Hardware-Efficient & High-Throughput VLSI-Architectures of Convolutional-Neural-Network Accelerator for Edge Applications

Thesis submitted in partial fulfillment of

the requirements for the degree of

DOCTOR OF PHILOSOPHY

by

Md. Najrul Islam

Enrollment No. D18064

Supervisors

Dr. Rahul Shrestha (Guide)

&

Prof. Shubhajit Roy Chowdhury(Co-Guide)

SCHOOL OF COMPUTING AND ELECTRICAL ENGINEERING

INDIAN INSTITUTE OF TECHNOLOGY MANDI

Oct. 2025

The best of people are those who are most beneficial to others.

----Prophet Mohammed PBUH

In the name of the Almighty, whose mercy surrounds all endeavors.

I dedicate this work to my family, whose love and sacrifices shaped my path.

Declaration by the Research Scholar

I hereby declare that the entire work embodied in this thesis entitled **"Hardware-Efficient & High-Throughput VLSI-Architectures of Convolutional-Neural-Network Accelerator for Edge Applications"** is the result of investigations carried out by me in the School of Computing and Electrical Engineering, Indian Institute of Technology Mandi, Mandi, Himachal Pradesh, India, under the supervision of **Dr. Rahul Shrestha** (Guide), Associate Professor, School of Computing and Electrical Engineering, Indian Institute of Technology Mandi, Mandi, Himachal Pradesh, India, and **Prof. Shubhajit Roy Chowdhury** (Co-Guide), Professor, School of Computing and Electrical Engineering, Indian Institute of Technology Mandi, Mandi, Himachal Pradesh, India

I also declare that it has not been submitted elsewhere for the award of any degree or diploma. In keeping with general practice, due acknowledgments have been made wherever the work described is based on the findings of other investigators. Any omissions that might have occurred due to oversight or error in judgement are regretted.

Md. Najrul Islam
Enrollment No. D18064
School of Computing and Electrical Engineering,
Indian Institute of Technology Mandi,
Mandi, Himachal Pradesh, India - 175075.

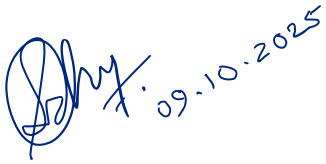
Date: 9th Oct. 2025
Place: IIT Mandi,
Himachal Pradesh, India

Declaration by the Research Advisors

It is certified that the entire work in this thesis entitled **"Hardware-Efficient & High-Throughput VLSI-Architectures of Convolutional-Neural-Network Accelerator for Edge Applications"** has been carried out by **Md. Najrul Islam,** Enrollment No. D18064, under our supervision and guidance for the degree of Doctor of Philosophy in the School of Computing and Electrical Engineering, Indian Institute of Technology Mandi, Mandi, Himachal Pradesh, India.

To the best of our knowledge and belief, present thesis completed by **Md. Najrul Islam** fulfils the requirements of the Ph.D. ordinance of the Indian Institute of Technology Mandi, Mandi, Himachal Pradesh, India. It contains original work of the candidate and no part of it has been submitted elsewhere for any degree or diploma.

Dr. Rahul Shrestha (Guide)
Associate Professor
School of Computing and Electrical Engineering,
Indian Institute of Technology Mandi,
Mandi, Himachal Pradesh, India - 175075.



Prof. Shubhajit Roy Chowdhury
(Co-Guide)
Professor, School of Computing and Electrical Engineering,
Indian Institute of Technology Mandi,
Mandi, Himachal Pradesh, India - 175075.

Acknowledgments

I bow to the *Almighty*, source of all knowledge and strength, without whose will nothing is possible. First and foremost, I would like to express my deepest and most sincere gratitude to my guide, Dr. Rahul Shrestha. His unwavering support, belief in my potential, and constant encouragement have played an instrumental role in shaping this research. He stood by me during challenging times, offering not only academic guidance but also personal support and reassurance. His patience, clarity of thought, and tireless efforts have inspired me throughout this journey. I consider myself extremely fortunate to have had him as my mentor, and this work would not have been possible without his constant presence and belief in me.

I would also like to sincerely thank my co-guide, Prof. Shubhajit Roy Chowdhury, for his support and presence throughout this research. His insights, feedback, and encouragement during various stages of the work have been valuable, and I am grateful for the time and consideration he offered during this journey.

I would like to express my heartfelt gratitude.....

To Dr. Hitesh Shrimali, for his constant encouragement, support and inspiration throughout the thesis journey. I am thankful to him for sharing his immense knowledge about VLSI design and valuable feedback towards my thesis work.

To Prof. Anup Dandapat for his invaluable guidance and encouragement at a crucial stage of my academic journey. It was his suggestion and support that inspired me to pursue a Ph.D. at IIT, a decision that has profoundly shaped my professional and personal growth.

To Dr. Ranjeet Jha and Dhaval Patel for their invaluable support in helping me understand the Python implementation of AI concepts. Dr. Jha's guidance in resolving my doubts and Dhaval Patel's detailed hands-on explanation videos both played a significant role in

strengthening the software implementation part of my work.

To Dr. Vipin Kizheppatt, whose insightful YouTube videos helped me clear many doubts and overcome hurdles related to hardware-software co-design.

To Dr. Sandeep Kumar Mishra for his guidance and support in helping me learn and navigate Electronic Design Automation (EDA) tools. His expertise and patient instruction provided a strong foundation for the technical aspects of my research.

To Dr. Rohit Chaurasiya for his selfless support, patience, and loving-kindness throughout my Ph.D. journey. He taught me the ASIC design flow, helped me understand numerous tools, and cleared countless doubts along the way. Our technical discussions not only deepened my knowledge but also made the learning process enjoyable. I truly cherish the valuable time spent together and the beautiful memories we created during this journey.

To Dr. Anuj Verma for his helpful guidance in clarifying my doubts related to inferring BRAM and managing clocks in FPGA design.

To Dr. Sushantha Gogoi for encouraging me to join IIT Mandi and for his insightful suggestion to explore hardware-software co-design in response to a specific challenge I was facing.

To the anonymous reviewers of IEEE TCAS-I, TVLSI, ISVLSI and VLSID, for their critical assessments towards the work carried out in my thesis.

To Shakti, Rahul Sharma, Nandit, Saurabh, Puneet Mishra, and Dr. Dinesh for technical and entertaining discussions.

To Sahil, Ayush, and Alip for their selfless help.

To all of my labmates for the joyful memories, shared laughter, and the sense of community that made the work environment both inspiring and enjoyable.

To Tarun Verma from Digital Analog Electronics Lab, Shivam Prajapati from Control System Laboratory and Sumit Maan from Signal Processing and Communication Lab for their kind cooperation.

To Nalini Ma'am, Rakhi Ma'am, Monu sir, Shiv, and all other staff from SCEE office and academic section, finance Section and library section for their kind-coperation and patience throughout the thesis journey.

To Ravi, Chaman Lal ji, and Narnder ji, caretaker of Dashir and BeasKund hostels, for

their cooperation in listening and addressing my concerns.

To Ashish Tiwari, Kanchan Singh Rana, Saurabh Dhiman, Shashwat, Shakti, Prashant, Monu, Shubham, and Aisharya for all the fruitful discussions during the course work and for their time in creating unforgettable memories.

To all staff of health centre, IIT Mandi, for their immense help during tough times.

To my mother, sister, brother, and wife for their unwavering love, support, and encouragement throughout this journey.

To all the sentient beings who knowingly or unknowingly have added value to my thesis work.

Md. Najrul Islam

Abstract

Rapid advancement of artificial intelligence (AI) and deep learning has driven the need for efficient hardware accelerators that are capable of handling complex convolutional neural network (CNN) computations. Though conventional processors like CPUs and GPUs are widely used, they are not ideally suited for the deployment of edge applications due to limitations in energy efficiency, data movement, and processing throughput. In this doctoral thesis, these challenges are addressed by presenting custom very-large scale-integration (VLSI) architectures tailored for CNN acceleration, optimized for both inference and training in resource-constrained environments. The proposed adaptive convolution mapping technique achieves a 1.71× improvement in the computation per multiply-accumulate (MAC) unit over the state-of-the-art designs. In addition, an uninterrupted processing strategy, leveraging a random-access line-buffer (RALB), delivers 2.55× higher throughput by eliminating memory stalls and maintaining a continuous dataflow. On further optimizing the energy efficiency, the suggested designs emphasize extensive local data reuse across all the CNN layers, leading to a 3.4× improvement in energy efficiency over state-of-the-art works. Furthermore, an unified CNN-accelerator design is presented in this thesis that supports both training and inference on a single architectural platform. It is incorporated with the optimized forward and backward passes, efficient data reuse, and scalable kernel mapping strategies to support a broad range of CNN workloads. Therefore, such unified CNN accelerator achieves 1.36× higher energy efficiency compared to reported training accelerators. The hardware architectures are implemented and evaluated on field-programmable gate-array (FPGA) platforms, confirming their ability to deliver high throughput, low power consumption, and scalability across a variety of CNN models.

Keywords: *CNN Accelerator, Deep Learning Hardware, FPGA Implementation, Energy-Efficient AI, Convolution Mapping, Data Reuse, VLSI Architecture, Inference and Training, Edge AI, Memory Optimization, Low-Power Computing, AI Hardware, Hardware Acceleration, Real-Time AI Processing.*

Contents

List of Figures

List of Tables

Abbreviations

1-9

1D: **O**ne **D**imensional.

3D: **T**hree **D**imensional.

A

ACCs: **Acc**umulators.

ACSU: **A**ctivation **S**earching **U**nit.

ADAS: **A**dvanced **D**river **A**ssistance **S**ystems.

AGU: **A**ddress-Generation **U**nits.

AI: **A**rtificial **I**ntelligence.

ASIC: **A**pplication **S**pecific **I**ntegrated **C**ircuit.

ASR: **A**utomatic **S**peech **R**ecognition.

***AvgPool*:** **Av**era**g**e **Pool**.

AXI: **A**dvanced **E**xtensible **I**nterface.

B

BERT: **B**idirectional **E**ncoder **R**epresentations from **T**ransformers.

BNN: **B**oltzmann **N**eural **N**etworks.

BRAM: **B**lock **RAM**.

C

CE: **C**ompute **E**fficiency.

***CN-DC*:** **C**lass **N**o. of the **D**etected object **C**lass.

CNG: **C**lass **N**o. **G**enerator.

CNNs: **C**onvolutional **N**eural **N**etworks.

***Conv*:** **Conv**olution.

CORDIC: **C**oordinate-**R**otations **D**igital-Computer.

CRC: **C**urrent **R**ead-Line-Select **C**ontroller.

CT: **C**ompute **T**ile.

CU: **C**lassify **U**nit.

CUC: **C**lassify **U**nit **C**ontroller.

CWC: **C**urrent **W**rite-Line-Select **C**ontroller.

D

DeMUX: **D**e-**MU**ltiple**X**er.

DL: **D**eep **L**earning.

DMA: **D**irect-**M**emory-**A**ccess.

DRAM: **D**ynamic **R**andom-**A**ccess-**M**emory.

DSR: **D**ata-&-**S**ignal **R**outer.

DWC: **D**epth-**W**ise Separable **C**onvolution.

E

EDA: **E**lectronics-**D**esign-**A**utomation.

F

***FC*$_{Last}$:** **Last F**ully **C**onnected .

FDSOI: **F**ully **D**epleted **S**ilicon **O**n **I**nsulator.

FP16: 16-bit **F**ixed-**P**oint Format.

FPGAs: **F**ield-**P**rogrammable **G**ate **A**rrays.

G

***G*:** **G**radient.

***G.B*:** **G**radients for **B**ias Values.

***G.I*:** **G**radients of **I**nput feature map.

***G.W*:** **G**radients of filter **W**eights.

GBCU: **G**lobal-**B**uffer cum **C**ontrol **U**nit.

***GCU*:** **G**radient **C**ompute **U**nit.

GPT: **G**enerative **P**re-training **T**ransformer.

GPU: **G**raphic **P**rocessing **U**nit.

H

HDL: **H**ardware-**D**escription-**L**anguage.

HGU: **H**orizontal **G**radient-**C**ompute **U**nit.

I

I: **I**nput feature map.

IDM: **I**nput **D**ata **M**onitor.

IEC: **I**nference **E**ngine **C**ontroller.

***IF-intr*:** **int**e**r**rupt signal from **IF** buffer .

ILA: **I**ntegrated **L**ogic **A**nalyzer.

iMC: **i**n-**M**emory **C**omputation.

IP: **I**ntellectual-**P**roperty.

ISS: **I**nput **S**election **S**witch.

K

KM: **K**ernel **M**emory.

KPC: **K**ernel **P**rocessing **C**ontroller.

KPU: **K**ernel **P**rocessing **U**nit.

L

***L*:** **L**oss Matrix.

LLMs: **L**arge **L**anguage **M**odels.

***Ln.S.C*:** **Li**ne-**S**election **C**ontrol-signal.

***Ls.*S_h:** **h**orizontal **Lo**ss **S**elector.

***Ls.*S_v:** **v**ertical **Lo**ss **S**elector.

LUT: **L**ook-**U**p **T**able.

M

MAC: **M**ultiply-**A**nd-**A**ccumulate.

***MaxPool*:** **Max Pool**ing operation.

ML: **M**achine **L**earning.

MLP: **M**ulti**L**ayer **P**erceptrons.

MUX: **MU**ltiple**X**ers.

N

NER: **N**amed **E**ntity **R**ecognition.

NLP: **N**atural **L**anguage **P**rocessing.

NN: **N**eural **N**etworks.

***Nxt.S*:** **Nxt S**tride signal.

O

OPs: **O**perations **P**er **s**econd.

P

PE: **P**rocessing **E**lement.

PMAU: **P**arallel **M**ultiply-&-**A**dd **U**nit.

PMOD: **P**eripheral-**MOD**ule interface.

PWC: **P**oint-**W**ise **C**onvolution.

R

RAG: **R**ead **A**ddress **G**enerator.

RALB: **R**andom-**A**ccess **L**ine-**B**uffer.

***Rd.S*:** **Rea**d **S**elector.

***RdPtr*:** **R**ead **P**ointers.

***rd-rdy*:** **rea**d **r**ea**dy** signal.

RE: **R**ead **E**nable signal.

REG: **REG**isters.

ReLU: **R**ectified **L**inear **U**nit.

ReLU6: A special **ReLU** that clips input values between 0 and **6**.

RGB: **R**ed **G**reen **B**lue color.

RL: **R**einforcement **L**earning.

RNN: **R**ecurrent **N**eural **N**etworks.

***Rs.S*:** **S**elect- for -**R**eu**s**e signal.

S

SD: **S**ecure **D**igital (Memory **C**ard).

***Str.Req*:** **Str**ide **Req**uest signal.

SZD: **S**ign-**A**nd-**Z**ero **D**etector.

T

TPUs: **T**ensor **P**rocessing **U**nits.

TTS: **T**ext- **T**o -**S**peech.

W

WAG: **W**rite **A**ddress **G**enerator.

WE: **W**rite **E**nable port.

***wr*$_{act}$:** **wr**ite-**act**ivation signal.

***Wr*:** **Wr**ite command signal.

***wr-dn*:** **wr**ite-**d**o**n**e indicator signal.

***Wt.S*:** **W**ri**t**e **S**elector.

Symbols

α	Width of a kernel.
$\hat{\alpha}$	Energy consumption associated with each $\hat{a}$ type data movement.
β	Height of a kernel.
$\hat{\beta}$	Energy consumption associated with each $\hat{b}$ type data movement.
δ	Index of the filter kernel.
ϵ	Energy efficiency, expressed as $\epsilon = \Theta_T / \mathcal{W}$.
η_{MAC}	Compute efficiency, calculated as the number of operations performed per MAC unit (ops/MAC).
η_{PE}	Throughput density, given as the number of operations performed by a single PE per unit time (i.e. Ops/PE).
η_{PE-N}	Hardware Efficiency at normalized clock frequency.
γ	Depth of the filter kernel.
$\hat{\gamma}$	Energy consumption associated with each $\hat{c}$ type data movement.
λ_c	Latency of conventional CNN accelerators, given as $\lambda_c = \sum_{i=1}^{n}(t_{f_i} + t_{c_i})$.
λ_p	Latency of RALB based CNN accelerators, given as: $\lambda_p = (\sum_{i=1}^{n} t_{c_i}) + t_{pf1}$.
Ω	PE efficiency, represented by $\Omega = (N_{PE-util.}/N_{PE})$.
ψ_m	Energy required to move the data between the computation unit of PE and storage locations at different hierarchy. Expressed as $\psi_m = a{\cdot}\alpha + b{\cdot}\beta + c{\cdot}\gamma$.
σ	Time efficiency, calculated as the ratio of time duration when PEs are active and total time duration required for the computation.
Rr:	Read signal from KPC for PE.
Θ_T	Computation throughput, given as $\Theta_T = 2(N_{PE} \times f_{clk} \times \Omega \times \sigma)$ number of operations per second.

ε_c	Computation cost for each data which depends on the design logic and its precision.
ξ	Number of horizontal strides to calculate $G.I$.
FC_{last}	Last fully connected layer.
A	Width of each row of I.
$\hat{a}$	Number of times a data has been loaded from DRAM to GBCU.
AC_{Max}	Largest activation.
AC-Data	Output feature-map matrix from PMAU.
AC	Output activation values.
B	Bias values.
$\hat{b}$	Number of times the a data has been loaded from GBCU-to-RALB.
BF16	16 bit brain-float format.
$\hat{c}$	Number of times the a data has been loaded from RALB-to-PE.
$C_{\alpha\beta}$	Computation share contributed by each filter of size $\alpha\times\beta$, which is calculated as $C_{\alpha\beta} = \left(\mathcal{M}_{\alpha\beta}/\mathcal{M}_{tot}\right)\times 100\ \%$.
CN-DC	Class id of detected object (classification result).
$CN\text{-}DC_{i-1}$	Class id of the object associated with $AC_{Max_{i-1}}$.
conv-type	Control signal to indicate the type of convolution.
f_{clk}	Operating Clock frequency.
F_i	i^{th} fetching event.
G.B	Gradients for bias values.
G.I	Gradients for input feature map.
G.W	Gradients for filter weights.
I	Input feature map.
I-Data	Output bus of *I* buffer.
IRC	*I* read-state controller.

I-shape	Dimension of I.
itr_i.	i^{th} iteration.
I-W	Control signal to chose between input feature-map and filter weight.
l	Index of current layer.
L	The loss matrix corresponding to a 3D filter.
M	Number of items that can be stored in an RALB.
m	Number of PEs in a rows.
$\breve{M}$	Number of neurons in the FC_{last} layer.
$\hat{M} \times \hat{N} \times \hat{O}$	Dimensions of 3D loss matrix L.
$\mathcal{M}_{\alpha\beta}$	Number of MACs contributed by filters of size $\alpha \times \beta$.
$\mathcal{M}_{tot}$	Number of MACs contributed by all the filters of a model.
$\max\{I(x:\alpha,\ y:\beta,\ z)\}$	Maxpool: extraction of largest element from the $\alpha \times \beta$ matrix window of I with an origin at x, y.
$max\{0,\ I_{x,y,z}\}$	Perform ReLU on $I_{x,y,z}$.
n	Number of PEs in a column.
N	Total number of classes in the model.
N_{PE}	Total number of PEs available in the KPU.
$N_{PE-util.}$	Number of PEs utilized.
n_l	Number of iterations required for l^{th} layer.
O_i	i^{th} output port of the line memory.
P_i	i^{th} Process event.
Pr_i	Probability score of i^{th} class.
$Psum$	Partial sum (Incomplete AC).
$Psum_i$	Input partial sum.
$Psum_o$	Output partial sum.
$Q_{n.m}$	n+m bit fixed point representation with n bits for integer part and m bits for the fractions.

r_l	Minimum number of data items that are required to begin the processing of l^{th} layer.
s	Stride size.
S-Ovd	Controll signal to override SZD for first layer of the model.
t	Duration of computation.
t_{c_i}	Data computation time of itr_i.
t_{f_i}	Data fetching time for itr_i.
t_{itr_i}	Total time for itr_i, computed as $t_{itr_i} = t_{f_i}+t_{c_i}$.
t_{pf_i}	Pre-fetch duration to load minimum number of data for itr_i.
U	Activity rate of PEs that is given by $U=t_{ci}/t_{itr_i}$.
W	Filter weights.
$\mathcal{W}$	Average power consumption, computed as $\mathcal{W} = (\varepsilon_c + \psi_m)/t$.
wr_{act}	Write-activation signal.
wr-dn	Write-done signal.
wr-ptr	Write pointer.
x,y,z	Lateral, vertical and spatial position of the AC value that is being processed.
xx	Current position of the data in lateral axis, that is being processed during current stride.
yy	Current position of the data in vertical axis, that is being processed during current stride.
z	Number of filter weights that can be stored in the kernel memory inside PE.
zz	Current position of the data in spatial axis, that is being processed during current stride.

Chapter 1

Introduction

1.1 Artificial Intelligence

Artificial intelligence (AI) is a transformative field of computer science that focuses on developing systems capable of simulating human intelligence. These systems are designed to perform cognitive tasks such as learning, reasoning, problem-solving, perception, decision-making, and language understanding. Over the past few decades, AI has progressed from theoretical concepts to real-world applications, shaping various industries and redefining technological possibilities [1].

The origins of AI can be traced back to the mid 20^{th} century, with early concepts emerging from philosophy, mathematics, and neuroscience. Alan Turing's foundational work on computation and intelligence laid the groundwork for the field, proposing the idea that machines could mimic human cognitive functions [2]. The development of symbolic AI in the 1950s and 1960s introduced rule-based systems designed to simulate logical reasoning [2]. However, these early AI systems were limited by their reliance on manually coded knowledge, which restricted their adaptability to complex real-world scenarios [3].

The evolution of AI has been heavily influenced by advances in computational power, data availability, and algorithmic improvements. The late 20^{th} century witnessed the rise of machine learning (ML), shifting the focus from hand-crafted rules to data-driven learning techniques [4]. This transition enabled AI systems to learn patterns from large datasets and improve performance over time without explicit programming. The advent of deep learn-

ing (DL) in the early 21^{st} century further revolutionized AI, enabling models with multiple layers of artificial neurons to achieve unprecedented accuracy for image and speech recognition [5].

In present time, AI has been deeply embedded in everyday life, powering applications such as virtual assistants, recommendation systems, autonomous vehicles, healthcare diagnostics and many more. The field continues to evolve with advancements in reinforcement learning, natural language processing, and computer vision [6].

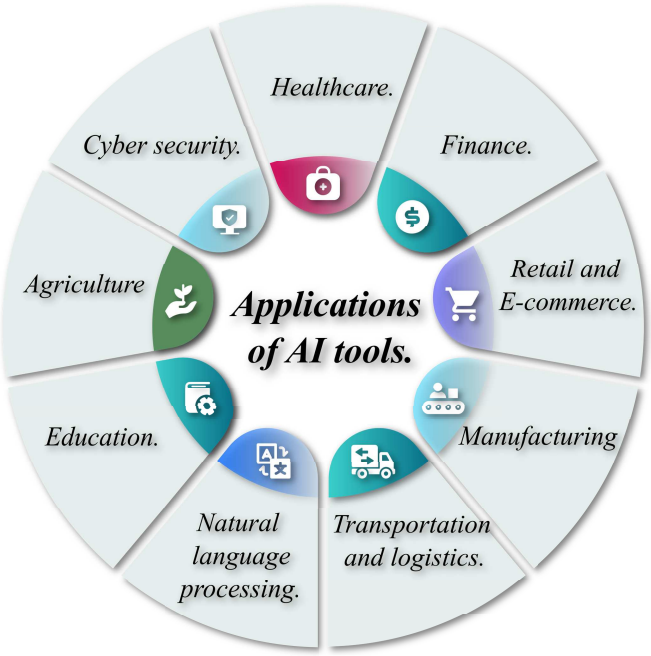


Fig. 1.1: An overview of used cases for AI in different applications [5] .

1.2 Applications and Impact of AI

AI has become a key driver of innovation, reshaping industries and everyday life by processing massive datasets, recognizing patterns, and supporting intelligent decision-making. Its applications span healthcare, finance, commerce, manufacturing, transportation, education, agriculture, and cybersecurity, where it continues to enhance efficiency, accuracy, and automation across diverse domains [6, 7].

In healthcare, AI enhances personalized medicine by analyzing genetic profiles and clinical data to tailor treatments for individual patients. Predictive analytics assist doctors in

identifying high-risk patients, monitoring vital signs, and detecting anomalies. It also supports medical imaging analysis, accelerates drug discovery, and enables remote patient monitoring, thereby improving clinical outcomes while reducing costs [8, 9].

In finance, AI does real-time fraud prevention by meticulously analyzing streams of transactional data to detect and block suspicious activities before they escalate. It also powers algorithmic trading, ensuring rapid order execution and the dynamic optimization of investment portfolios. In addition, AI can improve credit scoring, risk assessment, and personalized financial advisory services, making financial systems more secure and efficient [10, 11].

AI also plays a transformative role in retail and e-commerce, where it powers recommendation systems, demand forecasting, and inventory optimization. By predicting customer behavior and enabling dynamic pricing, it enhances supply chain efficiency while creating highly personalized shopping experiences [12, 13].

In manufacturing, AI enhances productivity through intelligent robotics, predictive maintenance, and adaptive process control. Automated quality inspection ensures consistency, while advanced decision-making systems reduce downtime and operational costs, making industries more flexible and resilient [5, 14].

Transportation and logistics are being reshaped by AI. Self-driving cars utilize AI-powered sensors, computer vision, and deep learning models to perceive their surroundings, make decisions, and navigate safely [15]. AI-based advanced driver-assistance systems (ADAS) improve road safety, while route optimization and intelligent traffic management enhance efficiency in logistics and urban mobility [16, 17].

Another major area is natural language processing (NLP), where AI enables real-time translation, chatbots, and sentiment analysis. These applications improve accessibility and customer service, while also allowing organizations to extract insights from vast collections of unstructured text [18].

Also, in education, AI can personalize learning experiences through adaptive learning platforms and intelligent tutoring systems that respond to student needs. Automated grading reduces routine workload for teachers, enabling them to focus on more impactful educational tasks [4].

AI is equally important in agriculture, where precision farming techniques rely on AI to monitor crop health, predict yields, and optimize the use of water and fertilizers. These innovations improve productivity and contribute to sustainable agricultural practices [19].

Another critical domain is cybersecurity, where AI enhances protection against cyber threats by detecting anomalies, predicting attack patterns, and automating incident responses. These systems provide stronger defense mechanisms for safeguarding sensitive data and critical infrastructures in real time [20].

Beyond these, AI is finding applications in areas such as surveillance, defense, creative industries, and many emerging fields that are still unfolding. The breadth of its impact illustrates how AI continues to reshape societies by enhancing decision-making, automating complex tasks, and unlocking new opportunities for innovation [1, 21].

Despite their diversity, these applications are ultimately grounded in a set of core AI components like *machine learning* (ML), *neural networks* (NN), *deep learning* (DL), *natural language processing* (NLP), *reinforcement learning (RL)*, etc., that collectively provide the foundation for intelligent behavior. A closer look at these key components will help clarify how AI systems achieve such versatility, which has been discussed in the following section.

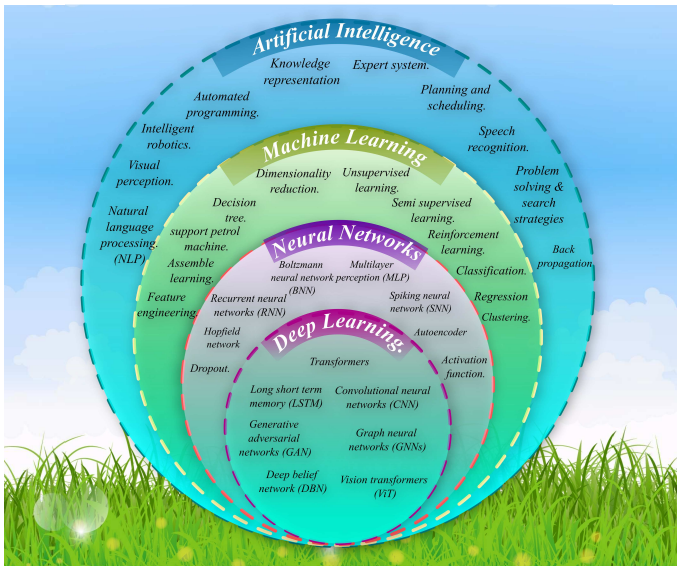


Fig. 1.2: Hierarchical relationship between artificial intelligence, machine learning, neural networks, and deep learning. Each inner circle represents a subset of the outer concept, with examples of models and methods associated at each level.

1.3 Key Components of AI

The relationship among AI, ML, NN, and DL can be better understood through a layered view, as illustrated in Fig. 1.2. This nested structure clarifies how different models and techniques are interrelated across levels of abstraction and complexity. AI is built on a set of interrelated components that collectively enable machines to learn, reason, perceive, and act in ways similar to human intelligence. Among these, ML forms the foundation, allowing systems to identify patterns and make predictions from data [4]. As illustrated in Fig. 1.2, ML encompasses various paradigms such as supervised, unsupervised, and reinforcement learning, each enabling automated knowledge extraction from different types of datasets.

DL, a specialized subset of ML, employs multi-layered neural networks to achieve superior performance in complex tasks such as image recognition, speech processing, and natural language understanding [3, 5]. At the core of DL are NNs, computational models inspired by the human brains interconnected neurons. NNs are particularly effective at learning non-linear relationships, with architectures ranging from feedforward multilayer perceptrons to recurrent, convolutional, and graph-based variants. Convolutional Neural Networks (CNNs), in particular, have revolutionized computer vision applications by automatically extracting spatial features from raw data, making them highly effective for image classification, object detection, and segmentation [22].

Beyond ML and DL, other AI components provide complementary capabilities. NLP enables machines to interpret and generate human language [18], while RL focuses on agents that learn optimal strategies by interacting with dynamic environments [21]. *Computer vision* (CV) interprets visual information for recognition and perception tasks, and *expert systems* encode domain-specific rules for automated decision-making [23].

As AI continues to advance, the demand for efficient computational resources grows, leading to the development of specialized hardware accelerators. The next sections will explore the role of hardware in AI, particularly in the optimization of deep learning workloads and the design of efficient CNN accelerators.

1.4 Motivation for CNN-Centric Hardware Acceleration

Within the diverse spectrum of AI techniques, several models have played central roles at different stages of development. Early approaches such as expert systems and rule-based reasoning demonstrated how symbolic logic could capture human expertise, yet they lacked scalability and adaptability in handling unstructured data [23]. Feedforward multilayer perceptrons improved upon this by learning patterns directly from data, but their fully connected nature led to a prohibitive number of parameters and limited capacity to exploit structural correlations.

Recurrent Neural Networks (RNNs) and their variants extended learning to sequential data, showing notable success in speech and language processing. However, their training suffers from vanishing gradients, long convergence times, and limited parallelization [3], constraining their scalability for modern large-scale applications. Similarly, the reinforcement learning has been proven to be powerful for decision-making tasks but is less effective in perception-heavy domains without integration with deep architectures.

By contrast, CNNs introduced two critical architectural advances: local connectivity and weight sharing, that drastically reduce parameter counts while preserving the ability to capture hierarchical features. This design enables CNNs to efficiently learn from spatial and temporal correlations in data, making them uniquely suited for image recognition, object detection, medical imaging, and other perception-driven applications [5, 22, 24]. Their layered feature hierarchy mirrors aspects of human perception, progressing from edges and textures to higher-level semantic features, which has driven their dominance in computer vision and beyond.

CNNs have also demonstrated versatility, extending beyond vision to speech recognition, natural language processing, and bioinformatics, underscoring their robustness across AI domains [3]. While newer architectures such as transformers are gaining prominence, CNNs remain indispensable due to their computational efficiency, scalability, and the maturity of the supporting ecosystem. For these reasons, CNNs are chosen as the focal point of this thesis, with a particular emphasis on designing hardware-efficient solutions to address their computational challenges.

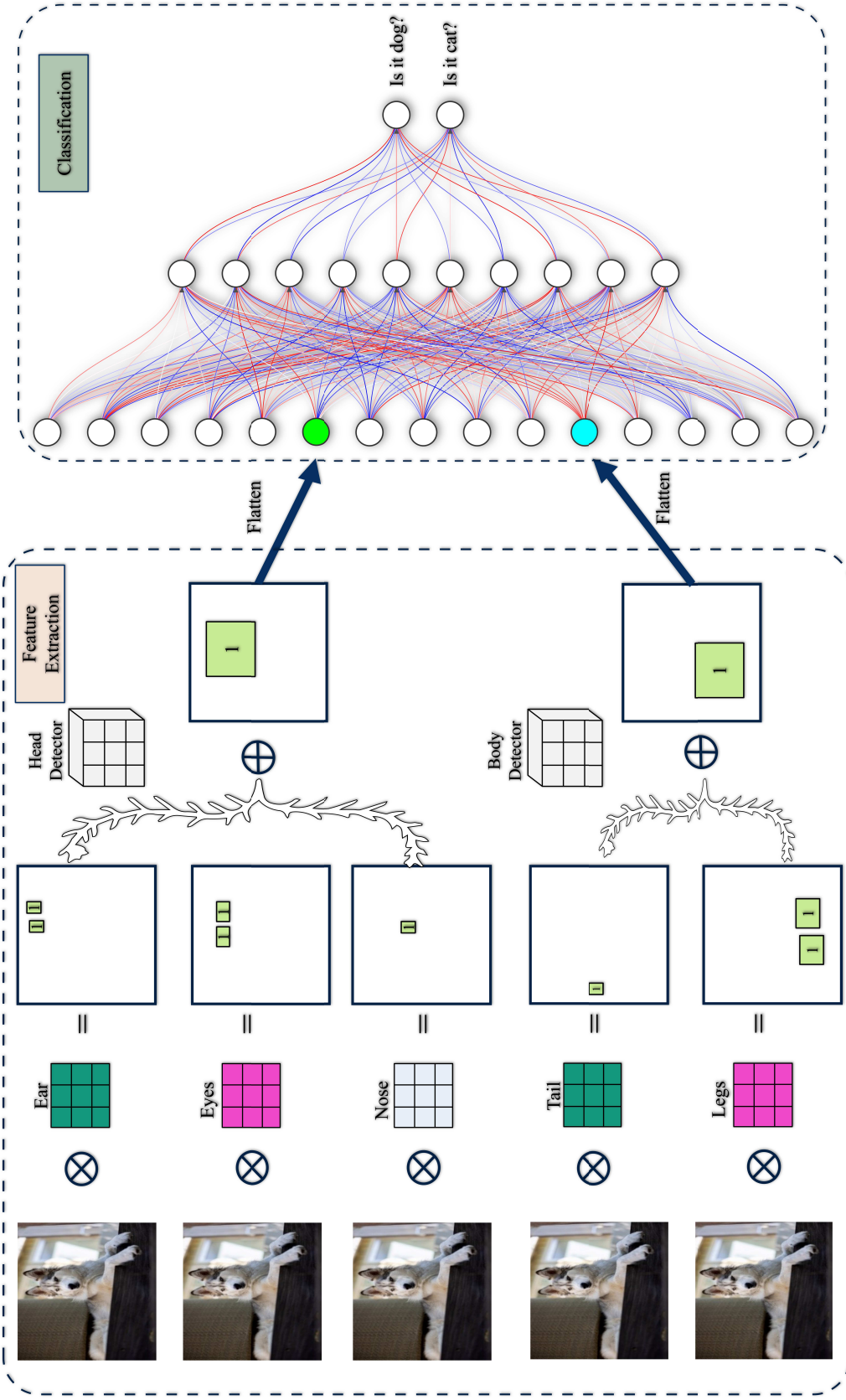


Fig. 1.3: Illustration of Human Object Recognition vs. CNN Processing.

1.5 Convolutional Neural Networks: From Intuition to Implementation

1.5.1 Human's Object Identification versus CNN Working Mechanism: An Analogy

To design an effective CNN accelerator, we first need to understand the way a CNN model works. Fig. 1.3 provides an analogy between the way humans identify objects and the functioning of a CNN. When humans identify an object, such as a dog, we rely on recognizing distinct features ears, eyes, nose, tail, and legs and their relative positions. If these features match our internal concept of a dog, we classify the object accordingly.

Now, imagine a system with specialized "filters" designed to detect each of these features. For example, one filter might look for ears, another for eyes, and another for the nose, and so on as shown in Fig. 1.3. Some filters might even focus on combinations of features like ears, eyes, and nose to detect the head while others could detect the body, such as the legs and tail. This hierarchical approach mirrors the following step by step workings of a CNN model.

1. **Feature Extraction through Filters (Convolutional Layers):**
    - CNNs apply small, learnable filters (kernels) across the image to detect low-level patterns like edges, textures, and simple shapes, similar to how we check for features like ears, eyes, and nose.
    - In the initial layers, filters capture basic visual elements without forming an understanding of the entire object.
2. **Building Higher-Level Representations (Deeper Convolutional Layers):**
    - When we combine features (ears, eyes, nose) to identify the head of a dog, deeper layers of the CNN combine low-level patterns to form more complex structures.
    - Intermediate layers might detect partial structures (like a dog's head or body), which later layers use to recognize the full object.

3. **Object Classification (Fully Connected Layers & Decision Making):**

    - Once the CNN layers have identified key features and their arrangements, the network assigns confidence scores to different possible objects.
    - Similar to how humans make a decision based on the overall arrangement of features, CNNs classify the object based on learned patterns.

This process closely mimics human analysis of visual information. However, unlike humans, CNNs learn which features to detect from large datasets, automatically discovering the most useful filters and patterns for distinguishing among different objects.

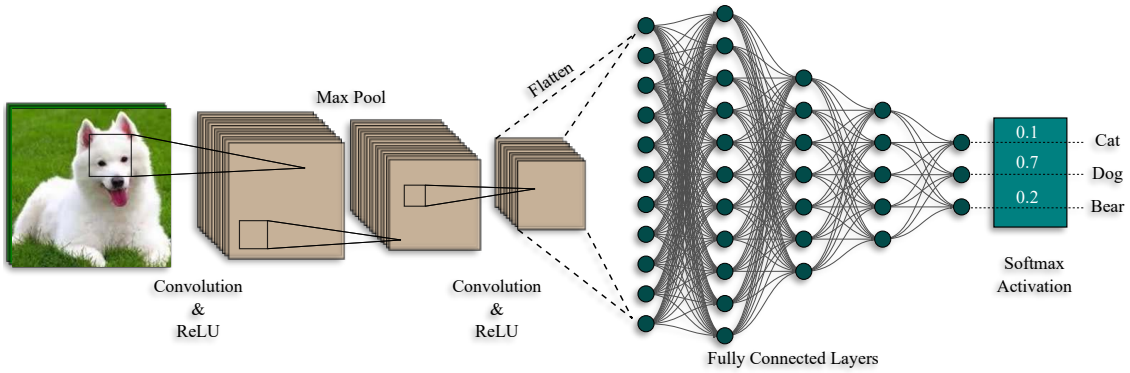


Fig. 1.4: Computations contributed by different kernel sizes in CNN models.

1.5.2 Understanding CNNs: A Hierarchical Approach to Image Recognition

A CNN processes images in a hierarchical fashion, progressively extracting features at various levels of abstraction. This approach is analogous to human object recognition, where individual features are first identified, then combined to form a complete understanding of the object. Fig. 1.4 provides a visual representation of one such CNN model.

The process starts at the input layer, where an image is represented as a three-dimensional tensor with dimensions $H_i \times W_i \times C_i$, where H_i, W_i, and C_i represent image-height, image-width and number of channels, respectively, three for an RGB image.

1.5.2.1 Feature Extraction: Convolutional Layers

CNNs utilize convolutional layers to extract spatial features. Small filters (kernels) of size $k \times k$ slide over the input image, computing dot products between filter values and local image regions. This operation generates feature maps, highlighting significant image structures. Early layers capture low-level features like edges and textures, while deeper layers extract higher-level representations, such as object parts (eyes, nose, ears for a dog).

In the convolution (*Conv*) layer of CNN, each element of the output feature map is computed as

$$AC_{x,y,z,\delta} = \sum_{a=1}^{\alpha} \sum_{b=1}^{\beta} \sum_{c=1}^{\gamma} \left(W_{a,b,c,\delta} \times I_{xx,yy,zz} \right) + B_{\delta}. \tag{1.1}$$

In the above expression, *AC, W, I*, and *B* denote output activation, filter weight, input feature-map, and bias values, respectively. Likewise, x, y, z and δ represent the positions of current value of output feature map over its four different dimensions. In addition, xx, yy, and zz are three dimensional positions of the current value of input feature map which is being processed. These values are computed as $xx = (s \times x + a)\,; yy = (s \times y + b)\,; zz = (s \times z + c)$ where s denotes the stride size. Furthermore, α, β, γ and δ are the sizes of four different dimensions of filters of a layer. Thus, $\alpha = \beta = 3$ for 3×3 convolution and $\alpha = \beta = 5$ for 5×5 convolution.

1.5.2.2 Non-Linearity: Activation Function

After convolution, an activation function, typically the rectified-linear-unit (*ReLU*), is applied to introduce non-linearity. *ReLU* is defined as:

$$AC_{x,y,z} = max\left\{0, \quad I_{x,y,z}\right\} \tag{1.2}$$

This non-linearity allows the network to learn complex patterns by discarding negative values and retaining positive feature responses.

On the other hand, some CNN models with lower bit precision [25] use a special type of *ReLU* called *ReLU6* wherein, network discards negative values and retaining positive

feature responses. However, if the numerical value of the positive feature responses is more than 6, they are clipped to 6. Operation of such *ReLU6* operation can be expresses as:

$$AC_{x,y,z} = min\{max\{0, I_{x,y,z}\}, 6\} \tag{1.3}$$

1.5.2.3 Dimensionality Reduction: Pooling Layers

To enhance computational efficiency and robustness, CNNs employ pooling layers. The most common method is *maxpool*, which selects the maximum value from non-overlapping regions of size $\alpha \times \beta$, where α and β are the dimensions (height and width, respectively) of the pool kernel. This reduces spatial dimensions while preserving the most important features, helping achieve translation invariance and enabling CNNs to recognize objects despite minor shifts or variations in the image. *Max-pooling* is mathematically expressed as:

$$AC_{x,y,z} = max\Big\{I(x : \alpha,\ \ y : \beta,\ \ z)\Big\} \tag{1.4}$$

where $max\Big\{I(x : \alpha,\ \ y : \beta,\ \ z)\Big\}$ denotes the extraction of largest element from the $\alpha \times \beta$ matrix window with an origin at (x, y) inside the z^{th} channel of I.

Some models [25], [26] also use average pooling (*avgpool*) wherein, the average of all elements of the $\alpha \times \beta$ region is taken, instead of taking only the maximum value. Such *avgpool* operation can be expressed as :

$$AC_{x,y,z} = \frac{\sum_{x}^{+\alpha} \sum_{y}^{+\beta} I_{x,y,z}}{\alpha \times \beta} \tag{1.5}$$

1.5.2.4 Higher-Level Feature Representation

As the network progresses through deeper convolutional and pooling layers, feature maps are further refined to detect more abstract structures, such as the overall shape of an object. This enables the network to develop a hierarchical understanding of the image, similar to how humans combine individual components to recognize an object.

1.5.2.5 Classification: Fully Connected Layers and Output

Once feature extraction is complete, the feature maps are flattened into a one-dimensional vector and passed to fully connected (FC) layers. For FC layers, each element of AC is the weighted sum of all elements of I and their corresponding biases. This operation is mathematically expressed as

$$AC_x = \sum_{n=1}^{N} I_n \times W_{n,x} + B_x. \tag{1.6}$$

These FC layers learn relationships between the extracted features, with the final output layer using a softmax activation function to produce a probability distribution across different object classes:

$$Pr_i = \frac{e^{CS_i}}{\sum_j^N e^{CS_j}} \tag{1.7}$$

Here, CS_i network's score for class i. AC from the last FC (FC_{last}) layer is given to softmax function as the class score (CS). Pr_i represents the network's score and probability for class i, respectively. The class with highest probability is then selected as the predicted label.

CNNs follow a structured approach for object recognition, starting with simple feature detection and progressing to complex pattern recognition. Such hierarchical method mirrors human perception, where objects are identified by combining distinct features. By automating the feature extraction and classification process, CNNs achieve profound accuracy in image recognition tasks.

