder Theorem and involves polynomial multiplication. One dimensional Winograd algorithm for computing m outputs using an r tap filter will require $m+r-1$ multiplications Lavin and Gray (2016). Two dimensional (2D) Winograd algorithm can be implemented from nested one dimensional (1D) Winograd algorithm. Output Y can be written as,

$$Y = A^T[(GgG^T) \odot (B^T dB)]A \tag{A.1}$$

The transform matrices A, B and G can be precomputed, once the value of m and r are known. Transform matrices for various Winograd tile sizes are computed offline and are given in the following sections.

A.1 $F(2 \times 2, 3 \times 3)$

$F(2 \times 2, 3 \times 3)$ is obtained by nesting $F(2, 3)$ 1D Winograd tiles. The coefficients for $F(2, 3)$ are given as:

$$A^T = \begin{pmatrix} 1 & 1 & 1 & 0 \\ 0 & 1 & -1 & -1 \end{pmatrix}$$

$$B^T = \begin{pmatrix} 1 & 0 & -1 & 0 \\ 0 & 1 & 1 & 0 \\ 0 & -1 & 1 & 0 \\ 0 & 1 & 0 & -1 \end{pmatrix}$$

$$G = \begin{pmatrix} 1 & 0 & 0 \\ \frac{1}{2} & \frac{1}{2} & \frac{1}{2} \\ \frac{1}{2} & -\frac{1}{2} & \frac{1}{2} \\ 0 & 0 & 1 \end{pmatrix}$$

A.2 $F(3 \times 3, 2 \times 2)$

$F(3 \times 3, 2 \times 2)$ is obtained by nesting $F(3, 2)$ 1D Winograd tiles. The coefficients for $F(3, 2)$ are given as:

$$A^T = \begin{pmatrix} 1 & 1 & 1 & 0 \\ 0 & 1 & -1 & 0 \\ 0 & 1 & 1 & 1 \end{pmatrix}$$

$$B^T = \begin{pmatrix} 1 & 0 & -1 & 0 \\ 0 & 1 & 1 & 0 \\ 0 & -1 & 1 & 0 \\ 0 & -1 & 0 & 1 \end{pmatrix}$$

$$G = \begin{pmatrix} 1 & 0 \\ \frac{1}{2} & \frac{1}{2} \\ \frac{1}{2} & -\frac{1}{2} \\ 0 & 1 \end{pmatrix}$$

A.3 $F(3 \times 3, 3 \times 3)$

$F(3 \times 3, 3 \times 3)$ is obtained by nesting $F(3, 3)$ 1D Winograd tiles. The coefficients for $F(3, 3)$ are given as:

$$A^T = \begin{pmatrix} 1 & 1 & 1 & 10 & \\ 0 & 1 & -1 & 2 & 0 \\ 0 & 1 & 1 & 4 & 1 \\ & & & & \end{pmatrix}$$

$$B^T = \begin{pmatrix} 2 & -1 & -2 & 1 & 0 \\ 0 & -2 & -1 & 1 & 0 \\ 0 & 2 & -3 & 1 & 0 \\ 0 & -1 & 0 & 1 & 0 \\ 0 & 2 & -1 & -2 & 1 \\ & & & & \end{pmatrix}$$

$$G = \begin{pmatrix} \frac{1}{2} & 0 & 0 \\ -\frac{1}{2} & -\frac{1}{2} & -\frac{1}{2} \\ -\frac{1}{6} & \frac{1}{6} & -\frac{1}{6} \\ \frac{1}{6} & \frac{1}{3} & \frac{2}{3} \\ 0 & 0 & 1 \\ & & \end{pmatrix}$$

A.4 $F(4 \times 4, 3 \times 3)$

$F(4 \times 4, 3 \times 3)$ is obtained by nesting $F(4, 3)$ 1D Winograd tiles. The coefficients for $F(4, 3)$ are given as:

$$A^T = \begin{pmatrix} 1 & 1 & 1 & 1 & 1 & 0 \\ 0 & 1 & -1 & 2 & -2 & 0 \\ 0 & 1 & 1 & 4 & 4 & 0 \\ 0 & 1 & -1 & 8 & -8 & 1 \\ & & & & & \end{pmatrix}$$

$$B^T = \begin{pmatrix} 4 & 0 & -5 & 0 & 1 & 0 \\ 0 & -4 & -4 & 1 & 1 & 0 \\ 0 & 4 & -4 & -1 & 1 & 0 \\ 0 & -2 & -1 & 2 & 1 & 0 \\ 0 & 2 & -1 & -2 & 1 & 0 \\ 0 & 4 & 0 & -5 & 0 & 1 \end{pmatrix}$$

$$G = \begin{pmatrix} \frac{1}{4} & 0 & 0 \\ -\frac{1}{6} & -\frac{1}{6} & -\frac{1}{6} \\ -\frac{1}{6} & \frac{1}{6} & -\frac{1}{6} \\ \frac{1}{24} & \frac{1}{12} & \frac{1}{6} \\ \frac{1}{24} & -\frac{1}{12} & \frac{1}{6} \\ 0 & 0 & 1 \end{pmatrix}$$

A.5 $F(6 \times 6, 3 \times 3)$

For $F(6 \times 3)$, the transform matrices consist of large coefficients and are given as:

$$A^T = \begin{pmatrix} 1 & 1 & 1 & 1 & 1 & 1 & 1 & 0 \\ 0 & 1 & -1 & 2 & -2 & 3 & -3 & 0 \\ 0 & 1 & 1 & 1 & 4 & 4 & 9 & 9 & 0 \\ 0 & 1 & -1 & 8 & -8 & 27 & -27 & 0 \\ 0 & 1 & 1 & 16 & 16 & 81 & 81 & 0 \\ 0 & 1 & -1 & 32 & -32 & 243 & -243 & 1 \end{pmatrix}$$

$$B^T = \begin{pmatrix} -36 & 0 & 49 & 0 & -14 & 0 & 1 & 0 \\ 0 & 36 & 36 & -13 & -13 & 1 & 1 & 0 \\ 0 & -36 & 36 & 13 & -13 & -1 & 1 & 0 \\ 0 & 18 & 9 & -20 & -10 & 2 & 1 & 0 \\ 0 & -18 & 9 & 20 & -10 & -2 & 1 & 0 \\ 0 & 12 & 4 & -15 & -5 & 3 & 1 & 0 \\ 0 & -12 & 4 & 15 & -5 & -3 & 1 & 0 \\ 0 & -36 & 0 & 49 & 0 & -14 & 0 & 1 \end{pmatrix}$$

$$G = \begin{pmatrix} \frac{-1}{36} & 0 & 0 \\ \frac{1}{48} & \frac{1}{48} & \frac{1}{48} \\ \frac{1}{48} & -\frac{1}{48} & \frac{1}{48} \\ -\frac{1}{120} & -\frac{1}{60} & -\frac{1}{30} \\ -\frac{1}{120} & \frac{1}{60} & -\frac{1}{30} \\ \frac{1}{720} & \frac{1}{240} & \frac{1}{80} \\ \frac{1}{720} & -\frac{1}{240} & \frac{1}{80} \\ 0 & 0 & 1 \end{pmatrix}$$

REFERENCES

1. **Abtahi, T.**, **A. Kulkarni**, and **T. Mohsenin**, Accelerating convolutional neural network with fft on tiny cores. *In 2017 IEEE International Symposium on Circuits and Systems (ISCAS)*. 2017. ISSN 2379-447X.