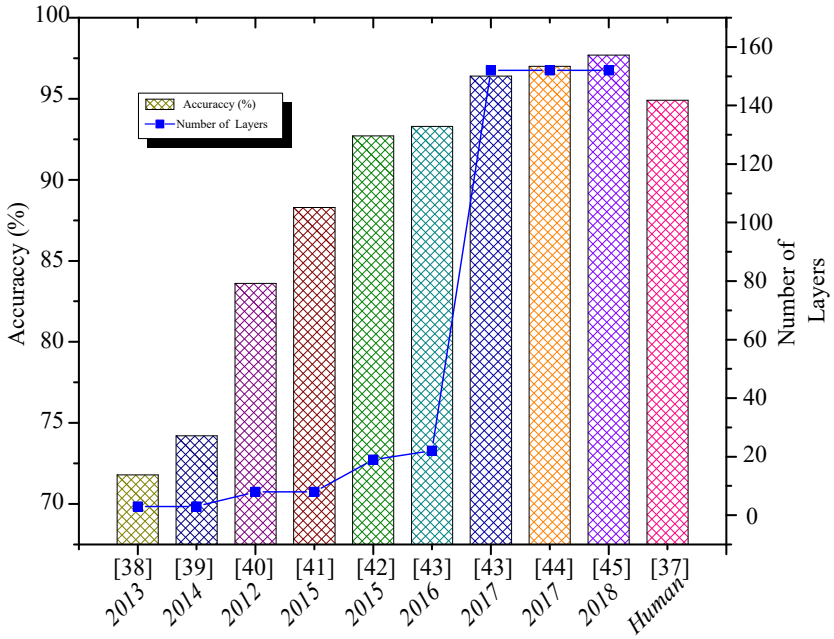


Fig. 1.5: ImageNet classification accuracy versus network depth for major CNN architectures.

1.6 Accuracy and Complexity Trade-offs in AI Models

In recent years, AI models especially deep-convolutional neural networks have demonstrated remarkable improvements in accuracy and generalization, making them highly suitable for the deployment of high-stakes domains such as healthcare, autonomous vehicles, and surveillance systems [5], [27]. These models have evolved from relatively shallow architectures, like AlexNet [28] in 2012-13, to extremely deep and sophisticated designs such as ResNet [29] and SENet [30], which incorporate over 150 layers to extract hierarchical features from data. As depicted in Fig. 1.5, CNN models have not only improved consistently in performance but, in some cases, have surpassed human-level accuracy in visual recognition tasks [31]. This trend underscores the increasing reliability of AI in perception tasks. However, it also reflects a growing complexity in model architecture, which presents challenges in terms of computational cost, energy efficiency, and interpretability. The trade-off between performance and complexity remains a central concern in the design and application of modern AI systems.

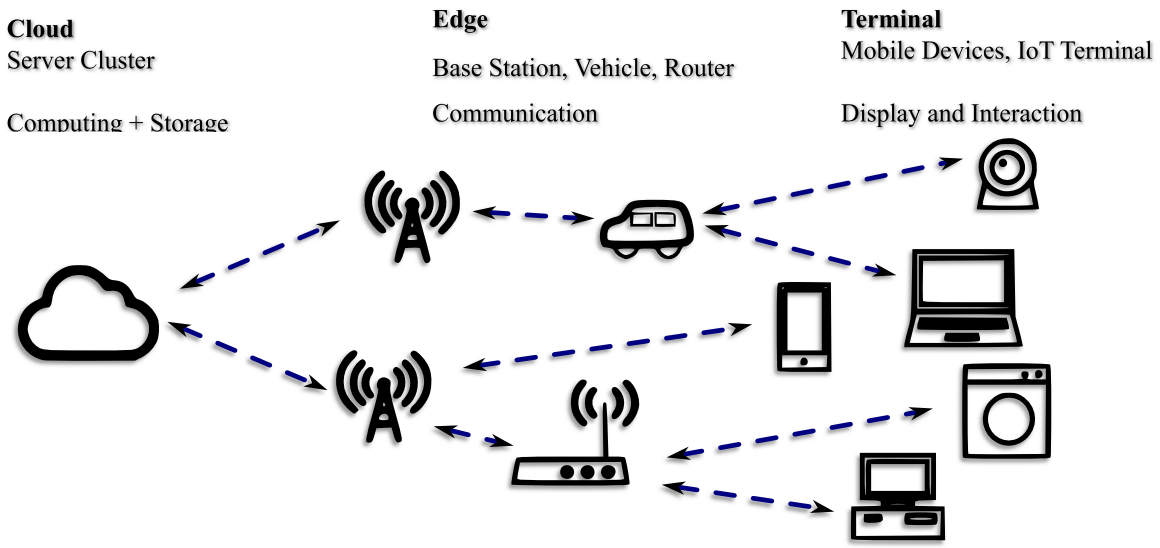


Fig. 1.6: An overview of a distributed deep learning framework showing the flow of data from terminal devices to the cloud, where computation-intensive tasks are executed.

With the increasing complexity, modern deep learning models, especially deep neural networks, have become increasingly computationally intensive, requiring substantial processing power and memory. These requirements often exceed the capabilities of mobile or embedded devices. To address this limitation, intelligent systems are now commonly deployed in a distributed architecture involving three main layers: terminals, edge, and cloud. User-end devices such as smartphones and IoT sensors (terminals) are primarily responsible for data collection and interface tasks. The data is then transmitted to more capable cloud servers via intermediate edge devices such as routers or base stations. In the cloud, large-scale computational resources perform the main inference and analysis operations. The results are subsequently returned to the terminal for real-time interaction. Fig. 1.6 shows an overview of such distributed deep learning framework depicting the flow of data from terminal devices (e.g., mobile phones, sensors) through edge infrastructure to the cloud, where computation-intensive tasks are executed. Such model enables services like voice assistants (e.g., Siri or Alexa) and smart applications (e.g., Google Lens) to operate efficiently despite hardware constraints at the user level. However, this distribution of computation introduces new challenges related to latency, bandwidth usage, and data privacy, which have become critical considerations in system design [36], [37], [38].

1.7 CNN Hardware Accelerators: Addressing Computational Challenges in AI

The rapid advancement of AI, particularly in deep learning, has led to an increasing demand for high-performance computation. CNNs are widely used in image processing, computer vision, and various other AI applications, require significant computational resources due to their reliance on large-scale matrix operations, convolutional computations, and iterative optimization techniques. General-purpose processors are often inefficient in handling these tasks due to their sequential processing nature and limited parallelism. While graphics processing units (GPUs) offer substantial parallelism, however, they face limitations in power efficiency and flexibility, making the need of dedicated AI accelerators more pressing [22, 39].

The inefficiencies of traditional architectures stem from their inability to efficiently execute the massive parallel computations required for CNN training and inference. Many AI workloads, particularly those deployed in real-time applications and edge devices, operate under strict power and thermal constraints. Conventional computing architectures struggle with balancing performance and energy efficiency, necessitating the development of specialized hardware solutions on application-specific integrated circuits (ASICs) and field-programmable gate arrays (FPGAs) platforms [40]. These accelerators improve performance by minimizing redundant computations, optimizing data movement, and leveraging hardware parallelism.

CNN accelerators are designed to optimize key operations commonly found in deep learning models:

- **Matrix Multiplication and Convolutions:** CNNs heavily rely on these operations, which can be significantly accelerated using dedicated hardware architectures [41].
- **Quantization and Sparsity Exploitation:** Specialized AI accelerators incorporate techniques such as reduced precision computation like 8 or 16 bit fixed point representation instead of 32 bit floating point representation of numbers, and sparsity-aware processing to enhance efficiency while maintaining accuracy [42].

- **Memory Hierarchy Optimization:** AI accelerators implement optimized memory hierarchies, including high-bandwidth on-chip caches and dataflow architectures, to minimize latency and improve computational throughput [43].

Additionally, real-time AI applications, such as autonomous vehicles, robotics, and medical diagnostics, demand ultra-low latency processing. Traditional processors struggle to meet these stringent timing constraints, making dedicated CNN hardware accelerators essential. These accelerators employ efficient pipeline architectures and memory optimizations to ensure high-speed computation [5].

Furthermore, scalability is another critical factor in AI acceleration. As deep learning models grow in complexity, conventional architectures face bottlenecks due to memory bandwidth limitations and inefficient resource utilization. AI hardware accelerators address these challenges by offering:

- **Scalable Architectures:** AI-optimized hardware, including tensor processing units (TPUs) and advanced FPGAs, enables efficient multi-chip scalability, allowing large-scale CNN training and inference [44].

- **Reconfigurability:** FPGAs provide adaptable hardware implementations, making it possible to support different CNN models and applications without requiring costly new chip fabrication [45].

On the other hand, the economic viability of large-scale AI deployment depends on cost-effective and energy-efficient hardware solutions. General-purpose computing clusters incur high operational costs due to inefficiencies in power consumption and hardware utilization. AI accelerators improve cost efficiency by enhancing processing throughput and reducing power consumption, making large-scale CNN applications more practical and accessible [46].

Given the growing complexity of CNN models and the increasing need for efficient, real-time processing, the development of specialized hardware accelerators is essential. By addressing the limitations of traditional processors, CNN accelerators significantly enhance computational efficiency, scalability, and power optimization. This motivation underpins the

necessity of designing CNN hardware accelerators to meet the ever-evolving demands of AI applications across cloud computing, edge devices, and real-time processing environments.

1.8 Literature Review and Research Gaps

1.8.1 Early Developments in CNN Acceleration

The evolution of CNN accelerators has been driven by the need to balance accuracy, throughput, and energy efficiency across diverse applications. Early efforts primarily relied on GPUs. While GPUs provided massive parallelism, they consumed hundreds of watts, which made them unsuitable for embedded and edge devices where energy and area are critical constraints [47,48]. Early CPU and GPU-based acceleration platforms demonstrated the potential of parallel architectures but highlighted the unsustainable overhead of general-purpose designs in power and latency-sensitive domains.

To address these issues, dedicated hardware accelerators such as DianNao [47] and Eyeriss [48] introduced domain-specific architectures. These designs exploited local data reuse, optimized on-chip buffers, and specialized interconnects, demonstrating that *data movement*, rather than arithmetic computation, dominates energy cost in CNN execution. Subsequent surveys and tutorials reinforced this finding, emphasizing the need for accelerator architectures that minimize memory traffic while sustaining high throughput. Several designs later built upon these foundations, employing systolic arrays, tiled matrix multiplication, and FPGA overlays to enhance scalability and throughput [39,49].

1.8.2 Adaptive Kernel Mapping

A recurring challenge in CNN accelerators lies in handling diverse convolutional kernels. While many CNN models rely heavily on 3×3 convolutions, kernels such as 1×1 (e.g., in GoogleNet and ResNet bottlenecks) and larger 5×5 kernels are also widely used. Fixed datapaths tuned only for 3×3 kernels often underutilize hardware resources when processing other sizes.

To mitigate this, several accelerators introduced flexible kernel mapping schemes. Zhang

et al. [49] demonstrated FPGA-based approaches for configurable convolution blocks, while Li et al. proposed SmartShard, a hardware-aware adaptive kernel mapping method that dynamically adjusts datapath utilization [50]. These designs improved utilization but typically incurred control and routing overheads, limiting scalability for deeper models. Thus, achieving both high utilization and efficient adaptability across varying kernels remains a partially unsolved problem.

1.8.3 Uninterrupted Processing and Dataflow Techniques

Memory latency continues to be a dominant bottleneck. Simple buffering schemes result in frequent stalls, reducing throughput. Double-buffering or ping-pong buffering techniques [51, 52] partially addressed this by overlapping computation and memory access. However, these still suffer from idle cycles when switching between contexts or loading new data.

More advanced solutions employ hierarchical buffers, random-access line buffers, or asynchronous data streaming [53]. While effective in convolution layers, many of these approaches do not extend to other operations such as pooling, normalization, and activation functions. Consequently, interruptions occur when transitioning between different stages of the model, undercutting the potential benefits of continuous pipelining.

1.8.4 Full-Model Data Reuse

Most accelerators emphasize data reuse within convolutional layers, particularly exploiting spatial and temporal reuse of activations and weights [48, 54]. While this substantially reduces memory traffic, other layers such as pooling, fully connected layers, and even nonlinear activation functions have received comparatively less attention.

Han et al. introduced compression techniques that reduced parameter storage through pruning, quantization, and Huffman coding [55], primarily targeting fully connected layers. Similarly, EIE [54] showed efficiency gains on compressed networks but did not address pooling or normalization layers. Holistic reuse across *all* CNN components remains rare. Without such strategies, redundant memory transfers persist, leading to suboptimal performance for complete inference pipelines.

1.8.5 Training Accelerators

While inference dominates current accelerator research, training CNNs is equally important, especially for enabling continuous learning at the edge. Training requires both forward and backward passes, weight updates, and storage of intermediate activations, which drastically increases memory and compute demand compared to inference.

Googles TPU [39] was initially optimized for inference, with training support added later through larger-scale datacenter deployments. FPGA-based frameworks such as Toolflow [56] and FlexFlow [57] demonstrated partial training capabilities but lacked fine-grained reuse strategies for activations and gradients. Some ASIC prototypes [58] explored reduced-precision training accelerators, highlighting mixed-precision arithmetic as a potential path to efficiency. Nevertheless, these approaches were either limited to small-scale models or focused only on convolutional layers.

A persistent limitation of most training accelerators is the under-utilization of reuse opportunities in training. Gradients, weights, and activations are often redundantly transferred between on-chip and off-chip memory, resulting in significant energy overheads. Moreover, most designs fail to unify training and inference efficiently, requiring separate architectures or modes of operation.

1.8.6 Research Gaps

Despite significant progress in CNN accelerator design, the reviewed literature suggests four critical gaps:

1. *Adaptive kernel mapping* that achieves high utilization across diverse convolution types without incurring excessive overhead.

2. *Uninterrupted processing* architectures that maintain continuous pipelining across all CNN Layers.

3. *Full-model local data reuse*, ensuring efficient reuse in pooling, normalization, activation, and fully connected layers in addition to convolution.

4. *Unified training and inference acceleration*, enabling efficient gradient computation, fine-grained reuse across passes, and scalable deployment for edge learning.

To maintain a focused discussion in this introductory chapter, only a consolidated overview of the major research gaps has been provided here. A more detailed literature review corresponding to each of these four gaps is presented in the subsequent chapters: Chapter 2 discusses adaptive kernel mapping, Chapter 3 examines uninterrupted processing techniques, Chapter 4 focuses on full-model data reuse, and Chapter 5 reviews works on unified training and inference acceleration. These chapter-specific reviews provide an in-depth background for the proposed methodologies.

1.9 Contributions of the Thesis

Motivated by the aforementioned gaps, this thesis presents hardware-efficient and high-throughput CNN architectures that unify inference and training within a single platform. By leveraging adaptive kernel mapping, uninterrupted processing, and extensive full-model data reuse, the proposed designs address the limitations of existing works and establish a scalable foundation for real-time, edge-deployed deep learning systems. The key contributions of this thesis are summarized as follows:

- A CNN hardware accelerator is designed to efficiently support multiple convolution filter sizes, optimizing resource utilization through an adaptive convolution mapping technique. Larger convolution kernels are mapped onto smaller processing units, significantly enhancing parallelism and computational efficiency. A novel *input feature buffer* architecture is introduced to maximize data reuse and reduce redundant memory access, improving computational efficiency. A multi-stage *parallel multiply-&-add unit* is developed to efficiently process various filter sizes, enhancing the hardware efficiency. The proposed memory hierarchy reduces energy consumption while maintaining high throughput, making the accelerator highly efficient for real-time AI applications.

- An uninterrupted processing technique has been introduced to eliminate memory stalls

and ensure continuous data flow, improving CNN processing speed and reducing latency bottlenecks. A *random-access line-buffer* architecture is developed to enhance local data reuse, avoiding unnecessary data shifts and significantly reducing redundant memory accesses. The proposed design is implemented on an FPGA-based CNN accelerator, showcasing high-speed performance and energy efficiency. The accelerator is tested on edge AI applications, proving its scalability across different CNN models and real-time deployment scenarios.

- A novel low-complexity classification unit is designed to optimize the computations of classification layer, significantly reducing computation cost while maintaining high classification accuracy. This optimization minimizes power consumption and computational overhead, making it adequate for resource-constrained edge devices requiring real-time decision-making.

- The accelerator design is optimized for maximum energy efficiency by exploiting local data reuse across all CNN operations, including convolution, ReLU activation, *maxpool*, and fully connected layers. By minimizing external memory access and reusing feature maps, weights, and activations, the accelerator achieves low power consumption while maintaining high processing throughput. The accelerator is synthesized on FPGA hardware-platform, demonstrating superior energy efficiency and hardware utilization. The accelerator is tested in edge AI applications, proving its adaptability to different CNN workloads.

- A unified CNN accelerator architecture is developed that supports both inference and training on the same hardware, significantly reducing design complexity and hardware costs. This proposed architecture exploits data reuse to compute weight and activation gradients efficiently, reducing computational overhead during backpropagation. It also maximizes data reuse during the forward pass and introduces an innovative computation-mapping strategy to handle large kernel sizes efficiently. The unified architecture is implemented on FPGA, demonstrating efficient real-time inference and training capabilities. The accelerator is validated in real-world AI applications, proving its scalability for on-device training and adaptive deep learning models.

These contributions collectively improve CNN accelerator performance, energy efficiency, and adaptability for both inference and training, making it well-suited for real-time, low-power, and scalable AI applications.

Chapter 2

Hardware-Efficient CNN Accelerator with Adaptive Convolution Mapping

2.1 Introduction

Superior accuracy of CNN models has spiked the development of modern AI applications, as discussed in Chapter 1. To improve the accuracy of CNN models, their sizes (i.e. depth and/or width) can be increased, provided large amount of labeled data is available for training such expanded-CNN models [59]. Otherwise, they become prone to overfitting, as large number of parameters downshifts the performance curve from the expected projection [59]. Furthermore, small increase in the number of filters between two consecutive convolution-layers in CNN model, quadratically increases the numbers of computations [59]. As the computational budget is always finite, an efficient distribution of computing resources is preferred to an indiscriminate increase of size, even when the main objective is to enhance the quality of performance [26]. Standard models like AlexNet [28] and VGG-16 [34] perform inference on RGB images using 61 million and 138 million parameters, respectively, consuming more than 500 MB of memory for storing only weights. Requirements of such sophisticated and massive computations that consume huge power for CNN models incur severe bottleneck in the widespread deployment of such applications on edge devices [60]. There are several reported works in the literature that aim to reduce the parameter size and computation complexity by pruning and compressing the parameters

and reducing the precision [42], [61] to achieve more hardware and energy efficiencies. The work presented in [62] has adopted approaches to increase the energy efficiency by optimizing the data movement and optimizing reuse of local data in the accelerator. However, as mentioned earlier, to improve the accuracy of CNN models, their computation complexities are increased. Hence, it necessitates the development of high-throughput and energy-efficient processing units for the implementation of such CNN applications. By using multiple processing elements (PEs) in parallel, CNN inference engines for these AI applications can achieve remarkable surge in speed [63]. CNN accelerator like [39, 40, 64–67], showed significant boost in the computation speed of CNN inference.

Chapter 1 provided an overview of the computational challenges associated with deep learning workloads and highlighted the importance of hardware accelerators in addressing these challenges. This chapter builds upon those discussions by proposing a high-performance CNN accelerator that optimally supports multiple convolution while maximizing hardware utilization. Highlights of our contributions in this chapter are as follows:

1. We identified inefficiencies in existing CNN accelerators, particularly in hardware utilization for different convolution filter sizes. To address this, we developed a CNN hardware accelerator that is optimized for multiple convolution-filter sizes, ensuring efficient hardware utilization and adaptability to evolving deep-learning architectures.

2. We proposed a novel data feeding and processing approach that mapped larger convolution filters (7×7 and 5×5) onto smaller 3×3 processing units, increasing parallelism and resource utilization. Additionally, we designed an efficient input feature buffer (*I Buffer*) architecture to maximize data reuse and minimize redundant memory access by introducing a new line-buffer mechanism.

3. To further improve efficiency, we implemented a multi-stage PMAU that effectively processed multiple filter sizes. Instead of using a single large convolution unit, we combined multiple smaller convolution units to enhance throughput.

4. For hardware validation, we synthesized the proposed accelerator on a Virtex-7 VC709 FPGA using 16-bit brain float (BF16) format and compared hardware efficiency with prior FPGA-based CNN accelerators.

Table 2.1: Brief Summary of Contemporary CNN models.

Models	AlexNet [28]	VGG16 [34]	ResNet -152 [29]	GoogleNet [26]	MobileNet [25]	EfficientNet-B7 [68]
Top 5 Error (%)	16.4	7.4	5.3	6.7	10.1	2.9
Conv Layers	5	16	59	21	28	812
Total Weights	61	138	25.5	6	4.2	66
Total MACs	724 M	15.5 G	39G	1.43 G	0.57 G	37

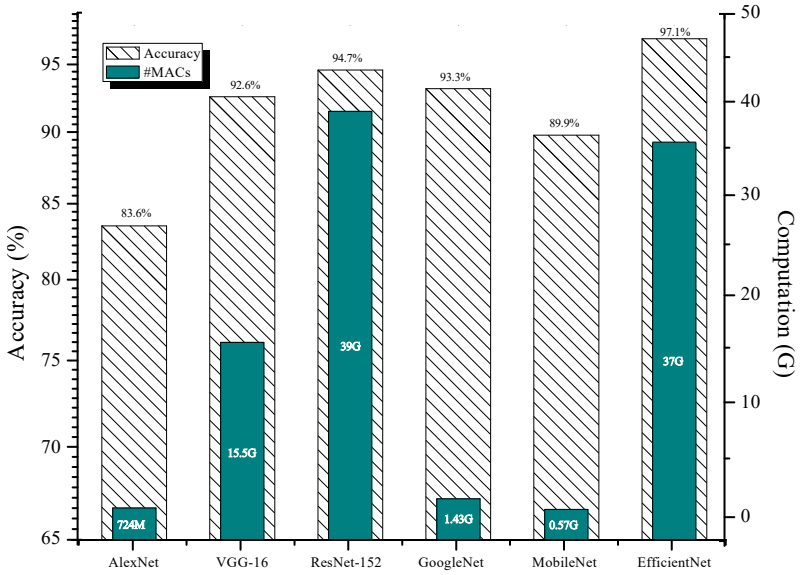


Fig. 2.1: Computation cost and accuracy of the state-of-the-art CNN models.

2.2 Hardware Inefficiency in Conventional CNN Accelerators

To achieve higher state-of-the-art accuracy, the number of CNN layers with different shapes and millions of weights are being prolifically adopted by wide range of contemporary applications [62]. Therefore, we analyzed some of the contemporary CNN models to understand the amount of computations that are contributed by different types of kernels in the CNN model. Table 2.1 summarizes the computation cost MACs (number of MAC operations), number of *conv* layers, number of filter weights, and the Top-5 error of some of the contemporary CNN models. Fig. 2.1 visualizes the computation costs and accuracies of such models. Summary of computation shares contributed by different filters in five most

Table 2.2: Contribution of Different Filters in Total Computation of Various CNN Models

	$C_{\alpha\beta}$ Values of Filters for Different CNN Models.				
CNN Model	1×1	3×3	5×5	7×7	11×11
AlexNet	-	42%	47%	-	11%
VGG-16	-	100%	-	-	-
GoogLeNet	22%	63%	8%	7%	-
MobileNet	10%	90%	-	-	-
EfficientNet	2.2%	63.9%	33.9	-	-

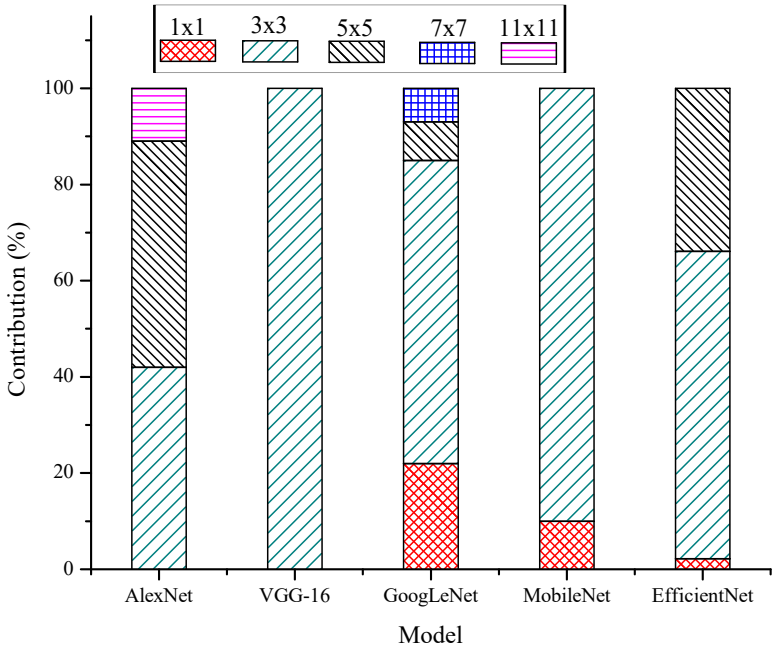


Fig. 2.2: Computations contributed by different kernel sizes in CNN models.

popular CNN-models have been presented in Table 2.2. Fig. 2.2 visualizes the same for ease fo understanding. Here, the computation share contributed by each filter of size $\alpha \times \beta$ is calculated as $C_{\alpha\beta} = \left(\mathcal{M}_{\alpha\beta}/\mathcal{M}_{tot}\right) \times 100$ % where $\mathcal{M}_{\alpha\beta}$ and $\mathcal{M}_{tot}$ represent number of MACs contributed by single filter (of size $\alpha \times \beta$) and all the filters, respectively. As presented in Table 2.2, majority of computations in the state-of-the-art CNN models are dominated by smaller filters, mostly 3×3 sized filter, rather than large filters with the size like 7×7 or 11×11.

As we know, the computation throughput (Θ_T) determines overall processing performance of a CNN accelerator, because Θ_T is directly proportional to the inference rate. The computation throughput is $\Theta_T \propto N_{PE} \times \Omega$ where N_{PE} is the number of PEs in CNN accelera-

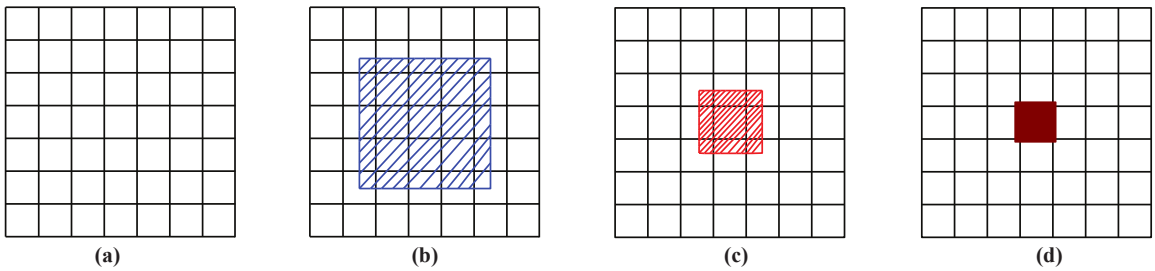


Fig. 2.3: Schematic representations of (a) a convolution processing unit for 7×7 kernels, (b) mapping 5×5 convolution, (b) mapping 3×3 convolution, and (c) mapping 1×1 convolution on 7×7 convolution processing units.

tor and Ω represents the PE efficiency which is given by $\Omega = N_{PE-util.}/N_{PE}$ such that $N_{PE-util.}$ denotes the number of PEs utilized. To achieve higher PE efficiency that maximizes the Θ_T value, PE array in the accelerator must be compatible with the filter shape of CNN model. On the other hand, conventional CNN accelerators do support only small set of filter sizes, for example, [40] supports only 3×3 kernel, [39], [66] support 1×1 and 3×3 kernel, and [65] and [67] do support 3×3 and 5×5 kernel. However, a CNN accelerator should support various filter sizes to efficiently handle different network architectures, maximize hardware utilization, adapt to evolving deep-learning models, and optimize computational performance across diverse convolutional operations [69]. The reason being, a CNN accelerator designed for a specific kernel size faces severe inefficiency when the computation of filter with different sizes are mapped on it. For better understanding, Fig. 2.3 shows a conventional way of processing smaller convolution tasks by fitting them as a part of larger convolution unit and filling rest of the portions with zeroes. Hence, such processing of 5×5, 3×3, and 1×1 convolutions using a convolution unit which is designed for 7×7 convolution has been illustrated in Fig. 2.3 (b), (c) and (d), respectively. It is evident from such processing that significant portions of the architecture are unutilized (around 49 %, 82%, and 98% of PE area would be wasted for processing 5×5, 3×3, and 1×1 convolutional task, respectively with 7×7 PE) that diminishes the benefit of hardware acceleration.

2.3 Proposed Architectures

To address the problem of inefficient mapping of convolution tasks to PE arrays of conventional CNN accelerator, this chapter presents a new hardware-architectural approach of efficiently mapping different types of filters in a PE array. Since this work focuses to develop a robust CNN accelerator that support as many filter sizes as possible, we have considered GoogleNet [26] as the target model. This is because, the GoogleNet requires four different types of kernels, as shown in Fig. 2.2. Therefore, the proposed architecture of hardware accelerator has been refereed as *'hardware accelerator for GoogLeNet CNN'*.

2.3.1 High-Level Architecture of Hardware Accelerator

In this chapter, we have proposed an approach for mapping larger convolutions (7×7 and 5×5) to smaller convolution (3×3) processing unit. It has been primarily carried out by using a new data feeding approach. To achieve maximum data reuse during every stride, this work presented a new *I Buffer* architecture. The proposed hardware accelerator architecture for GoogLeNet CNN has been shown in Fig. 2.4 (b). Its datapath includes a parallel multiply-&-add unit (PMAU), two local memories: (1) *I Buffer* that stores few lines of input feature-maps from one channel for 7×7 convolution and two or six different channels for 5×5 and 3×3 convolutions, respectively; (2) *Filter Buffer* for active filters (only one filter for 7×7 convolution and bank of 2 and 6 filters for 5×5 & 3×3 convolutions, respectively). A 1:2 de-multiplexer (DeMUX1) routes 128 bit (which is concatenated eight-parallel input data of 16 bit each) data from host controller to *I Buffers* or *Filter Buffers*, depending on the value of *I/W* control signal, as shown in Fig. 2.4 (b). It also shows that another 1-bit 1:2 de-multiplexer (DeMUX2) routes the write activation signal (wr_{act}) from the host controller to *I* or *Filter Buffers* for writing eight data in every clock cycle.

Here, *I Buffer* comprises of nine different memory-arrays/line-buffers and each of them stores upto 232 different elements of input feature matrix that enables *I buffer* to store nine rows of input feature matrix (*I*). As illustrated in Fig. 2.4 (b), host controller generates three configuration signals: *conv-type*, *I-shape*, and *s* to indicate convolution type, *I* size (height and width), and sizes of both horizontal & vertical strides, respectively. Therefore, after

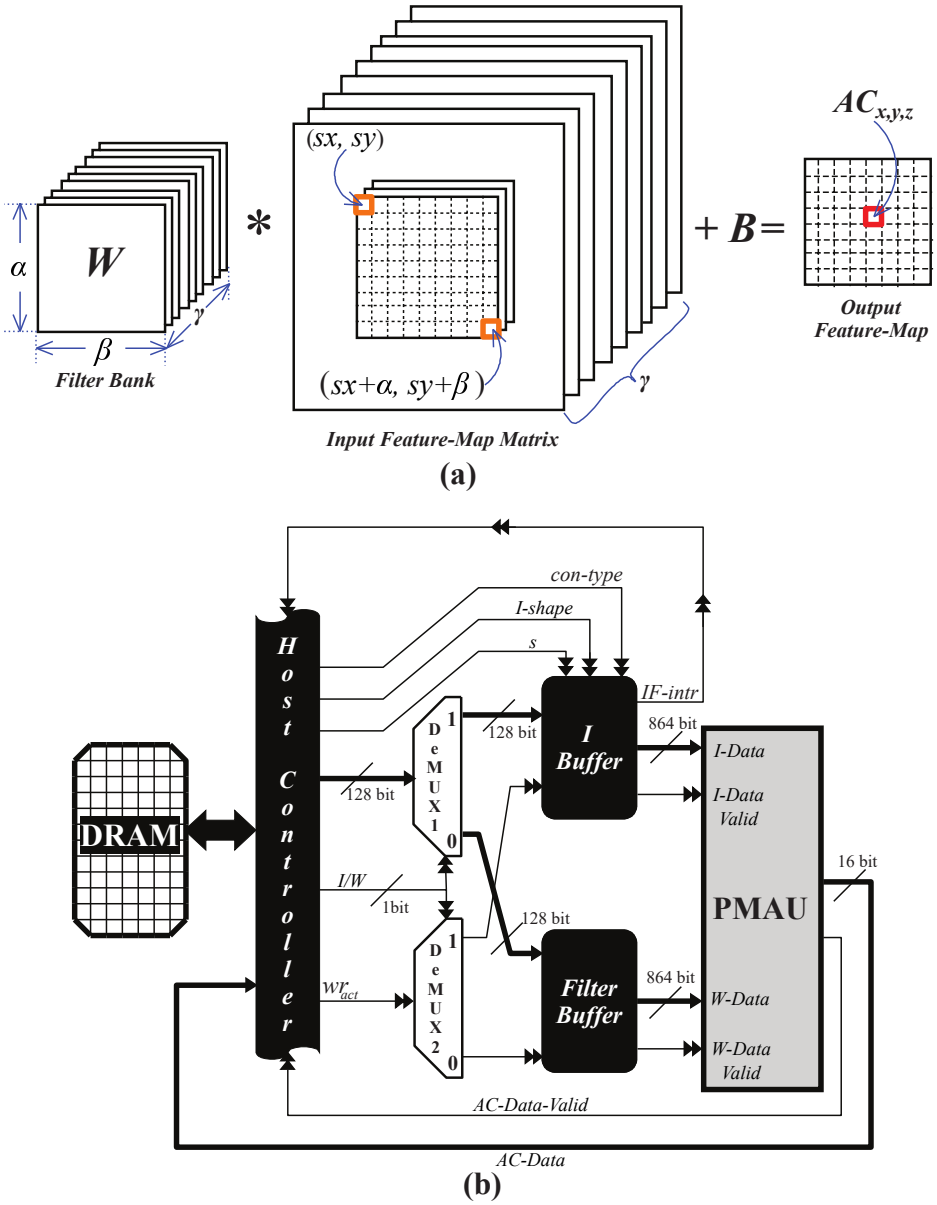


Fig. 2.4: (a) Schematic illustration of convolution process, and (b) proposed high-level of hardware accelerator to efficiently map different types of filters.

finishing horizontal stride on active rows (i.e. when reading finishes for first row), one vertical stride is carried out. Subsequently, *I Buffer* generates an interrupt (*IF-intr* signal) to host controller for sending new lines from *I* (i.e. one new-line for stride = 1 and two new-lines for stride = 2) and rest of the lines are reused from the *I Buffer* itself.

Subsequently, PMAU architecture of the proposed accelerator performs 54 parallel multiplications and adds their products to generate single output. As discussed earlier, the most commonly used CNN filters are 1×1, 3×3, 5×5, 7×7, and the proposed GoogLeNet accelerator requires all of them. Rather than implementing separate architectures for these convolutions or mapping 5×5 and 3×3 convolutions to single 7×7 convolution, we propose to split the larger convolutions to multiple 3×3 convolution units to perform convolutions and to sum-up their results to realize the outcome of larger convolution. Hence for 7×7, 5×5 and 3×3 convolutions, we perform convolutions to one (γ = 1), two (γ = 2) and three (γ = 3) channels, respectively, to enhance the hardware utilization. In the PMAU, there are nine multipliers and eight adders for single 3×3 convolution unit. Three such units are combined for realizing 5×5 convolution unit which is replicated twice to realize 7×7 convolution unit.

2.3.2 Proposed *I*-Buffer Architecture

Detailed VLSI architecture of *I Buffer* for the proposed hardware accelerator is shown in Fig. 2.5 where 128-bit data bus and three configuration signals (*conv-type*, *I-shape*, and *s*) are the primary inputs to each of the line-buffers. The write-activation (wr_{act}) signal from host controller is fed to *I*-valid port of *I Buffer*, indicating there are valid data for this *I Buffer* in the 128-bit input data-bus. Subsequently, the current write-line-select controller (CWC) activates write enable port for one of the line-buffers where data writing takes place. Thereafter, such line-buffer activates write-done (*wr-dn*) signal and transfers to CWC, which then deactivates the write enable (WE) port of current line-buffer and activates WE port of next line-buffer for writing the remaining data, as shown in Fig. 2.5. Thus, such process continues until the host controller deactivates wr_{act} signal.

In order to start reading the *I* data for feeding PMAU in the accelerator architecture, there must be sufficient number of such values stored in the *I Buffer*. Hence, *I* read-state controller (IRC) keeps track of the number-lines written in *I Buffer* and once *n* number

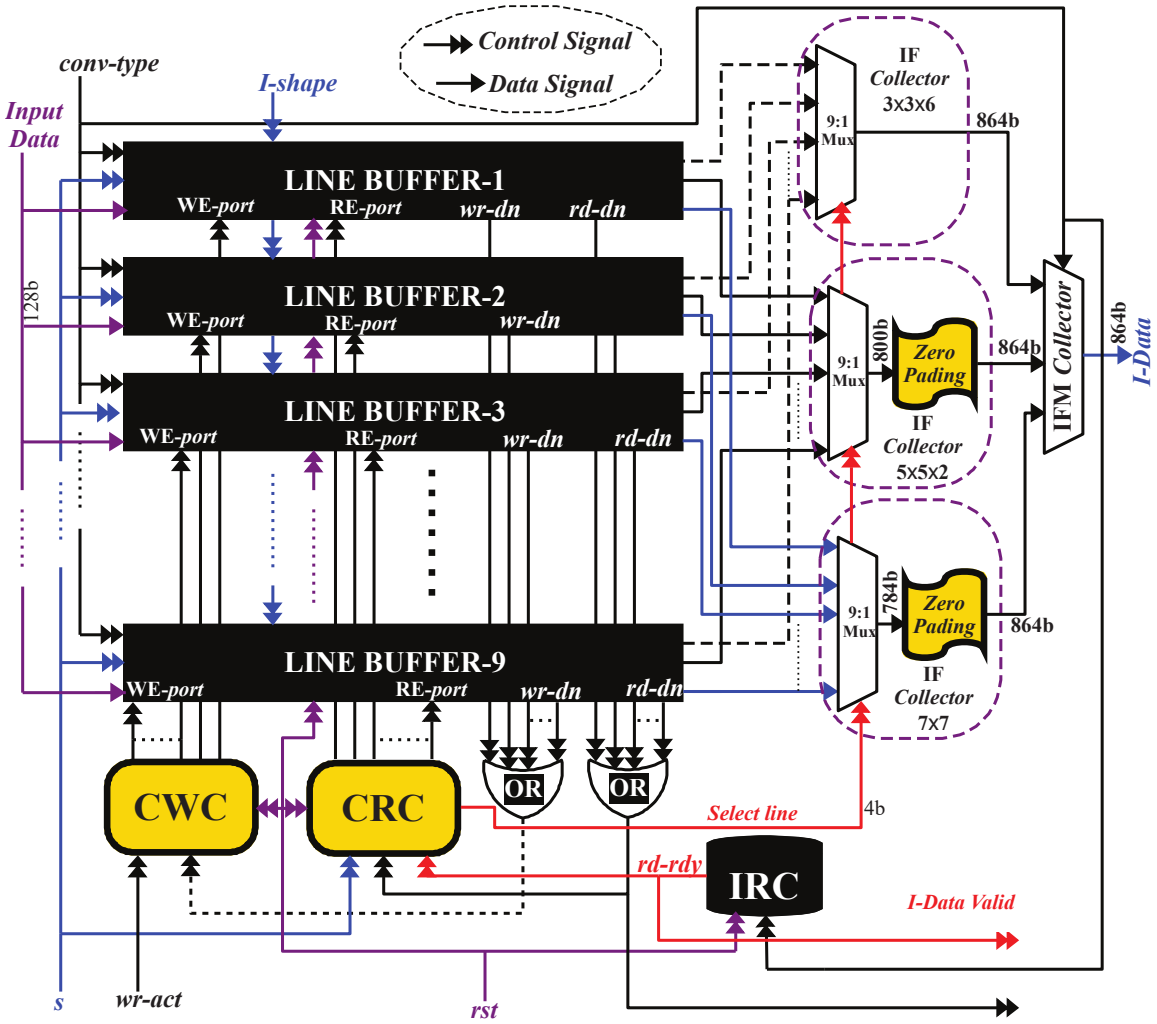


Fig. 2.5: Proposed VLSI architecture of *I Buffer* for the hardware accelerator.

of lines (n represents the size of convolution filter where n = 3 and 7 for 3×3 and 7×7 convolutions, respectively) are written in *I Buffer*, it activates read ready (*rd-rdy*) signal that initiates read operation, as shown in Fig. 2.5. It can be seen that *rd-rdy* signal has been connected to the current read-line-select controller (CRC) to indicate whether read operation should start or not. This *rd-rdy* signal is also tapped out from *I Buffer* architecture and it is connected to *I-Data Valid* port of PMAU for indicating that the valid I matrix is being fed to it. Furthermore, when *rd-rdy* signal from IRC sets high then CRC enables $n-1$ consecutive line-buffers. Thereby, once the read enable (RE) port of line-buffer is activated, all the line-buffers generate three different types of I-matrix based on the value of *conv-type* signal.

For 7×7 convolution, an active line-buffer generates seven I Data in each clock cycle. Similarly for 5×5 convolution, line-buffer generates total of 18 I Data per cycle. At every clock cycle, outputs from multiple (3, 5, and 7 lines for 3×3, 5×5, and 7×7 convolutions, respectively) such active line-buffers are aggregated to generate the final I-matrix. Therefore, our design uses three different collectors for accumulating 49, 50 and 54 different I data for 7×7, 5×5, and 3×3 convolutions, respectively. On the other side, PMAU architecture has been designed using 54 parallel multiply-and-add units which requires 54 different I data from *I Buffer*. Hence, we need to feed zero-values to some multipliers to avoid corrupting the computed *output feature-map* value that is also referred as activation value. To accomplish this, we incorporated I-collector in *I Buffer* architecture that collects the I data from preceding three I-collectors and adds extra zeroes to make them equivalent to 54 different I data for further processing in PMAU, as shown in Fig. 2.5. Based on the value of *conv-type* signal (indicating the convolution size), one of three busses from I collectors gets routed to the output bus (*I-Data*) of *I Buffer* that is fed to PMAU in our accelerator.

2.3.2.1 Micro-architecture of Line Buffer

The suggested micro-architecture of line-buffer that has been used in the design of *I Buffer* is presented in Fig. 2.6. Here, until the write-enable (WE) signal remains high, the 128-bit input data are routed to 232×16-bit memory array via de-multiplexer. Its each 128-bit output has been segregated into eight 16-bit values which are stored in memory array in every clock cycle. These values are written to the pointed location (i.e. represented as

wr-ptr) of the memory array that is generated by the write pointer, as shown in Fig. 2.6. Once the *wr-ptr* ≥ its maximum allowed value (i.e. decided by the size of original *I matrix*),

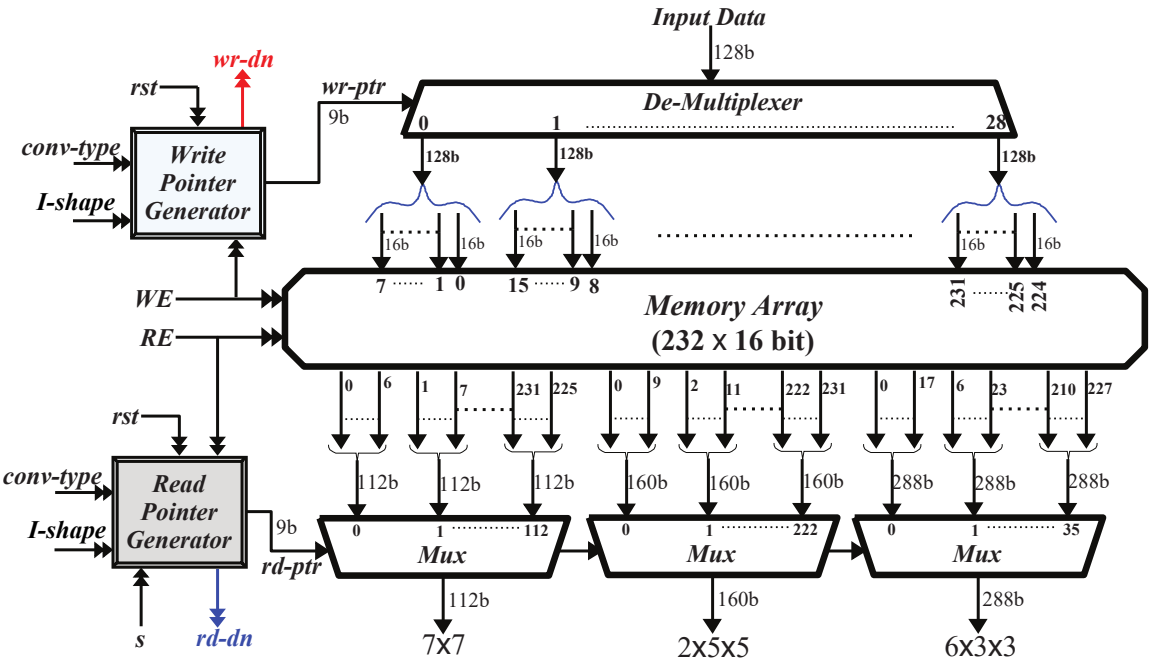


Fig. 2.6: Micro-architecture of line-buffer for the proposed *I Buffer* design.

wr-ptr resets and *wr-dn* signal is activated for one clock cycle that causes CWC of *I Buffer* to deactivate the WE signal of current line-buffer and the writing process switches to next line-buffer. On the other side, if read-enable (RE) signal of line-buffer is high then 7/10/18 parallel data (depending on the magnitude of *conv-type* signal) from the location pointed by *rd-ptr* are readout from the memory array in every clock cycle, as shown in Fig. 2.6. During this reading phase, *rd-ptr* value is incremented in every clock cycle by the stride size (i.e. 1 or 2). When *rd-ptr* value becomes ≥ its peak value (decided by the size of the original *I* matrix and the convolution size), the *rd-ptr* resets to zero and *rd-rdy* becomes high for single clock cycle that causes CRC in *I Buffer* to deactivate the RE signal of current line-buffer.

2.3.3 Suggested VLSI-Architectures for Filter Buffer and PMAU

The proposed filter-buffer architecture shown in Fig. 2.7 (a) comprises of a memory array for storing 56 different elements of filter matrix. Once the wr_{act} input-signal (from the host controller) is activated then the *Filter Buffer* stores eight filter-elements from 128

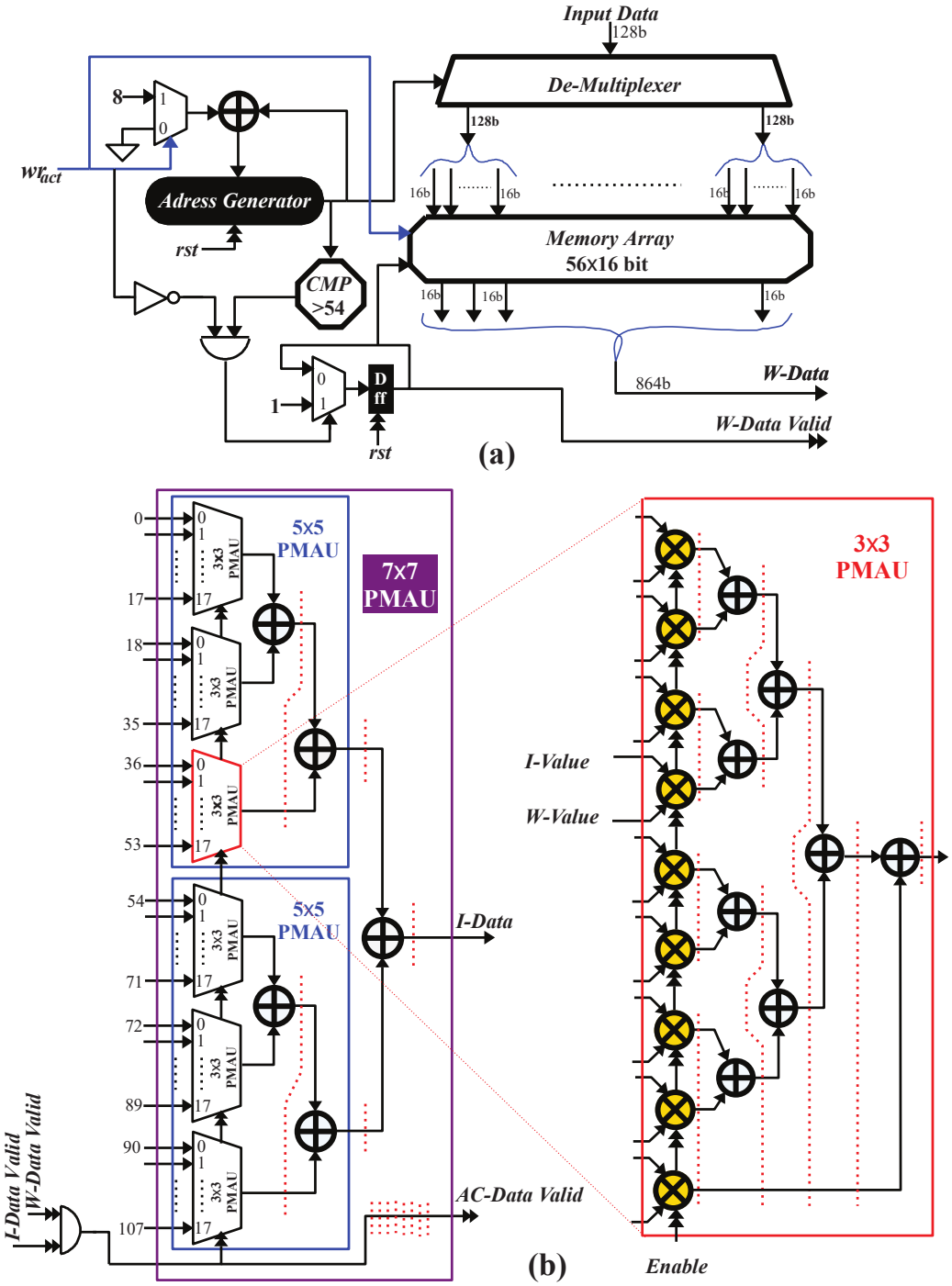


Fig. 2.7: Micro-architectures of (a) *Filter Buffer* and (b) PMAU for the proposed *I Buffer* design.

bit input-data bus in every clock cycle and *wr-ptr* value is incremented by eight. When the *Filter Buffer* stores sufficient data (i.e. 54 values), the read operation starts and the *W-Data Valid* signal is made high, indicating that the output *W-Data* is valid, as shown in Fig. 2.7 (a). Subsequently, the outputs from *I* and *Filter Buffers* are processed by PMAU in our accelerator and its VLSI architecture is shown in Fig. 2.7 (b). It shows that 7×7 PMAU has been designed using two 5×5 PMAUs where each consists of three 3×3 PMAUs (i.e. PMAU has total of six 3×3 PMAUs). Each 3×3 PMAU comprises of nine parallel multipliers and eight adders. The 3×3 PMAU processes nine *I-Data* and nine *W-Data* values from *I* and *Filter Buffers*, respectively. Similarly, 5×5 PMAU generates *AC-Data* value from 27 *I-Data* and 27 *W-Data* values. Furthermore, 7×7 PMAU generates *AC-Data* value from total 54 *I-Data* and 54 *W-Data* values. Therefore, while performing 3×3 convolution, six such 3×3 *I* and filter matrices from six different channels of *I* matrix and filter matrix are mapped to these six 3×3 PMAUs. For 5×5 convolution, 25 values of *I* and filter matrices from each of the two channels are mapped to these two 5×5 PMAUs, and two ports of each of them are fed with null values. Finally, while executing 7×7 convolution, 49 values of *I* and filter matrices are mapped to 49 ports of 7×7 PMAU and remaining five ports are fed with null values. For 1×1 convolution, it uses the same procedure of 3×3 convolution except, the *I-Data* and *W-Data* values are read using channel parallel approach, thus mapping 9 numbers of 1×1 convolution in a 3×3 PMAU. Hence, rest of the process for 3×3 and 1×1 convolutions are identical.

2.4 Experimental Results & Comparison

In this chapter, performance analysis of the proposed CNN accelerator has been carried out using Vivado 2018.2 RTL-simulator. To begin with, functional validation of our design is performed with the aid of simulation in MATLAB R2019a on a Windows operating-system based host computer with a Quad Core, 8 thread Intel i5-9300H CPU, 24 GB DDR4 RAM and NVIDIA GTX1650 GPU using FP32 representation. Subsequently, different types of convolution filter matrices and corresponding *I* matrices for different test images have been exported to two different groups of binary files. For the hardware validation, a Verilog

hardware-description-language (HDL) coded test-bench has been designed to act as an external host controller to the accelerator. As discussed earlier, the proposed CNN accelerator has been designed using BF16 format. These binary files for filters and I matrix have been read and converted to BF16 representation. The output waveforms of accelerator are validated from the simulation and at the same time, output values have been stored back to another binary file for analyzing the result in MATLAB.

Table 2.3: Comparison with prior works.

	[70]	[71]	[72]	[73]	**Proposed Implementations**			
FPGA	Kintex-7	ZC706	VC709	VC707	**Kintex-7**	**ZC706**	**VC709**	**VC707**
Precision (bit)	FP8	FP16	FP8	1 bit	**BF16**	**BF16**	**BF16**	**BF16**
f_{clk} (MHz)	200	150	200	200	**200**	**150**	**200**	**200**
Memory Utilization (%)	37	89	60	80.8	**0.44**	**0.41**	**0.21**	**0.30**
LUTs Utilization (%)	46.5	84	78	76.4	**6.71**	**6.26**	**3.16**	**4.51**
DSP Utilization (%).	61.4	8	80	84	**0**	**0**	**0**	**0**
Throughput (Θ_T)	342.18✠	137✠	137✠	2100✠	**21.6♣**	**16.2♣**	**21.6♣**	**21.6♣**
N_{PE}	1024	780	5754	9216	**54**	**54**	**54**	**54**
η_{PE} (MOPS/PE)	334.2	175.5	313.8	227.83	**400**	**300**	**400**	**400**
Accuracy Loss** (%)	3–6	0.3–0.5	3–6	12–30	**≃0**	**≃0**	**≃0**	**≃0**

♣ : GFLOPS, ✠ : GOPS, ** : Compared to FP32 implementation

The proposed accelerator has been synthesized and post-route implemented on the Virtex-7 VC709 FPGA and the results are presented in Table. 2.3. It has also been implemented on FPGA board used by compared works, and the results are presented in Table. 2.3. It shows the comparison of our accelerator results with the reported implementations of different approaches which are used for improving the efficiency of hardware utilization for CNN accelerators. Since our work focuses towards area-efficient design that uses limited hardware resources of only 54 multipliers, the entire accelerator design has been accommodated within 13.7k LUTs. It delivers lesser throughput compared to the reported designs in Table. 2.3. However, throughput density of our design that is calculated as $\eta_{PE} = \Theta_T/N_{PE}$ is better compared to reported approaches. Where, Θ_T, N_{PE}, and f_{clk} stand for the computation throughput, number of PE, and the operating clock frequency, respectively.

The improved results of the proposed design can be intuitively explained as follows. The architecture employs adaptive convolution mapping, which ensures efficient utilization of processing elements across varying kernel sizes. Earlier designs often left hardware underutilized when handling non-standard kernels, leading to wasted resources. Moreover, it

uses BFP16 format which maintains the full dynamic range of FP32 and has been proven to achieve accuracy comparable to FP32 models. Such innovations enable the architecture to achieve superior throughput density while maintaining FP32-level accuracy, outperforming prior accelerators that relied on narrower precisions at the cost of accuracy loss. Nevertheless, if the area budget of the design is higher then the proposed accelerator can be scaled up to enhance the throughput.

2.5 Summary

This chapter presented a versatile CNN accelerator designed to efficiently process four fundamental convolution tasks: 1×1, 3×3, 5×5, and 7×7 with optimized hardware resource utilization. The proposed architecture introduced a novel mapping approach where a 5×5 convolution was executed using three 3×3 convolution units, and a 7×7 convolution was implemented using two 5×5 processing units (equivalent to six 3×3 units). Moreover, nine 1×1 was implemented using one 3×3 convolution units This hierarchical decomposition enhanced the computational efficiency and flexibility.

To further optimize performance, we developed an efficient *I* and *Filter Buffer* architectures that maximized local data reuse, reducing memory access overhead. The accelerator design was implemented on the Virtex-7 VC709 FPGA board using a 16-bit brain-float (BF16) representation, consuming only 13.7k LUTs while achieving a throughput of 21.6 GFLOPS at 200 MHz. The proposed architecture demonstrates a balance between computational efficiency, and resource utilization, making it well-suited for edge computing applications requiring efficient CNN processing.

However, as mentioned in Section-2.4, our proposed accelerator delivered lesser throughput compared to the reported designs. Therefore, following chapter of this thesis will emphasize on improving the throughput of CNN accelerator with the help of efficient uninterrupted processing technique and suitable hardware scaling approach.

Chapter 3

Design of High-Throughput and Energy-Efficient CNN Accelerator: Techniques and Architectures

3.1 Introduction

As discussed extensively in chapter 1 and chapter 2, CNN has steered many of the modern AI applications. Superior performance of CNN is calibrated based on the way of recognizing the text or object or audio-key-frame in the input image or audio data by searching for the presence of millions to several billions of trained feature parameters [74]. The CNN carries out such feature search by sweeping and performing arithmetic operations (like multiplication and addition) with different filter weights that are associated with various features across the input data [75]. A detailed discussion on the working of CNN for image classification, and an analogy with human's object classification has been presented in Chapter 1. Nonetheless, the superior recognition accuracy of CNN can be achieved at the cost of extremely high computational-complexity and massive data movements, incurring performance degradation and surge in energy consumption, respectively [74, 75]. Such adverse consequences refrain CNN from its efficient hardware implementation for battery operated devices which are extensively used in contemporary edge applications like internet of thing, autonomous electric vehicle, and implanted biomedical devices [76]. Furthermore,

such contemporary applications demand higher computation throughput at lower power consumption [75]. Based on aforementioned discussion, substantial throughput and profound energy efficiency have become key requirements for the hardware implementation of CNN accelerators [77], [78]. On the other hand, when power hungry engine like graphic processing unit (GPU) is used as a core hardware for CNN processing [74], attempts are being made to accelerate CNN in an energy efficient way [79]. Similarly, dedicated parallel-processing platforms like mobile GPU [80], tensor processing unit (TPU) [81], field programmable gate arrays (FPGAs) [76, 79, 82, 83], application specific integrated circuit (ASIC) [62], and in-memory computation (iMC) [84] are used to efficiently process the computations in CNN. Note that GPUs and TPUs consume hundreds of watts which make them unfit for any edge applications. Subsequently, iMC and ASIC deliver highest energy efficiency; however, they have poor compatibility and scalability with the rapidly evolving CNN models due to the lack of re-programmability. In contrast, FPGA implementation of CNN accelerator offers re-programmability, and if its hardware architecture is well designed then it delivers adequate energy efficiency [75]. On the one side, GPU, TPU, ASIC and iMC implementations refrain from as a stand-alone system for any application, as they are instead required to be interfaced with an external computer/controller. On the other side, contemporary FPGA boards with on-board computer and logic elements (like Zynq-7000 SoC and Zynq Ultrascale+ MPSoC series of FPGA boards) can be implemented as complete stand-alone systems for various contemporary applications.

Energy efficiency of high-throughput parallel-computing CNN accelerator can be enhanced by extensively re-using the local data and minimizing the expensive read/write operations to and from the external memory. In addition, low precision computation can also be used to significantly increase the energy efficiency. In the reported work from [62], row-stationary data flow based architecture of CNN accelerator delivers promising energy efficiency by reducing the amount of off-chip memory access that requires a large number of shift registers to aid the local data re-use. However, such design is not a convenient choice for FPGA based applications [75]. In [85], Nguyen et al. could achieve higher energy efficiency by using lower precision at the cost of degradation in accuracy. Later in [86], authors have improved such accuracy by using the layer specific optimization for mixed data flow,

using the multiple precision. Similarly, [76] presents software-based reconfigurability by using dual-loop arithmetic module at the cost of throughput and energy efficiency, due to un-optimized data movement. In [75], the energy efficiency has been enhanced by reducing the off-chip memory access with the aid of kernel partitioning technique. However, it requires extra pre and post processings on the input and the output data. Furthermore, despite the presence of parallel computing, real world yield in computational throughput of most CNN accelerators falls below the theoretically estimated throughput due to under utilization of available resources and underlying problem of interrupted data supply [74]. Recent contribution from [87] showed that, in the worst case scenarios, a state-of-the-art complex model like GoogleNet utilizes only 6.6% of processing elements in an accelerator. In Chapter 2, a hardware-efficient CNN accelerator was introduced with an adaptive convolution mapping technique and an optimized buffer hierarchy. While these optimizations improved computational efficiency; however, performance bottlenecks still arise due to memory stalls and inefficient data movement, limiting overall throughput and energy efficiency. Hence, further enhancements are required to address memory stalls and inefficient data movement.

Therefore to circumvents the aforementioned issues and challenges, this chapter presents an energy as well as hardware efficient, high throughput, and reconfigurable FPGA-based CNN accelerator for object recognition application.

Our main contributions in this chapter are as follows.

- A new uninterrupted processing technique has been proposed to reduce the latency of CNN accelerator that enhances its throughput and energy efficiency.
- We suggest a new data-reusability process to achieve maximum reuse of local data without data shifting in CNN accelerator in order to suppress the data movement that eventually reduces its energy consumption.
- Furthermore, this chapter presented a new random-access line-buffer (RALB)-based hardware architecture of kernel processing unit (KPU) for CNN accelerator to realize the above proposed techniques.
- Following that, the proposed CNN accelerator has been hardware implemented in

various FPGA platforms and their results are compared with the state-of-the-art implementations.

- Eventually, hardware prototype of our CNN accelerator in Zynq-UltraScale+ MPSoC-ZCU102 FPGA-board has been functionally validated in real-world test-setup that classifies object from an input image with the aid of GoogLeNet CNN model.

This chapter has been organized as follows. In section 3.2, overall system level understanding of our work has been presented, along with brief mathematical background. It also comprehensively discusses the research challenges that are addressed in this chapter. Furthermore, section 3.3 presents the proposed technique and new VLSI architectures for the CNN accelerator. Following that, implementation results, discussion, comparison and validation are included in section 3.4. Eventually, section 3.5 summarizes this chapter.

3.2 Prerequisite and Research Challanges

3.2.1 System Model

An overview of the hardware-based system for an object detection application has been schematically shown in Fig. 3.1. It comprises of four major hardware components: (1) camera, (2) FPGA board (Zynq MPSoC evaluation-board), (3) external memory (secure digital (SD) card), and (4) display. Here, FPGA board consists of on-board computer that can run software programs, written in high-level language. Specifically, it has an ARM processor with six cores, DDR4 random-access-memory (RAM); and various peripheral controllers like USB controller (that interfaces camera with FPGA board), external SD-card controller, and display controller, as shown in Fig. 3.1. It also shows that the CNN accelerator is implemented using configurable logic blocks, multipliers, embedded block-RAMs of core FPGA chip on the board. Further, SD card contains the pre-trained model parameters and additional software programs. As illustrated in Fig. 3.1, the CNN accelerator has been designed using a global buffer cum control unit (GBCU) and a kernel processing unit (KPU). Here, GBCU manages the data flow between the on-board computer and KPU as well as manages the operation of KPU based on configuration signals from the on-board

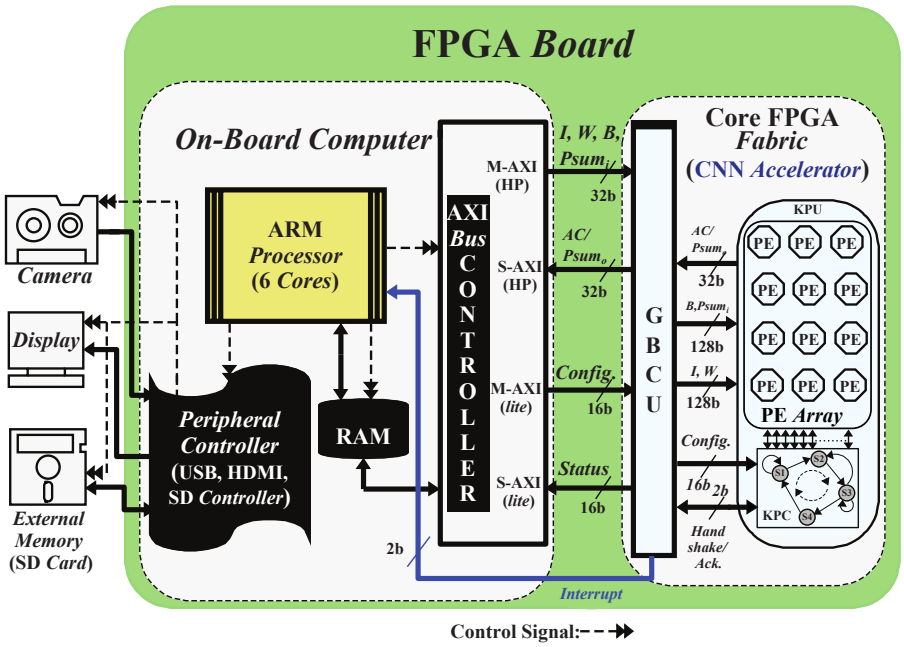


Fig. 3.1: Schematic representation of the hardware level system using FPGA board and other supporting peripherals.

computer. KPU is primarily made of an array of processing elements (PEs). Each of these PEs comprises of a multiply-&-accumulate (MAC) unit and a comparator unit. A CNN accelerator may have more than one KPU. Thus, the operation of such KPU is also locally managed by one kernel processing controller (KPC). It manages the operations of PE array and accordingly moderates the data flow between GBCU and KPU in the proposed CNN accelerator. Its operation is iteratively carried out via closed loop data-flow between KPC and PE-array. In every iteration, PE array receives the input feature matrix (I) and other parameters like filter weights (W) as well as biases (B) from KPC. Subsequently, PE array produces the output feature matrix (AC) that is passed back to KPC. In such iterative process, AC from one operation is re-used as I for the computation in subsequent layer. This process continues until the operations for all the layers are accomplished, as discussed further in section 3.2.1. As shown in Fig. 3.1, the CNN accelerator has been interfaced with on-board computer via high-performance AXI buses. Here, I for the first layer of CNN is the input image, and AC for the last layer of CNN is the computed probability of each class in the model. To begin with object detection process, when the system initially boots up, the on-board computer of FPGA transfers the model information along with trained W and

b parameters to configure the CNN accelerator. Thereafter, once the accelerator is fully configured with the model specification, on-board computer begins the processing of input images from external source (i.e. camera) and starts feeding/off-loading them to the CNN accelerator. Subsequently, it performs a number of operations on the images for all *Conv* and *FC* layers to search for different class features. Thus, it classifies the object that is present in the input image and computes its probability.

3.2.2 Research Challenges

3.2.2.1 Energy-Efficient Data Flow

The number of operations required by equations (1.1)−(1.7), in their corresponding hardware architectures, vary over a wide range that proliferates the computational complexity. From an implementation aspect, CNN accelerator must process such computationally complex operations in high speed and energy efficient manners by mapping them over a large array of PEs. Thus, it exploits high degree of parallelism and extensive reusability of local data within the periphery of PE array to achieve higher computation throughput and lower energy consumption, respectively. Both these performance factors are primarily dependent on the availability of data locally within the array of PEs. In [62], Chen et al. have achieved high energy efficiency from their implemented architecture with the aid of row stationary approach. Here, *W*, *I*, and *Psums* are reused in horizontal, diagonal, and vertical directions, respectively, in the PE array of CNN accelerator. It showed that the row stationary approach achieves better energy efficiency in comparison to the weight or output stationary approach [88, 89]. However, the row stationary data-flow requires continuous shifting of data over a large number of shift registers in all directions. The work reported in [75] demonstrated that the design with such a large number of shift registers is infeasible to be implemented on the FPGA platform. Thereby, [75] proposed a kernel partition technique that down-scaled the number of memory accesses to achieve adequate energy efficiency in FPGA-based re-configurable architecture, at the cost of highly complex pre and post operations. Furthermore, [86] showed that by proper optimization of data precision over different layers, all weights of CNN model can be stored in the BRAM of FPGA to avoid of-chip

DRAM access for an better energy efficiency. However, it requires to redesign the hardware architectures from extremely low level for each of the different CNN models and it must retrain the model to preserve accuracy. Therefore, our work presents RALB-based modified row-stationary data flow where both I and W values are maximally reused without shifting them from one PE to another PE that reduces the latency as well as conserves the energy. Hence, detailed working of the RALB based data flow has been clearly discussed in section 3.3.2.2.

3.2.2.2 Throughput Reduction due to Interruption

To directly map (1.1)−(1.7) equations on hardware, it requires enormous size of PE array in CNN accelerator. However, such mapping is not feasible due to various design-constraint limitations like memory bandwidth, area and power budgets. Therefore, computations in the CNN accelerator hardware takes place in phased manner over a number of processing iterations [62, 74]. During each iteration, the PE array of CNN accelerator computes partial sum (*Psum*) which is a portion of the output feature map that is later combined with other partial sums to generate final output feature map AC for equations (1.1)−(1.7) based on the configuration information from KPC, as shown in Fig. 3.1. In energy efficient data flow based conventional CNN accelerator like [62, 88, 89], between every two consecutive-iterations when the data is being fetched from the KPC to the local storage inside each PE, the PE array cannot operate and remains idle that consequents in frequent throttling. As a result, it increases the latency, and eventually degrades the achievable throughput of CNN accelerator. There have been various attempts to reduce the latency, for example, [90] uses pin-pong approach to reduce the latency. However, it does not support local reuse of data, incurring higher memory bandwidth. Consequently, result it is unable to completely minimize the idle time for small-sized filters due to bandwidth limitations. Furthermore, [91] combined a fine-grained column-based pipeline approach with ping-pong architecture-based filter storage to reduce the latency for filter loading. However it does not minimize the latency incurred for loading I and also refrains from supporting the re-use of local data. Moreover, ping-pong approach demands double the amount of required memory (for example, $2\times k_i$ kernel storage for k_i kernel groups in [91] and two buffers for W, I, and *Psums*/AC in [90]). Hence, this

chapter proposes new technique and hardware architecture for an energy-efficient CNN accelerator that mitigate such frequent throttling issue of PE array by efficiently optimizing the way data is fetched from KPC and stored within the periphery of PE array. Thus, enhancing throughput and energy efficiency of the proposed CNN accelerator.

3.3 Proposed Technique and Architecture

3.3.1 Uninterrupted-Processing Technique

As discussed in section 3.2.2, if t_{f_i} represents the data fetching time between KPC and local storage of PE in i^{th} iteration then Fig. 3.2 (a) and (b) shows the timing diagrams for conventional and proposed CNN accelerators, respectively. Here, t_{c_i} denotes data computation time of PE array in i^{th} iteration where each of these iterations consumes the time duration of t_{itr_i}. In the i^{th} iteration of conventional CNN accelerator, a process (denoted by P_i) in PE array that computes *Psums* begins only after completing the data fetching event (F_i) from KPC to PE array, as shown in Fig. 3.2 (a). During this t_{f_i} time of F_i, the PE array remains idle. Thus, effective duration of every i^{th} iteration is $t_{itr_i} = t_{f_i}+t_{c_i}$. On the other hand,

Algorithm 1 Proposed Uninterrupted-Processing Technique for CNN Accelerator

1: Determine n and r; ▷ n denotes number of required iterations and r denotes the minimum number of data required to begin P_i process.
2: **Initialization:** i=0 and j=0; ▷ i and j count the number of processed iteration and the number of data that has been fetched, respectively.
3: Begin $F_{i=1}$; ▷ Start pre-fetching of data for $itr_{i=1}$.
4: **for** $i \leq n$ **do**
5: **if** $j \leq r$ **then** ▷ Adequate data has not been fetched yet.
6: Continue $F_{i=1}$; ▷ Fetching continues for $itr_{i=1}$.
7: $j = j+1$; ▷ Count the number of data pre-fetched for $itr_{i=1}$.
8: Return to Step 13; ▷ Fetching continues for $itr_{i=1}$.
9: **else** ▷ Adequate data fetched to begin computation.
10: $i = i+1$;
11: Begin P_i; ▷ Computation begins for itr_i.
12: **if** F_i is Complete **then**
13: Begin F_{i+1}; ▷ Begin pre-fetching data for itr_{i+1}.
14: **else**
15: Continue F_i;
16: **if** P_i is Complete **then**
17: Return to Step 10. ▷ Begin computation for for itr_{i+1}.
18: **else**
19: Continue P_i.

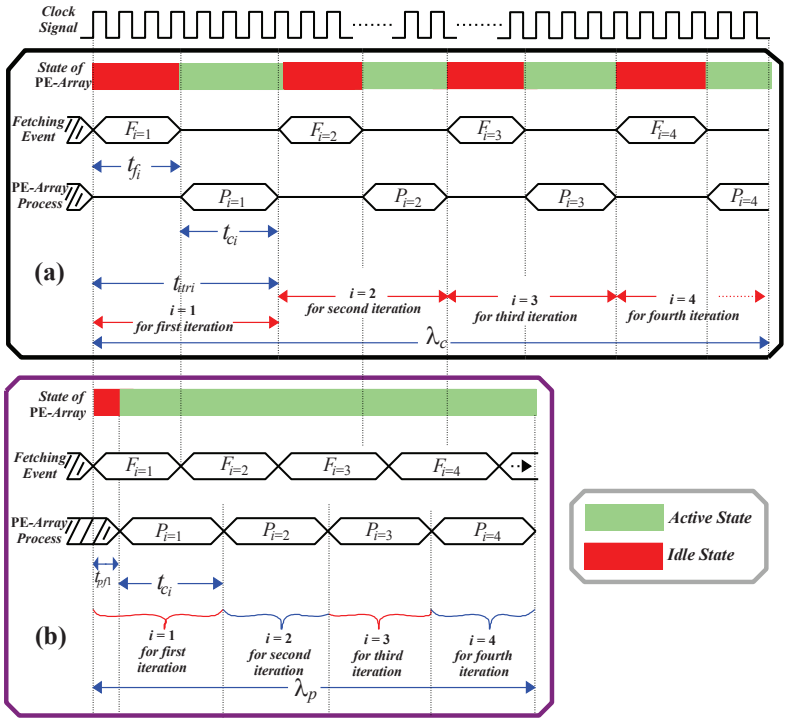


Fig. 3.2: Timing diagrams for (a) conventional and (b) proposed techniques.

the proposed uninterrupted-processing technique mitigates this t_{f_i} time and hence reduces the latency which consequently enhances the throughput of our CNN accelerator. Various steps of this technique have been hierarchically presented in Algorithm 1 and its primary notion is schematically illustrated in Fig. 3.2 (b). Its key idea is to concurrently perform both F_i and P_i, rather than processing them in sequential manner. To elaborate further, the computation of P_i can start once the minimum number of data, say r, has been fetched from KPC to PE array (referring lines 5−11 in Algorithm-1) that requires a pre-fetch duration of t_{pf_i}, as shown in Fig. 3.2 (b). This t_{pf_i} is a part of total fetch duration t_{f_i} of i^{th} iteration and note that $t_{pf_i} << \left(t_{f_i}/2\right)$. Subsequently, remaining data is fetched during the computation of i^{th} iteration, referring lines 12−15 in Algorithm-1.

Here, t_{f_i} is shorter duration than t_{c_i} and thereby, F_i finishes much earlier than the end of operation P_i. Thereby, we propose to start the pre-fetch for next $(i+1)^{th}$ iteration before the P_i operation ends, referring line no. 13 in Algorithm-1 and schematically represented in Fig. 3.2 (b). Since the minimum r data has been already pre-fetched for $(i+1)^{th}$ iteration during P_i of i^{th} iteration, P_{i+1} commences instantaneously at the end of P_i. Such phenomenon effectively mitigates the idle time of PE array during F_{i+1} (i.e. t_{f_i} is mitigated,

as shown in Fig. 3.2). As a result, the effective duration for each iteration becomes approximately equal to the computation time required by P_i (i.e. $t_{itr_i} \approx t_{c_i}$). Hence for the CNN model that requires n iterations, the effective latency reduces from $\lambda_c = \sum_{i=1}^{n}(t_{f_i} + t_{c_i})$ to $\lambda_p = (\sum_{i=1}^{n} t_{c_i}) + t_{pf1}$ where λ_c and λ_p represent latencies of conventional and proposed CNN accelerators, respectively; thus, conserving the time duration of $\approx\sum_{i=1}^{n}(t_{f_i})$.

3.3.2 Proposed Hardware Architectures

3.3.2.1 Low-Latency CNN-Accelerator Architecture

Referring to Fig. 3.2 (b), the PE array of CNN accelerator must simultaneously perform two operations: (1) store the new data fetched from KPC, and (2) supply data to the computation unit of PEs. Moreover, the proposed architecture for CNN accelerator needs to achieve *reuse of local data* without using large number of shift registers to conserve both energy and hardware resources. To incorporate these features in the suggested CNN accelerator, this work presents new architecture for locally storing and reusing the data within the periphery of PE array. Hence, the proposed generic architecture of CNN accelerator that supports $m\times n$-sized PE-array is shown in Fig. 3.3 (a). In conventional CNN accelerators like [62, 83] where both filter-weights and I values are stored in each of the PEs using single port SRAM. As a result, which the PE of such conventional design is incapable of operating when the data is being fetched. Unlike, our work changes the W storage inside the PE to dual port memory to achieve simultaneous read as well as write (for W pre-fetch) feature and changes the I storage by introducing RALB-based approach to achieve simultaneous read-write feature along with *local data reuse* without shifting I from one PE to another. Schematic representation of the RALB-based approach in the proposed CNN accelerator is depicted in Fig. 3.3 (a). It shows that each RALB is connected to a single or multiple row(s) of PEs, at any instance of time, to store their I values. Further, the suggested micro-architecture of RALB is presented in Fig. 3.3 (b) that incorporates steering logics and an $M\times k$ sized memory. It is capable of writing k bit of data to any of these M memory locations pointed by j write address that is steered via WA generator (i.e. address decoder), as illustrated in Fig. 3.3 (b). Once the value of j write address exceeds the magnitude of r (i.e.

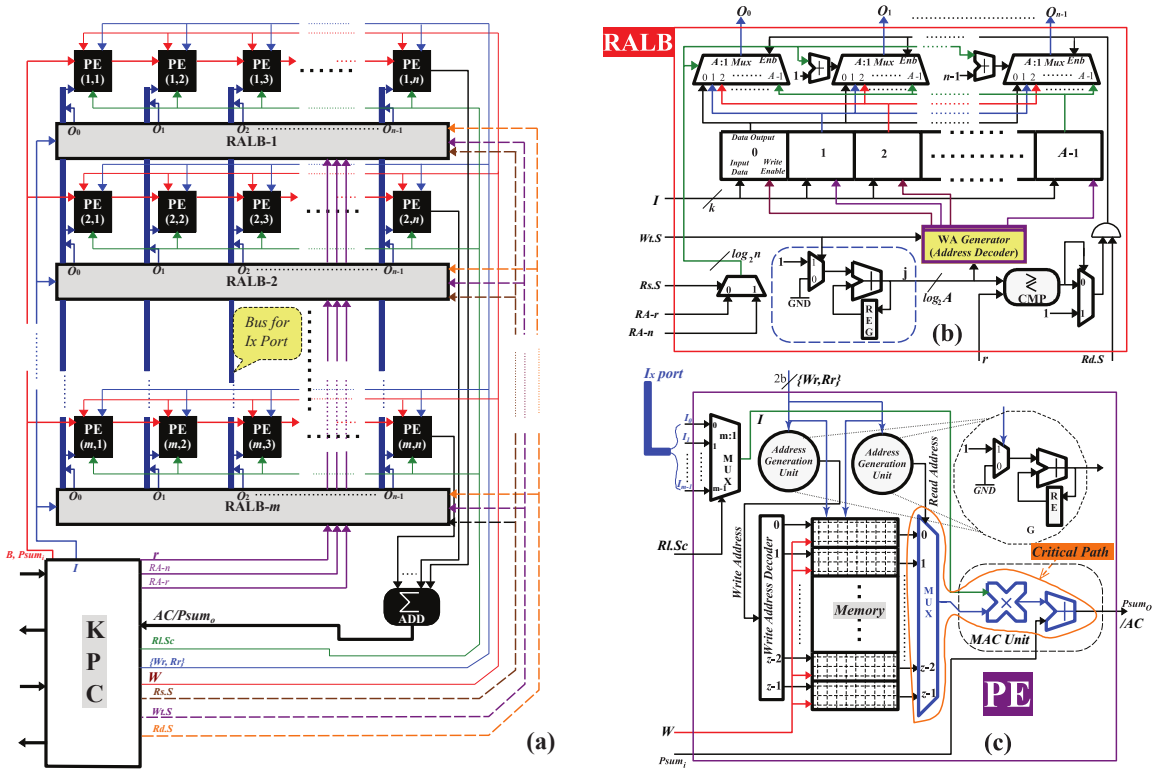


Fig. 3.3: (a) Proposed VLSI-architecture of CNN accelerator, (b) suggested internal micro-architecture of RALB, and (c) PE micro-architecture.

minimum number of data), RALB provides n number of k bit data to the computation units of n PEs, corresponding to the row with which RALB is connected to begin the P_i process, referring line 11 from Algorithm 1. Furthermore, RALB continues to store new data from outside the PE array, while simultaneously providing the data to n PEs. Thus, aforementioned process enables the proposed architecture to concurrently perform the operations of both F_i and P_i. In addition, it avoids the possible problem of read-before-write due to the fact that read rate (i.e. frequency at which read-address increases) is $z\times$ smaller than write-rate (rate at which write-address increases). The reason being, each element of I fed to PE is reused $z\times$ for MAC operation with z filter-weights that are stored inside the PE, as shown in Fig. 3.3 (c).

Since RALB stores M/A rows of I values and $A+r$ is less than M, RALB can pre-fetch $M-A$ number of data for the next iterations (i.e. F_{i+1} begins), referring line 13 in Algorithm 1. Note that A denotes width of each row of I. As the data is pre-fetched, P_{i+1} computation

for the next iteration can start once the P_i ends, under the condition that filter-weights are pre-fetched and stored in the filter storage of PE, as presented in Fig. 3.3 (c). It consists of a small $z \times k$-sized memory for storing z number of filter weights and a computation unit that can be configured for MAC or *max* operation. With the aid of bus-selection control signal to PE, its computation unit is routed with I values from the respective RALB which is connected with the PE, as shown in Fig. 3.3 (a) and (c). For performing MAC operations in PE, partial sum is fed as $Psum_i$ to PE architecture. Following that, MAC module is activated for computing $Psum_o$, as shown in Fig. 3.3 (c). Since both W and I are pre-fetched, P_{i+1} process instantaneously commences after the P_i process ends and hence incurring the time duration for an iteration to be $t_{itr_i} \approx t_{c_i}$, referring Fig. 3.2 (b).

3.3.2.2 Technique for Efficient Data Reuse and High-Throughput Computation

By using vertical routing bus-network in PE array from Fig. 3.3 (a) and bus-selection control signal for PEs in Fig. 3.3 (c), any row of PEs in the PE array can be connected to any RALB within its periphery. This allows the proposed CNN accelerator architecture to exploit the local reuse of I values throughout the period of horizontal strides in a row. Such I values stored in a RALB remains stationary while KPC increases the read address of RALB by s (note that s denotes the stride size, as discussed earlier in section II-A), from where the data is fed to PEs. Thus, horizontally reusing the I values without moving them from one PE to another. For vertical reuse of I values, let us assume x^{th} RALB is connected to i^{th} row of PE array and $(x + 1)^{th}$ RALB is connected to $(i + 1)^{th}$ row of PE array, and the remaining rows of PE array are connected in the similar fashion. Now, during each vertical strides, x index increments by s and hence $(x + s)^{th}$ RALB connects to i^{th} row of PE array and the same pattern is followed by rest of the rows in PE array. Since the number of RALBs is limited, x periodically resets to zero after it exceeds the maximum RALB count and such cycle continues till the end of all iterations. Note that during each of the vertical strides, KPC resets the read addresses of RALBs (whose data are to be reused by another row of PE array) to the new address where it was at the beginning of previous vertical stride. Furthermore, KPC increases the read addresses of those RALBs which contain the new lines of I for current vertical stride. Therefore, each I value is reused $z(\alpha - s)^2$ times and hence if z

is large enough to fit all the values of a filter weights in a layer then each I value needs to be fetched only once from the KPC. Moreover, the filter weights stored in local storage inside PE remains stationary throughout the period of all horizontal and vertical strides. Thus, each of these filter weights has been reused for $(A - s)^2$ times. Thereby, aforementioned process enables the proposed CNN accelerator to conserve its energy and latency.

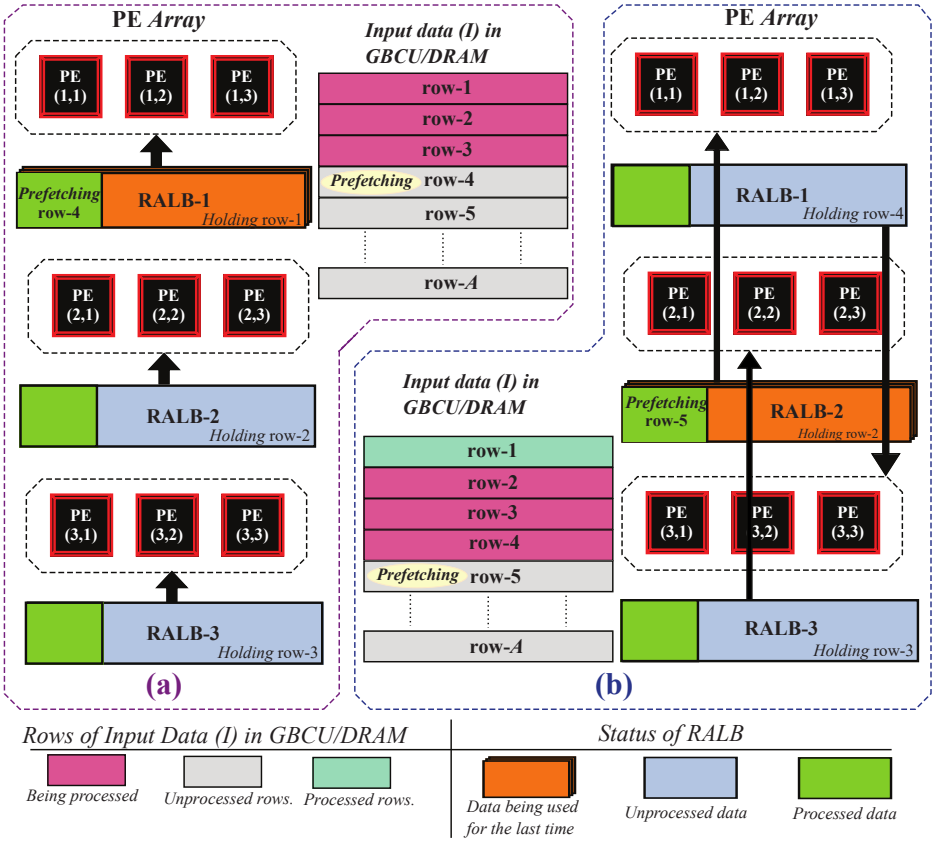


Fig. 3.4: Schematic representation of the suggested data-reusability process for 3×3 PE array in CNN accelerator.

For the better visualization of this technique, 3×3 PE array has been considered with three RALBs and a stride size $s = 1$ on an input feature map I of size $A \times A$, as presented by Fig. 3.4. It shows the process of reusing the data during vertical stride, and pre-fetching of data to perform uninterrupted computation.