2. **Aguilar-González, A.**, **M. Arias-Estrada**, **M. Pérez-Patricio**, and **J. L. Camas-Anzueto** (2019). An fpga 2d-convolution unit based on the caph language. *Journal of Real-Time Image Processing*, **16**(2), 305–319. ISSN 1861-8219. URL `https://doi.org/10.1007/s11554-015-0535-1`.

3. **Ahmad, A.** and **M. A. Pasha**, Towards design space exploration and optimization of fast algorithms for convolutional neural networks (cnns) on fpgas. *In 2019 Design, Automation Test in Europe Conference Exhibition (DATE)*. 2019. ISSN 1530-1591.

4. **Akin, B.**, **P. A. Milder**, **F. Franchetti**, and **J. C. Hoe**, Memory bandwidth efficient two-dimensional fast fourier transform algorithm and implementation for large problem sizes. *In 2012 IEEE 20th International Symposium on Field-Programmable Custom Computing Machines*. 2012. ISSN null.

5. **Aydonat, U.**, **S. O'Connell**, **D. Capalija**, **A. C. Ling**, and **G. R. Chiu**, An opencl™deep learning accelerator on arria 10. *In Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, FPGA '17. ACM, New York, NY, USA, 2017. ISBN 978-1-4503-4354-1. URL `http://doi.acm.org/10.1145/3020078.3021738`.

6. **Babionitakis, K.**, **V. Chouliaras**, **K. Manolopoulos**, **K. Nakos**, **D. Reisis**, and **N. Vlassopoulos** (2010*a*). Fully systolic fft architecture for giga-sample applications. *Journal Signal Process Syst (2010)*.

7. **Babionitakis, K.**, **V. A. Chouliaras**, **K. Manolopoulos**, **K. Nakos**, **D. Reisis**, and **N. Vlassopoulos** (2010*b*). Fully systolic fft architecture for giga-sample applications. *Journal Signal Process Syst (2010)*, **58**, 281–299.

8. **Bai, L.**, **Y. Zhao**, and **X. Huang** (2018). A cnn accelerator on fpga using depthwise separable convolution. *IEEE Transactions on Circuits and Systems II: Express Briefs*, **65**(10), 1415–1419. ISSN 1558-3791.

9. **Bangari, V.**, **B. A. Marquez**, **H. Miller**, **A. N. Tait**, **M. A. Nahmias**, **T. F. de Lima**, **H. Peng**, **P. R. Prucnal**, and **B. J. Shastri** (2020). Digital electronics and analog photonics for convolutional neural networks (deap-cnns). *IEEE Journal of Selected Topics in Quantum Electronics*, **26**(1), 1–13. ISSN 1558-4542.

10. **Baskin, C.**, **N. Liss**, **E. Zheltonozhskii**, **A. M. Bronshtein**, and **A. Mendelson** (2017). Streaming architecture for large-scale quantized neural networks on an fpga-based dataflow platform.

11. **Bergstra, J.**, **O. Breuleux**, **F. Bastien**, **P. Lamblin**, **R. Pascanu**, **G. Desjardins**, **J. Turian**, **DavidWarde-Farley**, and **Y. Bengio** (2010). Theano: a cpu and gpu math expression compiler. *SciPy*, **4**, 3.

12. **Blahut, R. E.**, *Fast Algorithms for Signal Processing*. Cambridge University Press, New York, 2010.

13. **Chakraborty, T. S.** and **S. Chakrabarti**, On output reorder buffer design of bit reversed pipelined continuous data fft architecture. *In APCCAS 2008 - 2008 IEEE Asia Pacific Conference on Circuits and Systems*. 2008. ISSN null.

14. **Chakradhar, S.**, **M. Sankaradas**, **V. Jakkula**, and **S. Cadambi**, A dynamically configurable coprocessor for convolutional neural networks. *In Proceedings of the 37th Annual International Symposium on Computer Architecture*, ISCA '10. ACM, New York, NY, USA, 2010. ISBN 978-1-4503-0053-7. URL `http://doi.acm.org/10.1145/1815961.1815993`.

15. **Chang, J.** and **J. Sha** (2017). An efficient implementation of 2d convolution in cnn. *IEICE Electronics Express*, **14**(1), 20161134–20161134.

16. **Chen, R.** and **V. K. Prasanna**, Energy optimizations for fpga-based 2-d fft architecture. *In 2014 IEEE High Performance Extreme Computing Conference (HPEC)*. 2014. ISSN null.

17. **Chen, Y.**, **J. Emer**, and **V. Sze**, Eyeriss: A spatial architecture for energy-efficient dataflow for convolutional neural networks. *In 2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*. 2016. ISSN 1063-6897.

18. **Chen, Y.**, **Y. Lin**, **Y. Tsao**, and **C. Lee** (2008). A 2.4-gsample/s dvfs fft processor for mimo ofdm communication systems. *IEEE Journal of Solid-State Circuits*, **43**(5), 1260–1273. ISSN 1558-173X.

19. **Chen, Y.**, **T. Luo**, **S. Liu**, **S. Zhang**, **L. He**, **J. Wang**, **L. Li**, **T. Chen**, **Z. Xu**, **N. Sun**, and **O. Temam**, Dadiannao: A machine-learning supercomputer. *In 2014 47th Annual IEEE/ACM International Symposium on Microarchitecture*. 2014. ISSN 2379-3155.

20. **Cheng, C.** and **K. K. Parhi** (2007). High-throughput vlsi architecture for fft computation. *IEEE Transactions on Circuits and Systems II: Express Briefs*, **54**(10), 863–867. ISSN 1558-3791.

21. **Chetlur, S.**, **C. Woolley**, **P. Vandermersch**, **J. Cohen**, **J. Tran**, **B. Catanzaro**, and **E. Shelhamer** (2014). cudnn: Efficient primitives for deep learning.

22. **Chidambaram, R.** (2005). *A Scalable and High-performance FFT processor, Optimized for UWB-OFDM*. Ph.D. thesis, M.S. thesis, Delft University of Technology.

23. **Chih-Peng Fan**, **Mau-Shih Lee**, and **Guo-An Su**, Efficient low multiplier cost 256-point fft design with radix-2^4 sdf architecture. *In IEEE Asia Pacific Conference on Circuits and Systems, APCCAS 2006*. 2006.

24. **Chin-Teng Lin**, **Yuan-Chu Yu**, and **Lan-Da Van**, A low-power 64-point fft/ifft design for ieee 802.11a wlan application. *In 2006 IEEE International Symposium on Circuits and Systems*. 2006*a*. ISSN 2158-1525.

25. **Chin-Teng Lin**, **Yuan-Chu Yu**, and **Lan-Da Van**, A low-power 64-point fft/ifft design for ieee 802.11a wlan application. *In 2006 IEEE International Symposium on Circuits and Systems*. 2006*b*. ISSN 2158-1525.

26. **Cho, T.** and **H. Lee** (2013). A high-speed low-complexity modified radix-2^5 fft processor for high rate wpan applications. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, **21**(1), 187–191. ISSN 1557-9999.

27. **Ciobanu, C. B.** and **G. N. Gaydadjiev**, Separable 2d convolution with polymorphic register files. *In* **H. Kubátová**, **C. Hochberger**, **M. Daněk**, and **B. Sick** (eds.), *Architecture of Computing Systems – ARCS 2013*. Springer Berlin Heidelberg, Berlin, Heidelberg, 2013. ISBN 978-3-642-36424-2.