Consider each vertical stride as an iteration (itr_i). Fig. 3.4 (a) shows that initially at $itr_{i=1}$ iteration, RALB-0, RALB-1 and RALB-2 are connected with 1^{st}, 2^{nd} and 3^{rd} rows of PE array, respectively. Referring Fig. 3.4 (a), three RALBs viz. RALB-1, RALB-2 and RALB-3 are holding row-1, row-2, and row-3, respectively, of the input feature map I. Since the

row-1 data fetched from RALB-1 is being used for the last time during the current vertical stride, initial r number of data in RALB-1 is not required anymore once the architecture performs r number of horizontal strides. Thus, this architecture pre-fetches initial r number of data item from row-4 of I to those initial r locations of RALB-1 (referring line nos. 12−13 from Algorithm 1). In second vertical stride ($itr_{i=2}$), now the computation is to be performed on row-2, row-3, and row-4 of I. Since row-2 and row-3 are already there in RALB-2 and RALB-3, and also enough r number of data from row-4 is pre-fetched in RALB-1, computation for second vertical stride can immediately start after the computation of first vertical stride. Furthermore, Fig. 3.4 (b) shows the suggested process of reusing the data of RALB-2 and RALB-3 by row-1 & row-2, and row-3, respectively, using new data of row-4 of I. Here, line-selection control signal of PEs in row-1, row-2 and row-3 changes their connections from I_1, I_2, and I_3 to I_2, I_3, and I_1, respectively, as shown in Fig. 3.4 (b). Thus, during each vertical stride, our PE array reuses the data of $\alpha-s$ RALBs and fetches data from external memory to other s RALBs. Therefore, data stored in a line memory is reused for $\alpha-s$ number of vertical strides. As discussed above, between every two consecutive vertical strides, data values are reused maximum of $z(\alpha-s)\times$. Thus, each of the data items stored in line memory has been used by up-to $z(\alpha-s)^2$ number of MAC operations, and each of the filter values is used for $(A-s)^2$ number of MAC operations.

3.4 Implementation Results, Comparisons and Hardware Validation

3.4.1 FPGA Implementation Results

In the proposed architecture of $m\times n$ PE-array, m determines three imperative factors: (1) length of vertical routing network, (2) size of bus selection multiplexer in PE, and (3) data movement latency between single RALB and PE that belongs to the farthest PE row (i.e. RALB-0 to m^{th} PE row). Similarly, n determines the width of vertical routing network. The computation throughput is given as $\Theta_T = (2\times N_{PE}\times f_{clk}\times\sigma\times\Omega)$ GOPs where N_{PE} is the number of PEs in CNN accelerator (i.e. $N_{PE} = m\times n$) and Ω represents the PE efficiency which

Table 3.1: FPGA and ASIC Implementation-Results Comparison of Proposed CNN Accelerator with Relevant State-of-the-Art Works.

	[62]	[79]	[83]	[76]	[75]	[75]	[86]	**Proposed**				
Platform	65 nm	ZC706	ZC706	VC707	VC709	VC709	VC707	**ZC706**	**VC707**	**VC709**	**ZCU102**	
Clk. Freq. (MHz)	200	200	150	200	200	200	200	**200**	**200**	**200**	**340**	**340**
LUTS (in kilo)	NA	–	182.616	–	121.472	121.472	280.4	**189.591**	**102.936**	**102.936**	**103.985**	**103.985**
FFs (in kilo)	NA	–	127.653	–	159.872	159.872	220.6	**12.721**	**20.474**	**20.474**	**20.631**	**20.631**
BRAMs (36kb)	NA	–	486	–	467	467	715.5	**480**	**640**	**640**	**640**	**640**
N_{PE}	168	576	780	64	664	664	2515	**864**	**864**	**864**	**864**	**864**
Precision (in bits)	16	16/8	16	16	16	16	Mixed(16-1)	**16**	**16**	**16**	**16**	**16**
CNN Model	AlexNet	Mod. AlexNet	VGG-16	VGG-16	AlexNet	VGG-16	RESNET152	**VGG-16**	**VGG-16**	**VGG-16**	**VGG-16**	**GoogLeNet**
Θ_T (GOPs)	46.1	198.1	187.8	12.5	220	230.1	726	**345.6**	**345.6**	**345.6**	**587.52**	**576.7**
η_{PE} (MOPs/PE)	274	343	241	195	331	347	288.7	**400**	**400**	**400**	**680**	**667**
Accuracy Loss** (%)	0.2–0.5	1-6	0.2–0.5	0.2–0.5	0.2–0.5	0.2–0.5	1-6	**0.2–0.5**	**0.2–0.5**	**0.2–0.5**	**0.2–0.5**	**0.2–0.5**
Power Consumption (W)	0.278	NA	NA	NA	9.61	9.61	–	**1.89**	**1.73**	**1.78**	**4.11**	**4.11**
Energy Effn. (GOPs/W)	166.2	NA	14.22	NA	22.9	22.9	–	**182.85**	**199.77**	**194.1573**	**142.95**	**140.31**

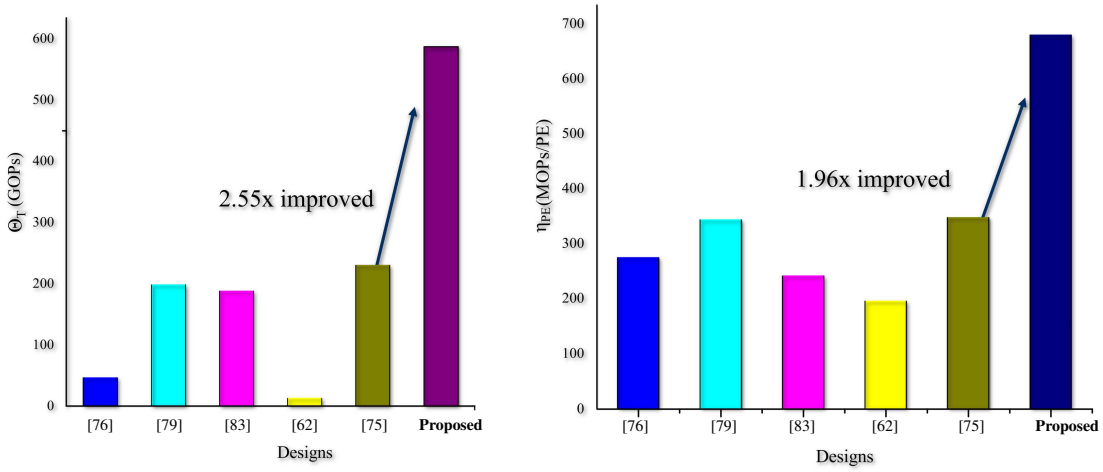
** : Compared to FP32 implementation, – : Not reported.

is given by $\Omega = N_{PE-util.}/N_{PE}$ such that $N_{PE-util.}$ denotes the number of PEs utilized. f_{clk} denotes the operating clock frequency of CNN accelerator, and GOPs refers to giga operations per second, and σ is the time efficiency of PEs that is given by the ratio of time duration when PEs are active and total time duration required for the computation (σ=t_{ci}/t_{itr_i}) [92]. Since each PE performs two operations: multiplication and addition in each clock cycle, referring Fig. 3.3 (c), a factor of 2 has also been incorporated in the aforementioned Θ_T expression [83], [93]. Since $\Theta_T \propto \Omega$, hence, to achieve higher PE efficiency that maximizes the Θ_T value, both m and n must be compatible with the filter shape of CNN model. As presented in Table 2.2, majority of computations in the state-of-the-art CNN models are dominated by smaller filters, mostly 3×3-sized filter, rather than larger filters with the size like 7×7 or 11×11.

Since a CNN accelerator can not be 100% efficient for all the filter shapes, m must be chosen in a way that it emphasizes only those filters which contribute highest computations, along with aforementioned hardware aspects. Hence, the proposed CNN accelerator consists of a dynamically configurable 36×24-sized PE array that is segregated into 16 smaller clusters to keep the effective value of m and n smaller. Each of these clusters comprises of a 9×6-sized PE array along with nine RALBs. Every single RALB can store 1024 pixels of I and thus effective values are m=9 and n=6 for these clusters. They are individually controlled by the KPC and each of them can simultaneously perform convolution operations for six 3×3-sized filters that delivers 100% of PE efficiency (Ω=1) or two 5×5-sized filters with 92% of PE efficiency (Ω=0.92) or 54 1×1-sized filters with 100% PE efficiency (Ω=1) or only one 7×7-sized filter with 91% PE efficiency (Ω=0.91).

Therefore, the proposed VLSI-architecture of RALB-based CNN accelerator has been hardware implemented on FPGA evaluation board (Xilinx Zynq-UltraScale+ MPSoC-ZCU102 board). Furthermore, gate-level synthesis and static-timing-analysis of suggested CNN accelerator indicate that its critical path lies in the PE micro-architecture and hence the longest-path delay includes two multiplexers, one adder and one multiplier delays, as shown in Fig. 3.3 (c). Thus, the proposed CNN accelerator is capable of operating at a maximum clock frequency of 340 MHz, when implemented on Zynq-UltraScale+ FPGA board. Eventually, FPGA hardware-resources consumed by the aforementioned implementation have

been presented in Table 3.1.



(a) (b)

Fig. 3.5: Performance gain compared to state-of-the-art works. (a)Throughput Θ_T, (b) Throughput density η_{PE}

.

3.4.2 Comparison with the State-of-the-Art Implementations

For fair comparison with other reported works in Table 3.1, we have also synthesized and post-route simulated the proposed CNN accelerator in same FPGA platforms used by these implementations from the literature. Due to slower block-RAMs (BRAMs) of Xilinx Virtex-7 (VC709 and VC707) and Xilinx ZYNQ-7000 (ZC706) boards, the proposed CNN accelerator when implemented on these FPGA platforms could operate at a maximum clock frequency of 200 MHz. As discussed above, Θ_T is an imperative performance metric that indicates the number of computations performed per second by the CNN accelerator. With the aid of proposed interrupt-free processing, $t_{itr_i} \approx t_{c_i}$ in our CNN accelerator and hence the value of σ=1.

Furthermore, energy efficiency is expressed as $\epsilon = \Theta_T/\mathcal{W}$ where $\mathcal{W}$ represents average power consumption that is computed as $\mathcal{W} = (\varepsilon_c + \psi_m)/t$ [62], [74]. Here, ε_c is the computation cost for each data which depends on the design logic and its precision. Similarly, ψ_m is the energy required to move the data from (or to) storage location, at different hierarchy, to (or from) the computation unit of PE. In our architecture, ψ_m can be expressed as $\psi_m = \hat{a}\times\hat{\alpha} + \hat{b}\times\hat{\beta} + \hat{c}\times\hat{\gamma}$ where $\hat{a}$ is the number of times a data has been loaded from DRAM to

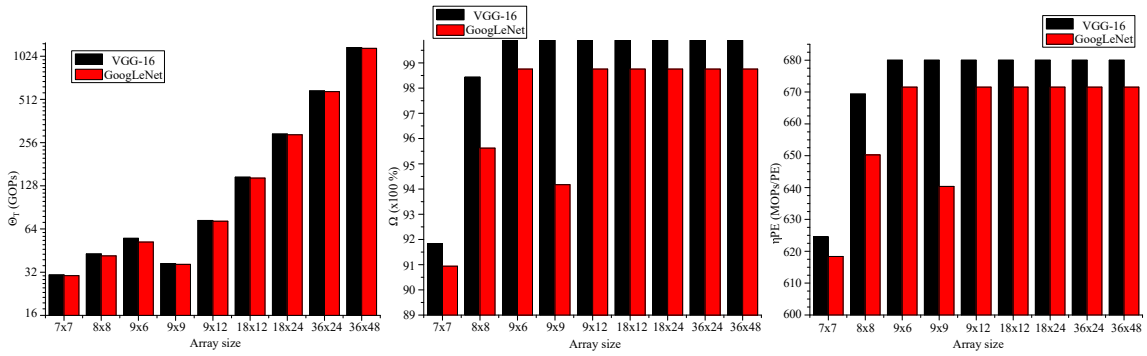


Fig. 3.6: Various achievable values of throughput (*Left*), PE efficiency (*Center*), and throughput density (*Right*) using different sizes of PE-array in the proposed CNN accelerator for VGG-16 and GoogLeNet CNN models.

GBCU and $\hat{\alpha}$ is the energy consumption associated with each of these data movement. Similarly, $\hat{b}$ and $\hat{c}$ are the number of times the same data has been loaded from GBCU-to-RALB and RALB-to-PE, respectively, and their respective energy costs are given by $\hat{\beta}$ and $\hat{\gamma}$. Note that B and c are inversely proportional to a and B, respectively. Among $\hat{\alpha}$, $\hat{\beta}$, and $\hat{\gamma}$, the smallest value is $\hat{\gamma}$; whereas, $\hat{\alpha} \approx 200\times\hat{\gamma}$, and $\hat{\beta} = 6\times\hat{\gamma}$ [62]. Since the magnitude of ψ_m is higher than ε_c, it is necessary to minimize the costly data movements which reduces $\hat{a}$ and $\hat{b}$ values that can be compensated by increasing the $\hat{c}$ value. In the proposed CNN accelerator, each I value is reused $z(\alpha - s)^2$ times and each of the filter weight is reused $(A - s)^2$ times, as discussed earlier in section 3.3.2.2. Thereby for I value, B is $z(\alpha - s)^2$ times smaller than c, and a is (number of filters)/z times smaller than B. Similarly for filter weights, both a and B are $(A - s)^2$ times smaller than c. Hence, the proposed RALB-based CNN accelerator consumes an average total power of 4.109 W while operating at a peak clock frequency of 340 MHz, when implemented on Zynq-UltraScale+ FPGA board. This total power consumption of 4.109 W is composed of dynamic and static powers of 3.383 W (i.e. 82%) and 0.726 W (i.e. 18%), respectively. Such power consumption of the proposed design is 2.34× lesser than the state-of-the-art FPGA implementation results from [75]. Based on this total power consumption, the energy efficiency (i.e. $\epsilon = \Theta_T/\mathcal{W}$) of our architecture is 140.95 GOPs/W which is 6.24× higher than the contemporary implementation from [75].

In order to demonstrate the reconfigurability of our design, it has been configured and implemented for both VGG-16 and GoogLeNet neural networks, as their implementation results are presented in Table 3.1. By using the proposed uninterrupted processing-technique

along with the RALB-based PE array mapping, while processing VGG-16 and GoogLeNet CNN models, 100% (i.e. Ω=1) and 98.16% (i.e. Ω=0.9816) of PEs, respectively, have been utilized by the proposed CNN accelerator architecture. Therefore, based on aforementioned quantifications, our design delivers throughput of 576.7 GOPs at a frame-rate of 202 fps and 587.52 GOPs at a frame-rate of 18.9 fps while processing GoogLeNet and VGG-16 neural networks, respectively, as presented in Table 3.1. This chapter also presents a FOM which is termed as throughput density (η_{PE}) that is computed as η_{PE}=Θ_T/N_{PE}. It indicates the number of operations performed in one second per PE and hence its higher value is desirable. Among all the relevant reported works [62], [79], [83], [76], [75] in Table 3.1, the state-of-the-art implementation (using Xilinx Virtex-7 VC709 FPGA-board) of [75] delivers highest throughput of Θ_T=230.1 GOPs and throughput density of η_{PE}=347 MOPs/PE. Moreover, with the aid of the suggested uninterrupted-processing technique, the proposed CNN accelerator architecture when implemented in the same FPGA platform delivers a throughput of 345.6 GOPs which is 33.42% better than the one reported by [75]. Furthermore, the suggested architecture achieves throughput density of 400 MOPs/PE which is 13.25% better than the η_{PE} of [75], as shown in Table 3.1. It also shows that the proposed CNN accelerator, when implemented on Xilinx Zynq-UltraScale+ ZCU102 FPGA-board, delivers highest throughput and throughput density of 587.52 GOPs and 680 MOPs/PE, respectively. In terms of hardware utilization, our design requires moderate consumption that is comparable to the reported implementations, as presented in Table 3.1.

The improvements in performance of the proposed design arises from its emphasis on uninterrupted processing through efficient buffering and dataflow scheduling. In contrast to conventional accelerators that suffer idle cycles during memory access or layer transitions, the design eliminates stalls that usually occur during data loading by using a random-access line buffer (RALB) that maintains a continuous dataflow. This ensures that the processing elements remain active instead of idling while waiting for memory. At the same time, local reuse of intermediate data across both convolutional and non-convolutional layers significantly reduces off-chip memory traffic. As a result, every cycle is effectively utilized, which allows the accelerator to sustain higher throughput and much better energy efficiency than prior works, while preserving near-FP32 accuracy levels.

On the other side, the proposed CNN accelerator is a scalable design, as its Θ_T increases with the size of PE array by maintaining nearly constant Ω and η_{PE} values. To demonstrate the same, Fig. 3.6 presents the variations of Θ_T, Ω, and η_{PE} with the increasing sizes of PE array from 7×7 to 36×48 for both VGG-16 and GoogLeNet CNN models. As discussed in section 3.4.1, following the adaptive convolution mapping from chapter 2, a 9×6 cluster of PE array with 54 PEs, perform convolution operations for six 3×3-sized filters that delivers Ω=1 or two 5×5-sized filters with Ω=0.92 or 54 1×1-sized filters with Ω=1 or only one 7×7-sized filter with Ω=0.91, hence achieves an effective Ω = 1 for VGG-16 model and 0.98 for GoogLeNet model. Therefore, as seen in Fig. 3.6, 9×12, 18×12 18×24, 36×24, and 36×48-sized PE arrays containing 2, 4, 8, 16 and 32 such clusters linearly increase the Θ_T while maintaining constant Ω and η_{PE}. On the other hand, PE arrays like 7×7, 8×8, and 9×9 gets affected due to inefficiency convolution mapping for different filter sizes. Hence, the value of both Ω and η_{PE} for such arrays remain lesser compared to earlier mentioned ones.

3.4.3 Peak Throughput Issues due to DRAM Bandwidth Limitation

Majority of the CNN accelerators like [90,94,95] face difficulties in achieving their peak throughput due to limitation in the sustainable DRAM bandwidth of the FPGA boards. For example, [95] requires to transfer more than 64 byte parallel data from external memory to achieve its peak throughput. On the other hand, reported architecture from [94] compromises the performance due to bandwidth limitation and even using 8 bit quantization. Further, the reported work from [90] is unable to hide all the external communication time under the computation time due to bandwidth limitations. As a result of extensive reuse of local data, as discussed earlier in section 3.3.2.2, the proposed CNN accelerator requires to transfer a maximum of 16-byte parallel data to/from the external memory (i.e. DRAM) at its peak throughput for the tested 16-bit precision. Therefore, the maximum required bandwidth of the proposed accelerator is significantly lesser than the sustainable DRAM bandwidth of the FPGA board that has been used for the implementation.

3.4.4 Compatibility with State-of-the-Art CNN Models

As discussed in section 3.4.1, the proposed architecture natively supports different filter sizes like 1×1, 3×3, 5×5, and 7×7. Thus, it can be used for state-of-the-art CNN models like MobileNet-V1, MobileNet-V2, EfficentNet and many more. Based on our earlier discussion, the proposed CNN accelerator architecture consists of 16 clusters of 9×6 PE array. Hence for state-of-the-art models, some of the clusters can be configured for depth-wise separable convolution (DWC) with larger filter like 3×3 or 5×5, while the remaining clusters can be configured for point-wise convolution (PWC) with 1×1 kernels. These clusters can also be configured to work in a pipelined manner to directly perform PWC on the output of DWC to reduce off-chip data access. Further, the residual connection can be achieved by sending I through the $Psum_i$ port.

3.4.5 Hardware Validation of the Proposed CNN Accelerator

3.4.5.1 Hardware Test Setup and Validation Method

Schematic and real-world views of the test setup for validating the hardware prototype of the proposed CNN accelerator are shown in Fig. 3.7 (a) and (b), respectively. They comprise of three major components: (1) host computer, (2) Zynq-UltraScale+ MPSoC-ZCU102 FPGA-board, and (3) external SD memory-card. The host computer has been installed with high-end tools like MATLAB® R2019a, Xilinx® Vivado 2018.2 and Xilinx® SDK 2018.2. To begin the validation process, a CNN model has been initially imported in the MATLAB® environment that is followed by extraction of model parameters like filter weights and biases for different layers of the CNN model. Thereafter, these parameters are converted into 16-bit fixed-point (FP16) format and are stored in the external SD memory-card, as multiple binary files. Similarly, pixel data of the input image is also converted to FP16 format and stored in the SD memory-card. Consecutively, this SD card is unplugged from the host computer and reconnected to the FPGA board, as shown in Fig. 3.7.

With the aid of Vivado 2018.2 tool, the Verilog HDL code of the proposed CNN accelerator is packed as an AXI-compatible IP. It is then interfaced with direct-memory-access (DMA) controller which is further connected with the AXI-HP port of onboard computer of

the FPGA board (for high-speed data transfer from onboard-DRAM to CNN accelerator), as illustrated earlier in Fig. 3.1. It also shows that the configuration and handshake signals between onboard-computer and CNN accelerator has been established using AXI-peripheral registers which are connected with the AXI-*lite* port of the onboard computer. Furthermore, an interrupt signal informs the onboard computer whether the GBCU is ready to receive new data from onboard-DRAM or it requires to offload the *AC* data to this DRAM, as shown in Fig. 3.1.

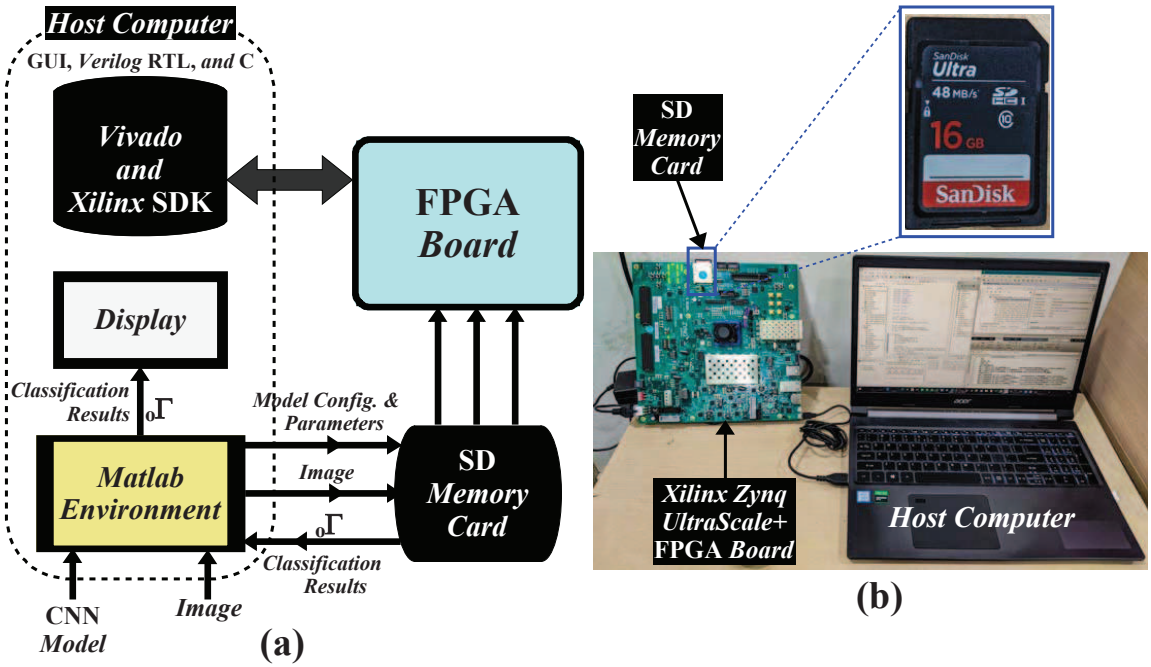


Fig. 3.7: (a) Schematic representation, and (b) real-world snapshot of the test setup for validating the proposed CNN accelerator prototype.

On the other hand, synthesized hardware information along with bit-stream of CNN accelerator is exported to Xilinx® SDK 2018.2 tool, as shown in Fig. 3.7 (a). Here, a stand-alone software application has been written in high-level C-language which loads the input image and model parameters from the SD memory-card to the onboard-DRAM of FPGA board. Following that, it sends input image and model parameters of a layer to the KPC of CNN accelerator using the HP port, as shown in Fig. 3.1. Simultaneously, it sets the configuration bits according to layer specification. Further, this application software off-loads the *AC* data of the layer from KPC to onboard-DRAM and returns them *I* during the processing of subsequent layer. Additionally, it stores a part of *AC* for each layer and full

AC of the last FC layer in SD memory-card. Thereafter, the SD memory-card is unplugged from the FPGA board and plugged back to the host computer, in order to visualize the results in MATLAB® environment.

3.4.5.2 Experimental Results

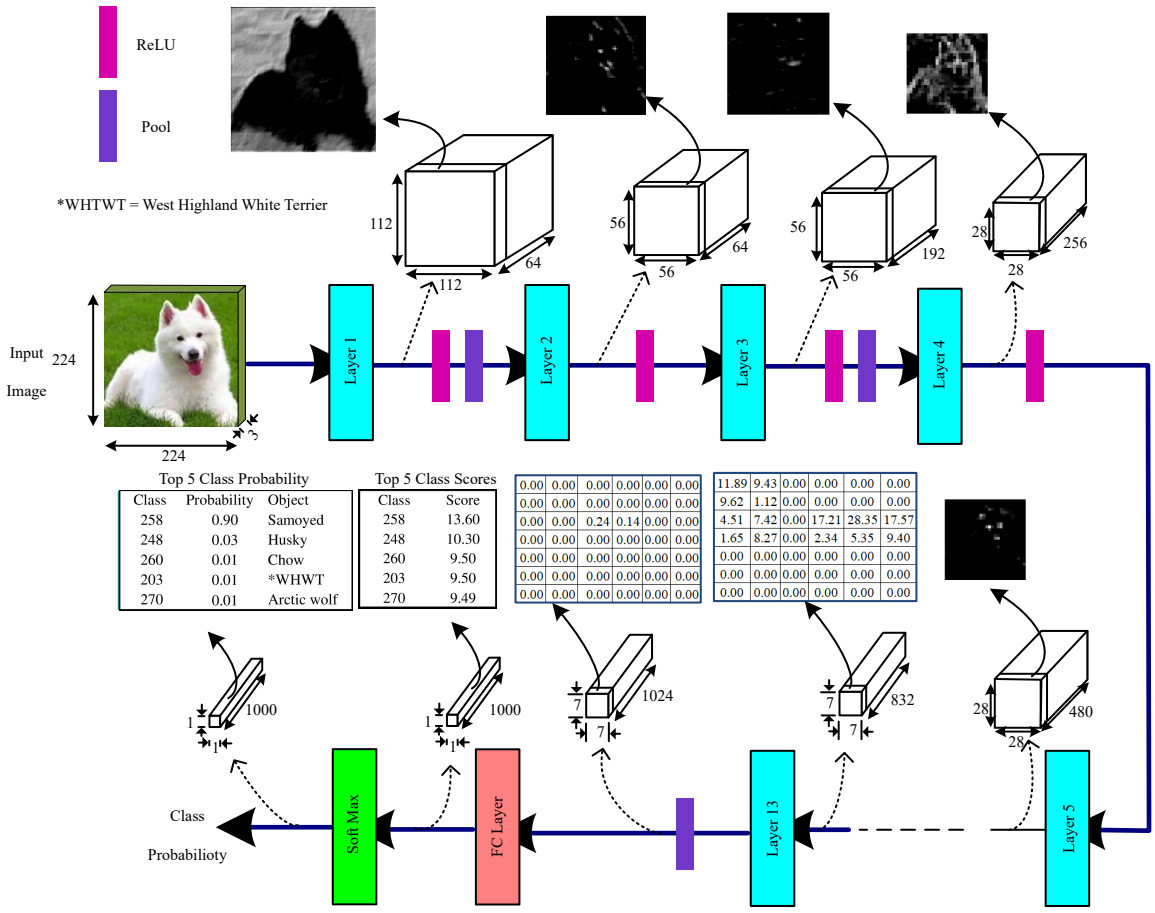


Fig. 3.8: Schematic representation of layer-wise processing of an image in the proposed CNN accelerator for GoogleNet model.

This chapter presents layer-wise output data for a red green blue (RGB) color image of the dimension 224×224×3 (this dimension has been specified by GoogLeNet CNN-model [26]) that is processed by the proposed CNN accelerator to classify the object from input image by using GoogLeNet CNN-model, as illustrated in Fig. 3.8. In the first layer, *Conv* operation has been performed with 64 different filters of dimension 7×7×3 (3 for three different R, G and B channels of input image) with strides s=2. Such operation enables to search and extract the score for 64 different high-level features in the form of *AC*, containing

64 feature score matrix of *height*×*width* dimension of 112×112. Here, both these height and width are scaled to (input-height)/s and (input-width)/s, respectively. Since 112×112×64 dimension of AC is extremely large to visualize here as a matrix, we have plotted one out of these 64 channels as an 112×112 image, mapping each value of the feature scores in AC as an image pixel. Thereafter, ReLU and max-pooling with s=2 have been applied on it that further reduces the dimension to 56×56×64, which is then fed back to the CNN accelerator for processing layer-2. In this second layer, 64 different filters of dimension 3×3×64 (64 for 64 different channels of I) with s=1 is applied that produces AC of same dimension, as shown in Fig. 3.8. In order to visualize the AC, one out of these 64 channels is plotted as an image. Certain features that have been detected produces bright pixel while dark part denotes the absence or very low score for the searched feature in that region, as shown in Fig. 3.8. Thereafter, *ReLU* activation and *maxpool* with s=1 is applied on the AC, before passing it to next layer. As a result, in those layers where $s>I$ for *Conv* or pool operation, height and width dimensions of AC reduces from the same value of I by a factor of s. Since after the layer-12 on-wards, *height*×*width* dimension of AC is small enough (7×7 or smaller) to be shown as a matrix. Hence, we have presented these values in the tabular format and the exact numerical value of detection scores for a certain feature are also shown in Fig. 3.8. Eventually, we have applied the soft-max activation on the output of FC layer which contains the detection score for all the 1000 classes from ImageNet data-set [96] to convert these class scores into class probabilities. Hence, the probabilities of Top-5 class have been presented in Fig. 3.8. All of these Top-5 classes are different breeds of dogs or wolf with Top-1 class, showing the correct breed of object in the input image. Thereby, aforementioned process validates the correct functionality of the proposed CNN accelerator hardware-prototype.

3.4.6 Accuracy of the Proposed CNN Accelerator

The suggested uninterrupted processing-technique and the corresponding CNN-accelerator architecture are precision independent. Both of them can be used for any data precision like floating point or highly quantized fixed-point representation by simply changing the precision of the computation unit in each PE of our accelerator to the desired precision. Furthermore, throughput of the design also scales up with the reduction in precision. However,

for evaluating the suggested architecture, we have designed the data path by considering 16 bit precision. As a result, this work uses both BF16 as well as FP16 representations for the data and filter weights. Even though BF16 representation preserves the accuracy of 32 bit floating point representation, it requires higher design efforts to realize the PE unit using the DSP blocks on FPGA and requires extra hardware resources. On the other hand, FP16 representation enables to directly map the PE unit into the in-built DSP core blocks of FPGA; however, it reduces the accuracy of our design by 0.5%.

3.5 Summary

This chapter introduced a novel uninterrupted processing approach for CNN accelerators aimed at improving both throughput and energy efficiency. The core idea centers on enabling simultaneous execution of PE computations and data fetching, which significantly reduces processing latency and boosts the achievable throughput. To facilitate this, a low-latency VLSI architecture was proposed, featuring a PE array that is based on a new RALB structure. Such RALB-enabled design ensured continuous data flow and higher throughput density without interruption.

To further optimize performance, the architecture leverages enhanced local data-reuse within the PE array. This exploitation of data locality incurred considerable improvement in energy conservation. Nonetheless, the current design focused solely on accelerating convolution operations within CNNs. On the other hand, there remains a substantial opportunity to expand the architecture's efficiency by extending local data reuse techniques to support a broader range of CNN operations.

The hardware implementation of the proposed accelerator was carried out on the Zynq UltraScale+ MPSoC ZCU102 FPGA platform. The system operated at a maximum frequency of 340 MHz and consumed a total power of 4.11 W. With an array of 864 PEs, the design achieved a peak throughput of 587.52 GOPs and an energy efficiency of 142.95 GOPs/W. Comparative analysis with existing state-of-the-art designs revealed that the proposed accelerator delivered a 33.42% increase in throughput and 6.24× improvement in per-PE energy efficiency.

The functional correctness and practical utility of the proposed design were validated through a real-world test scenario, involving object detection using the GoogLeNet CNN model, confirming the accelerator's capability to efficiently handle practical inference tasks.

Chapter 4

Efficient CNN Inference-Engine with Classify Unit based on New Memory-Sharing and Data-Reusing Techniques

4.1 Introduction

Though superior accuracy of CNN models has spiked the development of modern AI applications, requirements of sophisticated and massive computations that consume huge power for CNN models incur severe bottleneck in the widespread deployment of such applications on edge devices [60]. This necessitates the development of high-throughput and energy-efficient processing units for the implementation of such CNN applications. By using multiple PEs in parallel, CNN accelerators for these AI applications can achieve remarkable surge in speed. However, inefficient mapping of CNN kernels in PE array significantly affect the achievable throughput. Moreover, parallel processing using multiple PEs is restricted due to limited bandwidth of off-chip DRAM [90, 94, 95]. As a result, PEs in large array frequently starve for the data that often causes interruption in their processing and hence, limits the capability of sustaining peak throughput of CNN inference engine [93, 94]. Therefore, chapter 2 presented an optimized mapping technique to efficiently map different

CNN kernels of different sizes while chapter 3 presented uninterrupted processing based techniques and architectures for high throughput CNN accelerator.

However, conventionally, with the increase in the number of PEs, their energy consumption also increases which is still a major concern from an implementation aspect. Recent study [62], [74] has shown that significant portion of total power consumption in CNN computation is largely contributed due to frequent and massive movement of data between off-chip DRAM and processing unit. Such power dissipation is rather more than the power required by the computational processing units. Thus, chapter 3 also presented techniques and architecture to maximize reuse of local data to enhance the energy efficiency of processing *Conv* layer, hence presenting design of high-throughput and energy-efficient CNN accelerator. Several reported works in the literature also proposed various hardware architectures to perform computation for CNN models with higher throughput as well as better energy-efficiency [62], [74], [97]. These reported implementations mostly exploited the reuse of local data, at the expense of extra hardware requirements for complex routing and additional memory blocks. Moreover, alike chapter 3, these designs also mainly focused on the architectural optimization for the computation of *Conv* operation in feature extraction layers.

Since, energy consumption is significantly contributed by data movements, optimizing both processing and data-movement of other computations in the feature extraction layer, like *ReLU*, *maxpool* and *avgpool*, are equally important. Thus, it is necessary to minimize the data movement for the computation of all layers in CNN model to scale-up the energy efficiency of any CNN-based applications. Therefore, this chapter caters such needs by proposing reconfigurable PE-architecture that performs all types of computations in the feature extraction layers for the state-of-the-art CNN models, that enables to reuse the local data for all the computations of feature extraction layers. On the other hand, conventional energy-efficient architecture of CNN inference engine like [74, 90, 94, 95] utilizes large amount of hardware for memory as well as routing-network to reuse the local data and such CNN accelerator often encounters interruptions in their computation which is periodically halted during data movement in the PE array. This further affects the sustainable throughput of CNN inference engine. To circumvent such problems, this chapter extends the RALB based

architecture from chapter 3 to design a new line-memory based KPU to achieve the reuse of local data and uninterrupted processing while utilizing lesser memory and computational hardware resources.

On the other hand, conventional softmax-based classification layer [98, 99] implementing computation of (1.7) apparently requires N number of complex exponential and the same number of divisive operations to compute the probability of each class for an N-class model. Thus, while majority of CNN inference engines [99] rely on off-chip processor for the computation of classification layer, various attempts have been made to accelerate the computation of classification layer by using dedicated hardware. However, these works use coordinate-rotations digital-computer (CORDIC) based method [100] or look-up table (LUT)-based powers-2 approximation [87, 98, 99, 101, 102] to realize exponential and divisive computations. Nonetheless, such implementations still require significant amount of hardware resources, and these hardware designs are not integrated with CNN accelerator to deliver complete on-chip solution [103]. Based on analysis towards the working of classification layer and empirical realizations, we propose a simplified classification approach and hardware efficient classify unit where the computation of classification layer has been carried out using only comparators and multiplexers. Thus, this design excludes the requirement of any complex hardware for exponential and divisive computations. In addition, the suggested classify unit has been integrated with the kernel processing unit of CNN accelerator to build a complete on-chip CNN inference engine (CNN accelerator for inference). Highlights of all the contributions presented in this chapter are enumerated as follows.

- A dynamically reconfigurable hardware-friendly and high-speed VLSI-algorithm for the complete CNN inference-engine has been proposed in this chapter. It especially includes memory-efficient uninterrupted processing and all computations of feature-extraction as well as classification layers.

- A new multi-purpose hardware-architecture for PE has been suggested here. It is capable of performing all types of computations for the feature extraction layers of the state-of-the-art CNN models. In addition, this chapter also presents hardware-efficient architecture of the line-memory for achieving efficient data-reuse and uninterrupted

processing in CNN inference engine.

- Using the aforementioned PE and line-memory architectures, we have presented high-throughput and energy-efficient architecture of kernel processing unit. Subsequently, hardware efficient design of classify unit has been suggested here that is integrated with kernel processing unit to realize complete design of efficient CNN inference-engine.

- Comprehensive analyses of accuracy (using standard data-sets) and hardware resources for both the proposed algorithm and architectures, respectively, are carried out. Finally, we presented FPGA implementation of the proposed CNN inference engine and its functional validation, using real-world test setup. It is also ASIC synthesized and post-layout simulated in 28 nm fully depleted silicon-on-insulator (FD-SOI) technology node.

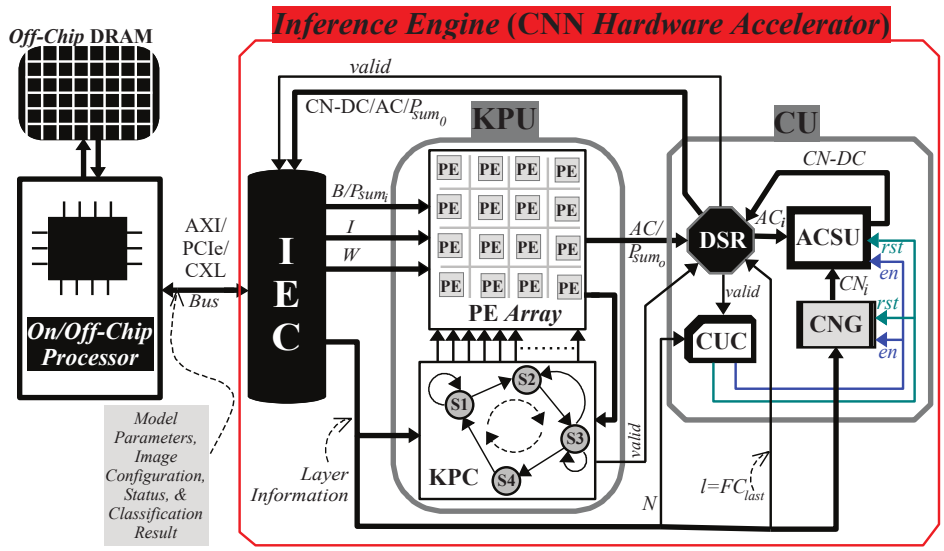


Fig. 4.1: Overall architecture of an objection classification system based on the proposed CNN inference engine.

4.2 System Model and Proposed Algorithm

4.2.1 System Model

While the hardware level system for object detection application presented in Fig. 3.1 of chapter 3 is purely from the perspective of FPGA implementation, a generic overview of the proposed object-classification system is shown in Fig. 4.1, consisting of three major parts: an on/off-chip processor, an off-chip DRAM, and a CNN inference engine. As discussed in chapter 3, the primary role of on/off-chip processor is to handle the software-based tasks like loading model parameters from off-chip non-volatile memory, input the image data from external source like camera as well as store them using off-chip DRAM, and provide configuration signals to the inference engine, as presented in Fig. 4.1. It also shows that inference engine and on/off-chip processor are interfaced via high data-rate links like CXL or PCIe or AXI buses for transferring data between off-chip DRAM and inference engine. The key role of inference engine is to perform dedicated high-speed computations for the CNN model. Since such computation types and processing requirements for different layers of CNN model vary over a wide range, the inference engine has been designed with three major blocks: updated KPU, updated GBCU, and classify unit (CU), as shown in Fig. 4.1. Here, GBCU communicates with on/off-chip processor, KPU, and CU, in order to configure and manage the flow of data like I, W, $Psum_i$, B, AC, class number of detected class (CN-DC) and many more signals. Furthermore, GBCU is responsible for managing the data flow between inference engine and off-chip DRAM, as illustrated in Fig. 4.1.

To begin with, GBCU first configures both KPU and CU with the layer information of CNN model, and also provides input image as I along with W as well as B of the active layer of model to KPU. Similar to chapter 3, depending on the size of PE array in KPU and parameter size of the current layer, here also KPU either generates the final AC value of the layer or only a portion of AC, (i.e. $Psum_o$). These partial sums are used in successive iterations of computation to generate the final AC and eventually, output from KPU is fed to CU. If the current layer (l) is the last fully-connected layer (i.e. l=FC_{last}) of the CNN model then CU starts the computation of classification layer. Finally, either CU returns the class

number (*CN*) of the object detected in the input image back to GBCU or CU directly returns the output *AC* from KPU back to GBCU in the form of $Psum_o$ or *AC*, as shown in Fig. 4.1. Here, GBCU reuses these values during successive iterations of computation for the same or next layer. Comprehensive discussion on the proposed efficient architectures of KPU and CU are presented in sections 4.3.1 and 4.3.2, respectively.

4.2.2 Proposed Algorithm

The proposed implementation-friendly algorithm for complete CNN inference-engine has been presented in Algorithm 2. Its salient features are to dynamically configure the inference engine based on the layer information, to uninterruptedly perform the computations, and to classify the object in CNN inference engine itself while processing the last FC layer. To begin with, values must be provided for three input parameters: (1) layer number of the FC_{last} layer, (2) number of iterations required for a layer (n_l), and (3) minimum number of data items that are required to begin the processing of that layer (r_l), as presented in line numbers 2-3 of Algorithm 2. Subsequently, selection of computation type that is required by the layer is performed in line number 4 of Algorithm 2. As discussed earlier, *W*, *I*, *B*, and *AC* denote filter weight, input feature map, bias value, and output activation, respectively. In addition, *xx*, *yy*, and *zz* represent the current position(s) in one, two, and three dimensional spaces of the data, respectively, that is being processed during current stride. Here, $\alpha \times \beta \times \gamma \times \delta$ is the size of a multi-dimensional filter, referring line number 4 in Algorithm 2, where width, height and depth of a filter are represented by α, β and γ, respectively. Further, δ denotes the number of filters.

To achieve uninterrupted processing, similar to Algorithm 1 of chapter 3, the partial fetching of r_l data is first completed and thereafter, the computation begins simultaneously with the fetching of remaining data, as presented in line numbers 5-15 of Algorithm 2. Once the data loading is over for the current iteration, pre-fetching of data commences for the subsequent iteration, even in the case of layer switching, as shown in line numbers 14-23 of Algorithm 2. After the processing completes for iteration, next iteration immediately starts. A visual illustration of such processing timeline, in comparison with conventional architectures, was presented earlier in Fig. 3.2 of Chapter 3. Here, if the processing of

Algorithm 2 Proposed Implementation-Friendly Algorithm for Inference Engine

1: **Initialization:** i=1, j=0, l=0, $AC\text{-}Max$=0, & $CN\text{-}DC$=0; ▷ i, j, and l count the number of processed iteration, the number of data that has been fetched, and the number of processed layer, respectively.
2: Determine FC_{last}; ▷ FC_{last} is layer number of the last FC layer.
3: Determine n_l and r_l; ▷ n_l, and r_l are number of iterations required for $Layer_l$ & minimum number of data required to begin the computation of $Layer_l$.
4: Determine Computation (P^l) as:

$$P^l \Rightarrow AC = \begin{cases} \sum_{a=1}^{\alpha}\sum_{b=1}^{\beta}\sum_{c=1}^{\gamma}\left(W_{a,b,c,\delta}\times I_{xx,yy,zz}\right)+B_\delta, & Layer_l = Conv \\ \sum_{x=1}^{N} I_n \times W_{n,x} + B_x, & Layer_l = FC \\ max\{I(x{:}\alpha, y{:}\beta,\ z)\}, & Layer_l = maxpool \\ \frac{\sum_{x}^{+\alpha}\sum_{y}^{+\beta} I_{x,y,z}}{2^{Log_2^{\lfloor \alpha \times \beta \rfloor}}}, & Layer_l = avgpool \\ max\{0, I_{x,y,z}\}, & Layer_l = ReLU \\ min\{max\{0, I_{x,y,z}\}, 6\}, & Layer_l = ReLU6 \end{cases}$$

5: Begin $F^l_{i=1}$; ▷ Start pre-fetching of data for $itr^l_{i=1}$.
6: **while** $i \le n_l$ **do**
7: **if** $j \le r_l$ **then** ▷ Adequate data has not been fetched yet.
8: Continue $F^l_{i=1}$; ▷ Fetching continues for $itr^l_{i=1}$.
9: $j = j$+1; ▷ Count the number of data pre-fetched for $itr^l_{i=1}$.
10: Return to Step 13; ▷ Fetching continues for $itr^l_{i=1}$.
11: **else** ▷ Adequate data fetched to begin computation.
12: **Process** P^l_i; ▷ Compute AC_i.
13: **if** F^l_i is not complete **then**
14: Continue F^l_i;
15: **else**
16: **if** $i < n_l$ **then**
17: Begin F^l_{i+1}; ▷ Start pre-fetching data for itr^l_{i+1}.
18: **else**
19: set j = 0
20: Begin $F^{l+1}_{i=0}$; ▷ Start pre-fetching data for $itr^{l+1}_{i=0}$.
21: **if** P^l_i is Complete **then**
22: **if** $l = FC_{last}$ **then**
23: $$AC_{Max_i} = \begin{cases} AC_i\ , & AC_i > AC_{Max_{i-1}}; \\ AC_{Max_{i-1}}, & AC_i \le AC_{Max_{i-1}}; \end{cases}$$
24: $$CN\text{-}DC_i = \begin{cases} CN\text{-}DC_{i-1}\ , & AC_i < AC_{max_{i-1}}; \\ CN_i, & AC_i \ge AC_{Max_{i-1}}; \end{cases} \quad ;$$
25: $$Output = \begin{cases} AC_i\ , & l \ne FC_{last}; \\ 0\ , & l = FC_{last}\ \&\ i \ne n_l; \\ CN\text{-}DC_i\ , & l = FC_{last}\ \&\ i{=}n_l; \end{cases}$$
26: $$l = \begin{cases} l+1\ , & i{=}n_l\ \&\ l \ne FC_{last}; \\ 0\ , & i{=}n_l\ \&\ l = FC_{last}; \\ l\ , & i < n_l; \end{cases}$$
27: **if** $i = n_l$ **then**
28: Set i = 0;
29: Return to Step 9. ▷ Start processing the next layer.
30: **else**
31: Set $i = i + 1$
32: Return to Step 18. ▷ Start computation for itr^l_{i+1}.
33: **else**
34: Return to Step 18 ▷ Continue P^l_i.

current layer (in a given iteration) is the last FC layer (i.e. $l{=}FC_{last}$) then the computation for classification layer is performed in pipeline where the AC_i is fed as input and eventually, returns the class number (CN) of the detected class as an output, referring line numbers 24-28 from Algorithm 2. On the other side, if $l{\neq}FC_{last}$ then AC_i is returned as output that is recycled during the computation of subsequent layer, as illustrated by line number 28 of Algorithm 2.

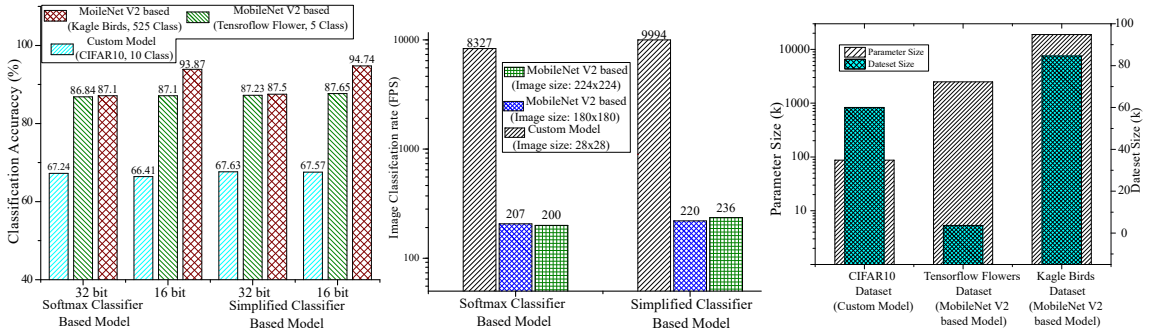


Fig. 4.2: Comparative accuracy and performance analyses of the proposed implementation-friendly algorithm for CNN inference engine.

To perform the computation of average pooling layer, the denominator of division has been replace from $\alpha{\times}\beta$ to $2^{\log_2^{\lfloor \alpha \times \beta \rfloor}}$ that enables to perform the division operation using simple bit shifting, rather than using complex hardware for division operation. Since all values of the activation are scaled at same ratio, no degradation has been noticed in the classification accuracy of the model. Furthermore, for the computation of classification layer, proposed algorithm employs a simplified classification technique wherein it generates the classification result directly from the activation of last FC layer, and avoids all complex exponential and divisive computations that are used in conventional softmax-based classification layer processor. This new approach is based on an analysis towards the working of classification layer and empirical realizations. As a result, it is found that the probability Pr_i of an object class is maximum when the magnitude of activation AC_i, associated with that class from the last FC layer, is at its peak. This method processes N (i.e. $N{=}n_{FC_{last}}$) number of AC_i and it subsequently searches as well as determines the largest activation (AC_{Max}) and its associated CN, as presented in line numbers 25-29 of Algorithm 2. Finally, it returns the CN associated with AC_{Max} as the detected class, referring line number 26-28 in Algorithm 2. Aforementioned proposed technique uses simple comparison and sorting method.

Thereby, its corresponding hardware architecture (i.e. classify unit from Fig. 4.1) requires only comparator and multiplexer.

Comparative performance analysis of both the proposed and the conventional techniques are presented in Fig. 4.2. It shows the comparison of classification accuracy in Fig. 4.2 (a), and computation frame rate in Fig. 4.2 (b) of three different CNN models for both conventional and proposed approaches. Here, three CNN models have been developed for three different datasets. Furthermore, Fig. 4.2 (c) shows the size of dataset used for training these models and their parameter size. Referring Fig. 4.2, unlike parameter and dataset size, choosing proposed hardware-friendly approach over the conventional approach refrains from causing noticeable impact on the classification accuracy. Moreover, it significantly improves the processing speed and the classification frame rate.

4.3 Proposed Hardware Architectures, Implementation Results and Comparisons

Based on the proposed Algorithm 2, this section presents efficient hardware architectures of an updated KPU and CU for the complete CNN inference-engine, as discussed earlier in section 4.2 with the aid of Fig. 4.1. In addition, their implementation results are also presented and compared with the state-of-the-art works.

4.3.1 Energy and Memory Efficient Architecture of KPU

As discussed earlier in chapter 3 as well as in section 4.1, an energy-efficient and high-speed inference engine must have the capability to minimize the costly off-chip memory access by maximizing the efficient reuse of local data within the chip to reduce power consumption and mitigate interruptions in the data processing to sustain the peak achievable throughput. For such processing, the proposed KPU architecture shown in Fig. 4.3 has been designed by extending the *RALB*-based line-stationary approach from chapter 3. Unlike, the proposed approach in this chapter focuses on further enhancing the hardware efficiency of CNN inference engine by using a specially designed line-memory to store I data within

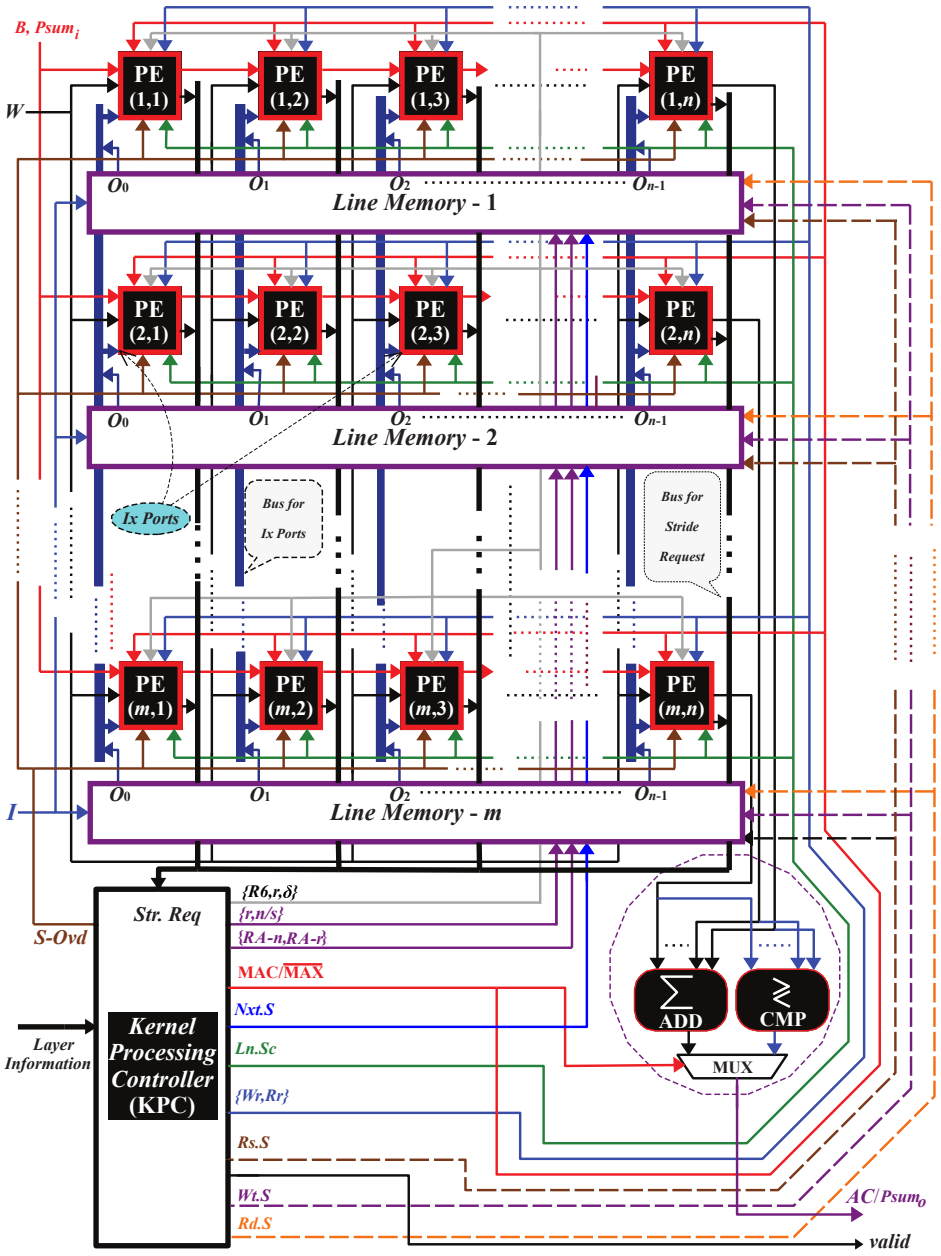


Fig. 4.3: Proposed hardware architecture of energy and memory efficient KPU.

the PE array, and W value inside the PE. As shown in Fig. 4.3, this suggested KPU also consists of $m \times n$ PE matrix similar to chapter 3, wherein m line-memory units are used for three key purposes: (1) to store the incoming data, (2) for providing data to PEs, and (3) to reuse the data within the PE array. In order to efficiently configure the PE array, for different filter shapes, and achieve efficient data-reuse, the routing of data between line memories and PEs has been controlled with the aid of KPC, as presented in Fig. 4.3. At a given time instance, any PE in the array can be dynamically routed to and fetch the data from any of the m line memories in the proposed design. At a time, one line-memory can provide data to n different PEs. Thus, suggested KPU architecture that contains PE array and line memories can perform MAC and pool operations for varying filter sizes, ranging from 1×1 to $m \times n$, at different level of parallelism. Comprehensive discussion on the proposed architectures of PE and line-memory are carried out in the following sections 4.3.1.1 and 4.3.1.2, respectively. The techniques which allow the proposed KPU to achieve energy-efficient data reuse is similar to one that has been discussed earlier in section 3.3.2.2 of chapter 3. Furthermore, detailed explanation of the techniques which allow this KPU to achieve chapter 3 like uninterrupted processing is presented in sections 4.3.1.3 of this chapter.

4.3.1.1 Multipurpose PE Architecture

Referring line number 4 of Algorithm 2, the inference engine must dynamically configure itself for different types of computations viz. *Conv*, *FC*, *maxpool*, *avgpool*, *ReLu*, and *ReLu6*, which are required by various layers of the CNN model. Therefore, KPU of CNN inference engine requires a PE which supports all these computation types. Hence, Fig. 4.4 shows the proposed micro-architecture of such PE that consists of MAC unit for processing both *Conv* and *FC* layers; a MAX module to performs the *maxpool* operation. As illustrated in Fig. 4.4, depending upon the value of $MAC/\overline{MAX}$ select signal, one of either MAC or *maxpool* operation is selected. In order to perform *ReLu* activation on I_l (i.e. A_{l-1}) as well as to conserve energy for null values, the suggested PE also consists of sign-and-zero detector (SZD). On detecting a negative or null value of I, SZD turns off the MAC unit and produces a null value at the output. In this way, it performs *ReLu* activation on A_{l-1} and conserves energy by avoiding unnecessary switching of MAC unit for such input values. This plays

a significant role in energy conservation, as the major portion of I values are zeros towards the later layers of CNN models [63]. Moreover, SZD is overridden by the *S-Ovd* signal for first layer of the model to allow I_1, irrespective of their polarity. Furthermore, to perform *Relu6* on the outgoing activation, a MIN module is used which clips the output activation below 6 when required. To perform *avgpool* operation, multiplier in the MAC unit acts as a pass-through buffer, thus MAC unit performs as an adder only. The accumulated results are bit shifted to achieve the division without using a real hardware for divider (referring line number 4 of Algorithm 2). As discussed in section 4.3.1, the proposed PE can fetch data from any of the m different line memories. Data from the desired line-memory is selected by using the *Ln.Sc* signal, as shown in Fig. 4.4.

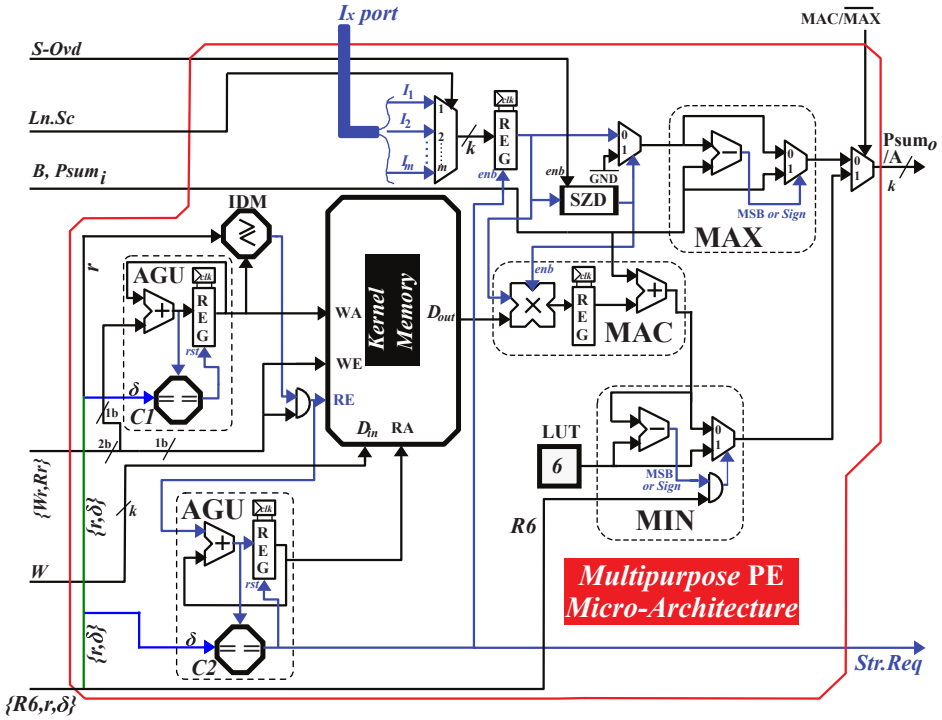


Fig. 4.4: Proposed architecture of multipurpose PE that is used in the design of PE-array for KPU.

Since CNN models apply several filters on the same input data, the proposed PE architecture also incorporates a $z{\times}k$-sized kernel memory, which stores z filter weights of k-bit each, in order to minimize the off-chip data access and maximize the local data reuse. Here, read and write operations are enabled with the aid of two independent address-generation units (AGUs) for read and write address ports: RA and WA ports, as shown in Fig. 4.4. However, read operation is supervised by the input data monitor (IDM) module which dis-

ables the read operation until r number of filter weights are not stored in the memory. Here, each of the I values that is fed to PE is reused upto $z\times$ by z different filter weights. As a result, rather than fetching a new I data in each clock cycle, the proposed PE uses stride request-based mechanism. Once in every δ clock cycles when AGU reads the last address of the filter memory, comparator *C2* in the proposed PE architecture sends the stride request signal (i.e. *Str.Req*) to KPC module for the next I data, as well as activates input data register to store the incoming data and holds it for next δ clock cycles, as shown in Fig. 4.3 and 4.4.

4.3.1.2 Hardware-Efficient Line-Memory Architecture

As discussed comprehensively in chapter 3, conventional energy-efficient designs of CNN accelerators [62], [74] use large memories inside the PE to store W, I as well as *Psums* and continuously move these data in all directions for their local reuse. Such implementation requires significant hardware and energy overheads for storage and routing purposes. It also faces frequent interruptions in the processing while moving the data in and out of PEs. Chapter 3 introduced RALB-based KPU to address this challenges. In every clock cycle such RALB was able produce n parallel data. Due to the feature of pure random parallel read mechanism in it's architecture, all M locations of RALB were ready to produce data in each clock cycle. However, as discussed in section 3.3.2.2 and 4.3.1.1, due to extensive reuse of local data, PE architecture of the proposed KPU in this chapter requires n numbers of data after every δ clock cycles. Among them, only s numbers of I data are new, and n-s number of data from previous read cycle can be reused using some local output buffer. Hence, this chapter uses a new design of line buffer which has been referred as *line-memory* that requires significantly reduced number of resources for the line buffer. Detailed discussion of the new *line-memory* is as follows:

As discussed in section 4.3.1, proposed KPU architecture uses the line stationary approach that simultaneously shares single line-memory with n different PEs, as presented in Fig. 4.3. The proposed architecture of a line-memory is shown in Fig. 4.5. It comprises of $k\times A$-sized memory which is adequate for storing at least A number of input data where A represents the size of a single row of the input feature map I. Here, read and write operations in such line-memory are independently governed by two different control signals from

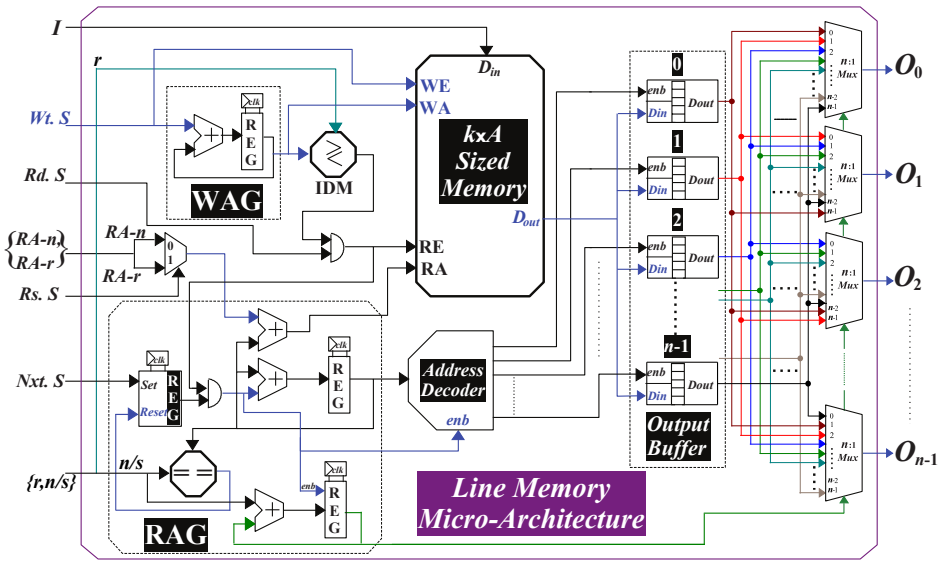


Fig. 4.5: Proposed architecture of single line-memory, used in the design of PE array for KPU.

the KPC module: *Rd.S* and *Wt.S*, as shown in Fig. 4.5. These signals decide whether the line-memory stores the incoming data or provides the data to PEs. In addition, they locally manage read and write addresses of the memory by controlling two different address generator units: write address generator (WAG) and read address generator (RAG), as shown in Fig. 4.5. Furthermore, read operation is supervised by IDM which ensures such operation is being performed only when the adequate number of data (i.e. r) has been fetched from the memory (referring line numbers 7-13 of Algorithm 2). Whenever the *Wt.S* signal selects a line-memory for writing the incoming data, WAG sequentially generates a new write address in every clock cycle to store each of the incoming data items. During every stride, line-memory receives two different read pointers (*RdPtr*): RA-r and RA-n from the KPC module of KPU to denote reading locations for reused and new data, respectively. Hence, if the data stored in the line-memory is being reused during a vertical stride then it uses RA-r (to indicate read address for reuse) as the read pointer, that follows the same addressing scheme which was adopted in the previous vertical stride. Otherwise, it uses RA-n as the read pointer if the data of line-memory is being used for the first time. Here, the *Rs.S* signal from KPC module has been generated to indicate whether the line-memory is selected for reused or fresh data. When a line-memory is selected for read operation and if IDM indicates that sufficient data has been written into the memory, then every time RAG receives a

Nxt.S trigger-signal, it generates n/s consecutive read addresses that start from the address pointed by *RdPtr*, over the next n/s clock cycles.

In addition, line-memory has an output buffer that comprises of n k-bit registers and the same number of n:1 multiplexers, as shown in Fig. 4.5. Apart from generating read addresses for the $k \times A$ memory, RAG also generates write address for the registers of this output buffer, and select line bits for the multiplexers. When read operation is being performed in the line-memory, during each clock cycle, address decoder module decodes these write addresses and activates one of the n registers to store the output of $k \times A$ memory, as shown in Fig. 4.5. In this way, every time there is a trigger in the *Nxt.S* signal, n/s number of new data items from n/s different locations of $k \times A$ memory are buffered in n registers of the output buffer. Thus, between every two horizontal strides, output buffer holds $n-s$ number of old data for reuse and s number of new data. Here, select-line bits for the multiplexer also increments by s to switch connection of output ports. As discussed earlier in section 4.3.1 and shown in Fig. 4.3 as well as Fig. 4.4, n parallel outputs (O_0 to O_{n-1}) from n registers of the output buffer of a single line-memory is connected to any one of the m different *Ix* ports of n different PEs. Note that *Ix* port is represented as the concatenation of I_1, I_2, $I_3 \cdots I_m$ (each of k bit-width) which are fed to m:1 multiplexer in the proposed PE architecture in Fig. 4.4. Here, outputs from m line-memories are connected to every PE via such *Ix* port, as shown in Fig. 4.3. Referring Fig. 4.4, every time there is a stride request (once in every δ clock cycles), data from *O* port of line-memory is transferred to the input register of the PE (via *Ix* port and m:1 multiplexer of PE architecture). Thus, registers of output buffer in the line-memory can again store the data which are required in the next stride. Therefore, the proposed line-memory architecture uses only single $k \times A$-sized memory for n number of PEs, compared to n number of such memories in conventional architectures from [74].

4.3.1.3 Technique for High-Throughput Computation

Computation in the conventional inference engine stops while moving the data within an array of PEs and when the data are loaded from external memory [62], [74]. Since the data movements in inference engine takes place as frequently as the computations, achievable peak throughput of accelerator is adversely affected due to such frequent interruptions

in the computation process that are mitigated by the proposed technique, as presented in Algorithm 2. Referring line numbers 7−13 in this Algorithm 2, instead of waiting for all the data for a vertical stride to be fetched in the line-memory, suggested technique starts the computation after a partial fetching where a minimum r number of data are fetched and the remaining data will be fetched during the computation run time, referring line numbers 14−15 from Algorithm 2. This process is realized in the hardware architecture of KPU with the aid of suggested architectures of PE and line-memory, as presented in section 4.3.1.1 and 4.3.1.2.

Here, the proposed design of PE requires new data only once in every δ clock cycles, and the proposed line-memory needs s clock cycles to generate the new data for s number of PEs. Therefore, it uses the remaining duration of $\delta - s$ clock cycles to fetch the remaining data from external memory. Thus, the proposed KPU performs both data fetching and computation, simultaneously. Furthermore, as discussed in section 3.3.2.2, suggested KPU architecture also pre-fetches the minimum number of data, required for subsequent vertical stride, and it immediately starts the computation of next vertical stride at the end of current vertical stride, without any interruption. Therefore, the proposed KPU architecture performs computation with higher throughput and sustains this peak throughput during the entire computation period.

4.3.1.4 KPU Implementation Results and Comparison

The proposed KPU architecture has been hardware implemented on FPGA platform (using AMD-Xilinx Zynq-Ultrascale+ ZCU102 MPSoC board). Detail implementation results of the proposed KPU are presented in Table 4.1. For fair comparison, proposed KPU architecture has been additionally implemented in other FPGA platforms (viz. ZC706, VC707 and VC709) which are adopted by the reported state-of-the-art implementations. Note that the fixed-point bit-quantization representation ($Q_{n.m}$) of 16 bit has been used in all the aforementioned implementations.

Static timing analysis of the our KPU architecture indicates that it's critical path lies in the PE and such path includes a single multiplier. Thus, the proposed KPU implementation operates at a peak clock frequency of 360 MHz when implemented in aforementioned AMD-

Table 4.1: Comparison of Proposed KPU Implementations with Relevant State-of-the-Art Works.

	[79]	[83]	[76]	[75]	[97]	[93]	**Proposed Implementations**						
Platform	ZC706	ZC706	VC707	VC709	VC709	ZCU102	**ZC706**	**VC707**	**VC709**		**ZCU102**		
Clock Frequency (MHz)	200	150	200	200	200	340	**389**	**407**	**382**	**382**	**360**	**360**	**360**
LUTS (in kilo)	–	182.616	–	121.472	107.325	103.985	**93.683**	**93.683**	**93.683**	**93.683**	**96.676**	**96.676**	**96.676**
FFs (in kilo)	–	127.653	–	159.872	74.430	20.631	**27.666**	**27.666**	**27.666**	**27.666**	**27.725**	**27.725**	**27.725**
BRAMs (36kb)	–	486	–	467	390	640	**64**	**64**	**64**	**64**	**64**	**64**	**64**
DSP Usage	576	780	64	664	0	864	**864**	**864**	**864**	**864**	**864**	**864**	**864**
N_{PE}	576	780	64	664	1312	864	**864**	**864**	**864**	**864**	**864**	**864**	**864**
Precision (in bits)	16/8	16	16	16	4	16	**16**	**16**	**16**	**16**	**16**	**16**	**16**
CNN Model	A.N*	VGG16	VGG16	VGG16	M.N.V2	VGG16	**VGG16**	**VGG16**	**VGG16**	**M.N.V2**	**VGG16**	**M.N.V2**	**R.N.50**
Accuracy Loss** (%)	0.5-3	0.2-0.5	0.2-0.5	0.2-0.5	8-15	0.2-0.5	**0.2-0.5**	**0.2-0.5**	**0.2-0.5**	**0.2-0.5**	**0.2-0.5**	**0.2-0.5**	**0.2-0.5**
Θ_T (GOPs)	198.1	187.8	12.5	230.1	413.9	587.52	**672.192**	**703.23**	**660.01**	**660.01**	**622.08**	**622.08**	**619.21**
η_{PE} (MOPs/PE)	343	241	195	347	320	680	**778**	**814**	**764**	**764**	**720**	**720**	**716.68**
Power Consumption (W)	–	–	–	9.61	6.10	4.11	**2.169**	**1.968**	**1.91**	**1.91**	**2.99**	**2.99**	**2.99**
Energy Efficiency (GOPs/W)	–	14.22	–	22.9	67.85	140.31	**309.91**	**357.33**	**345.55**	**345.55**	**208.05**	**208.05**	**207.09**

A.N* = Modified AlexNet, M.N.V2 = MobileNet V2, R.N.50 = ResNet 50, ** : Compared to FP32 implementation

Xilinx Zynq-Ultrascale+ ZCU102 MPSoC FPGA-board. As shown in Table 4.1, suggested KPU can operates at 1.79×, 1.91×, 1.91×, and 1.06× higher clock frequency compared to the reported implementations from [79], [75], [97], and [93], respectively, when implemented in their respective FPGA boards. As discussed earlier in section 4.3.1.3, the proposed KPU architecture performs uninterrupted computations. Thus, PEs in the proposed KPU architecture are never idle and carries out their computations with 100% of time efficiency (i.e. σ=1). Thus, the proposed KPU architecture is capable of delivering a throughput that is 3.39×, 2.87×, 1.6×, and 1.06× higher than the throughput achieved by [79], [75], [97], and [93] reported implementations, respectively, on their respective FPGA board, as illustrated in Table 4.1. On the other hand, throughput density η_{PE} is an important figure-of-merit that determines the hardware efficiency of inference engine and it is given by the number of operations performed by a single PE per unit time (i.e. OPs/PE). Improvements in Ω, σ and f_{clk} values of the proposed KPU are responsible for enhancing the η_{PE} value which is 2.26×, 2.20×, 2.3875× and 1.06× higher than the η_{PE} values of [79], [75], [97], and [93] reported implementations, respectively, as presented in Table 4.1.

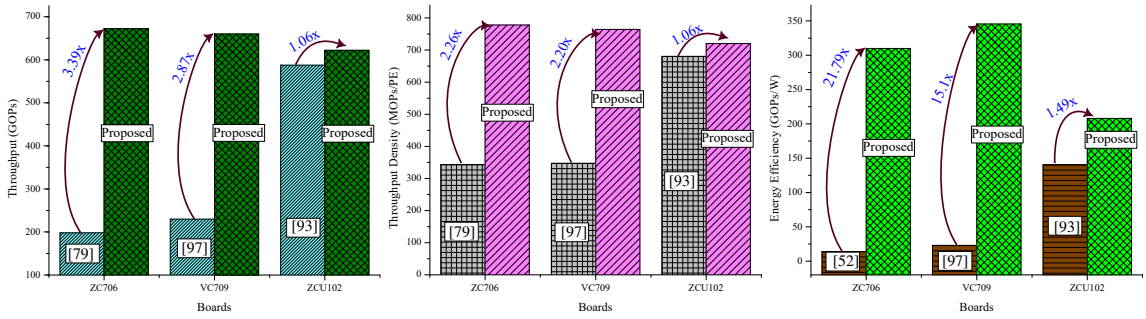


Fig. 4.6: Gains (a) throughput, (b) throughput density, and (c) energy efficiency of the proposed KPU compared to state-of-the-art works

As discussed in section 4.1, memory footprint of inference engine mainly determines its overall power consumption. Hence, it is desirable to have an architecture with reduced memory footprint and efficient reuse of local data to reduce the power requirement. Referring to section 4.3.1 where it is described that the proposed KPU architecture simultaneously shares a single line-memory with n numbers of PEs as well as it reuses each input data for $z(\alpha-s)^2\times$ and filters the values for $(A-s)^2\times$. As a result, the suggested KPU architecture significantly reduces the memory footprint in the form of BRAM usage in the FPGA. It uses a total of 64

BRAMs where each of them has a size of 36 kb and such memory utilization is 7.59×, 7.29×, 6× and 10× lesser than the utilizations reported by [83], [75], [97] and [93], respectively, as presented in Table 4.1. It also shows that the energy efficiency (i.e. number of computations performed per watt of total power consumed) of the proposed KPU is 21.79×, 15.09×, 5.09×, and 1.49× higher compared to the reported works from [83], [75], [97] and [93], respectively. Furthermore, suggested KPU consumes significantly lesser LUTs and flip-flops compared to the reported works, with an exception of flip-flop usage of [93] where the proposed KPU utilizes slightly more flip-flops in comparison to [93]. However, the proposed KPU has a notably lesser utilization of area-expensive BRAMs and LUTs compared to [93], as presented in Table 4.1.

4.3.2 Proposed Hardware-Efficient Architecture of Classify Unit

As discussed earlier in section 4.2, to determine the class number of detected object in the input image, computation of classification layer from the proposed Algorithm 2 involves searching and finding the largest activation value (i.e. AC_{Max}) from the FC_{last} layer, and CN associated with AC_{Max}. Hence, the hardware design for such activation search operation can be carried out by adopting either parallel or resource-shared approach. For a CNN model with N classes, the former parallel approach can be beneficial, provided all N different activation values from the FC_{last} layer of the CNN model are simultaneously available. Such approach requires an array of $N-1$ comparators as well as $2(N-1)$ number of 2:1 multiplexers in order to perform the search operation, and a highly-complex class number generator (CNG) to produce N parallel CNs. However, for a CNN model with $\breve{M}$ neurons in the FC_{last} layer, it would require $\breve{M}$ number of PEs in the KPU to compute one AC item per clock cycle. In addition, a model with N classes typically comprises of N neurons in the FC_{last} layer. Thus, it requires $N \times \breve{M}$ number of PEs in the PE-array of KPU for computing all N number of AC values. The proposed KPU architecture in this chapter has been designed using 864 PEs. Hence, this KPU is incapable of generating more than one AC value per clock cycle for a 1000 class image-classification models like VGG, GoogleNet and MobileNet, while computing the FC_{last} layer. Such adverse incapability is also present in the contemporary implementations of KPU architecture, reported in the literature [75,76,79,83,93]. Therefore,

rather adopting the parallel approach, this chapter focusses on the resource-shared approach for designing CU architecture, as shown in Fig. 4.7. It represents the proposed hardware-efficient micro-architecture of CU for the suggested simplified classification-technique, referring Algorithm 2. Major submodules of this architecture are data-&-signal router (DSR), classify unit controller (CUC), CNG and activation searching unit (ACSU). Instead of using the approximated hardware for complex exponential and divisive operations [87,98,99,102], the proposed CU architecture has been designed using only comparators and steering logics (i.e. multiplexers). Suggested micro-architectures of the aforementioned submodules are comprehensively presented in the remaining portion of this subsection.

4.3.2.1 Micro-architecture for DSR

Based on the discussion from section 4.2, KPU output has been directly fed to CU, irrespective of the layer being processed. In the proposed CU architecture, the key role of DSR is to collect both output ($AC/Psum_o$) and validation (*valid*) signals from PE-array and KPC of KPU, respectively. Here, if the current layer is FC_{last} layer of CNN model (i.e. $l = FC_{last}$) then DSR routes them as AC_i activation-value to ACSU through multiplexer MUX1 and input-valid port of the CUC (referring line number 25 of Algorithm 2), as shown in Fig. 4.7. It also shows that DSR collects the *CN-DC*, as the final classification result from ACSU, and the output validation signal from CUC that are routed through MUX2 and MUX3, respectively, to GBCU of the inference engine. On the other hand, if $l \neq FC_{last}$ then it directly returns the incoming output and validation signals from KPU to GBCU via MUX2 and MUX3, respectively, referring line number 28 of Algorithm 2.

4.3.2.2 CUC and CNG Micro-architectures

The suggested CU architecture operates only when the KPU is processing FC_{last} layer of CNN model; otherwise, DSR directly returns activation or $Psum_o$ to GBCU, and all other modules of CU remain idle. Here, CUC manages the operations of two modules: CNG and ACSU by continuously monitoring the layer information, validity of activation from KPU, and the status of CNG. Such CUC architecture has been designed with a subtractor-based comparator and a 2:1 multiplexer, as shown in Fig. 4.7. When DSR indicates (using the

valid signal) that KPU is processing FC_{last} layer by producing a valid activation AC_i value to CU and subsequently, the output of MUX4 in CUC activates the ACSU to process such AC_i value. In parallel, CUC also activates CNG to generate a *CN* value for the same input AC_i value. Further, comparator of CUC continuously compares the count of *CN* passed from CNG to ACSU with the total number of classes (i.e. *N*) in the CNN model that is derived from the layer information. Once CUC detects that all *N* classes have been covered, it deactivates both ACSU and CNG, as well as resets these modules to prepare them for subsequent inference task.

While processing the FC_{last} layer, KPU is configured in a way that it computes AC_i values in the same order of classes as in the ImageNet dataset that is used in this chapter. Thus, as presented in Fig. 4.7, CNG has been designed using a synchronous up-counter, wherein it linearly generates a CN_i for each of the AC_i values from FC_{last} layer.

4.3.2.3 ACSU Architecture

Referring the discussion from section 4.2, the computation of classification layer in Algorithm 2 involves searching and finding the AC_{Max} value, from the FC_{last} layer, and *CN* value associated with AC_{Max}. Here, the suggested ACSU architecture has been designed to generate *CN* value of the detected object in input image. Our ACSU micro-architecture comprises of a comparator (realized using subtractor), three 2:1 multiplexers (MUX5, MUX6 and MUX7), and two feedback registers: REG1 and REG2, as shown in Fig. 4.7. To begin with, both REG1 and REG2 are reset to null value that initializes both AC_{Max} and *CN-DC* values to zero, referring line number 1 of Algorithm 2. While KPU architecture is processing the FC_{last} layer, if ACSU receives a valid AC_i then it is fed to both MUX5 and comparator. At the same time, CNG produces a class number (CN_i) associated with the AC_i and is fed to MUX6, as presented in Fig. 4.7. In addition, the values of $AC_{Max_{i-1}}$, and $CN\text{-}DC_{i-1}$ from REG1 and REG2 are also fed to other terminals of MUX5 and MUX6, respectively. If the comparator finds current AC_i value is higher than the earlier largest value of activation $AC_{Max_{i-1}}$, then comparator output drives MUX5 and MUX6 to store AC_i and CN_i in REG1 and REG2 as the largest found-value of $AC_{Max_{i-1}}$ and the class number associated with largest activation (i.e. *CN-DC*), respectively. On the other hand, if the comparator

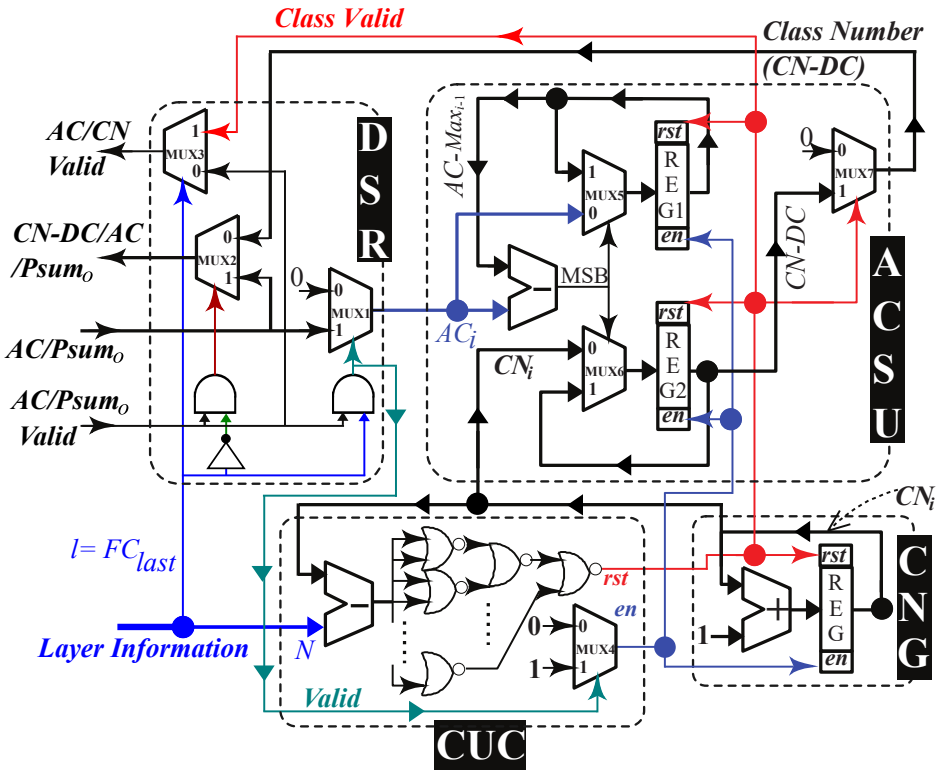


Fig. 4.7: Proposed micro-architecture of classify unit for the proposed CNN inference engine.

determines new activation value to be smaller than previously found largest activation value (i.e. AC_i<$AC_{Max_{i-1}}$), then it retains the previous value of REG1 and REG2, respectively, referring line numbers 25-26 of Algorithm 2. This process continues until CUC stops ACSU at the end of comparing all N activation values that are being fed from KPU. Eventually, it returns the CN associated with the largest activation to the DSR, which produces it as the classification result (i.e. class number of the detected object *CN-DC*) and routes to GBCU of the proposed inference engine. In the aforementioned way, the proposed CU architecture classifies the object from the activation of FC_{last} layer without requiring any complex exponential and divisive computations.

4.3.3 Hardware Resources and Latency Analysis

FPGA implementation results of the proposed CU architecture and their comparison with the state-of-the-art works are presented in Table 4.2. As discussed earlier in section 4.2.2, the complexity of classification layer surges with the number of classes (i.e. N) in the model. Thus, in the conventional state-of-the-art designs of CU for such classification layer

Table 4.2: Comparison of Proposed Classify Unit with Relevant State-of-the-Art FPGA based Works.

Work	FPGA Board	Prec. (bit)	N	Clock Freq. (MHz)	Θ_T (MIPs)	Resource LUTs, FFs	LUTs +FFs	$\breve{\alpha}$
[99]	ZC706	16	10 (1000)	500	500	128, 97 (561, 111)✠	225 (672)	2.22 (0.74)✠
[102]	KC705	16	10	154	154	2229, 224	2453	0.06
[98]	ZC706	16	8	500	500	395, 498	893	0.56
[87]	Zynq-7	16	10	-	-	1746, 1386	3180	-
Prop.	**ZCU102** **ZC706** **KC705**	**16**	**2–1000**	**1250** **389** **381**	**1250** **389** **381**	**59, 49**	**108**	**11.57** **3.6** **3.53**

$\breve{\alpha}$=Θ_T/{LUTs + FFs} MIPS/Resource where Θ_T: Computation Throughput.
✠: Quantifications of Hardware Resources and Throughput Density ($\breve{\alpha}$) for N=1000.

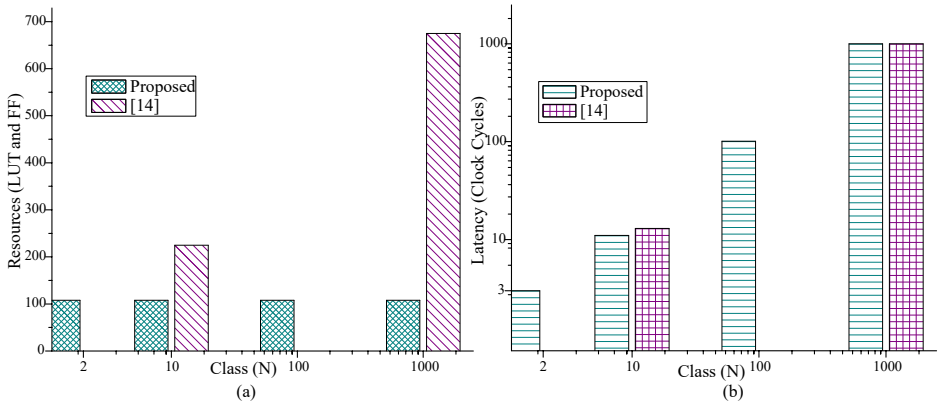


Fig. 4.8: Comparison analysis plots of (a) hardware resources and (b) latency, consumed by proposed and state-of-the-art CU architectures for different values of N.

[87, 98, 101, 102], both hardware-complexity and latency gradually increases with the N value. Hence, most of these reported works have fixed the magnitude of N between 8 to 10. Nevertheless, only [99] has reported the implementation results for N values of 10 and 1000, as listed in Table 4.2. Therefore, comparison plots of the proposed CU architecture with respect to this state-of-the-art work from [99] has been presented in Fig. 4.8. It shows the comparison analysis in terms of latency and hardware consumption for different values of N.

Since the proposed CU architecture from Fig. 4.7 is independent of N, its hardware resources remain unchanged until N exceeds 2^k where k denotes the bit precision. Unlike, for state-of-the-art work [99], hardware consumption increases from 225 units for N=10 to 672 units for N=1000 (i.e. 2.99× surge), as shown in Fig. 4.8 (a). As a result, throughput density (i.e. $\breve{\alpha}$) falls from 2.22 to 0.74. Moreover, $\breve{\alpha}$ value of the proposed CU architecture is 38.33%

and 79.44% higher than $\breve{\alpha}$ values of [99] for N=10 and N=1000, respectively, as presented in Table 4.2. However, classification latency of the proposed architecture increases linearly with N, as shown in Fig. 4.8 (b). Since the ACSU of the proposed CU architecture contains single comparator, this resource-shared architecture demands a classification latency of N+1 clock cycles and it is still comparable to the latency of N+3 clock cycles, which is delivered by [99], as shown in Fig. 4.8 (b). Even though, there is such linear dependency of latency with N, the proposed architecture classifies images at the frame rate of 1.25×10^6 fps when N=1000 that is 41670× faster than the conventional video frame-rate of 30 fps.