28. **Cohen, D.** (1976). Simplified control of fft hardware. *IEEE Transactions on Acoustics, Speech, and Signal Processing*, **24**(6), 577–579. ISSN 0096-3518.

29. **Collobert, R.**, **K. Kavukcuoglu**, and **C. Farabet**, Torch: A matlab-like environment for machine learning. *In BigLearn, NIPS Workshop*. 2011.

30. **Cortes, A.**, **J. F. Sevillano**, **I. Velez**, and **A. Irizar**, An fft core for dvb-t/dvb-h receivers. *In 2006 13th IEEE International Conference on Electronics, Circuits and Systems*. 2006. ISSN null.

31. **D'Alberto, P.**, **P. A. Milder**, **A. Sandryhaila**, **F. Franchetti**, **J. C. Hoe**, **J. M. F. Moura**, **M. Puschel**, and **J. R. Johnson**, Generating fpga-accelerated dft libraries. *In 15th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM 2007)*. 2007. ISSN null.

32. **Deng, L.**, **C. . Yu**, **C. Chakrabarti**, **J. Kim**, and **V. Narayanan**, Efficient image reconstruction using partial 2d fourier transform. *In 2008 IEEE Workshop on Signal Processing Systems*. 2008. ISSN 2162-3570.

33. **DiCecco, R.**, **G. Lacey**, **J. Vasiljevic**, **P. Chow**, **G. Taylor**, and **S. Areibi**, Caffeinated fpgas: Fpga framework for convolutional neural networks. *In 2016 International Conference on Field-Programmable Technology (FPT)*. 2016. ISSN null.

34. **Dou, Y.**, **S. Vassiliadis**, **G. K. Kuzmanov**, and **G. N. Gaydadjiev**, 64bitt floating-point fpga matrix multiplication. *In Proceedings of the 2005 ACM/SIGDA 13th International Symposium on Field-programmable Gate Arrays*, FPGA '05. ACM, New York, NY, USA, 2005. ISBN 1-59593-029-9. URL `http://doi.acm.org/10.1145/1046192.1046204`.

35. **Esmaeilzadeh, H.**, **E. Blem**, **R. S. Amant**, **K. Sankaralingam**, and **D. Burger**, Dark silicon and the end of multicore scaling. *In 2011 38th Annual International Symposium on Computer Architecture (ISCA)*. 2011. ISSN 1063-6897.

36. **F. Kristensen, A. O., P. Nilsson**, Flexible baseband transmitter for ofdm. *In IASTED Conf. Circuits Signals Syst.*. 2003.

37. **Frigo, M.** and **S. G. Johnson**, Fftw: an adaptive software architecture for the fft. *In Proceedings of the 1998 IEEE International Conference on Acoustics, Speech and Signal Processing, ICASSP '98 (Cat. No.98CH36181)*, volume 3. 1998. ISSN 1520-6149.

38. **Garrido, M.**, **J. Grajal**, and **O. Gustafsson** (2011). Optimum circuits for bit reversal. *IEEE Transactions on Circuits and Systems II: Express Briefs*, **58**(10), 657–661. ISSN 1558-3791.

39. **Garrido, M.**, **S. Huang**, **S. Chen**, and **O. Gustafsson** (2016). The serial commutator fft. *IEEE Transactions on Circuits and Systems II: Express Briefs*, **63**(10), 974–978. ISSN 1558-3791.

40. **Gold, B.** and **L. R.Rabiner**, *Theory And Application of Digital Signal Processing*. Prentice Hall, 1975.

41. **Gonzalez, R. C.** and **R. E. Woods**, *Digital Image Processing*. 3rd Edition, Prentice Hall, 2008.

42. **Guan, Y.**, **H. Liang**, **N. Xu**, **W. Wang**, **S. Shi**, **X. Chen**, **G. Sun**, **W. Zhang**, and **J. Cong**, Fp-dnn: An automated framework for mapping deep neural networks onto fpgas with rtl-hls hybrid templates. *In 2017 IEEE 25th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. 2017. ISSN null.

43. **Guichang Zhong**, **Fan Xu**, and **A. N. Willson** (2006). A power-scalable reconfigurable fft/ifft ic based on a multi-processor ring. *IEEE Journal of Solid-State Circuits*, **41**(2), 483–495.

44. **Guo, K.**, **L. Sui**, **J. Qiu**, **J. Yu**, **J. Wang**, **S. Yao**, **S. Han**, **Y. Wang**, and **H. Yang** (2018). Angel-eye: A complete design flow for mapping cnn onto embedded fpga. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, **37**(1), 35–47. ISSN 1937-4151.

45. **Guo, Y.**, **A. Yao**, and **Y. Chen** (2016). Dynamic network surgery for efficient dnns.

46. **Gupta, S.**, **A. Agrawal**, **K. Gopalakrishnan**, and **P. Narayanan**, Deep learning with limited numerical precision. *In Proceedings of the 32Nd International Conference on International Conference on Machine Learning - Volume 37*, ICML'15. JMLR.org, 2015. URL `http://dl.acm.org/citation.cfm?id=3045118.3045303`.

47. **Hailesellasie, M.**, **S. R. Hasan**, **F. Khalid**, **F. A. Wad**, and **M. Shafique**, Fpga-based convolutional neural network architecture with reduced parameter requirements. *In 2018 IEEE International Symposium on Circuits and Systems (ISCAS)*. 2018. ISSN 2379-447X.

48. **Han, S.**, **H. Mao**, and **W. J. Dally** (2015). Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding.

49. **Han, X.**, **D. Zhou**, **S. Wang**, and **S. Kimura**, Cnn-merp: An fpga-based memory-efficient reconfigurable processor for forward and backward propagation of convolutional neural networks. *In 2016 IEEE 34th International Conference on Computer Design (ICCD)*. 2016. ISSN null.

50. **Hasan, M.**, **T. Arslan**, and **J. S. Thompson** (2003). A novel coefficient ordering based low power pipelined radix-4 fft processor for wireless lan applications. *IEEE Transactions on Consumer Electronics*, **49**(1), 128–134. ISSN 1558-4127.

51. **He, K.**, **X. Zhang**, **S. Ren**, and **J. Sun**, Deep residual learning for image recognition. *In 2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 2016. ISSN 1063-6919.

52. **Hsiang-Sheng Hu**, **Hsiao-Yun Chen**, and **Shyh-Jye Jou**, Novel fft processor with parallel-in-parallel-out in normal order. *In 2009 International Symposium on VLSI Design, Automation and Test*. 2009. ISSN null.

53. **Hsiao, C.**, **Y. Chen**, and **C. Lee** (2010). A generalized mixed-radix algorithm for memory-based fft processors. *IEEE Transactions on Circuits and Systems II: Express Briefs*, **57**(1), 26–30. ISSN 1558-3791.

54. **Huang, S.** and **S. Chen** (2012). A high-throughput radix-16 fft processor with parallel and normal input/output ordering for ieee 802.15.3c systems. *IEEE Transactions on Circuits and Systems I: Regular Papers*, **59**(8), 1752–1765. ISSN 1558-0806.

55. **Huang, S.** and **S. Chen**, A high-parallelism memory-based fft processor with high sqnr and novel addressing scheme. *In 2016 IEEE International Symposium on Circuits and Systems (ISCAS)*. 2016. ISSN 2379-447X.

56. **Huang, Y.**, **J. Shen**, **Y. Qiao**, **M. Wen**, and **C. Zhang** (2018). Malmm: A multi-array architecture for large-scale matrix multiplication on fpga. *IEICE Electronics Express*, **15**(10), 20180286–20180286.