4.3.4 ASIC Design and Comparison

This part of the thesis presents the comparison of ASIC design (synthesized and post-layout simulated) results of the proposed classify unit with the state-of-the-art works on classifier layer. The proposed classify unit architecture has been synthesized and post-layout simulated using the standard electronics-design-automation (EDA) tools in 28 nm-FDSOI technology node. Specifically, Verilog HDL-code of the proposed design is functionally verified, gate-level synthesized, and timing analyzed, using the NCSim EDA-tool from Cadence. Furthermore, the gate level netlist, obtained from the Genus tool, is imported in Cadence Innovus EDA-tool, using the library exchange format (LEF) files of 28 nm-FDSOI technology. In this platform, physical design of the proposed classify-unit netlist has been carried out, that includes floor-planning, placement, power planning-&-routing, clock tree synthesis, timing verification and signal routing of the proposed digital-circuit layout. Hence, the final chip-layout (using 5 metal layers) of the proposed classify unit is presented in Fig. 4.9. Furthermore, these results and their comparison with the state-of-the-art implementation results are presented in Table 4.3. It shows that proposed design delivers a maximum throughput (Θ_T) of 2.5 GIPS, occupying a core area of 236 μm^2 with a hardware efficiency ($\hat{\eta}$) of 5.05 GIPS/mw/mm^2 that is 12.54× better compared to the state-of-the-art work from [101]. In addition, the proposed classify unit occupies 24.1× lesser area and consumes 83.9% lower power, compared to [101], when implemented on a same technology node. Referring Table 4.3, it seems that the proposed classify unit delivers comparatively lower throughput. However, while operating at the maximum clock frequency of 2.5 GHz,

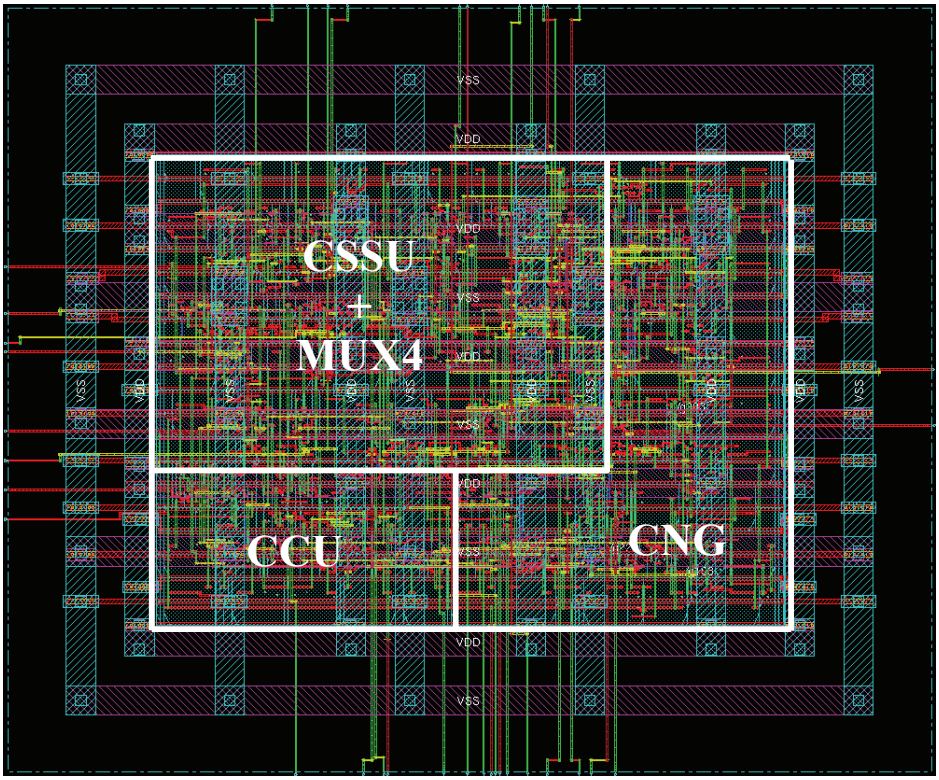


Fig. 4.9: Core ASIC layout (with 5 metal layers) of the proposed classify unit in 28 nm-FDSOI technology node.

the suggested classify unit processes the information at a frame rate of 2.5×10^6 frames per second, provided KPU generates single class score in every clock cycle. As discussed earlier in section 4.3, major bottleneck is due to KPU because an accelerator with 1026 PEs can only generate one class score per clock cycle (when it is processing the last FC layer).

4.4 Hardware Development and Validation of Proposed CNN-Inference Engine

Referring to Fig. 4.1, the proposed efficient hardware-architectures of KPU and CU from section 4.3 are aggregated into a complete CNN inference-engine which is implemented on FPGA board, and ASIC synthesized as well as post-layout simulated in 28 nm FD-SOI technology node. Here, fixed-point bit-quantization of $Q_{n.m}$=16 bit has been used for both ASIC and FPGA implementations. Table 4.4 presents the quantifications of various design parameters of the proposed CNN inference-engine in both FPGA and ASIC platforms. On

Table 4.3: Comparison of Proposed Classify Unit with Relevant State-of-the-Art Works where all these Implementations are carried out in 28 nm-FDSOI Technology Node with 16 bits of Data Precision.

Works	No. of Classes (N)	Clock Frequency (GHz)	Throughput Θ_T (GIPS)	Area (μm^2)	Total Power (mW)	$\hat{\eta}$ (GIPS/(mW·mm^2))
[98]	8	2.78	22.24	10081	-	-
[101]	10	3	30	5676	13.12	402.85
[104]	8	2.8	22.4	15000	51.6	28.94
[87]	10	0.58	-	3818	1.528	-
Proposed	**2–1000**	**2.5**	**2.5**	**236**	**2.1**	**5.05 k**

Hardware Efficiency ($\hat{\eta}$) = Θ_T /{Total Power × Area} GIPS/(mW·mm^2).

Table 4.4: Implementation Results of the Complete Inference Engine in FPGA and ASIC platforms.

ZCU102 FPGA (16 nm FinFET Technology)		**ASIC (28 nm FD-SOI Technology)**	
Clock Frequency	360 MHz	Clock Frequency	3.85 GHz
N_{PE}	864	N_{PE}	864
Θ_T	622.08 GOPS	Θ_T	6.65 TOPS
LUT Count	97589	Cell Count	9964573
FF Count	27790	Core Area	9.5146 mm^2
BRAM Count	64	Core Dimension	3.084×3.085 mm
Total Power	2.99W	Total Power	19.748 W
η_{PE}	720 MOPS/PE	η_{PE}	7.7 GOPS/PE
Energy Efficiency	208.05 GOPs/W	Energy Efficiency	336.74 GOPs/W

observing Table 4.4, three major modules of the proposed inference engine: CU, GBCU and KPU consume 0.086%, 0.693% and 99.219% of the total hardware requirement, respectively. In addition, ASIC die-layout of the proposed inference engine in 28 nm FD-SOI technology node is shown in Fig. 4.10. Furthermore, this ASIC chip-layout is capable of delivering a throughput of 6.65 TOPs, as illustrated in Table 4.4. On the other side, performing computation with the aid of suggested CU architecture (in the FPGA platform) reduces operation time for classification layer from 2.05 ms in the ARM Cortex A-53 CPU to 0.8 μs in the CU for a N=1000 class CNN-model, at the expense of 0.086% extra hardware. Such complete implementation of CNN inference engine that specially includes the hardware version of CU, along with KPU which performs all type of computation for the feature extraction layers of state-of-the-art CNN models, has been first time reported here. Furthermore, the FPGA prototype of such CNN inference engine is used for the object classification and validated for its correct functionality.

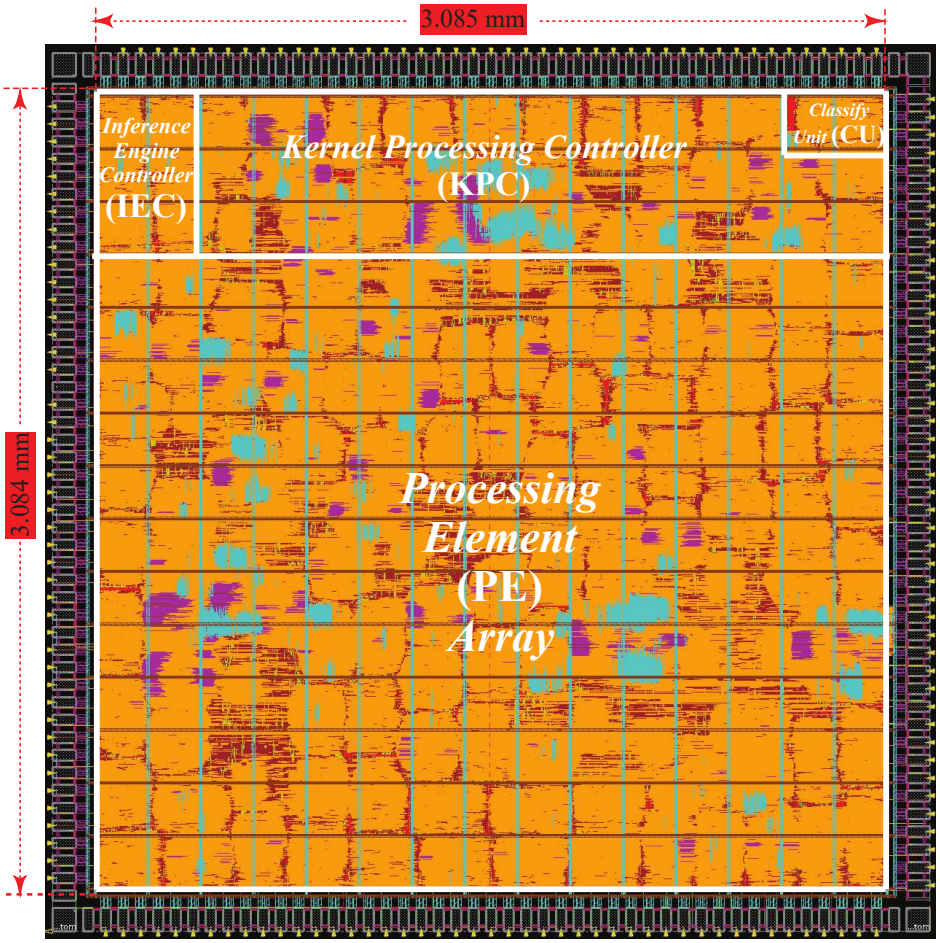


Fig. 4.10: ASIC chip-layout of the proposed inference engine for objection detection application in 28 nm FD-SOI technology node.

4.4.1 Real-World Test Setup for Functional Validation

As discussed in section 4.2.1, an object classification system comprises of the proposed CNN inference engine, on/off-chip processor, and DRAM. For implementing such system, this chapter adopted the Xilinx ZCU102 MPSoC FPGA-board that has a processing system (containing an ARM-cortex processor with six cores), a 4.5 GB DDR4 DRAM, and a user-programmable FPGA fabric inside the same system-on-chip. Here, Fig. 4.11 (a) and (b) show a schematic representation of the test setup and a snapshot of the actual prototype, respectively. Our test setup contains host computer, SD memory card, FPGA-board, and Keysight-16861A 32-channels logic-analyzer, along with connecting probe (Keysight N2140A probe), as shown in Fig. 4.11 (b). Here, the host computer is used for developing the HDL code for suggested hardware architectures, integrating them, synthesizing

and generating the bit-stream. It is also used for creating necessary software to program and operate the FPGA board. Furthermore, host computer imports the CNN model and provides extracted model parameters as well as input image to the FPGA board. For the aforementioned purposes, host computer has been installed with two Xilinx tools: Vivado 2018.2 and SDK 2018.2, and a Python tool (JupyterLab 3.0.14), as shown schematically in Fig. 4.11 (a). Using such Vivado tool, an AXI4-stream compatible IP has been developed from the Verilog HDL code of the proposed inference engine that is interfaced with an IP of a memory-mapped AXI DMA module. Such module is interfaced with one of the accelerated cache-coherent AXI-HP ports of the processing system to form a high band-width link between on-chip processor and inference engine for high speed access of on-board DDR4 DRAM. Configuration bits and status signals of the inference engine are interfaced using the general purpose AXI-lite ports of the processing system with the aid of memory-mapped AXI4 registers. Output of the inference engine, implemented on FPGA board, are also tapped via its peripheral module interface (PMOD) in order to show the classification result on the screen of logic analyzer.

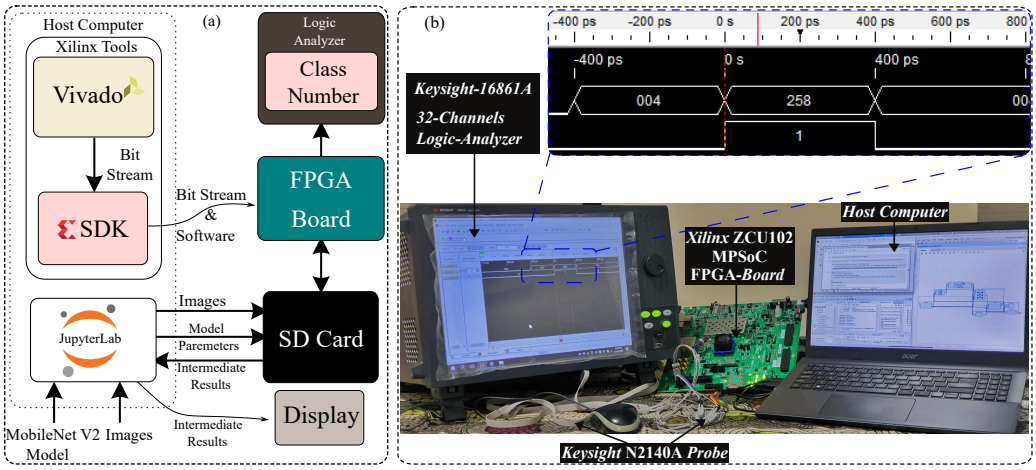


Fig. 4.11: Real-world test setup of object detection system for functional validation of the proposed CNN inference engine.

Furthermore, an interrupt mechanism has been implemented to enable communication between the processing system and the inference engine, allowing the processor to remain free instead of being halted during inference computations. Referring Fig. 4.11 (a), synthesized bit stream of inference engine is exported to Xilinx SDK 2018.2 tool in order to

develop necessary software that programs the FPGA board. Simultaneously, a CNN model has been imported into the Python environment of JupyterLab 3.0.14 tool that optimizes the model and extracts the model parameters (like weights and biases), and stores them as binary files (with '.bin' format) in the SD memory, as presented in Fig. 4.11 (a). Additionally, using the aforementioned Python environment, pixel information of input images are extracted and stored as binary files in the same SD memory card, as illustrated in Fig. 4.11 (a). Following that, such SD memory card is removed from the host computer and installed in the FPGA board. To read these model parameters and input images from the SD memory card in the FPGA board, we have developed a stand-alone software for the FPGA board in the Xilinx SDK tool. When executed in the FPGA board, after reading the data from SD memory card, this stand-alone software loads them in the DDR4 DRAM of FPGA board and also sets the configuration bits for inference engine through the memory-mapped AXI4 registers, connected to the AXI4-lite ports. In order to visualize the intermediate results while processing the computation of a CNN model, instructions have been included in this software to store some of the activations from intermediate layers of model into the SD memory card. One such example has been shown in Fig. 4.12 where the intermediate results from a CNN model are also visualized.

Once the software is ready, FPGA board is connected to the host computer. Subsequently, PMODs of FPGA board are connected to the channels of Keysight-16861A 32-channels logic-analyzer, via Keysight N2140A probe, as shown in Fig. 4.11 (b). With the aid of Xilinx SDK tool in the host computer, FPGA board has been programmed and the compiled file (with '.elf' format) of software has been executed on one of the hexa cores of ARM processor in the processing system of ZCU102 MPSoC FPGA-board. Hence, the final classification results are displayed on the logic analyzer in the form of class number of the detected class (i.e. *CN-DC*). Class labels of the class numbers that are detected for three of the tested images are presented in the rightmost 3×3 matrix in Fig. 4.12, and the class labels for all one-thousand different class numbers for 1000-class ImageNet models are provided here [105]. Furthermore, activations of the intermediate layers are stored in the SD memory card that are later plotted using Python simulation environment in the host computer, as shown in Fig. 4.12.

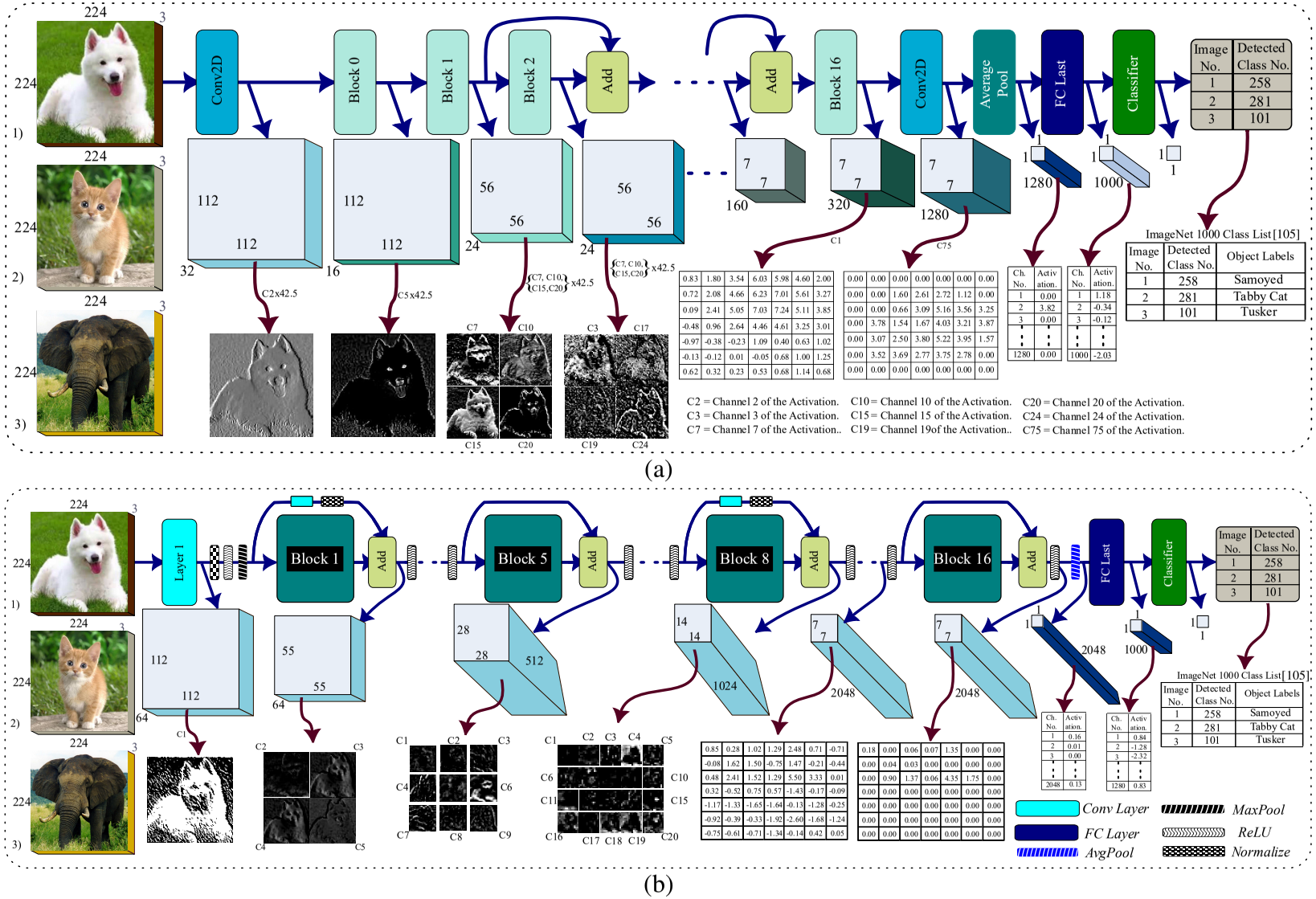


Fig. 4.12: Schematic representations of layer-wise processing of images in the proposed CNN inference engine using (a) MobileNet-V2, and (b) ResNet50 models.

4.4.2 Model Compatibility

As discussed earlier in section 4.3.1.1, the proposed architecture intrinsically supports 1×1, 3×3, 5×5, and 7×7 kernels, as well as all computations for *Conv*, *FC*, *maxpool*, *avg-pool*, *ReLu*, and *ReLu6* layers. Therefore, the proposed architecture can process any state-of-the-art CNN models that use any combination of these kernel sizes and computation layers. Models that have been used to validate the proposed inference engine include VGG-16, ResNet, GoogLeNet, MobileNet, and EfficientNet. Following section discusses the detailed implementation flows for two of the extremely-complex CNN models: MobileNet-V2 and ResNet50.

4.4.3 Validation and Implementation Results

4.4.3.1 MobileNet-V2 Implementation Flow

In MobileNet-V2 model architecture, except first convolution, last convolution, average pool and fully-connected layers, the remaining layers use a Block-based approach with or without residual connections. Here, each Block consists of a 1×1 expansion layer which is followed by ReLU6, a 3×3 depth-wise separable layer that is also followed by ReLU6, and a 1×1 projection layer. Here, 1×1 expansion layer expands the number of channels to multiple fold, before feeding the activation to 3×3 depth-wise separable layer that produces activation with same number of channels and finally, projection layer compresses the number of channels. Thus, if the CNN inference engine processes one complete layer at a given time then it requires massive data movement between off-chip DRAM and on-chip inference engine. Such data movement is comparatively lesser on processing the entire Block together, as the number of channels for input and output of a Block is significantly smaller. Since, PE-array size and memory inside the proposed CNN inference engine is limited to fit all computations and data of a Block of MobileNet V2 model, we have adopted the pipelined approach where the PE array is segregated into three groups. Now, a group of PEs has been configured to perform the computation of 1×1 expansion layer and returns the activations to GBCU of the proposed inference engine (refer Fig. 4.1) that stores them to the line memories of second group in the PE-array of KPU. Simultaneously, such second group of PEs performs the

computation for 3×3 depth-wise separable layer, while the third group of PEs carries out computation for 1×1 projection layer. Finally, GBCU stores the activations of that Block, back to DRAM when all the line memories are occupied. As a result, it significantly reduces the number of off-chip data movements. The number of PEs allotted to each of these groups are decided according to the ratio of their computational load. However, such ratio between these groups varies from Block to Block. Thus, we have also included additional instructions in software to set the configuration bits for dynamically allotting different number of PEs to these groups, according to the variations of computation ratios in different Blocks.

To validate the functionality of the proposed CNN inference-engine, several images have been fed into the FPGA prototype of an object classification system that uses the proposed CNN inference engine. Three of such input images and their classification results are presented in Fig. 4.12 (a). As discussed in section 4.4.1, the final classification result has been displayed on the keysight logic analyzer in the form of *CN-DC*, while activations of the intermediate layers are stored in the SD memory card. However, for better clarity, we have presented detailed view of these layer-wise results for one image in Fig. 4.12 (a) , and the snapshot of its final classification result on the screen of logic analyzer in Fig. 4.11. Nevertheless, final classification result for all three images are included in Fig. 4.12 (a). As discussed in section 4.4.3.1, MobileNet V2 model uses Blocks to combine 1×1 expansion layer, 3×3 depth-wise separable layer, and 1×1 projection layer. Therefore, Fig. 4.12 (a) displays the output of only projection layer, as the final output of a Block, rather than displaying for each layer inside the Block. Here, dimension of each channel of the output activation for Layer-0, Block-0, Block-1, Block-2, and Block-3 are 112×112, 112×112, 56×56, 56×56 and 56×56, respectively. Thus, these activations are too large to visualize as a matrix in Fig. 4.12 (a). Therefore, we plotted them as a single-channel gray-scale images. Furthermore, each convolution and expansion layer use ReLU6 at the output to clip the output values below the numerical value of 6. Thereby, we used a scaling factor of 42.5 in the Python environment to scale up the values before plotting them as grey-scale pixels. Nevertheless, each output channel of Block-13 to Block-16 and the last convolution layer is a 7×7 matrix. Since the dimensions of these channels are small enough to visualize as matrix, Fig. 4.12 (a) displays their values in the form of a 7×7 matrix. Here, each value in the matrix

formed by a channel of the output activation denotes the extracted score for the presence of a feature on the input image that has been searched by a fitter associated with that feature. The activations of average-pool and fully-connected layers are one dimensional array whose preview is presented in Fig. 4.12 (a). Therefore, output of the classification layer is a single value which is the class number of the object that has been detected by the proposed CNN inference-engine from the input image. Hence, the human readable labels that are associated with these class numbers are provided with the ImageNet class-1000 dataset [105].

4.4.3.2 ResNet50 Implementation Flow

Except the first convolution layer and the last FC-layer, remaining computations of the ResNet50 model is distributed over 16 different computational Blocks, with each Block having a residual connection between its input and output. While majority of the residual connections simply add the input of a Block with its output. Some residual connections also have a convolution layer embedded in it (like Blocks 1, 4, 8 and 14). Each of these 16 computational Blocks is made of a 1×1 convolution layer, followed by a 3×3 convolution layer, which is followed by another 1×1 convolution layer. Computation of each of the convolution layers (both inside a Block, outside a Block, and in residual connections) is followed by a normalization and a *ReLU* operation. While computation of *Conv*, *FC*, *ReLU* and *maxpool* operations are handled by dedicated hardware Blocks in the PE array, offset subtraction for the normalization operation here has been achieved by supplying 2's complement form of the offset values through B, Psum_i port of the PE array in Fig. 4.3. Here, scaling operation is performed using the bit shifting approach, similar to the way of average pooling, as discussed in section 4.2.2.

When processing a Block, significant portion of the output feature map (up to 2.25 Mb) is reused within PE array for the subsequent layer. Towards the end of computation of a Block, it adds the output feature map with the output of residual connection. Since the residual connections require whole output feature map of the previous Block/layer, and the output of residual connection is used only after the computation of eight different computations in a Block. Therefore, we copy the final output feature-map of a Block to outside the PE array and hold it until we need to send them back to the PE array for residual additions. Therefore,

in Fig. 4.12 (b), output of a complete Block after the residual addition has been presented rather than showing the output of each layer inside these Blocks.

In the ResNet50 model, dimension of each channel of the output feature map is 112×112 for layer 1. It is 55×55 for Blocks 1−3, 28×28 for Blocks 4−7, 14×14 for Blocks 8−13, and 7×7 for Blocks 14−16. Thus, some channels of the output feature map for layer 1 to Block 16, as a grayscale image, has been shown in Fig. 4.12 (b). Average pool after Block 16 reduces the 7×7 dimension to 1×1. As a result, both input and output feature maps of the FC layer are one dimensional (1D) arrays. A preview of the same is presented in Fig. 4.12 (b). Finally, output of the classification layer is a single value, which is the class number of the object that has been detected by the proposed CNN inference engine from the input image. Human readable labels that are associated with these class numbers are provided with the ImageNet class-1000 dataset [105]. For both models, Fig. 4.12 shows that the detected class numbers of three images along with their object labels. These class numbers produced by the proposed inference engine correctly depicts the object present in the input image, as depicted in Fig. 4.12.

4.5 Summary

This chapter presented the development of a complete CNN inference engine/accelerator that supports all major computational layers typically found in state-of-the-art models, including convolution, activation, pooling, and classification operations. While chapters 2 and 3 were restricted to convolutional computations, chapter 2 introduced an adaptive convolution mapping technique for efficient resource utilization, and chapter 3 focused on high-throughput and energy-efficient architectures for convolution layers. These designs, though effective, were limited in scope to convolution operations and refrained from supporting the full computational pipeline of CNN models. This chapter addressed this limitation by proposing an enhanced and optimized hardware design that enables efficient processing across all components of CNN inference.

The advantage of the proposed architecture lies in its ability to extend local data reuse beyond convolutional layers. Prior accelerators primarily exploited reuse in convolution,

leaving pooling, normalization, activation, and fully connected layers to repeatedly access off-chip memory. The proposed design incorporates a reuse-aware scheduling strategy that allows intermediate results from all layers to be retained and reused locally, thereby reducing memory bandwidth pressure and energy consumption. This holistic reuse approach, combined with lightweight control overhead, explains the significant gains in throughput and efficiency while ensuring negligible accuracy loss compared to FP32 baselines.

Though chapters 2-4 were tailored exclusively for inference tasks, a major objective of the next chapter would be to extend our contributions by addressing the training, with a particular focus on efficient reuse of local data in back propagation. This will lead to a unified accelerator framework capable of efficiently handling both inference and training workloads.

Chapter 5

Unified-CNN Accelerator for Training and Inference

5.1 Introduction

As discussed extensively in chapter 1–4, highly-reliable accuracy of CNN models has paved its way to various AI applications. These CNN models work in two different phases: firstly the training phase where these models learn various statistical parameters which are required for identifying specific object(s) in the input data. In the second phase of inference, such trained model uses the learned parameters from the training process to identify objects in previously unseen data. However, both of these phases require massive and sophisticated computations. Specially the training phase, as it needs the model to perform forward pass on millions of input data, determine the loss, back propagate the gradients in order to learn different statistical features and update them. In addition, it must perform several epochs of such forward and backward passes to achieve the desired level of accuracy. Due to the requirement of such gigantic and sophisticated computations, majority of the CNN models are primarily being trained on extremely power-hungry servers.

On the other side, it is necessary to have high-throughput, hardware-efficient and energy-efficient training accelerators to achieve high-speed training at the edge devices while reducing the carbon footprint of such AI applications (for both training and inference). While many applications rely on cloud-based servers for both training and inference. As discussed

in chapter 2-4, utilizing multiple PEs in parallel allows the CNN accelerator to achieve significant amount of improvement in speed for such AI applications. However, this also leads to increased energy consumption, which remains a critical challenge from an implementation perspective. As discussed in chapters 3 and 4, substantial portion of the total power consumption in CNN computation stems from the frequent and large-scale data transfers between the off-chip DRAM and the processing unit. Notably, such data movement consumes more power than the actual computational operations [74]. Since the data movement accounts for the majority of power consumption, previous chapters focused on maximizing energy efficiency for CNN inference by exploiting full-model data reuse across all operation. Similarly, numerous works from literature have also reported various hardware architectures that are primarily aimed at improving the throughput and energy efficiency of CNN models [62, 74, 90, 94, 95, 97]. However, most of the existing approaches focus solely on the forward-pass inference, limiting their applicability. Nevertheless, it is crucial to develop the CNN-accelerator hardware architecture that efficiently exploits the local data reuse for both forward and backward passes during the training phase.

Even though the primary focus on the literature has been around accelerating the inference operation, attempts have been made to design energy-efficient training accelerators. For example, [106] presented a detailed performance analysis of convolutional and non-convolutional operations in CNN training on ASIC accelerators. Furthermore, [107] introduced on-device CNN training with energy-efficient gradient computations using a grouped PE architecture and masking for activation storage reduction. In addition, [108] introduced a balanced resource utilization between convolutional and fully connected layers, to improve training efficiency. The work from [109] implemented a shared MAC unit to improve energy efficiency of CNN training process. However, such work excluded the full exploitation of local data reuse to enhance energy efficiency. On the other hand, a recent work from [110] presented a new architecture that interleaved gradients of both weights and activations for MLP models to enhance the energy efficiency. This work has been extended in [111] where it supports the training of CNN models. However, these works did not explore the forward pass, rather they are limited to gradient computation for backward pass only.

In order to bridge this research gap that circumvents the tradeoff challenge between

energy efficiency and achievable throughput of CNN accelerator, this chapter presents a unified-CNN accelerator that maximizes local data reuse in both forward and backward passes. By efficiently managing data movement, the proposed architecture minimizes interruptions and ensures to sustain high-throughput. A dedicated gradient compute unit (GCU) optimizes backward-pass computations by efficiently mapping large kernels and reusing loss values to enhance gradient filter weight ($G.W$) and activation gradient ($G.I$) computations, reducing redundancy and improving efficiency. The accelerator is implemented on an FPGA platform and validated using standard CNN models in a real-world test setup, demonstrating its feasibility and correctness. Key contributions of this chapter are summarized as follows:

- **Unified-CNN Accelerator with Maximal Data Reuse:** Proposes a novel accelerator that unifies both forward and backward passes of CNN models while maximizing local data reuse. This reduces redundant memory access and enhances computational efficiency.
- **Optimized Backward Pass with Efficient Gradient Computation:** Introduces a dedicated GCU architecture to efficiently handle large kernel computations in the backward pass. Additionally, it enhances the computation of $G.W$ and $G.I$ by effectively reusing loss values.
- **Hardware Implementation and Real-world Validation:** Demonstrates the practicality of the proposed accelerator through FPGA-based implementation. The hardware prototype is functionally verified using standard CNN models in a real-world test setup to assess performance and correctness.

This chapter is structured as follows. Section 5.2 covers the mathematical foundations of CNN inference and training, along with a detailed discussion of the research challenges addressed in this chapter. Thereafter, section 5.3 provides an overview of the system-level design, proposed technique, and novel VLSI architectures for the unified-CNN accelerator. Following that, section 5.4 presents the implementation results, discussions, comparisons, and validation. Finally, section 5.5 summarizes the chapter.

5.2 Mathematical Prerequisite and Research Problem

5.2.1 Mathematical Backgrounds of CNN Inference and Training

As discussed extensively in chapter 1, the primary computation carried out during the forward pass (inference phase) of a CNN models is the feature search, wherein the presence of millions to several billions of trained features are searched in the input data, over different model layers. Subsequently, such feature scores are used for identifying particular object in the input data. Computation of such feature search operation is conventionally referred as *Conv* operation and it is mathematically expressed as

$$AC_{x,y,z,\delta} = \sum_{a=1}^{\alpha} \sum_{b=1}^{\beta} \sum_{c=1}^{\gamma} \left(W_{a,b,c,\delta} \times I_{xx,yy,zz} \right) + B_{\delta}. \tag{5.1}$$

Along with the forward pass, another major computation involved in the training phase is the calculation of gradients for filter weights ($G.W$), gradients for bias values ($G.B$), and gradients for input feature map ($G.I$) of each layer. Computations of $G.W$ and $G.B$ of all filter weights and biases are necessary to update them during each pass. Since input feature map (I) is the bridge between two subsequent layers, computation of $G.I$ for each activation are also necessary to back propagate the loss, till the first layer of CNN model.

Computation of the gradients of such input feature map can be expressed as

$$G.I_{x,y,z} = \sum_{a=1}^{\alpha} \sum_{b=1}^{\beta} \sum_{c=1}^{\gamma} \left(W_{\alpha\text{-}a,\beta\text{-}b,\gamma\text{-}c,\delta} \times L_{xx,yy,zz} \right). \tag{5.2}$$

Similarly, computation of the gradient of a filter bias value can be expressed as

$$G.B_{\delta} = \sum_{x=1}^{\hat{M}} \sum_{y=1}^{\hat{N}} \sum_{z=1}^{\hat{O}} L_{x,y,z,\delta}. \tag{5.3}$$

Likewise, computation of the gradients of filter weights can be expressed as

$$G.W_{\alpha,\beta,\gamma,\delta} = \sum_{x=1}^{\hat{M}} \sum_{y=1}^{\hat{N}} \left(L_{x,y,\gamma,\delta} \times I_{x+\alpha\text{-}1,y+\beta\text{-}1,\gamma,\delta} \right) \tag{5.4}$$

where L is the loss matrix corresponding to a 3D filter for the current layer. Here, $\hat{M}, \hat{N}$ and $\hat{O}$ are the sizes of three dimensions of such 3D loss matrix (L). For the last layer of the model, L is calculated as the difference between the outcome of model and the expected (actual/correct) output of the last layer. Unlike, for the other layers, it is the $G.I$ of subsequent layer. Eventually, weights and bias values are updated using (5.5) and (5.6), respectively, as shown below.

$$W_{\alpha,\beta,\gamma,\delta} = W_{\alpha,\beta,\gamma,\delta} - L.R \times G.W_{\alpha,\beta,\gamma,\delta}, \tag{5.5}$$

$$B_{\delta} = B_{\delta} - L.R \times G.B_{\delta} \tag{5.6}$$

where $L.R$ stands for the learning rate of the training process. Following that, the gradients of input feature map $G.I$ for each layer is back propagated to previous layer as the loss matrix L. Henceforth, computations based on these equations continue for each layer until all filter weights and bias values of each of the layers are updated.

5.2.2 Research Problem

Referring mathematical equations (5.1), (5.2) and (5.4), the computation of forward pass as well as calculations of both $G.I$ and $G.W$ are analogous. However, the CNN accelerators designed to process fixed set of smaller kernel sizes face severe inefficiency to map the computation of gradient calculations [75, 76, 86]. Specially, the most important one is computation of $G.W$, due to huge variation in the dimensions of I and L for different layers. For example, in CNN model like VGG-16 [34], dimensions of both I and L are 114×114×64 and 112×112×128 for layer-2, 58×58×128 and 56×56×256 for layer-5, 16×16×512 and 14×14×512 for layer-11, respectively. This is the trend for majority of CNN models. Therefore, conventional CNN accelerators that are designed for inference with fixed or small range of kernel dimensions are immensely inefficient to compute the gradients for filter weights (i.e. $G.W$).

Furthermore, as discussed extensively in chapters 2-4, energy consumption in the com-

putation of such CNN models is profoundly contributed by the data movement, several local reuse techniques like weight stationary, output stationary and row stationary approaches have been used to reduce a number of off-chip accesses, and maximize the local reuse [63,74]. However, each of these techniques requires extra hardware for complex routing and additional memory blocks. Moreover, these architectures are limited to only inference with a smaller dimension of kernels. To circumvent aforementioned issues, this chapter presents a new unified architecture of CNN accelerator extending the line stationary approach from chapter 3 and 4 to maximizes the reuse of local data during (i) forward pass of both inference and training, and (ii) backward pass of training. Furthermore, the proposed architecture of unified-CNN accelerator is capable of efficiently handling the mapping of large kernels for the computation of $G.W$.

5.3 Proposed Technique and Hardware Architectures

5.3.1 System Model

A system-level overview of the proposed CNN engine for inference and training has been presented in Fig. 5.1. Similar to the system models of chapter 3 & 4, this also consists of three major blocks: off-chip DRAM, off/on-chip software processor, and the suggested unified-CNN accelerator. Similar to chapter 4, this accelerator also has been interfaced with on/off-chip software processor via high-speed links like CXL or PCIe or AXI buses for transferring the data between CNN-accelerator and DRAM, as shown in Fig. 5.1. In this system, during the inference/forward pass, the software processor acts like a master controller that loads the model parameters from off-chip storage, takes the input data from external sources and stores them in DRAM. It also provides the necessary configuration signals to the unified-CNN accelerator. In the similar way, during the training/backward pass, such software processor loads the training data from external non-volatile memory to DRAM in batch-wise fashion. It also generates necessary configuration signals to the unified-CNN accelerator for other vital operations.

In this CNN engine, major role of the proposed unified CNN-accelerator is to perform

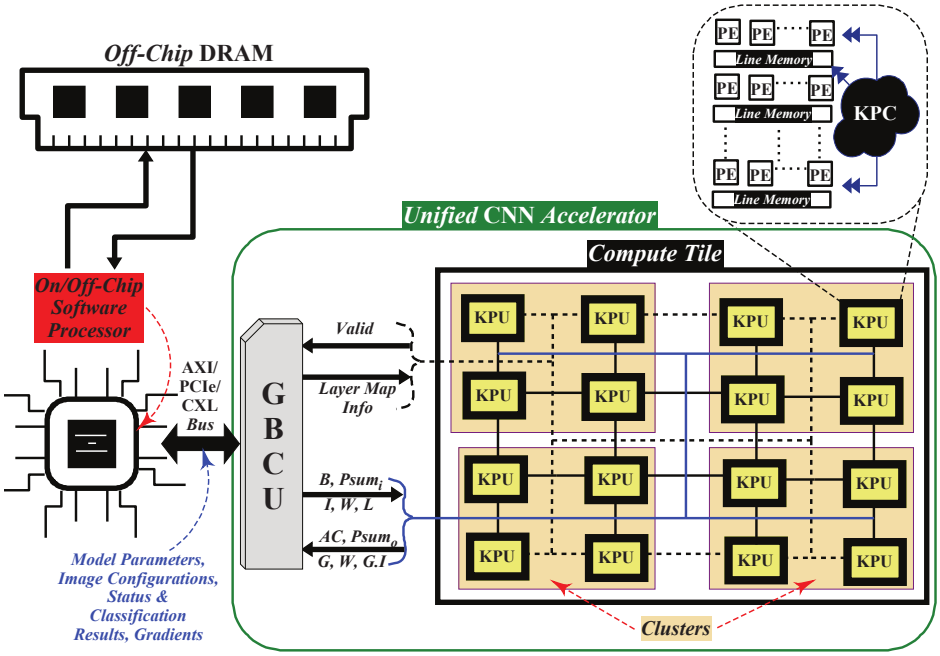


Fig. 5.1: Overall system-level design of the proposed CNN engine for inference and training of CNN models.

dedicated mathematical computations for the feature search and the gradient calculation during inference and training processes, respectively. Since the CNN algorithm can reuse a significant portion of data, our CNN accelerator has been designed with two major blocks: a GBCU and a compute tile (CT), as shown in Fig. 5.1. Since the size of kernels varies over a wide range during the inference and training processes, the CT architecture has been designed with multiple clusters of KPU, as presented in Fig. 5.1. Each KPU contains a PE array, multiple line memories, and KPC. The software processor manages the operation of GBCU which in turn manages the flow of data like I, W, B, AC, $Psum$, $G.W$ and $G.I$. Furthermore, this processor also manages every other configuration and control signals between DRAM and CT, as well as buffers some amount of data within the buffer memory of GBCU. To begin with the computations for inference or training pass, software processor provides layer information and data to the GBCU. Following that, GBCU configures CT according to the layer information and the kernel size. Depending on the kernel size, GBCU either maps it into one cluster of KPU, thus processing multiple kernels in parallel using all of the different clusters of KPU, or splits the computation of large kernel among multiple KPUs. Finally, GBCU either stores the computed AC or G within its buffer memory if they can be

reused soon; otherwise, transfers them to DRAM .

5.3.2 Proposed Technique

The proposed technique for unified CNN-accelerator has been presented in Algorithm 3. In addition, we have empirically determined that the overall efficiency of KPU degrades for very large size of PE array. A detailed discussion on the impact of varying size of PE array on the efficiency of KPU has been presented in section 5.3.4. Therefore, number of PEs in such KPUs are not sufficient to accommodate the large kernels during the calculation of $G.W$ in some of the initial layers of large-input CNN models. Hence, instead of mapping the computation of $G.W$ in the KPU in one-go for parallel processing, this chapter combines the computations of both $G.W$ and $G.I$ in the KPU. Consequently, this exploits the reuse of L data for the calculations of both (5.2) and (5.4), as well as avoids hardware inefficiency. A comprehensive discussion on the architectural design of such approach for computing $G.W$ and $G.I$ has been carried out in section 5.3.3.4. However, for the substantial filter kernels in (5.1) and (5.2), their computations are segregated and mapped on multiple KPUs, referring line numbers 2-5 of Algorithm 3. Here, multiple smaller kernels are mapped in a single KPU whenever possible, referring line numbers 6-7 of Algorithm-3. On the other side, line numbers 9-43 of Algorithm-3 brings the uninterrupted processing technique from Algorithm 1 and 2, enhancing the computational throughput Θ_T of KPU by minimizing the interruption due to data fetching. The key idea in such uninterrupted processing is to simultaneously perform both fetching F_i and computation P_i events, rather than sequentially. A comparative depiction of such processing timeline with respect to conventional architectures has been provided earlier in Fig. 3.2 of Chapter 3.

5.3.3 Proposed Hardware Architectures

5.3.3.1 Energy-Efficient Micro-Architecture of KPU

As discussed earlier in section 5.3.1 of this chapter, CT of the unified CNN accelerator comprises of multiple KPU clusters. These clusters and GBCU are organized in a mesh network wherein each KPU receives the $Psum$ from other KPU in CT, as configured by GBCU.

Algorithm 3 Proposed Implementation-Friendly Algorithm for the Unified CNN-Accelerator Design

1: Determine M and N ▷ M and N denote the number of *PEs* in a *KPU* and number of elements in the kernel which is to be processed
2: **if** $N>M$ **then**
3: Find κ: $\kappa\times M\geq N$ & $(\kappa\text{-}1)\times M<N$
4: Split the kernel to make κ smaller kernels with number of element $\leq M$
5: Map the computation of the kernel to κ numbers of *KPUs*.
6: **else**
7: Map τ kernels in a *KPU* : $\tau\times N\leq M$ & $(\tau+1)N>M$.
8: **for** computation in each *KPU* **do**
9: Determine n and r; ▷ n, and r are number of iterations to be performed and the least number of data that need to be fetched to begin the computation, respectively.
10: Initialize: i=1, j=0; ▷ i, j record the count of completed iteration, and the count of fetched data, respectively.
11: Begin $F_{i=1}$; ▷ Start pre-fetching of data for $itr_{i=1}$.
12: **while** $i \leq n$ **do**
13: **if** $j \leq r$ **then** ▷ Adequate data has not been fetched yet.
14: Continue $F_{i=1}$; ▷ Fetching continues for $itr_{i=1}$.
15: $j = j+1$; ▷ Count the number of data pre-fetched for $itr_{i=1}$.
16: Return to Step 13; ▷ Fetching continues for $itr_{i=1}$.
17: **else** ▷ Adequate data fetched to begin computation.
18: **Process** P_i; ▷ Compute AC_i.
19: **if** F_i is not complete **then**
20: Continue F_i;
21: **else**
22: **if** $i < n$ **then**
23: Begin F_{i+1}; ▷ Start pre-fetching data for itr_{i+1}.
24: **else**
25: set j = 0
26: Begin $F_{i=0}$; ▷ Start pre-fetching data for $itr_{i=0}$.
27: **if** P_i is Complete **then**
28: **if** $i = n$ **then**
29: Set $i = 0$;
30: Return to Step 9. ▷ Start processing the next layer.
31: **else**
32: Set $i = i + 1$
33: Return to Step 18. ▷ Start computation for itr_{i+1}.
34: **else**
35: Return to Step 18 ▷ Continue P_i.

This section presents the proposed KPU architecture for unified-CNN accelerator based on Algorithm 3. As mentioned earlier in section 5.2.2, CNN accelerator must minimize the number of off-chip accesses and maximize the re-use of on-chip local data to enhance the energy efficiency. In addition, such accelerators need to avoid interruptions in the processing due to data movement in order to achieve high-speed computations. Moreover, an efficient unified-CNN accelerator shall maintain these features during both, inference and training. Referring (5.1) and (5.2) for a CNN model, dimension of I during the forward pass (for both inference and training) as well as L are identical. They also perform *Conv* operation with same filters of same dimension and depth, both during feature search in forward pass, and computation of $G.I$ during backward pass, except W are 180° rotated for the computation of $G.I$ during backward pass. Furthermore, during the backward pass from (5.4), same L needs to perform *Conv* operation with I that had been used earlier during the forward pass to calculate $G.W$. While reusing W and I values during forward pass, we need to exploit resuse of the same L value for the computation of both (5.2) and (5.4) to achieve maximum reuse of local data during backward pass. To cater this notion, the proposed KPU architecture has been designed using an approach where during the forward pass, I and W values remain stationary inside the KPU, and $Psum$ is shared among multiple KPUs, as shown in Fig. 5.2. Similarly, during the backward pass, L and W values remain stationary inside the KPU, and $Psum$ is shared among multiple KPUs.

Similar to chapter 4, proposed KPU also uses a specially adapted $m \times n$-array of PEs, along with m number of line memories to store I or L value and provide the same to PEs, while facilitating its local reuse. To achieve these features, suggested KPU uses a specialized routing network managed by the KPC, as presented in Fig. 5.2. Such routing network consists of n vertical data-buses. Each of them are connected to Ix ports of m PEs in a column and one of the o ports in same column of m line memories, as shown in Fig. 5.2. As a result, each line memory at any moment can produce n data, making $m \times n$ parallel data available to n vertical buses for Ix ports of PEs. Therefore, whenever KPC configures via the $Ln.S.C$, any PE in the KPU can receive data from any line-memory in the KPU, as illustrated in Fig. 5.2. Thus, KPU achieves vertical strides using these $Ln.S.C$ bits in association with the $Nxt.S$ signal. Comprehensive explanation of these routing processes using $Ln.S.C$ bit has

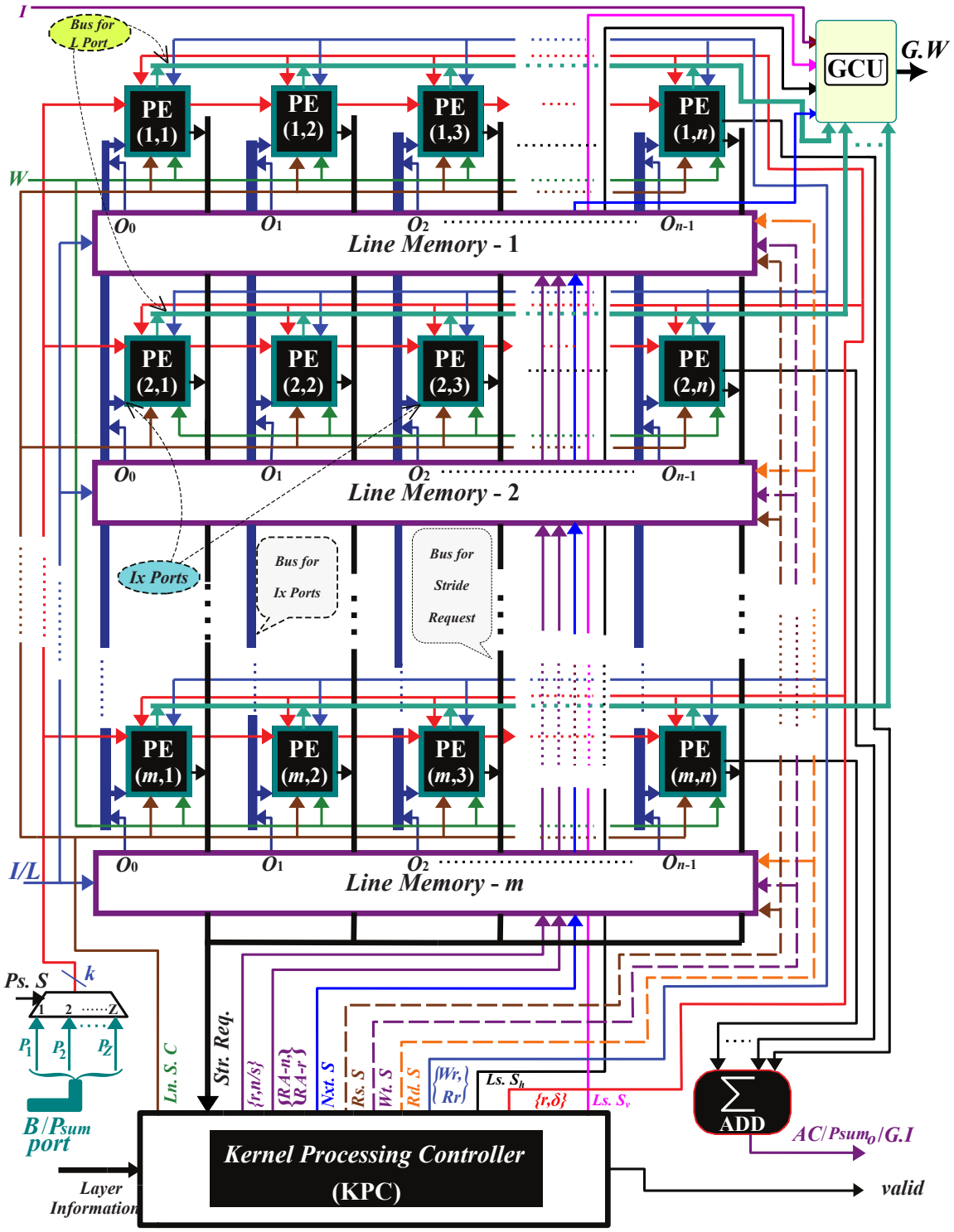


Fig. 5.2: Proposed energy-efficient micro-architecture of KPU.

been presented further in section 5.3.3.2, while discussing the PE architecture. Furthermore, KPC carries out the horizontal stride with the aid of *Nxt.S*, *RA-n*, and *RA-r* signals. More details about these signals have been covered in section 5.3.3.3.

During the backward pass, to reuse the elements of L matrix for the calculation of $G.W$, gradient calculate unit (GCU) for the activation has been incorporated in the proposed KPU architecture, as shown in Fig. 5.2. It shows that by using *Bus for L port* in row-wise manner, the same L values used by PEs are also simultaneously fed to GCU to perform *Conv* operation with I data to calculate $G.W$. Here, KPC manages the operation of GCU using three control signals: horizontal loss selector ($Ls.S_h$), vertical loss selector ($Ls.S_v$), and $Nxt.S$ signals. A detailed discussion on the reuse of L in the computation of $Psum$ for $G.W$ in the GCU has been presented in section 5.3.3.4.

5.3.3.2 Micro-architecture of PE

Since the primary focus of this KPU is performing feature search using (5.1) and gradient calculation using (5.2) and (5.4), hence, the proposed PE architecture has been designed using a MAC unit, a kernel memory (KM), two address-generation units: AGU1 and AGU2 for write and read operations, respectively, an input selection switch (ISS), and an IDM module, as shown in Fig. 5.3. Based on earlier discussion in section 5.3.3, both I or L and W values are kept stationary within the suggested KPU architecture. To cater this, suggested architecture of PE has been designed to hold the kernel values stationary within its KM for the whole iteration; whereas, I or L (loss value L during backward pass) is fed from the line memories. Since CNN models apply multiple kernels in each I or L value, KM stores up-to z kernel-values (W). Note that KPC manages the operation of PEs using the write (Wr) and read (Rr) signals, as shown in Fig. 5.2 and 5.3. By setting the value of Wr = high, whenever a PE is selected to write the incoming W values, AGU1 sequentially generates the write locations for such W values as well as it provides the count of W values stored in KM to IDM module, as shown in Fig. 5.3. Similarly, whenever PE is selected for MAC operation by setting Rr = high, AGU2 generates one read address in every clock cycle for KM to produce kernel element for the MAC unit. Here, the operation of AGU2 has been locally supervised by IDM module, which stops AGU2 from generating new W values from KM to

MAC unit, until adequate (i.e. r) number of kernel elements are stored in KM.

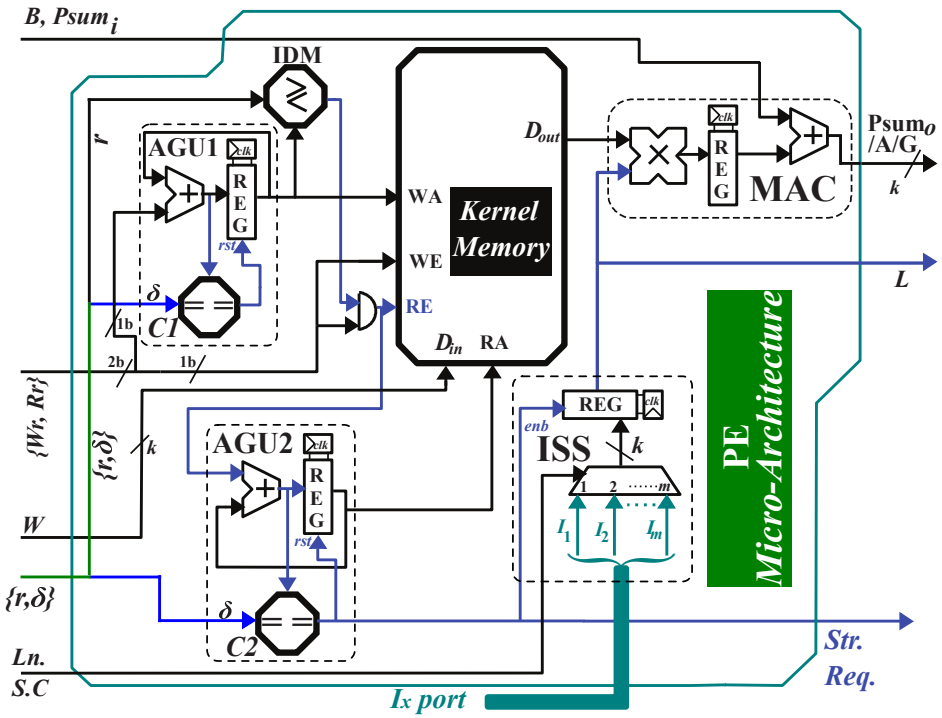


Fig. 5.3: Proposed micro-architecture of PE that is used in the design of PE-array for KPU.

Furthermore, *Ln.S.C* signal from KPC also manages the operation of ISS to select I or L from one of the m different line memories. Every time ISS selects an I or L value, it remains stationary in PE for δ clock cycles, whereas AGU1 generates δ number of W values. Subsequently, after every δ clock cycles, AGU2 also generates a stride request (i.e. *Str.Req*) signal requesting KPC to provide next value of I or L from line memory for a horizontal or vertical stride, as depicted in Fig 5.2. In addition, *Str.Req* signal activates the output register of ISS to store the newly incoming values of I or L for the next δ clock cycles and resets the read location of KM. Thus, PE applies δ number of kernels on each element of input I or L to calculate output activation (AC) or $G.I$, respectively, using (5.1) and (5.2). Till this instant, the operation of PE is identical for both forward and backward passes. However, as discussed in section 5.3.3.1, L values used by the PEs in KPU are also fed to GCU to be reused for the computation of $G.W$. Therefore, our PE shares currently selected value of L from the ISS through L port, as shown in Fig. 5.3.

5.3.3.3 Micro-architecture of Line Memory

Note that the operation of line memory is identical for both forward and backward passes. Nonetheless, during the forward pass, input feature map data I, and during backward pass, loss matrix data L is fed through I port of the line memory. Thus holding them throughout $\alpha - s$ numbers of vertical strides. Such architecture of the line memory has been extensively discussed in section 4.3.1.2 of chapter 4. Hence, it is also used in the design of the unified CNN accelerator presented in this chapter.

5.3.3.4 Micro-architecture of Gradient Calculate Unit

Dimensions of $G.I$ and I in (5.2) and (5.4), respectively, are identical for a *Conv* layer. Thus, after performing necessary padding, number of horizontal strides (say ξ) in (5.2) is equal to the number of horizontal $Psum$ required for each value of $G.W$ in (5.4). Similarly, number of vertical strides in (5.2) is equal to the number of such row of $Psum$ required for each value of $G.W$ in (5.4). However, computing $G.W$ for a filter with width (α) and height (β), it requires α copies of similar horizontal $Psum$, with L shifted by s between two consecutive such batch of ξ number of horizontal $Psum$. Same applies for the vertical $Psum$ as well. Hence, during horizontal strides of (5.4), each value of I needs to be multiplied with α number of L values from a row of the loss matrix L. This must be repeated for β number of subsequent (separated by s) rows of L for β number of vertical strides. Therefore, the proposed GCU architecture has been designed with m number of horizontal gradient-compute units (HGUs), as shown in Fig. 5.4 (a). Furthermore, the micro-architecture of HGU has been presented in Fig. 5.4 (b). Since each value of I needs to be used for β number of rows of L, thus each of the m number of HGU simultaneously receives the same I value. However, depending on the value of β, KPC activates β number of such HGUs, using $Ls.S_v$ signal, as illustrated in Fig. 5.2.

The suggested micro-architecture of HGU consists of one multiplier, n accumulators (ACCs), one input buffer (i.e. REG), one n:1 multiplexer, and an address decoder. As mentioned in section 5.3.3.2, PEs apply δ number of filter kernels in each value of I or L. Thus, each such value remains in REG of ISS (in PE architecture of Fig. 5.3) for δ clock

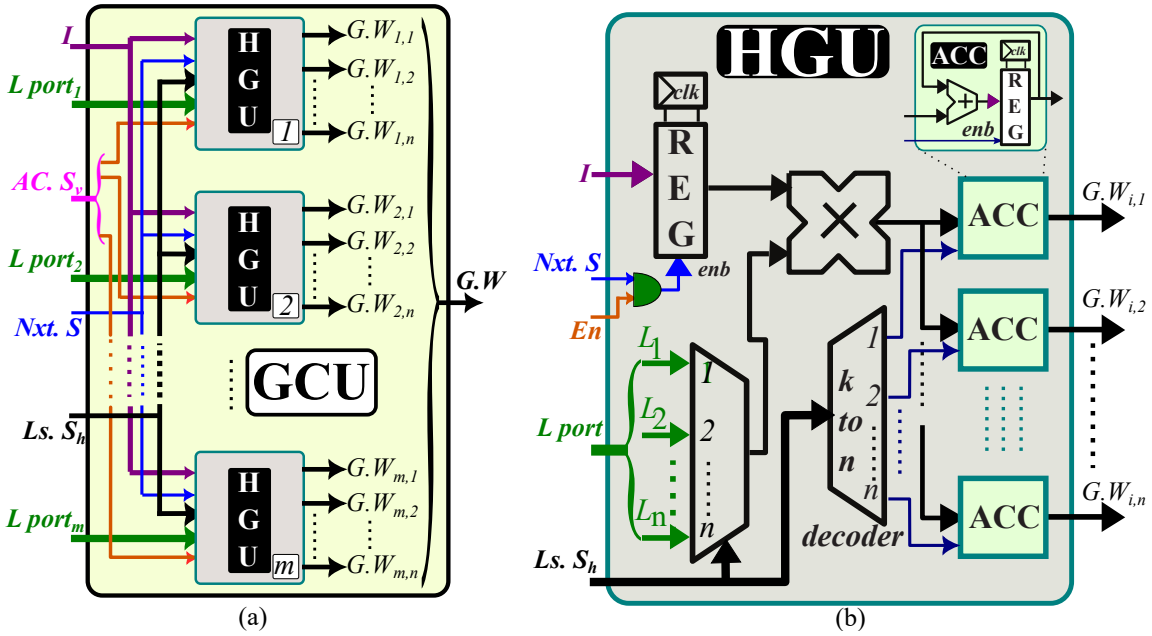


Fig. 5.4: Proposed (a) architecture of GCU and (b) micro-architecture of HCU.

cycles. Therefore, when enabled by $Ls.S_v$ signal, every time HGU receives a new value of I, it buffers the same in REG for δ clock cycles until next horizontal stride (when $Nxt.S \rightarrow$ high). Since each value of I needs to be multiplied with α number of L values from a row of L matrix, each of the HGUs receives n number of L values through L *port*, available in REG of the ISS module of each PE in a row, as shown in Fig. 5.2 and Fig. 5.4. Though each of the n number of L values are available at the L *port* of the HGU, it takes one of the L values in every clock cycle (decided by $Ls.S_h$ signal) and multiplies this value with the I value stored in REG. Subsequently, the results of such multiplication are fed to one of the n ACCs that is activated by the decoded address from $Ls.S_h$, instead of rushing and processing all of them together. The reason being, contents of both L *port* and REG of HGU will remain unchanged for δ clock cycles (until $Nxt.S \rightarrow$ high again), and conventionally $\alpha<\delta$. Further, each ACC continuously adds and accumulates every $Psum$ that it receives until the end of all $\xi \times \xi$ vertical and horizontal strides for a single loss matrix. Eventually, producing α columns and β rows of the $\alpha \times \beta$ sized $G.W$. In this way, such loss (L) matrix has been simultaneously used by both (5.2) and (5.4) without being fetched from the off-chip memory twice.

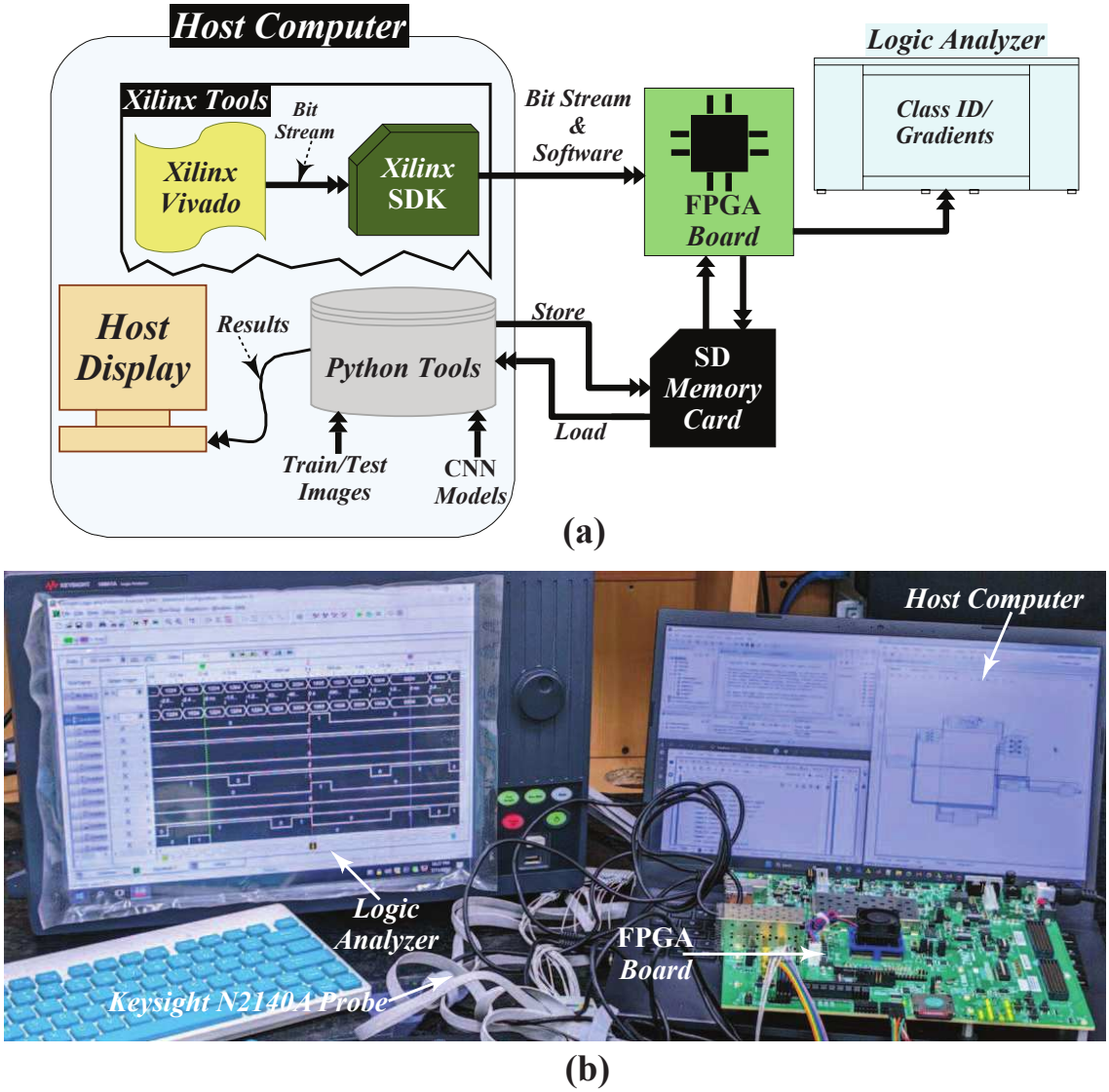


Fig. 5.5: (a) Schematic and (b) real-world snapshot of the test setup for the hardware validation of the proposed unified-CNN accelerator for inference and training.

5.3.4 Effect of PE-array Size on Efficiency

In the proposed KPU architecture, factors that depend on the size of m and n of the $m \times n$ PE-array are (i) complexity of *ISS* inside each PE, (ii) complexity of the KPU routing network, (iii) time required for the data movement between line memory and PE, and (iv) overall complexity of the KPC module. The reason being, larger is the size of m, more number of I values will be placed in each vertical bus for the Ix port. Thus, the size of such vertical bus must be wider for such Ix port. Therefore, complexity of *ISS* surges to select the data from a larger bus. Similarly, the distance between PE and line-memory, placed

farthest at the opposite direction, also increases. Due to such increase in distance as well as complexity of *ISS*, the speed of data movement enhances between line-memory and PE. Furthermore, with the increase in the size of n, complexities of *Data Router*, *Output Buffer*, and data storage (i.e. $K{\times}A$ *Sized Memory*) of the line-memory architecture increases, referring Fig. 4.5 from earlier chapter 4. As a result, the speed of data movement between line-memory and PE is adversely impacted. On the other side, throughput (Θ_T) is directly proportional to the number of PEs in KPU (N_{PE}), i.e. $\Theta_T \propto N_{PE}$ where $N_{PE} = m{\times}n$. Therefore, this is an optimization problem that trades the size of $m{\times}n$ with achievable throughput, hardware complexity, and data-movement latency.

5.4 Hardware Implementation, Validation and Comparison

Note that the proposed unified-CNN accelerator is independent of bit precision. However, back-propagation and gradient-descent processes use very small magnitudes of gradients. Thereby, the suggested accelerator has been implemented using BF16 bit format for achieving higher dynamic range with optimum hardware efficiency [112]. In this section, we present the hardware implementation of our unified-CNN accelerator and its validation with the aid of real-world test setup. Eventually, the implementation results are compared with the reported implementations in literature.

5.4.1 Test Setup for Hardware Validation

Schematic representation of the test setup for validating the hardware prototype of CNN engine that includes the proposed unified-CNN accelerator is presented in Fig. 5.5 (a). As discussed earlier in section 5.3.1, a software processor has been used for loading and storing the data between DRAM and CNN-accelerator. In addition, such processor provides configuration signals to the unified-CNN accelerator for other vital operations. For validating the functionality of suggested unified-CNN accelerator, we have used ZCU102 Zynq-UltraScale+ MPSoC FPGA-board which has an on-chip hexa-core ARM processor along

with the programmable FPGA fabric in the same SoC, and a large on-board DDR4 DRAM. Furthermore, other components used in this test setup are (i) a host computer, (ii) a memory card (SDHC), (iii) a 32-channels logic-analyzer (Keysight-16861A), and (iv) a connecting probe (Keysight N2140A), as presented in Fig. 5.5. Here, the snapshots of the real-world test setups for validating inference and training operations are shown in Fig. 5.5 (b). Rest of the interface of this test setup is similar to the one presented in section 4.4.1 of chapter 4. Here, only the software program, step by step process and the type of data differs from chapter 4.

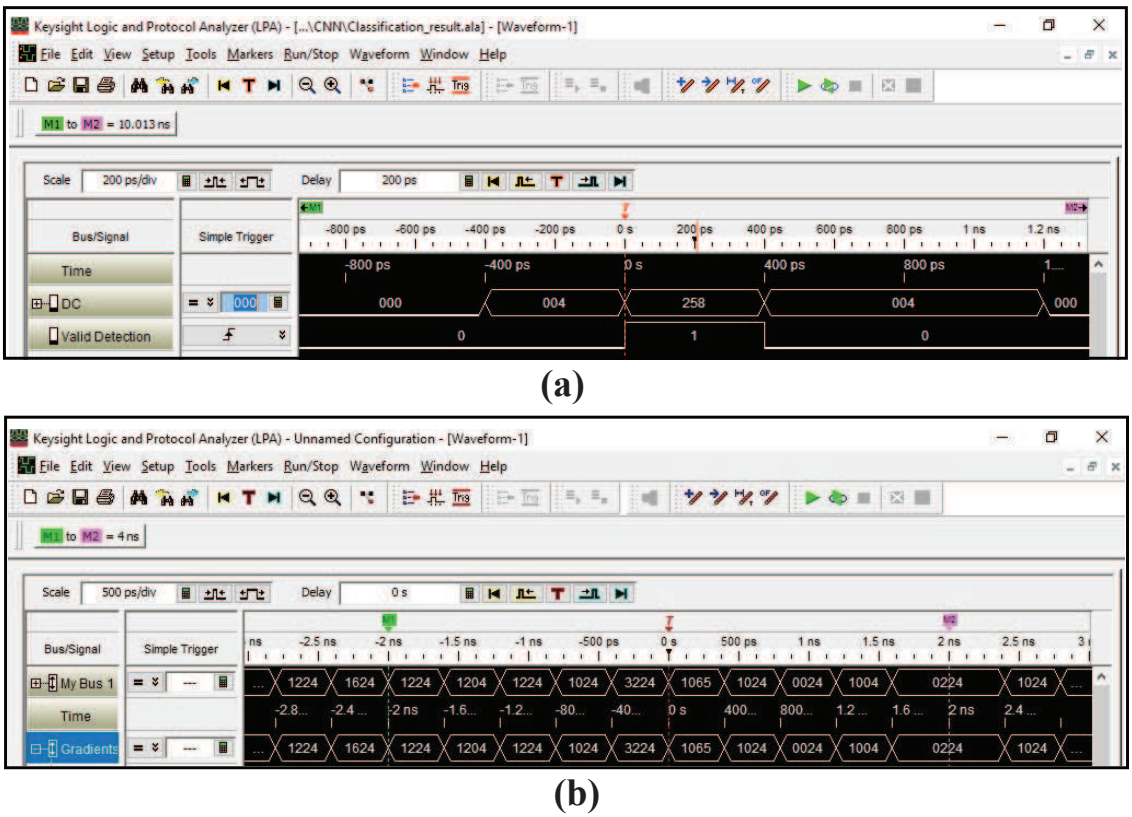


Fig. 5.6: Measured outputs from the FPGA prototype of the proposed unified-CNN accelerator for (a) inference and (b) training processes.

In order to perform inference with our CNN engine with the aid of Python tool in the host computer, a CNN model has been imported, optimized, and its parameters (like weights and biases) are extracted and stored in the memory card. Input images are also saved as binary files in this memory card which is then inserted in the FPGA board. To access these stored model parameters as well as input images and configure the system, we have developed a stand-alone software application using the Xilinx SDK tool. On the FPGA board, this

software reads the data from memory card and transfers the same to DDR4 DRAM, and it also sets the configuration bits according to the layer specifications. Towards the end of inference process, results of the classification are displayed on screen of logic analyzer in the form of class ID of the detected object, as shown in Fig. 5.6 (a). For better understanding, layer-wise processing of images are shown in Fig. 5.7 and its detailed discussion for the implementation of VGG16 model for the object recognition is covered in section 5.4.2.

On the other hand, to perform training with the CNN engine, training images from a dataset are imported using Python tool in the host computer. They are organized batch-wise and stored in the memory card as binary files, along with their labels. Description of the model which is to be trained is also exported in the host computer and stored in the memory card, which is then inserted in the FPGA board. Measured outputs (displayed on the logic analyzer screen) of the training process that are generated by the proposed unified-CNN accelerator - implemented on the FPGA platform - has been presented in Fig. 5.6 (b). Rest of the data flow has been managed by the software that is executed by on-chip software processor of the FPGA board. Here, Fig. 5.8 (a) shows the schematic representation of a custom made CNN model that has been used in this chapter to validate the training functionality of the proposed unified-CNN accelerator. Furthermore, Fig. 5.8 (b) shows the layer-wise processing for gradient calculation during such training process. A comprehensive discussion regarding the implementation of training for the aforementioned model has been presented in section 5.4.3.

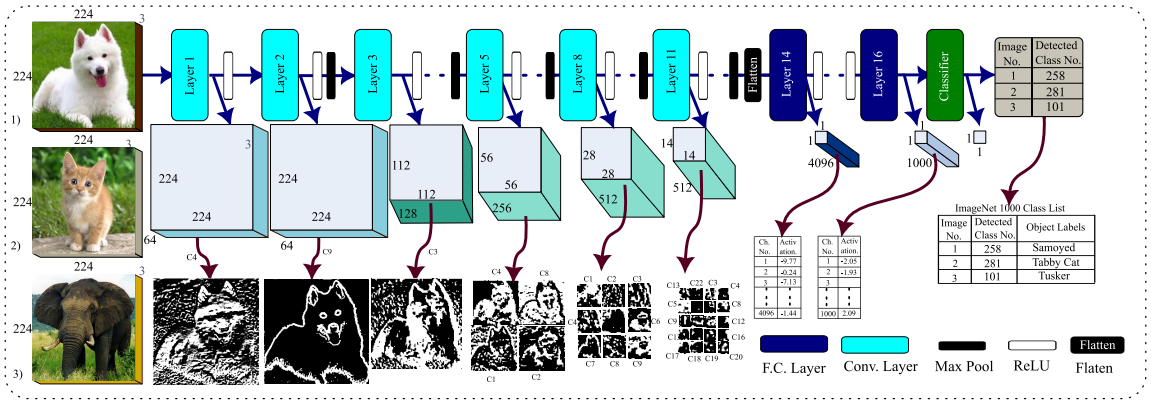


Fig. 5.7: Schematic representations of layer-wise processing of images in the proposed unified-CNN accelerator during the forward pass of inference using VGG-16 CNN-model.

5.4.2 Inference Implementation for Hardware Validation

To demonstrate inference capability of the proposed unified-CNN accelerator, inference with multiple CNN models has been carried out where a number of test images are processed through the FPGA prototype of the proposed unified-CNN engine. One such example where the classification outcomes for three sample images classified by the VGG-16 CNN-model, processed in our CNN engine, are presented in Fig. 5.7. As described in section 5.4.1, the final classification result is displayed on the screen of logic analyzer in the form of a class ID, while the activations from intermediate layers are stored in memory card. For clarity, a detailed layer-wise analysis of one sample image has been illustrated in Fig. 5.7, along with a snapshot of its final classification output on the logic analyzer screen in Fig. 5.5 (b). Furthermore, the classification results for all three test images are summarized in Fig. 5.7.

Here, the VGG-16 model uses linear layer connections where the activation AC of each layer is treated as the input I of immediate subsequent-layer. Therefore, it does not require complete off-loading of the activation of any layer, outside the PE array. However, we have completely off-loaded the output of some layers in order to store and visualize them, as shown in Fig. 5.7. When switching between the layers, this chapter reuses as much activation that can be stored in all the line memories within PE-array and in the storage of GBCU. Since these data have been generated towards the end of processing the previous layer, and are being utilized at the beginning of processing the subsequent layer, orientation of data processing thus flips between two subsequent layers. For this model, lateral and vertical dimensions of the output data does not change until a pool of operations have been used. Therefore, for layers 1 and 2, dimension of each channel of the output feature map is 224×224. It is 112×112 for the layers 3 and 4; furthermore, the dimension of each channel for layers 5−7 is 56×56. Similarly, dimension of each channel for layers 8−10 is 28×28, and for layers 11−13 the dimension of each channel is 14×14. Thus, we have plotted some channels of the output feature map for layers 1−13, as a gray scale image in Fig. 5.7. For layers 14−16, the output feature map is 1D array, a preview of them is presented in Fig. 5.7. Finally, the output of classification layer is a single value, which is the class number of the object that has been detected by the proposed CNN-inference engine from the input image.

Human readable labels that are associated with these class numbers are provided with the ImageNet class-1000 dataset [105]. Here, we have shown the detected class numbers of three images along with their object labels. It can be seen that the class number generated by our CNN engine correctly depicts the object present in the input image.

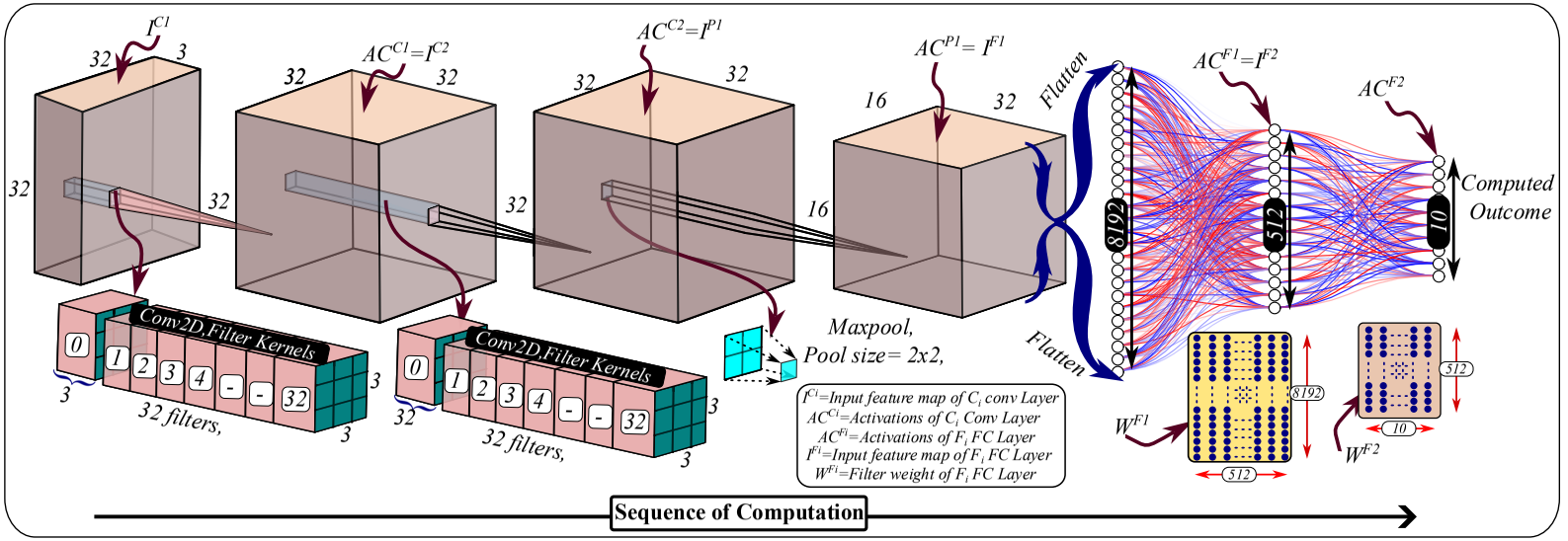
5.4.3 Implementation of Training for Hardware Validation

In this chapter, the proposed unified-CNN engine has been tested with multiple CNN models to validate its training capability. For better understanding, schematic representations of layer wise connection of data for both forward and backward passes of one such model are presented in Fig. 5.8 (a) and (b), respectively. Usually, CNN engine supports substantially complex CNN-models, the example CNN-model used in Fig. 5.8 are less complex for the ease of visualization. This is a simple custom-made CNN-model with ten classes, and the input size of 32×32×3 has been developed using the Python environment in the host computer. It consists of two *Conv* layers with kernel size of 3×3 for both, followed by a *maxpool*, and two FC layers. Since the input and output dimensions of this model matches with the requirement of CIFAR-10 dataset [113], we have thus trained it on the CIFAR-10 dataset.

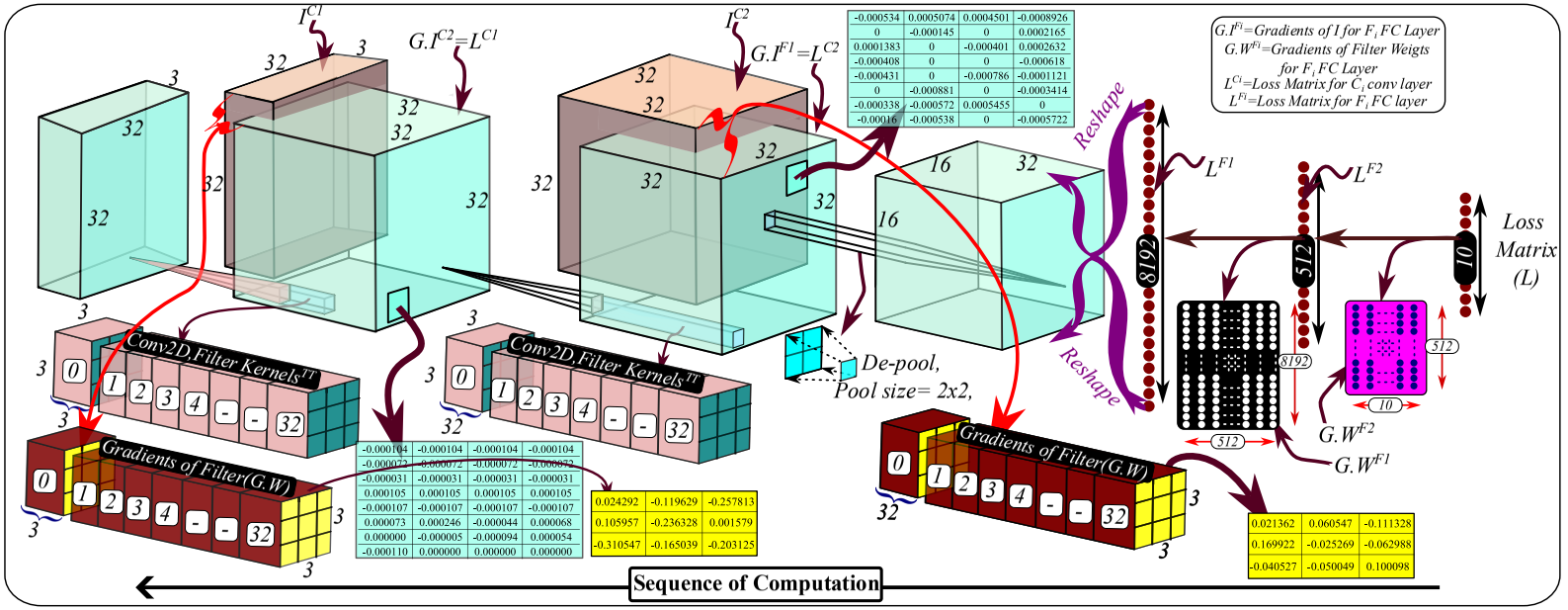
To begin with the training process, using the Python tool in the host computer, the layer information and randomly initialized weights as well as biases of the chosen CNN-model are exported in the memory card. As mentioned in section 5.4.1, we have also exported train images for the data set along with their class labels in the memory card. Thereafter, following steps are repeated for the desired number of epochs to proceed with the training process, referring Fig. 5.8.

1. **Forward Pass:**

   (a) **Forward Pass on Single Image:** Using the software executed on the on-chip processor of FPGA board, configure the unified-CNN engine with untrained initial weights. Then, transfer a random train image from a batch to the CNN engine to perform forward pass. During this forward pass, a copy of computed activation for all the layers are stored back to DRAM and these activations are used



(a) Connections during forward pass.



(b) Gradient computation during backward propagation.

Fig. 5.8: Schematic representations of a custom CNN model for CIFAR10 data set.

during the backward pass to calculate *G.W* from (5.4). Further, the activation of classification result has been used to calculate total loss *L*.

(b) **Complete Batch Processing:** Repeat the above step (1a) one-by-one for all images in the batch.

2. **Backward Pass:**

   (a) **Gradient Calculation:** Configure the unified-CNN engine with 180° rotated version of the same weights used in step 1. Send both the loss matrices of a layer for a specific image (via *I/L* port of KPU) and the activation of previous layer calculated in step (1a) as the input feature map (via *I* port of KPU) to calculate *G.I* and *G.W*, respectively, using (5.2) and (5.4). Here, the processing occurs in reverse order to propagate the loss in backward direction. Thus, *G.I* and *G.W* of the last layer is first calculated. Subsequently, this *G.I* is used as *L* matrix for its previous layer.

   (b) **Complete Batch Processing:** Repeat step (2a) for all images in the batch in one-by-one fashion to generate *b* set of *G.W* for a batch size of *b*.

3. **Weight Update After Batch Processing:** Compute the average of such *b* set of *G.W* and *G.B* to determine the final gradients which are denoted as $G.W_f$ and $G.B_f$, respectively, for the current batch. Consecutively, update the weights and bias with $\mathrm{G.W}_f$ and $G.B_f$ using (5.5) and (5.6), respectively.

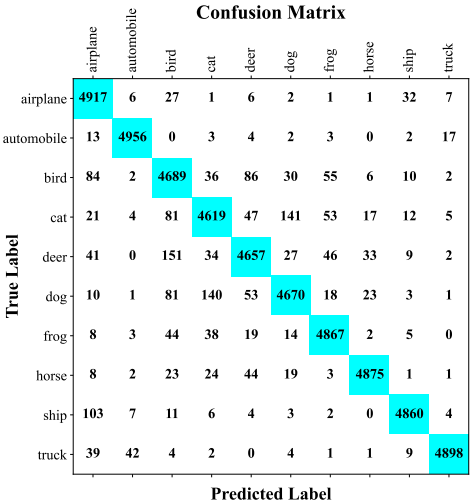
4. **Complete Epochs/Training:** Repeat all the above steps 1−3 for either all the epochs or until the desired level of accuracy has been achieved.