57. **Huggett, C.**, **K. Maharatna**, and **K. Paul**, On the implementation of 128-pt fft/ifft for high-performance wpan. *In 2005 IEEE International Symposium on Circuits and Systems*. 2005. ISSN 2158-1525.

58. **Huimin Li**, **Xitian Fan**, **Li Jiao**, **Wei Cao**, **Xuegong Zhou**, and **Lingli Wang**, A high performance fpga-based accelerator for large-scale convolutional neural networks. *In 2016 26th International Conference on Field Programmable Logic and Applications (FPL)*. 2016. ISSN 1946-1488.

59. **Iandola, F. N.**, **D. Sheffield**, **M. J. Anderson**, **P. M. Phothilimthana**, and **K. Keutzer**, Communication-minimizing 2d convolution in gpu registers. *In 2013 IEEE International Conference on Image Processing*. 2013. ISSN 2381-8549.

60. **J.-C. Kuo**, **C.-H. Wen**, and **An-Yeu (Andy) Wu**, Implementation of a programmable 64-2048-point fft/ifft processor for ofdm based communication systems. *In 2003 IEEE International Symposium on Circuits and Systems*. 2003.

61. **Jeesung Lee**, **Hanho Lee**, **Sang-in Cho**, and **Sang-Sung Choi**, A high-speed, low-complexity radix-2^4 fft processor for mb-ofdm uwb systems. *In 2006 IEEE International Symposium on Circuits and Systems*. 2006. ISSN 2158-1525.

62. **Jia, Y.**, **E. Shelhamer**, **J. Donahue**, **S. Karayev**, **J. Long**, **R. Girshick**, **S. Guadarrama**, and **T. Darrell**, Caffe: Convolutional architecture for fast feature embedding. *In Proceedings of the 22Nd ACM International Conference on Multimedia*, MM '14. ACM, New York, NY, USA, 2014. ISBN 978-1-4503-3063-3. URL `http://doi.acm.org/10.1145/2647868.2654889`.

63. **J.W.Cooley** and **J.W.Tukey.** (1965). An algorithm for machine computation of complex fourier series. *Math Comput*, **9**, 297–301.

64. **Kala, S.**, **S. Nalesh**, **A. Maity**, **S. K. Nandy**, and **R. Narayan**, High throughput, low latency, memory optimized 64k point fft architecture using novel radix-4 butterfly unit. *In 2013 IEEE International Symposium on Circuits and Systems (ISCAS)*. 2013*a*. ISSN 2158-1525.

65. **Kala, S.**, **S. Nalesh**, **S. K. Nandy**, and **R. Narayan**, Design of a low power 64 point fft architecture for wlan applications. *In 2013 25th International Conference on Microelectronics (ICM)*. 2013*b*. ISSN 2159-1679.

66. **Kee, H.**, **S. S. Bhattacharyya**, **N. Petersen**, and **J. Kornerup**, Resource-efficient acceleration of 2-dimensional fast fourier transform computations on fpgas. *In 2009 Third ACM/IEEE International Conference on Distributed Smart Cameras (ICDSC)*. 2009. ISSN null.

67. **Kim, H.**, **H. Nam**, **W. Jung**, and **J. Lee**, Performance analysis of cnn frameworks for gpus. *In 2017 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. 2017. ISSN null.

68. **Kim, J. S.**, **C. Yu**, **L. Deng**, **S. Kestur**, **V. Narayanan**, and **C. Chakrabarti**, Fpga architecture for 2d discrete fourier transform based on 2d decomposition for large-sized data. *In 2009 IEEE Workshop on Signal Processing Systems*. 2009. ISSN 2162-3570.

69. **Kim, Y. J.** and **H. S. Lee**, The implementation of 2d fft using multiple topology on 4ÃŮ4 torus. *In 2009 9th International Symposium on Communications and Information Technology*. 2009. ISSN null.

70. **Kolala Venkataramanaiah, S.**, **Y. Ma**, **S. Yin**, **E. Nurvithadhi**, **A. Dasu**, **Y. Cao**, and **J. Seo**, Automatic compiler based fpga accelerator for cnn training. *In 2019 29th International Conference on Field Programmable Logic and Applications (FPL)*. 2019. ISSN 1946-147X.

71. **Krizhevsky, A.**, **I. Sutskever**, and **G. E. Hinton**, Imagenet classification with deep convolutional neural networks. *In* **F. Pereira**, **C. J. C. Burges**, **L. Bottou**, and **K. Q. Weinberger** (eds.), *Advances in Neural Information Processing Systems 25*. Curran Associates, Inc., 2012, 1097–1105. URL `http://papers.nips.cc/paper/4824-imagenet-classification-with-deep-convolutional-neural-networks.pdf`.

72. **Kyriakos, A.**, **V. Kitsakis**, **A. Louropoulos**, **E. Papatheofanous**, **I. Patronas**, and **D. Reisis**, High performance accelerator for cnn applications. *In 2019 29th International Symposium on Power and Timing Modeling, Optimization and Simulation (PATMOS)*. 2019. ISSN 2474-5456.

73. **Lavin, A.** and **S. Gray**, Fast algorithms for convolutional neural networks. *In 2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 2016. ISSN 1063-6919.

74. **Lee, S.**, **D. Kim**, **D. Nguyen**, and **J. Lee** (2019). Double mac on a dsp: Boosting the performance of convolutional neural networks on fpgas. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, **38**(5), 888–897. ISSN 1937-4151.

75. **Lenart, T.** (2008). *Design of Reconfigurable Hardware Architectures for Real-time Applications, Modeling and Implementation*. Ph.D. thesis, PhD Thesis, Lund University, Sweden.

76. **Lenart, T.**, **M. Gustafsson**, and **V. wall** (2008). A hardware acceleration platform for digital holographic imaging. *Journal Signal Process Syst (2008)*.

77. **Li, N.** (2008). *ASIC FFT Processor for MB-OFDM UWB System*. Ph.D. thesis, MSc. thesis, Delft University of Technology.

78. **Li, S.**, **H. Xu**, **W. Fan**, **Y. Chen**, and **X. Zeng**, A 128/256-point pipeline fft/ifft processor for mimo ofdm system ieee 802.16e. *In Proceedings of 2010 IEEE International Symposium on Circuits and Systems*. 2010. ISSN 2158-1525.

79. **Li, X.**, **G. Zhang**, **H. H. Huang**, **Z. Wang**, and **W. Zheng**, Performance analysis of gpu-based convolutional neural networks. *In 2016 45th International Conference on Parallel Processing (ICPP)*. 2016. ISSN 2332-5690.

80. **Lian, X.**, **Z. Liu**, **Z. Song**, **J. Dai**, **W. Zhou**, and **X. Ji** (2019). High-performance fpga-based cnn accelerator with block-floating-point arithmetic. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, **27**(8), 1874–1885. ISSN 1557-9999.

81. **Liao, Q.**, **W. Liu**, **F. Qiao**, **C. Wang**, and **F. Lombardi**, Design of approximate fft with bit-width selection algorithms. *In 2018 IEEE International Symposium on Circuits and Systems (ISCAS)*. 2018. ISSN 2379-447X.

82. **Lin, C.-T.** and **Y.-C. Yu**, Cost-effective pipeline fft/ifft vlsi architecture for dvb-h system. *In Proceedings of the National Symposium on Telecommunications 2007*. 2007.

83. **Lin, H. .**, **H. Lin**, **R. C. Chang**, **S. . Chen**, **C. . Liao**, and **C. . Wu**, A high-speed highly pipelined 2n-point fft architecture for a dual ofdm processor. *In Proceedings of the International Conference Mixed Design of Integrated Circuits and System, 2006. MIXDES 2006.*. 2006. ISSN null.

84. **Lu, L.**, **Y. Liang**, **Q. Xiao**, and **S. Yan**, Evaluating fast algorithms for convolutional neural networks on fpgas. *In 2017 IEEE 25th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. 2017. ISSN null.

85. **Lu, L.**, **J. Xie**, **R. Huang**, **J. Zhang**, **W. Lin**, and **Y. Liang**, An efficient hardware accelerator for sparse convolutional neural networks on fpgas. *In 2019 IEEE 27th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. 2019. ISSN 2576-2613.

86. **Ma, Y.**, **Y. Cao**, **S. Vrudhula**, and **J. Seo**, An automatic rtl compiler for high-throughput fpga implementation of diverse deep convolutional neural networks. *In 2017 27th International Conference on Field Programmable Logic and Applications (FPL)*. 2017*a*. ISSN 1946-1488.

87. **Ma, Y.**, **Y. Cao**, **S. Vrudhula**, and **J. Seo** (2018). Optimizing the convolution operation to accelerate deep neural networks on fpga. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, **26**(7), 1354–1367. ISSN 1557-9999.

88. **Ma, Y.**, **M. Kim**, **Y. Cao**, **S. Vrudhula**, and **J. Seo**, End-to-end scalable fpga accelerator for deep residual networks. *In 2017 IEEE International Symposium on Circuits and Systems (ISCAS)*. 2017*b*. ISSN 2379-447X.

89. **Ma, Y.**, **N. Suda**, **Y. Cao**, **S. Vrudhula**, and **J. sun Seo** (2018). Alamo: Fpga acceleration of deep learning algorithms with a modularized rtl compiler. *Integration*, **62**, 14 – 23. ISSN 0167-9260. URL `http://www.sciencedirect.com/science/article/pii/S0167926017304777`.

90. **Ma, Z.**, **X. Yin**, and **F. Yu** (2015). A novel memory-based fft architecture for real-valued signals based on a radix-2 decimation-in-frequency algorithm. *IEEE Transactions on Circuits and Systems II: Express Briefs*, **62**(9), 876–880. ISSN 1558-3791.

91. **Maharatna, K.**, **E. Grass**, and **U. Jagdhold** (2004). A 64-point fourier transform chip for high-speed wireless lan application using ofdm. *IEEE Journal of Solid-State Circuits*, **39**(3), 484–493. ISSN 1558-173X.

92. **Mahmood, F.**, **M. Toots**, **L. ÃŰfverstedt**, and **U. Skoglund**, 2d discrete fourier transform with simultaneous edge artifact removal for real-time applications. *In 2015 International Conference on Field Programmable Technology (FPT)*. 2015. ISSN null.

93. **Manolopoulos, K.**, **K. Nakos**, **D. Reisis**, **N. Vlassopoulos**, and **V. A. Chouliaras**, High performance 16k, 64k, 256k complex points vlsi systolic fft architectures. *In 2007 14th IEEE International Conference on Electronics, Circuits and Systems*. 2007. ISSN null.

94. **Marti-Puig, P.** and **R. Reig Bolano**, Radix-4 fft algorithms with ordered input and output data. *In 2009 16th International Conference on Digital Signal Processing*. 2009. ISSN 2165-3577.

95. **Mathieu, M.**, **M. Henaff**, and **Y. LeCun** (2013). Fast training of convolutional networks through ffts.

96. **Mohammad, K.** and **S. Agaian**, Efficient fpga implementation of convolution. *In 2009 IEEE International Conference on Systems, Man and Cybernetics*. 2009. ISSN 1062-922X.

97. **Motamedi, M.**, **P. Gysel**, **V. Akella**, and **S. Ghiasi**, Design space exploration of fpga-based deep convolutional neural networks. *In 2016 21st Asia and South Pacific Design Automation Conference (ASP-DAC)*. 2016. ISSN 2153-697X.

98. **Nguyen, D. T.**, **T. N. Nguyen**, **H. Kim**, and **H. Lee** (2019). A high-throughput and power-efficient fpga implementation of yolo cnn for object detection. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, **27**(8), 1861–1873. ISSN 1557-9999.

99. **Oh, J. Y.** and **M. S. Lim** (2005). New radix-2 to the 4th power pipeline fft processor. *IEICE Transactions on Electronics*, **8**, 1740–1746.

100. **Peemen, M.**, **A. A. A. Setio**, **B. Mesman**, and **H. Corporaal**, Memory-centric accelerator design for convolutional neural networks. *In 2013 IEEE 31st International Conference on Computer Design (ICCD)*. 2013. ISSN 1063-6404.

101. **Perri, S.**, **M. Lanuzza**, **P. Corsonello**, and **G. Cocorullo** (2005). A high-performance fully reconfigurable fpga-based 2d convolution processor. *Microprocessors and Microsystems*, **29**(8), 381 – 391. ISSN 0141-9331. URL http://www.sciencedirect.com/science/article/pii/S0141933104001413. Special Issue on FPGAs: Case Studies in Computer Vision and Image Processing.

102. **Podili, A.**, **C. Zhang**, and **V. Prasanna**, Fast and efficient implementation of convolutional neural networks on fpga. *In 2017 IEEE 28th International Conference on Application-specific Systems, Architectures and Processors (ASAP)*. 2017. ISSN 2160-052X.

103. **Proakis, J. G.** and **D. G. Manolakis**, *Digital Signal Processing Principles, Algorithms and Applications*. Prentice Hall, 1996.

104. **Puschel, M.**, **J. M. F. Moura**, **J. R. Johnson**, **D. Padua**, **M. M. Veloso**, **B. W. Singer**, **Jianxin Xiong**, **F. Franchetti**, **A. Gacic**, **Y. Voronenko**, **K. Chen**, **R. W. Johnson**, and **N. Rizzolo** (2005). Spiral: Code generation for dsp transforms. *Proceedings of the IEEE*, **93**(2), 232–275. ISSN 1558-2256.

105. **Qiu, J.**, **J. Wang**, **S. Yao**, **K. Guo**, **B. Li**, **E. Zhou**, **J. Yu**, **T. Tang**, **N. Xu**, **S. Song**, **Y. Wang**, and **H. Yang**, Going deeper with embedded fpga platform for convolutional neural network. *In Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, FPGA '16. ACM, New York, NY, USA, 2016. ISBN 978-1-4503-3856-1. URL http://doi.acm.org/10.1145/2847263.2847265.

106. **Rao, L.**, **B. Zhang**, and **J. Zhao** (2016). Hardware implementation of reconfigurable 1d convolution. *Journal of Signal Processing Systems*, **82**(1), 1–16. ISSN 1939-8115. URL https://doi.org/10.1007/s11265-015-0969-5.

107. **Reisis, D.** and **N. Vlassopoulos** (2008). Conflict-free parallel memory accessing techniques for fft architectures. *IEEE Transactions on Circuits and Systems I: Regular Papers*, **55**(11), 3438–3447. ISSN 1558-0806.

108. **Rodrguez-Ramos, J. M.**, **E. M. Castell**, **C. D. Conde**, **M. R. Valido**, and **J. Marichal-Hernndez** (2008). 2d-fft implementation on fpga for wavefront phase recovery from the cafadis camera. *Proc. SPIE 7015, Adaptive Optics Systems, 701539*, **7015**, 1–11.

109. **Russakovsky, O.**, **J. Deng**, **H. Su**, **J. Krause**, **S. Satheesh**, **S. Ma**, **Z. Huang**, **A. Karpathy**, **A. Khosla**, **M. Bernstein**, **A. C. Berg**, and **L. Fei-Fei** (2015). Imagenet large scale visual recognition challenge. *Int. J. Comput. Vision*, **115**(3), 211–252. ISSN 0920-5691. URL http://dx.doi.org/10.1007/s11263-015-0816-y.

110. **Russo, L. M.**, **E. C. Pedrino**, **E. Kato**, and **V. O. Roda**, Image convolution processing: A gpu versus fpga comparison. *In 2012 VIII Southern Conference on Programmable Logic*. 2012. ISSN null.

111. **S, K.** (2013). *ASIC Implementation of a high throughput, low latency, memory optimized FFT Processor*. Ph.D. thesis, MS Thesis, Indian Institute of Science (IISc) Bangalore, India.

112. **S.-S. Wang** and **C.-S. Li** (2008). An area-efficient design of variable-length fast fourier transform processor. *Journal of Signal Processing Systems*, **51**(245).

113. **Saponara, S.**, **M. Rovini**, **L. Fanucci**, **A. Karachalios**, **G. Lentaris**, and **D. Reisis** (2012). Design and comparison of fft vlsi architectures for soc telecom applications with different flexibility, speed and complexity trade-offs. *Circuits Syst Signal Process*, **31**, 627–649.

114. **Sermanet, P.**, **D. Eigen**, **X. Zhang**, **M. Mathieu**, **R. Fergus**, and **Y. LeCun**, Overfeat: Integrated recognition, localization and detection using convolutional networks. *In Proceedings of ICLR*. 2014.

115. **Shen, J.**, **Y. Qiao**, **Y. Huang**, **M. Wen**, and **C. Zhang**, Towards a multi-array architecture for accelerating large-scale matrix multiplication on fpgas. *In 2018 IEEE International Symposium on Circuits and Systems (ISCAS)*. 2018. ISSN 2379-447X.

116. **Shih, X.**, **H. Chou**, and **Y. Liu** (2018). Vlsi design and implementation of reconfigurable 46-mode combined-radix-based fft hardware architecture for 3gpp-lte applications. *IEEE Transactions on Circuits and Systems I: Regular Papers*, **65**(1), 118–129. ISSN 1558-0806.

117. **Shirazi, N.**, **P. M. Athanas**, and **A. L. Abbott**, Implementation of a 2-d fast fourier transform on an fpga-based custom computing machine. *In Proceedings of the 5th International Workshop on Field-Programmable Logic and Applications*, FPL '95. Springer-Verlag, London, UK, UK, 1995. ISBN 3-540-60294-1. URL `http://dl.acm.org/citation.cfm?id=647922.741016`.

118. **Shousheng He** and **M. Torkelson**, A new approach to pipeline fft processor. *In Proceedings of International Conference on Parallel Processing*. 1996. ISSN null.

119. **Shousheng He** and **M. Torkelson**, Design and implementation of a 1024-point pipeline fft processor. *In Proceedings of the IEEE 1998 Custom Integrated Circuits Conference (Cat. No.98CH36143)*. 1998*a*. ISSN null.

120. **Shousheng He** and **M. Torkelson**, Designing pipeline fft processor for ofdm (de)modulation. *In 1998 URSI International Symposium on Signals, Systems, and Electronics. Conference Proceedings (Cat. No.98EX167)*. 1998*b*. ISSN null.

121. **Simonyan, K.** and **A. Zisserman**, Very deep convolutional networks for large-scale image recognition. *In ICLR 2015*. 2015.

122. **Sorokin, H.** and **J. Takala**, Conflict-free parallel access scheme for mixed-radix fft supporting i/o permutations. *In 2011 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. 2011. ISSN 1520-6149.

123. **StrÃűm, H.** (2016). *A Parallel FPGA Implementation of Image Convolution*. Ph.D. thesis, Master of Science Thesis in Electrical Engineering, Department of Electrical Engineering, LinkÃűping University.

124. **Suda, N.**, **V. Chandra**, **G. Dasika**, **A. Mohanty**, **Y. Ma**, **S. Vrudhula**, **J.-s. Seo**, and **Y. Cao**, Throughput-optimized opencl-based fpga accelerator for large-scale convolutional neural networks. *In Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, FPGA '16. ACM, New York, NY, USA, 2016. ISBN 978-1-4503-3856-1. URL `http://doi.acm.org/10.1145/2847263.2847276`.

125. **Swartzlander, E. E. J.** (2007). Systolic fft processors: A personal perspective. *Journal of VLSI Signal Processing*, **53**, 3–14.

126. **Szegedy, C.**, **Wei Liu**, **Yangqing Jia**, **P. Sermanet**, **S. Reed**, **D. Anguelov**, **D. Erhan**, **V. Vanhoucke**, and **A. Rabinovich**, Going deeper with convolutions. *In 2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 2015. ISSN 1063-6919.

127. **T., D.** (2001). Two virtex-ii fpgas deliver fastest, cheapest, best high-performance image processing system. *Xilinx Xcell Journal*, **41**, 70–71.

128. **Tang, S.**, **C. Liao**, and **T. Chang** (2012). An area- and energy-efficient multimode fft processor for wpan/wlan/wman systems. *IEEE Journal of Solid-State Circuits*, **47**(6), 1419–1435.

129. **Tang, S.**, **J. Tsai**, and **T. Chang** (2010). A 2.4-gs/s fft processor for ofdm-based wpan applications. *IEEE Transactions on Circuits and Systems II: Express Briefs*, **57**(6), 451–455. ISSN 1558-3791.

130. **Tsai, P.** and **C. Lin** (2011). A generalized conflict-free memory addressing scheme for continuous-flow parallel-processing fft processors with rescheduling. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, **19**(12), 2290–2302. ISSN 1557-9999.

131. **Tu, F.**, **S. Yin**, **P. Ouyang**, **S. Tang**, **L. Liu**, and **S. Wei** (2017). Deep convolutional neural network architecture with reconfigurable computation patterns. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, **25**(8), 2220–2233. ISSN 1557-9999.

132. **Uzun, I. S.**, **A. Amira**, and **A. Bouridane** (2005). Fpga implementations of fast fourier transforms for real-time signal and image processing. *IEE Proceedings - Vision, Image and Signal Processing*, **152**(3), 283–296. ISSN 1350-245X.

133. **Wang, H.**, **J. Lin**, **Y. Xie**, **B. Yuan**, and **Z. Wang**, Efficient reconfigurable hardware core for convolutional neural networks. *In 2018 52nd Asilomar Conference on Signals, Systems, and Computers*. 2018*a*. ISSN 1058-6393.

134. **Wang, H.**, **W. Liu**, **T. Xu**, **J. Lin**, and **Z. Wang**, A low-latency sparse-winograd accelerator for convolutional neural networks. *In ICASSP 2019 - 2019 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. 2019. ISSN 1520-6149.

135. **Wang, J.**, **J. Lin**, and **Z. Wang**, Efficient convolution architectures for convolutional neural network. *In 2016 8th International Conference on Wireless Communications Signal Processing (WCSP)*. 2016. ISSN 2472-7628.

136. **Wang, J.**, **J. Lin**, and **Z. Wang** (2018*b*). Efficient hardware architectures for deep convolutional neural network. *IEEE Transactions on Circuits and Systems I: Regular Papers*, **65**(6), 1941–1953. ISSN 1558-0806.

137. **Wang, M.** and **Z. Li**, A hybrid sdc/sdf architecture for area and power minimization of floating-point fft computations. *In 2016 IEEE International Symposium on Circuits and Systems (ISCAS)*. 2016. ISSN 2379-447X.

138. **Wang, W.**, **B. Duan**, **C. Zhang**, **P. Zhang**, and **N. Sun**, Accelerating 2d fft with non-power-of-two problem size on fpga. *In 2010 International Conference on Reconfigurable Computing and FPGAs*. 2010. ISSN 2325-6532.

139. **Wang, X.**, **C. Wang**, and **X. Zhou**, Work-in-progress: Winonn: Optimising fpga-based neural network accelerators using fast winograd algorithm. *In 2018 International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*. 2018*c*. ISSN null.

140. **Winograd, S.** (1980). Arithmetic complexity of computations. *Society for Industrial and Applied Mathematics*.

141. **Wu, D.**, **Y. Zhang**, **X. Jia**, **L. Tian**, **T. Li**, **L. Sui**, **D. Xie**, and **Y. Shan**, A high-performance cnn processor based on fpga for mobilenets. *In 2019 29th International Conference on Field Programmable Logic and Applications (FPL)*. 2019. ISSN 1946-147X.

142. **Xia, K.**, **B. Wu**, **X. Zhou**, and **T. Xiong**, A generalized conflict-free address scheme for arbitrary 2k-point memory-based fft processors. *In 2016 IEEE International Symposium on Circuits and Systems (ISCAS)*. 2016. ISSN 2379-447X.

143. **Xia, L.**, **L. Diao**, **Z. Jiang**, **H. Liang**, **K. Chen**, **L. Ding**, **S. Dou**, **Z. Su**, **M. Sun**, **J. Zhang**, and **W. Lin**, Pai-fcnn: Fpga based inference system for complex cnn models. *In 2019 IEEE 30th International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, volume 2160-052X. 2019. ISSN 2160-0511.

144. **Xiao, Q.**, **Y. Liang**, **L. Lu**, **S. Yan**, and **Yu-Wing Tai**, Exploring heterogeneous algorithms for accelerating deep convolutional neural networks on fpgas. *In 2017 54th ACM/EDAC/IEEE Design Automation Conference (DAC)*. 2017. ISSN null.

145. **Xing, Q.**, **Z. Ma**, and **Y. Xu** (2017). A novel conflict-free parallel memory access scheme for fft processors. *IEEE Transactions on Circuits and Systems II: Express Briefs*, **64**(11), 1347–1351. ISSN 1558-3791.

146. **Yang, C.**, **Y. Wang**, **X. Wang**, and **L. Geng** (2019). Wra: A 2.2-to-6.3 tops highly unified dynamically reconfigurable accelerator using a novel winograd decomposition algorithm for convolutional neural networks. *IEEE Transactions on Circuits and Systems I: Regular Papers*, **66**(9), 3480–3493. ISSN 1558-0806.

147. **Yang, K.**, **S. Tsai**, and **G. C. H. Chuang** (2013*a*). Mdc fft/ifft processor with variable length for mimo-ofdm systems. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, **21**(4), 720–731.

148. **Yang, K.**, **S. Tsai**, and **G. C. H. Chuang** (2013*b*). Mdc fft/ifft processor with variable length for mimo-ofdm systems. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, **21**(4), 720–731. ISSN 1557-9999.

149. **Yin, X.**, **F. Yu**, and **Z. Ma** (2016). Resource-efficient pipelined architectures for radix-2 real-valued fft with real datapaths. *IEEE Transactions on Circuits and Systems II: Express Briefs*, **63**(8), 803–807. ISSN 1558-3791.

150. **Yu, C.**, **C. Chakrabarti**, **S. Park**, and **V. Narayanan**, Bandwidth-intensive fpga architecture for multi-dimensional dft. *In 2010 IEEE International Conference on Acoustics, Speech and Signal Processing*. 2010. ISSN 1520-6149.

151. **Yu, C.**, **M. Yen**, **P. Hsiung**, and **S. Chen** (2011). A low-power 64-point pipeline fft/ifft processor for ofdm applications. *IEEE Transactions on Consumer Electronics*, **57**(1), 40–40. ISSN 1558-4127.

152. **Yu, C.-L.**, **J.-S. Kim**, **L. Deng**, **S. Kestur**, **V. Narayanan**, and **C. Chakrabarti** (2011). Fpga architecture for 2d discrete fourier transform based on 2d decomposition for large-sized data. *Journal Signal Process Syst (2011)*, **64**(1), 109–122.

153. **Yu, J.**, **Y. Hu**, **X. Ning**, **J. Qiu**, **K. Guo**, **Y. Wang**, and **H. Yang**, Instruction driven cross-layer cnn accelerator with winograd transformation on fpga. *In 2017 International Conference on Field Programmable Technology (ICFPT)*. 2017. ISSN null.

154. **Yu, Y.**, **C. Wu**, **T. Zhao**, **K. Wang**, and **L. He** (2019). Opu: An fpga-based overlay processor for convolutional neural networks. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 1–13. ISSN 1557-9999.

155. **Yu-Wei Lin**, **Hsuan-Yu Liu**, and **Chen-Yi Lee** (2004). A dynamic scaling fft processor for dvb-t applications. *IEEE Journal of Solid-State Circuits*, **39**(11), 2005–2013. ISSN 1558-173X.

156. **Yu-Wei Lin**, **Hsuan-Yu Liu**, and **Chen-Yi Lee** (2005). A 1-gs/s fft/ifft processor for uwb applications. *IEEE Journal of Solid-State Circuits*, **40**(8), 1726–1735. ISSN 1558-173X.

157. **Yufei Ma**, **N. Suda**, **Yu Cao**, **J. Seo**, and **S. Vrudhula**, Scalable and modularized rtl compilation of convolutional neural networks onto fpga. *In 2016 26th International Conference on Field Programmable Logic and Applications (FPL)*. 2016. ISSN 1946-1488.

158. **Yutian Zhao**, **A. T. Erdogan**, and **T. Arslan**, A novel low-power reconfigurable fft processor. *In 2005 IEEE International Symposium on Circuits and Systems*. 2005.

159. **Zeng, H.**, **R. Chen**, **C. Zhang**, and **V. Prasanna**, A framework for generating high throughput cnn implementations on fpgas. *In Proceedings of the 2018 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, FPGA '18. ACM, New York, NY, USA, 2018. ISBN 978-1-4503-5614-5. URL `http://doi.acm.org/10.1145/3174243.3174265`.

160. **Zhang, C.**, **P. Li**, **G. Sun**, **Y. Guan**, **B. Xiao**, and **J. Cong**, Optimizing fpga-based accelerator design for deep convolutional neural networks. *In Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, FPGA '15. ACM, New York, NY, USA, 2015. ISBN 978-1-4503-3315-3. URL `http://doi.acm.org/10.1145/2684746.2689060`.

161. **Zhang, C.**, **D. Wu**, **J. Sun**, **G. Sun**, **G. Luo**, and **J. Cong**, Energy-efficient cnn implementation on a deeply pipelined fpga cluster. *In Proceedings of the 2016 International Symposium on Low Power Electronics and Design*, ISLPED '16. ACM, New York, NY, USA, 2016. ISBN 978-1-4503-4185-1. URL `http://doi.acm.org/10.1145/2934583.2934644`.

162. **Zhang, C.**, **Zhenman Fang**, **Peipei Zhou**, **Peichen Pan**, and **Jason Cong**, Caffeine: Towards uniformed representation and acceleration for deep convolutional neural networks. *In 2016 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. 2016. ISSN 1558-2434.

163. **Zhuge, C.**, **X. Liu**, **X. Zhang**, **S. Gummadi**, **J. Xiong**, and **D. Chen**, Face recognition with hybrid efficient convolution algorithms on fpgas. *In Proceedings of the 2018 on Great Lakes Symposium on VLSI*, GLSVLSI '18. ACM, New York, NY, USA, 2018. ISBN 978-1-4503-5724-1. URL http://doi.acm.org/10.1145/3194554.3194597.

PUBLICATIONS BASED ON THESIS

Patent filed

1. Title of invention: **A Method for Implementing Convolutional Neural Networks using Winograd Minimal Filtering and GEMM**
   Inventors: Kala S, Babita Roslind Jose, Jimson Mathew, Nalesh S
   Indian Patent Application No.201841038043 A

International Journals

1. Kala S., Babita R Jose, Jimson Mathew, Nalesh S., **High Performance CNN Accelerator on FPGA Using Unified Winograd-GEMM Architecture**, *IEEE Transactions on Very Large Scale Integration Systems*, Vol. 27, No. 12, pp 2816-2828, 2019.
   DOI: 10.1109/TVLSI.2019.2941250

   This paper is an extension of VLSID 2019 conference paper. This paper gives the detailed description of *UniWiG* architecture and the details of FPGA implementation. Both fixed point and single precision floating point implementations of *UniWiG* has been described here. The performance analysis of various CNN models like AlexNet and VGG-16 on *UniWiG*, explained in Chapter 6 has been presented in this paper.

2. Kala S., Jimson Mathew, Babita R Jose, Nalesh S., **Radix-4^3 based Two Dimensional FFT Architecture with Efficient Data Reordering Scheme**, *IET Computers and Digital Techniques*, 2019, Vol. 13 Iss. 2, pp. 78-86,*DOI: 10.1049/iet-cdt.2018.5075*

   This paper is an extension of DTIS 2018 conference paper. This paper details about the Radix-4^3 based two dimensional FFT architecture which is presented in Chapter 3. ASIC and FPGA implementation details of 64×64 FFT and comparisons with state-of-art implementations are also discussed in this paper.

Book Chapter

1. Kala S., Nalesh S, Babita R Jose, Jimson Mathew., **Image reconstruction using novel two dimensional Fourier transform**, *Advances in Soft Computing and Machine Learning in Image Processing, Springer*, 2019, pp.699-718,
   https://doi.org/10.1007/978-3-319-63754-9_31

   A preliminary version of radix-4^3 as presented in initial section of Chapter 3 and ASIC synthesis results are discussed in this paper.

International Conferences

1. Kala S., Babita R Jose, Jimson Mathew, Nalesh S., **Efficient hardware acceleration of convolutional neural networks**, in *32nd IEEE International System-On-Chip Conference (SOCC)* 2019, Singapore, pp. 191-192, September 3-6, 2019.

   Discussions on inference and training of various CNN models in Chapter 4 and the architecture in Chapter 5 are presented in this paper.

2. Kala S., Jimson Mathew, Babita R Jose, Nalesh S. **UniWiG: Unified Winograd-GEMM Architecture for Accelerating CNN on FPGAs**, *32nd IEEE International Conference on VLSI Design & 18th International Conference on Embedded Systems (VLSID)*, New Delhi, pp. 209-214, Jan 5-9, 2019.

   A Unified Winograd-GEMM based CNN architecture, called as UniWiG, as discussed in Chapter 4, is presented in this paper. Performance model of *UniWiG* and extra BRAM requirements are also explained in this paper. Here, 32-bit floating point implementation of AlexNet using the proposed architecture on Virtex-7 FPGA is described.

3. Kala S, Babita R Jose, Jimson Mathew, Nalesh S., **Accelerating Convolutional Neural Networks on FPGA**, in *Student Research Forum, 33rd IEEE International Conference on VLSI Design & 19th International Conference on Embedded Systems (VLSID)*, Bangalore, 2020.
   (**Best Paper in Student Research Forum (Runner Up)**)

   An extended abstract of the PhD thesis is presented in the Student Research Forum.

4. Kala S., Debdeep Paul, Babita R Jose, Jimson Mathew, Nalesh S., **Performance Analysis of Convolutional Neural Network Models**, *9th IEEE International Conference on Advances in Computing & Communications (ICACC)*, Kochi, Kerala, November 2019.

   Details of inference and training of CNNs in GPU, CPU and Jetson TX2 in Chapter 4 is discussed in this paper. Hardware architecture for implementing AlexNet on FPGA is also presented in this paper.

5. Kala S., Babita R Jose, Debdeep Paul, Jimson Mathew. **A Hardware Accelerator for Convolutional Neural Network Using Fast Fourier Transform**, in *22nd International Symposium on VLSI Design And Test (VDAT)*, Tamil Nadu, pp. 28-36, June 2018.

   This paper discusses about an FFT based CNN implementation on FPGA, as presented in Chapter 4.

6. Kala S., Debdeep Paul, Babita R Jose, Nalesh S. **Design Space Exploration of Convolution Algorithms to Accelerate CNNs on FPGA**, in *8th IEEE International Symposium on Embedded System Design (ISED)*, Kochi, pp. 21-25, December 2018.

   Details of various convolution schemes for implementation of convolutional layers in CNN and their comparisons as presented in Chapter 4 is discussed here.

7. Kala S., Nalesh S., Babita R Jose, Jimson Mathew, Marco Ottavi. **Two Dimensional FFT Architecture Based on Radix-4^3 Algorithm with Efficient Output Reordering**, in *IEEE 13th Int'l Conference on Design & Technology of Integrated Systems in Nanoscale Era (DTIS)*, Italy, April 2018.

   An output reordered two dimensional FFT architecture for image processing applications, which is discussed in Chapter 3 is explained here. ASIC implementation results are also discussed.

CURRICULUM VITAE

Name:	**Kala S**
Gender:	Female
Date of birth:	19^{th} September 1984

Educational Qualifications

- 2002 - 2006
  B.Tech. (Bachelors of Technology) degree in Electronics and Communication Engineering
  Sree Narayana Gurukulam College of Engineering, Mahatma Gandhi University, Kerala,India.

- 2007 - 2009
  M.E. (Master of Engineering) degree in VLSI Design
  Anna University, TamilNadu, India.

- 2010 - 2013
  Master of Science (Engineering) by research
  Center for Nanoscience and Engineering (CeNSE),
  Indian Institute of Science (IISc) Bangalore, India

- 2016 - Present
  Ph.D (Doctor of Philosophy)
  Division of Electronics, SOE,
  Cochin University of Science & Technology, Kerala, India.
  Reg Date : 07-11-2016

Research Experience

- February 2013 to November 2013
  Project Associate
  Computer Aided Design Laboratory
  Supercomputer Education and Research Centre
  Indian Institute of Science, Bangalore
  Bangalore - 560 012

- November 2016 to November 2018
  University Junior Research Fellow (UJRF)

Division of Electronics, SOE
Cochin University of Science & Technology, Kochi-22

- December 2018 to December 2019
  Woman Scientist Research Fellow, under Woman Scientist Division (WSD), Kerala State Council for Science, Technology and Environment (KSCSTE)
  Division of Electronics, SOE
  Cochin University of Science & Technology, Kochi-22