During the training process, all the intermediate data (activations and gradients) movement takes place between DRAM and unified-CNN engine. Moreover, in order to visualize such results, we have off-loaded some values of *G.W* and *G.I* for few layers. Therefore, *G.W* for some filters of two layers are shown as 3×3 matrix in Fig. 5.8 (b). Since the *lateral* × *vertical* dimension of *G.I* for these layers is large 32×32-matrix, a 4×8 chunk from one channel of two *G.I* has been shown as 4×8 matrix in Fig. 5.8 (b). In order to validate the correct training ability of the proposed unified-CNN engine, we must evaluate the training

result using a standard data set. Therefore, once the training is complete after 30 epochs, accuracy of the trained model has been evaluated in the Python environment using various test images of CIFAR-10 data set. Thereafter, the trained weights and biases of the model is exported from the DRAM of FPGA board to the memory card. It is then inserted in the host computer where the trained weights are imported to evaluate the model in test images of the data set. Finally, Fig. 5.9 shows a snapshot of the evaluation result in terms of Top-1 detection accuracy of the model that has been trained in our FPGA-based unified-CNN engine, when evaluated on the test images of CIFAR-10 data set in the Python environment. The model achieves an average Top-1 accuracy of 96.025%, as shown in Fig 5.9 (a). In addition, the confusion matrix of the same CNN model has been presented in Fig. 5.9 (b). Hence, the aforementioned process validates the training capability of the CNN engine that is based on the proposed unified-CNN accelerator.

	precision	recall	f1-score	support
airplane	0. 94	0. 98	0. 96	5000
automobile	0. 99	0. 99	0. 99	5000
bird	0. 92	0. 94	0. 93	5000
cat	0. 94	0. 92	0. 93	5000
deer	0. 95	0. 93	0. 94	5000
dog	0. 95	0. 93	0. 94	5000
frog	0. 96	0. 97	0. 97	5000
horse	0. 98	0. 97	0. 98	5000
ship	0. 98	0. 97	0. 98	5000
truck	0. 99	0. 98	0. 99	5000
accuracy			0. 96	50000
macro avg	0. 96	0. 96	0. 96	50000
weighted avg	0. 96	0. 96	0. 96	50000

(a)



(b)

Fig. 5.9: Evaluation accuracy analysis of the CNN-model that has been trained using the proposed unified-CNN engine: (a) precision, recall, and f1-score, and (b) confusion matrix.

5.4.4 Results and Comparison

The proposed unified-CNN accelerator architecture has been implemented on an FPGA platform using the AMD-Xilinx Zynq UltraScale+ ZCU102 MPSoC board. The detailed implementation results of our unified-CNN accelerator are presented in Table 5.1. For a fair comparison, the unified-CNN accelerator has also been implemented on additional FPGA

Table 5.1: Comparison of Proposed Unified CNN Accelerator Implementations with Relevant State-of-the-Art Works.

	[114]	[115]	[116]	[117]	[118]	[119]	**Proposed Implementations**			
Platform	ZCU104	ZCU104	VC709	ZCU102	KCU1500	VC709	**ZCU102**	**VC709**	**KCU1500**	**ZCU104**
Precision (in bits)	BFP16	FP32	INT16	FP32	PINT8	PINT8	**BFP16**	**BFP16**	**BFP16**	**BFP16**
Accuracy Loss** (%)	≃0	0	0.3-0.6	0	6-10	6-10	**≃0**	**≃0**	**≃0**	**≃0**
Clock Frequency (MHz)	200	100	NR	200	250	200	**354**	**200**	**250**	**343**
LUTS (in kilo)	73.66	169.14	NR	329.29	199	132	**206.83**	**210.89**	**206.88**	**206.75**
FFs (in kilo)	26.83	219.372	466.04	NR	NR	NR	**67.97**	**67.85**	**67.97**	**67.97**
BRAMs (36kb)	220	304	NR	174	1060	240	**64**	**64**	**64**	**64**
DSP Usage	1285	12	NR	1500	1030	1728	**960**	**960**	**960**	**960**
Θ_T (GOPs)	102.43	4.39	1022	86.12	641.1	610.98	**679.68**	**384**	**480**	**658.56**
η_{MAC} (MOPs/MAC)	79.71	365.83	NR	57.41	622.43	353.427	**708**	**400**	**500**	**686**
Power Consumption (W)	6.44	0.67	32	14.2	26.8	8.44	**3.78**	**3.89**	**3.40**	**3.87**
Energy Efficiency (GOPs/W)	15.89	6.56	31.94	6.06	23.92	72.39	**179.81**	**98.71**	**141.02**	**170.17**

NR = not reported

platforms, namely VC709, KCU1500, and ZCU104, which are used in compared state-of-the-art implementations. As elaborated in section 5.4, all our implementations employ the BF16 format.

Static timing analysis reveals that the critical path of the unified-CNN accelerator lies within the *HGU*, involving a multiplexer, a multiplier, and an adder. Consequently, the accelerator achieves a peak clock frequency of 354 MHz when implemented on the AMD-Xilinx Zynq UltraScale+ ZCU102 MPSoC FPGA.

As discussed in section 5.3.2, the proposed KPU architecture ensures uninterrupted computation, keeping all PEs fully active (σ=1). This results in improved compute efficiency (η_{MAC}), measured as the number of operations performed per MAC unit per unit time (OPs/MAC). Compared to prior works [119], [115], and [117], our architecture demonstrates a η_{MAC} improvement of 1.13×, 1.87×, and 12×, respectively, as shown in Table 5.1.

As discussed extensively in chapters 3 and 4, power consumption in CNN engines is largely influenced by data transfers between off-chip DRAM and the accelerator. Reducing off-chip data movement and improving data reuse are crucial for energy efficiency. As detailed in Section 5.3.3.1, the proposed KPU architecture enables shared access to a single line memory across n PEs and one HGU. During the forward pass, each input (I) is reused $\delta(\alpha-s)^2$ times, while during the backward pass, each loss value (L) is reused $\{\delta(\alpha-s)^2 + (\alpha-s)\}$ times. Additionally, filter values are reused $(A-s)^2$ times in both passes. This significantly reduces off-chip memory transactions, leading to improved energy efficiency. Specifically, the proposed unified-CNN accelerator achieves 1.36×, 5.89×, 10.70×, and 29.64× higher energy efficiency (OPs/watt) compared to the architectures reported in [119], [118], [114], and [117], respectively.

The improved results in this chapter stem from the unified architecture that supports both training and inference within the same hardware framework. Conventional designs either optimize inference alone or provide partial training support, often duplicating hardware or underutilizing resources. The proposed architecture eliminates such redundancy by reusing the same processing elements for forward propagation, backward propagation, and weight updates. Furthermore, it exploits reuse of gradients, activations, and weights across training iterations, thereby reducing redundant memory transfers. Moreover, it uses BFP16 format,

which maintains the full dynamic range of FP32. Hence, this unified and reuse-driven approach along with high dynamic range allow the accelerator to deliver high throughput and energy efficiency for both training and inference at FP32-level accuracy, which prior designs could not achieve simultaneously.

5.5 Summary

This chapter presented an unified-CNN accelerator architecture designed to efficiently support both inference and training workloads. Our accelerator architecture was incorporated with design-level optimizations to improve the utilization of PEs and maximized local data reuse, which in turn enhanced both computational performance and energy efficiency. Central to the design is a *gradient compute unit* that managed large kernel operations and used backpropagated loss values to more effectively compute gradient filter weights and activation gradients. The complete architecture was implemented on a Zynq UltraScale+ ZCU102 FPGA board, where it achieved a peak throughput of 679.7 GOPs at 354 MHz. Its performance was validated using standard CNN models under real-world test conditions. When compared with the state-of-the-art accelerators, the proposed design demonstrated substantial gains, achieving up to 12.3× higher hardware efficiency and 10.7× better energy efficiency making it well-suited for deployment in edge computing scenarios.

Chapter 6

Summary, Conclusion and Future Directions

6.1 Thesis Summary

AI has become a cornerstone of innovation across numerous sectors, from healthcare and autonomous systems to industrial automation and cybersecurity. A significant contributor to this progress is the use of CNNs, particularly in the domain of computer vision and pattern recognition. However, these models incur increasing computational and memory demands, making their efficient deployment on constrained platforms such as edge devices particularly challenging. The growing depth and complexity of modern CNNs, while beneficial for accuracy and generalization, impose burdensome costs in terms of power, memory access, and throughput requirements. While general-purpose processors and GPUs offer flexibility, they fall short in energy efficiency and scalability, especially under edge constraints. Consequently, there is a clear need for tailored hardware accelerators that can strike a balance between performance, energy efficiency, and resource optimization. This thesis explored a comprehensive framework for the design and development of high-throughput, hardware-efficient CNN accelerators, with an emphasis on real-time operation and edge compatibility. The solutions in our thesis cover a wide range of architectural challenges, including convolution flexibility, data reuse, memory bandwidth optimization, and unified support for both training and inference.

This thesis proposed a sequence of architectural enhancements aimed at improving the hardware efficiency, energy consumption, and functional scope of CNN accelerators, with a particular focus on edge deployment. Our work began by addressing the challenge of filter-size diversity in modern CNN models. To improve compatibility and hardware utilization, an adaptive convolution mapping approach was introduced. This technique enabled large filters, such as 5×5 and 7×7, to be restructured into multiple 3×3 operations, which are processed using a common PE array. As a result, the design supported variety of convolutional layers without requiring separate processing pipelines for each kernel size. This architectural feature has been discussed in chapter 2 and was supported by Fig. 6.1. It presents the resulting gain in PE efficiency relative to conventional fixed-kernel designs.

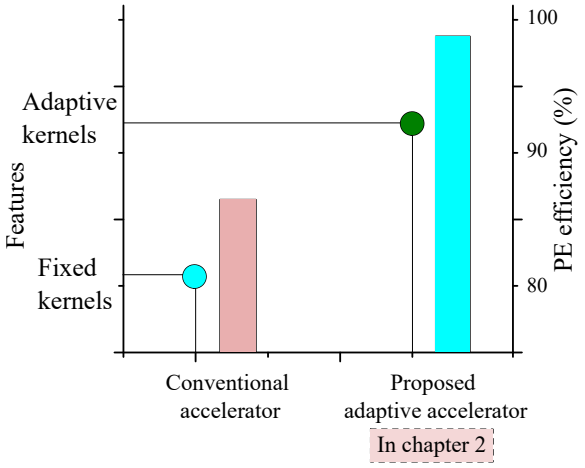


Fig. 6.1: PE efficiency comparison.

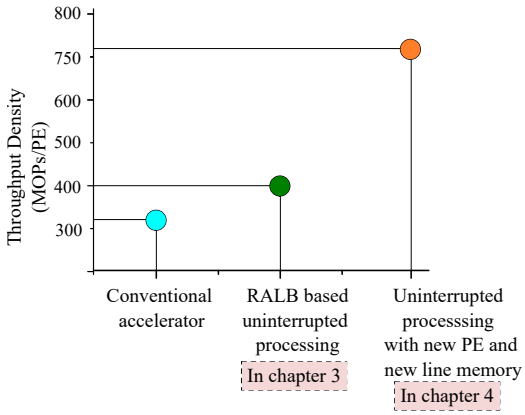


Fig. 6.2: Throughput density improvement.

Furthermore, this thesis explored ways to sustain computation without interruption from memory bottlenecks. In chapter 3, a line-stationary processing technique based on a RALB proposed to minimize computation stalling during data movements. This approach has been expanded in chapter 4 through the introduction of a redesigned PE micro-architecture and an optimized line-memory structure. These two stages collectively improved throughput density, as illustrated in Fig. 6.2, where the proposed designs demonstrate stepwise improvements over a conventional baseline.

Energy efficiency was also addressed by reducing redundant memory transfers. Chapter 3 demonstrated that the local data reuse via RALB lowers external memory dependence, which contributes to energy-aware design. Chapter 4 has built on this idea by introducing

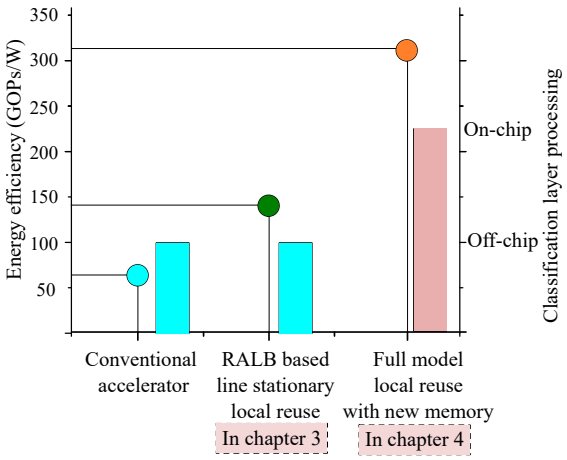


Fig. 6.3: Energy efficiency of baseline, locally reused, and fully reused CNN accelerator architectures.

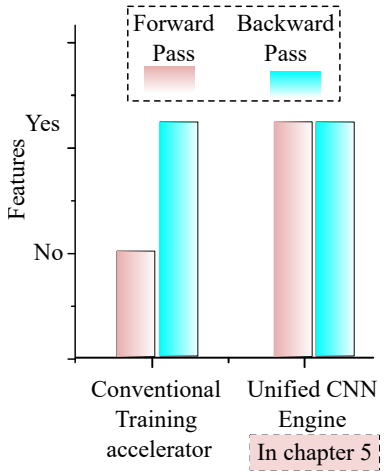


Fig. 6.4: Comparison of support for forward and backward pass in conventional and unified CNN training accelerators.

full-model data reuse and on-chip classification, thereby extending the reuse benefits to the entire inference pipeline. These modifications minimized the overall energy footprint, as reflected in Fig. 6.3, where the measured energy efficiency shows consistent improvement with each architectural iteration.

To validate the architectural components in a real-world scenario, a complete inference engine was designed by integrating the kernel and classification units into a single processing pipeline. Efficient memory sharing schemes were used to reduce resource duplication, and the proposed CNN engine has been evaluated using widely adopted models such as MobileNetV2 and ResNet50. The system was implemented and tested on FPGA and ASIC platforms, demonstrating model compatibility and consistent performance under constrained hardware conditions.

In addition to inference, the thesis introduced a unified accelerator design in chapter 5 that supported both training and inference using a common hardware backend. By incorporating a gradient computation unit and reusing shared resources across training phases, our design eliminated the need for a separate inference pipeline during training. This enables on-device training capability that is an important requirement for applications involving continual learning or model updates. The functional distinction between this unified accelerator and conventional training-only designs is summarized in Fig. 6.4, which indicates the extended support for both forward and backward passes.

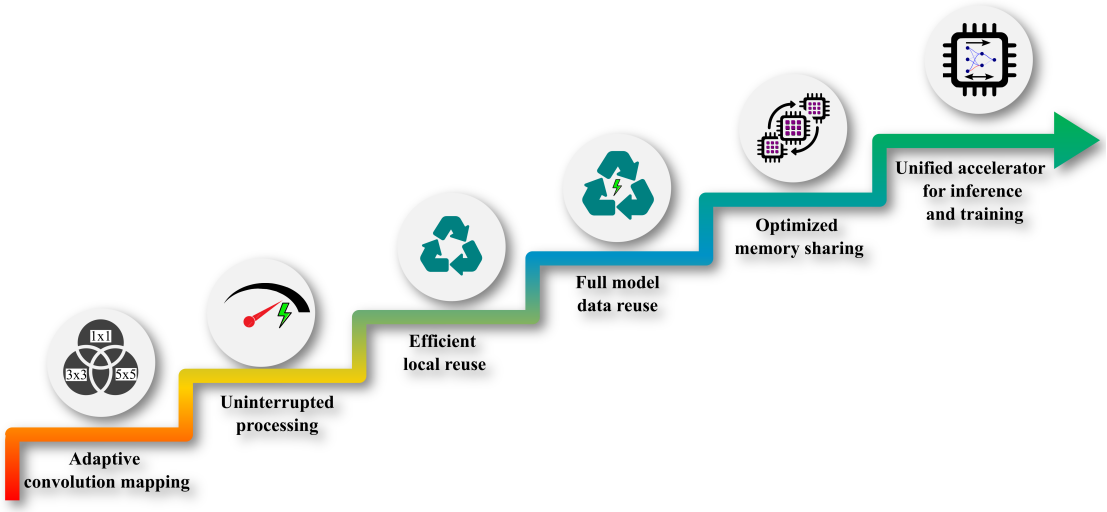


Fig. 6.5: Summary of architectural contributions across chapters showing progression from convolution mapping to unified training support.

The architectural decisions presented in this work followed a stepwise design methodology, where each chapter introduced an additional layer of optimization to the previous framework. These include progressive support for adaptive filter mapping, uninterrupted processing, energy-aware data reuse, optimized memory sharing, and full training support. The combined effect of these contributions is summarized in Fig. 6.5, which outlines the interconnected nature of the enhancements across the thesis.

In nutshell, our contributions offer a practical and modular architecture that addresses the computation, memory, and adaptability limitations of conventional CNN accelerators. The proposed solutions support scalable CNN execution with improved hardware utilization, reduced energy cost, and extended training capabilities, making them well-suited for deployment in real-time and edge-based AI systems.

6.2 Conclusion

CNNs are poised to play an increasingly vital role in the future edge intelligence-systems. As computer vision applications proliferate across mobile and embedded platforms, the need for compact and efficient CNN processing at the edge becomes one of the essential requirements. With their ability to extract rich and hierarchical features from raw sensory data, CNNs offer significant potential for enabling intelligent and real-time decision-making di-

rectly on edge devices without relying on the cloud connectivity. Edge platforms are fundamentally limited by area, power, and memory constraints, while CNNs typically demand high computational throughput and substantial data movement. Furthermore, the growing interest in on-device learning introduces additional complexity that demands the support not only for the inference, rather for resource-intensive training operations as well. Therefore, it becomes necessary to design an efficient high-speed CNN-accelerator that delivers excellent inference and training performances in real-time scenario. Furthermore, it should be cost-effective, must consume low-power and deliver higher data-rate. Hence, based on aforementioned contemporary requirements, our thesis presented new techniques for efficient and adaptive mapping of CNN kernels with corresponding hardware architectures for uninterrupted processing and energy efficient local reuse for both forward and backward passes of CNN inference and training. The overall contributions from our thesis has been shown in Fig. 6.5. Furthermore, our research works presented in this thesis concludes that the combination of adaptive convolution mapping with uninterrupted processing better utilizes the available processing elements in CNN accelerator architecture. Hence, it maximizes the throughput density in comparison to other alternatives. Additionally, on exploiting on-chip local reuse using the line stationary approach for all data of both inference and training significantly minimizes the power consumption for CNN processing over other implementation approaches from the literature. Therefore, it is necessary to explore CNN accelerator design from the perspective of adaptive kernel mapping, uninterrupted processing and maximum reuse of on-chip local data in order to conceive the best CNN accelerator for intelligence systems edge-applications.

6.3 Future Directions

While this thesis has addressed several foundational challenges in the development of hardware-efficient and high-throughput CNN accelerators, there remain many promising avenues for further exploration. The architectural strategies introduced here can be extended to accommodate emerging trends in neural network design, memory systems, and hardware scalability.

One potential direction is the support for transformer-based architectures. With the rising adoption of Vision Transformers (ViT), Swin Transformers, and similar attention-driven models in the field of computer vision, extending the proposed accelerator to handle attention-based computations could significantly expand its applicability. Efficiently mapping components such as multi-head self-attention and positional encoding onto hardware remains an open challenge, particularly for edge platforms where energy and resource constraints are critical.

Another important extension involves the incorporation of mixed-precision computation and quantization-aware training. By enabling layer-wise precision control and training-time awareness of quantization effects, future versions of the architecture could achieve better trade-offs between accuracy, energy consumption, and area footprint. These optimizations are particularly relevant for deployment on ultra-low-power or battery-operated devices.

Scalability across multiple accelerator instances is also worth investigating. As deep learning models grow in size and complexity, supporting distributed execution across multiple chips or cores becomes essential. Future designs could incorporate interconnect and synchronization mechanisms to efficiently parallelize CNN processing while maintaining real-time performance guarantees.

In safety-critical applications like autonomous driving and medical diagnostics, the robustness and trustworthiness of AI hardware are paramount. Enhancing the fault tolerance and security of the accelerator is therefore a key direction for future work. This may include error detection and correction in PE arrays, secure memory access, and lightweight encryption techniques to protect model parameters and intermediate activations from unauthorized access or data corruption.

Finally, a full ASIC implementation and tape-out of the proposed accelerator could yield valuable insights into post-silicon behavior. Although ASIC synthesis was performed as part of this work, actual fabrication and testing would enable thermal, parasitic, and signal-integrity-aware validation under real-world conditions. Such an evaluation would not only benchmark the architecture against industrial standards but also help guide iterative refinements toward commercialization.

Bibliography

[1] S. Russell and P. Norvig, *Artificial Intelligence: A Modern Approach*. Pearson, 2020.

[2] Turing and A. M., "Computing machinery and intelligence," *Mind*, vol. LIX, no. 236, pp. 433–460, 1950.

[3] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. MIT Press, 2016.

[4] T. Mitchell, *Machine Learning*. McGraw-Hill, 1997.

[5] Y. LeCun, Y. Bengio, and G. Hinton, "Deep learning," vol. 521, no. 7553, pp. 436–444, 2015.

[6] L. Floridi and J. Cowls, "A unified framework of five principles for ai in society," *Harvard Data Science Review*, 2019.

[7] A.-A. Christian. (2024, November) Discover the best ai tools: Top practices for safe and effective use. Accessed: 2025-04-13. [Online]. Available: https://www.ssl2buy.com/cybersecurity/ai-tools-best-practices-to-use-safely

[8] E. Topol, "High-performance medicine: the convergence of human and artificial intelligence," *Nature Medicine*, vol. 25, pp. 44–56, 2019.

[9] BioRender Editorial Team, "Revolutionizing our healthcare [infographic]," 2024, accessed on April 11 and 2025. [Online]. Available: https://www.biorender.com/blog/revolutionizing-our-healthcare-infographic

[10] A. Agrawal, J. Gans, and A. Goldfarb, *Prediction Machines: The Simple Economics of Artificial Intelligence*. Harvard Business Review Press, 2018.

[11] M. D. Smith and E. Brynjolfsson, "Recommendation engines," *Harvard Business Review*, 2017.

[12] OrangeMantra. (2025) How ai in manufacturing industry is reshaping businesses? Accessed: 2025-04-13. [Online]. Available: https://www.orangemantra.com/blog/ai-in-manufacturing-industry/

[13] Qulix Systems. (2025) The future of ai in the retail industry. Qulix Systems. Accessed: 2025-04-12. [Online]. Available: https://www.qulix.com/about/ai-in-retail/

[14] H. Kaur, "Role of ai in education: Use cases and examples and challenges and and future," December 2024, accessed: 2025-04-13. [Online]. Available: https://www.signitysolutions.com/blog/role-of-ai-in-education

[15] S. Thrun, "Toward robotic cars," *Communications of the ACM*, vol. 53, no. 4, pp. 99–106, 2010.

[16] ChipsAway. (2025) Advanced driver assistance systems (adas). Accessed: 2025-04-12. [Online]. Available: https://www.chipsaway.biz/blog/advanced-driver-assistance-systems-adas/

[17] T. Gabani. (2024, Mar.) Benefits of artificial intelligence in supply chain and logistic. Bigscal Technologies. Accessed: 2025-04-12. [Online]. Available: https://www.bigscal.com/blogs/logistics-industry/benefits-artificial-intelligence-supply-chain-logistic/

[18] D. Jurafsky and J. H. Martin, *Speech and Language Processing*, 3rd ed. Pearson, 2021.

[19] GeeksforGeeks. (2025, April) Ai in cybersecurity. Accessed: 2025-04-13. [Online]. Available: https://www.geeksforgeeks.org/ai-in-cybersecurity/

[20] A. L. Buczak and E. Guven, "A survey of data mining and machine learning methods for cybersecurity intrusion detection," *IEEE Communications Surveys & Tutorials*, vol. 18, no. 2, pp. 1153–1176, 2016.

[21] R. S. Sutton and A. G. Barto, *Reinforcement Learning: An Introduction*. MIT Press, 2018.

[22] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," pp. 1097–1105, 2012.

[23] Russell, S. J., Norvig, and Peter, *Artificial Intelligence: A Modern Approach*, 3rd ed. Upper Saddle River and NJ and USA: Prentice Hall, 2010.

[24] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278–2324, 1998.

[25] Sandler, Mark, Howard, Andrew, Zhu, Menglong, Zhmoginov, Andrey, Chen, and Liang-Chieh, "Mobilenetv2: Inverted residuals and linear bottlenecks," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2018, pp. 4510–4520.

[26] C. S. et al., "Going deeper with convolutions," pp. 1–9, 2015.

[27] Esteva, Andre, Kuprel, Brett, Novoa, R. A, Ko, Justin, Swetter, S. M, Blau, H. M, Thrun, and Sebastian, "Dermatologist-level classification of skin cancer with deep neural networks," *Nature*, vol. 542, no. 7639, pp. 115–118, 2017.

[28] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," pp. 1097–1105, 2012.

[29] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," pp. 770–778, 2016.

[30] Hu, Jie, Shen, Li, Sun, and Gang, "Squeeze-and-excitation networks," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2018, pp. 1915–1923.

[31] Russakovsky, Olga, Deng, Jia, Su, Hao, Krause, Jonathan, Satheesh, Sanjeev, Ma, Sean, Huang, Zhiheng, Karpathy, Andrej, Khosla, Aditya, Bernstein, Michael *et al.*, "Imagenet large scale visual recognition challenge," *International Journal of Computer Vision*, vol. 115, no. 3, pp. 211–252, 2015.

[32] Lin, Min, Chen, Qiang, Yan, and Shuicheng, "Network in network," *arXiv preprint arXiv:1312.4400*, 2013.

[33] Zeiler, M. D, Fergus, and Rob, "Visualizing and understanding convolutional networks," in *European Conference on Computer Vision (ECCV)*. Springer, 2014, pp. 818–833.

[34] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," 2015.

[35] Xie, Saining, Girshick, Ross, Dollár, Piotr, Tu, Zhuowen, He, and Kaiming, "Aggregated residual transformations for deep neural networks," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2017, pp. 1492–1500.

[36] X. Li, K. Ota, and M. Dong, "Ai and edge computing: A perfect match for emerging ai applications," *IEEE Network*, vol. 33, no. 2, pp. 36–41, 2020.

[37] W. Shi, J. Cao, Q. Zhang, Y. Li, and L. Xu, "Edge computing: Vision and challenges," *IEEE Internet of Things Journal*, vol. 3, no. 5, pp. 637–646, 2016.

[38] M. Satyanarayanan, "The emergence of edge computing," *Computer*, vol. 50, no. 1, pp. 30–39, 2017.

[39] N. P. Jouppi *et al.*, "In-datacenter performance analysis of a tensor processing unit," *Proceedings of ISCA*, pp. 1–12, 2017. [Online]. Available: https://doi.org/10.1145/3079856.3080246

[40] Y.-H. Chen *et al.*, "Eyeriss: An energy-efficient reconfigurable accelerator for deep convolutional neural networks," in *IEEE Journal of Solid-State Circuits*, 2017, pp. 127–138.

[41] W.-M. W. Hwu, *GPU Computing Gems*. Morgan Kaufmann, 2017.

[42] S. Han, H. Mao, and W. J. Dally, "Deep compression: Compressing deep neural networks with pruning and trained quantization and huffman coding," 2016.

[43] V. Sze, Y.-H. Chen, T.-J. Yang, and J. S. Emer, "Efficient processing of deep neural networks: A tutorial and survey," *Proceedings of the IEEE*, vol. 108, no. 4, pp. 673–691, 2020.

[44] G. Research, "Tensor processing unit (tpu)," Google Cloud TPU Documentation, 2021, available at: https://cloud.google.com/tpu.

[45] K. Venieris and C.-S. Bouganis, "Fpga-based acceleration of deep neural networks: A survey," *ACM Transactions on Reconfigurable Technology and Systems*, vol. 11, no. 3, pp. 1–23, 2018.

[46] S. Markidis, J. L. Trff, E. Laure, S. Benkner, and P. Gschwandtner, "Nvidia tensor core gpus: Performance and programmability," *Supercomputing Frontiers and Innovations*, vol. 5, no. 1, pp. 52–68, 2018.

[47] Chen, Tianshi, Du, Zidong, Sun, Ningyi, Wang, Jiawang, Wu, Chengyong, Chen, Yunji, Temam, and Olivier, "Diannao: A small-footprint high-throughput accelerator for ubiquitous machine-learning," in *ACM SIGARCH Computer Architecture News*, vol. 42, no. 1. IEEE Press, 2014, pp. 269–284.

[48] Chen, Yu-Hsin, Krishna, Tushar, Emer, J. S, Sze, and Vivienne, "Eyeriss: An energy-efficient reconfigurable accelerator for deep convolutional neural networks," in *IEEE International Solid-State Circuits Conference (ISSCC)*. IEEE, 2016, pp. 262–263.

[49] Zhang, Chen, Li, Peng, Sun, Guangyu, Guan, Yijin, Xiao, Bingjun, Cong, and Jason, "Optimizing fpga-based accelerator design for deep convolutional neural networks," in *ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA)*, 2015, pp. 161–170.

[50] Li, Ang, Chen, Yifan, Li, Jason, Chen, and Deming, "Smartshard: Adaptive kernel mapping for cnn accelerators," in *IEEE/ACM Design Automation Conference (DAC)*, 2016.

[51] Zhang, Chen, Sun, Zhenman, Li, Peng, Guan, Yijin, Xiao, Bingjun, Cong, and Jason, "Caffeine: Towards uniformed representation and acceleration for deep convolutional neural networks," in *IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, 2016, pp. 1–8.

[52] Qiu, Jiantao, Wang, Jie, Yao, Song, Guo, Kaiyuan, Li, Boxun, Zhou, Erjin, Yu, Jincheng, Tang, Tianqi, Xu, Ningyi, Song, Sen *et al.*, "Going deeper with embedded fpga platform for convolutional neural network," in *ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA)*, 2016, pp. 26–35.

[53] Ma, Yao, Cao, Yu, Vrudhula, Sarma, Seo, and Jae-sun, "Optimizing loop tiling and memory hierarchy for cnn accelerators," *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, vol. 10, no. 4, pp. 1–23, 2017.

[54] Han, Song, Liu, Xingyu, Mao, Huizi, Pu, Jing, Pedram, Ardavan, Horowitz, M. A, Dally, and W. J, "Eie: Efficient inference engine on compressed deep neural network," *ACM SIGARCH Computer Architecture News*, vol. 44, no. 3, pp. 243–254, 2016.

[55] Han, Song, Mao, Huizi, Dally, and W. J, "Deep compression: Compressing deep neural networks with pruning and trained quantization and huffman coding," *International Conference on Learning Representations (ICLR)*, 2016.

[56] Venieris, S. I, Kouris, Alexandros, Bouganis, and Christos-Savvas, "Toolflow for mapping diverse cnns on fpgas," in *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, vol. 11, no. 4, 2018, pp. 1–23.

[57] Zhang, Lei, Wang, Hongbo, Zhang, and Weimin, "Flexflow: A flexible dataflow accelerator architecture for convolutional neural networks," in *IEEE International Conference on Field Programmable Logic and Applications (FPL)*, 2017, pp. 1–8.

[58] Lee, Jungwook, Sampson, Jack, Narayanasamy, and Satish, "Training deep neural networks with reduced precision: Techniques and hardware architectures," *ACM Computing Surveys*, vol. 51, no. 6, pp. 1–36, 2018.

[59] A. Khan, A. Sohail, and U. Zahoora, "A survey of the recent architectures of deep convolutional neural networks," *Artif. Intell. Rev.*, vol. 53, pp. 5455–5516, 2020.

[60] M. N. Islam, R. Shrestha, , and S. R. Chowdhury, "A new hardware-efficient vlsi-architecture of googlenet cnn-model based hardware accelerator for edge computing applications," pp. 414–417, 2022.

[61] S. H. et al., "Eie: Efficient inference engine on compressed deep neural network," pp. 243–254, 2016.

[62] Y. Chen, T. Krishna, J. S. Emer, and V. Sze, "Eyeriss: An energy-efficient reconfigurable accelerator for deep convolutional neural networks," *IEEE J. Solid-State Circuits*, vol. 52, no. 1, pp. 127–138, 2017.

[63] M. N. Islam, R. Shrestha, , and S. R. Chowdhury, "Energy-efficient and high-throughput cnn inference engine based on memory-sharing and data-reusing for edge applications," *IEEE Trans. Circuits Syst. I and Reg. Papers*, vol. 71, no. 7, pp. 3189–3202, 2024.

[64] T. Chen *et al.*, "Diannao: A small-footprint high-throughput accelerator for ubiquitous machine-learning," in *Proceedings of ASPLOS*, 2014, pp. 269–284. [Online]. Available: https://doi.org/10.1145/2541940.2541967

[65] *Xilinx Deep Learning Processor Unit (DPU)*, Xilinx, 2021. [Online]. Available: https://www.xilinx.com/products/intellectual-property/dpu.html

[66] Google, "Google edge tpu," 2019. [Online]. Available: https://coral.ai/docs/edgetpu/

[67] H. Esmaeilzadeh *et al.*, "Bitfusion: An eda accelerator for deep neural networks," in *Proceedings of ISCA*, 2016, pp. 21–32. [Online]. Available: https://doi.org/10.1109/ISCA.2016.12

[68] Tan, Mingxing, Le, and Quoc, "Efficientnet: Rethinking model scaling for convolutional neural networks," in *Proceedings of the International Conference on Machine Learning (ICML)*. PMLR, 2019, pp. 6105–6114.

[69] V. Sze, Y. Chen, T. Yang, and J. S. Emer, "Efficient processing of deep neural networks: A tutorial and survey," *Proc. IEEE*, vol. 105, no. 12, pp. 2295–2329, 2017.

[70] Y. Yu, C. Wu, T. Zhao, K. Wang, and L. He, "Opu: An fpga-based overlay processor for convolutional neural networks," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 28, no. 1, pp. 1–13, 2019.

[71] K. G. et al., "Angel-eye: A complete design flow for mapping cnn onto embedded fpga," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 37, no. 1, pp. 35–47, 2018.

[72] S. Y. et al., "A high throughput acceleration for hybrid neural networks with efficient resource management on fpga," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 38, no. 4, pp. 678–691, 2019.

[73] Y. Duan, S. Li, R. Zhang, Q. Wang, J. Chen, and G. E. Sobelman, "Energy-efficient architecture for fpga-based deep convolutional neural networks with binary weights," pp. 1–5, 2018.

[74] V. Sze, Y.-H. Chen, T.-J. Yang, and J. S. Emer, *Efficient processing of deep neural networks*. Synthesis Lectures on Computer Architecture, 2020, vol. 15, no. 2.

[75] J. Li, K.-F. Un, W.-H. Yu, P.-I. Mak, and R. P. Martins, "An fpga-based energy-efficient reconfigurable convolutional neural network accelerator for object recognition applications," *IEEE Transactions on Circuits and Systems II: Express Briefs*, vol. 68, no. 9, pp. 3143–3147, 2021.

[76] Y.-X. Chen and S.-J. Ruan, "A throughput-optimized channel oriented processing element array for convolutional neural networks," *IEEE Transactions on Circuits and Systems II: Express Briefs*, vol. 68, no. 2, pp. 752–756, 2020.

[77] M. Z. A. et al., "A state-of-the-art survey on deep learning theory and architectures," *Electronics*, vol. 8, no. 3, p. 292, 2019.

[78] K. S. et al., "Memory requirements for convolutional neural network hardware accelerators," pp. 111–121, 2018.

[79] A. A. Gilan, M. Emad, and B. Alizadeh, "Fpga-based implementation of a real-time object recognition system using convolutional neural network," *IEEE Transactions on Circuits and Systems II: Express Briefs*, vol. 67, no. 4, pp. 755–759, 2019.

[80] S. S. L. O. et al., "Cnndroid: Gpu-accelerated execution of trained deep convolutional neural networks on android," pp. 1201–1205, 2016.

[81] N. P. J. et al., "In-datacenter performance analysis of a tensor processing unit," pp. 1–12, 2017.

[82] S. M. et al., "A resource limited hardware accelerator for convolutional neural networks in embedded vision applications," *IEEE Trans. Circuits Syst. II and Exp. Briefs*, vol. 64, no. 10, pp. 1217–1221, 2017.

[83] Q. J. et al., "Going deeper with embedded fpga platform for convolutional neural network," pp. 25–35, 2016.

[84] H. Jia, H. Valavi, Y. Tang, J. Zhang, , and N. Verma, "A programmable heterogeneous microprocessor based on bit-scalable in-memory computing," *IEEE J. Solid-State Circuits*, vol. 55, no. 9, pp. 2609–2621, 2020.

[85] D. T. Nguyen, T. N. Nguyen, H. Kim, , and H. J. Lee, "A high-throughput and power-efficient fpga implementation of yolo cnn for object detection," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 27, no. 8, pp. 1861–1873, 2019.

[86] D. T. Nguyen, H. Kim, , and H.-J. Lee, "Layer-specific optimization for mixed data flow with mixed precision in fpga design for cnn-based object detectors," *IEEE Transactions on Circuits and Systems for Video Technology*, vol. 31, no. 6, pp. 2450–2464, 2020.

[87] P. Darbani, N. Rohbani, H. Beitollahi, , and P. Lotfi-Kamran, "Rasht: A partially reconfigurable architecture for efficient implementation of cnns," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, 2022, early Access (DOI: 10.1109/TVLSI.2022.3167449).

[88] L. C. et al., "Origami: A convolutional network accelerator," pp. 199–204, 2015.

[89] Z. D. et al., "Shidiannao: Shifting vision processing closer to the sensor," pp. 92–104, 2015.

[90] C. Wu, J. Zhuang, K. Wang, , and L. He, "Mp-opu: A mixed precision fpga-based overlay processor for convolutional neural networks," pp. 33–37, 2021.

[91] J. Z. et al., "A low-latency fpga implementation for real-time object detection," pp. 1–5, 2021.

[92] C. Zhang, 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)*, Monterey and CA and USA, 2015, pp. 161–170.

[93] M. N. Islam, R. Shrestha, , and S. R. Chowdhury, "An uninterrupted processing technique-based high-throughput and energy-efficient hardware accelerator for convolutional neural networks," *IEEE Trans. Very Large Scale Integr. (VLSI)*, vol. 30, no. 12, pp. 1891–1901, 2022.

[94] F. Spagnolo, S. Perri, F. Frustaci, , and P. Corsonello, "Reconfigurable convolution architecture for heterogeneous systems on-chip," pp. 1–5, 2020.

[95] D. Wang, K. Xu, J. Guo, , and S. Ghiasi, "Dsp-efficient hardware acceleration of convolutional neural network inference on fpgas," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 39, no. 12, pp. 4867–4880, 2020.

[96] Deng, Jia, Dong, Wei, Socher, Richard, Li, Li-Jia, Li, Kai, Fei-Fei, and Li, "Imagenet: A large-scale hierarchical image database," in *2009 IEEE Conference on Computer Vision and Pattern Recognition*. IEEE, 2009, pp. 248–255. [Online]. Available: https://image-net.org/papers/imagenet_cvpr09.pdf

[97] L. Xuan, K.-F. U-den, C.-S. Lam, , and R. P. Martins, "An fpga-based energy-efficient reconfigurable depthwise separable convolution accelerator for image recognition," *IEEE Trans. Circuits Syst. II and Exp. Briefs*, vol. 69, no. 10, pp. 4003–4007, 2022.

[98] D. Zhu, S. Lu, M. Wang, J. Lin, , and Z. Wang, "Efficient precision-adjustable architecture for softmax function in deep learning," *IEEE Transactions on Circuits and Systems II: Express Briefs*, vol. 67, no. 12, pp. 3382–3386, 2020.

[99] F. Spagnolo, S. Perri, and P. Corsonello, "Aggressive approximation of the softmax function for power-efficient hardware implementations," *IEEE Transactions on Circuits and Systems II: Express Briefs*, vol. 69, no. 3, pp. 1652–1656, 2021.

[100] Y. Cao, J. Jia, D. Wu, , and W. Zhou, "Cordic-based softmax acceleration method of convolution neural network on fpga," pp. 66–70, 2020.

[101] Y. Zhang, Y. Zhang, L. Peng, L. Quan, S. Zheng, Z. Lu, , and H. Chen, "Base-2 softmax function: Suitability for training and efficient hardware implementation," *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 69, no. 9, pp. 3605–3618, 2022.

[102] Y. Gao, W. Liu, , and F. Lombardi, "Design and implementation of an approximate softmax layer for deep neural networks," pp. 1–5, 2020.

[103] M. N. Islam, R. Shrestha, and S. R. Chowdhury, "Low-complexity classification technique and hardware-efficient classify-unit architecture for cnn accelerator," in *IEEE International Conference on VLSI Design and Embedded Systems (VLSID)*, 2024, pp. 210–215.

[104] M. Wang, S. Lu, D. Zhu, J. Lin, , and Z. Wang, "A high-speed and low-complexity architecture for softmax function in deep learning," pp. 223–226, 2018.

[105] "Kaggle dataset: Imagenet-1000-classes," [Online]. Available: https://www.kaggle.com/datasets/skyap79/imagenet-classes.

[106] J. Hong, S. Arslan, T. Lee, and H. Kim, "Performance analysis of cnn inference/training with convolution and non-convolution operations on asic accelerator," in *Proc. IEEE Int. Symp. Circuits Syst. (ISCAS)*, 2023, pp. 1–5.

[107] J. Kim, H. Lee, and Y. Park, "An energy-efficient deep convolutional neural network training accelerator for in situ personalization on smart devices," in *Proc. IEEE/ACM Int. Conf. Comput.-Aided Des. (ICCAD)*, 2021, pp. 1–6.

[108] J. Li, G. Yan, W. Lu, S. Jiang, S. Gong, J. Wu, J. Yan, and X. Li, "Tnpu: An efficient accelerator architecture for training convolutional neural networks," in *Proc. Asia South Pac. Des. Autom. Conf. (ASPDAC)*, 2019, pp. 1–6.

[109] J. Hong, S. Arslan, T. Lee, and H. Kim, "Design of power-efficient training accelerator for convolution neural networks," *Electronics*, vol. 10, no. 7, p. 787, Mar. 2021.

[110] N. Unnikrishnan and K. K. Parhi, "A gradient-interleaved scheduler for energy-efficient backpropagation for training neural networks," pp. 1–5, 2020.

[111] N. K. Unnikrishnan and K. K. Parhi, "Intergrad: Energy-efficient training of convolutional neural networks via interleaved gradient scheduling," *IEEE Trans. Circuits Syst. I and Reg. Papers*, vol. 70, no. 5, pp. 1949–1962, 2023.

[112] K. et al., "A study of bfloat16 for deep learning training," 2019, arXiv preprint arXiv:1905.12322.

[113] A. Krizhevsky, "Learning multiple layers of features from tiny images," 2009, [Online]. Available: https://www.cs.toronto.edu/~kriz/cifar.html.

[114] T. H. Tsai and D. B. Lin, "An on-chip fully connected neural network training hardware accelerator based on brain float point and sparsity awareness," *IEEE Open J. Circuits Syst.*, vol. 4, pp. 85–98, 2023.

[115] Li, Jiajun, and et al., "Tnpu: An efficient accelerator architecture for training convolutional neural networks," 2019.

[116] T. G. et al., "Fpdeep: Acceleration and load balancing of cnn training on fpga clusters," pp. 81–84, 2018.

[117] Z. Liu, Y. Dou, J. Jiang, Q. Wang, , and P. Chow, "An fpga-based processor for training convolutional neural networks," pp. 207–210, 2017.

[118] S. D. et al., "A highly parallel fpga implementation of sparse neural network training," pp. 1–4, 2018.

[119] J. Lu, C. Ni, and Z. Wang, “Eta: An efficient training accelerator for dnns based on hardware-algorithm co-optimization,” *IEEE Trans. Neural Netw. Learn. Syst.*, vol. 34, no. 10, pp. 7660–7674, 2023.

List of Publications

Refereed Journal

1. M. N. Islam, R. Shrestha, and S. Roy Chowdhury, "Energy-Efficient and High-Throughput CNN Inference Engine Based on Memory-Sharing and Data-Reusing for Edge Applications," ***IEEE Trans. Circuits Syst. I, Reg. Papers***, vol. 71, no. 7, pp. 3189-3202, 2024.

2. M. N. Islam, R. Shrestha, and S. Roy Chowdhury, "An uninterrupted processing technique-based high-throughput and energy-efficient hardware accelerator for convolutional neural networks," ***IEEE Trans. Very Large Scale Integr. (VLSI)***, vol. 30, no. 12, pp. 1891-1901, 2022.

Peer-Reviewed Conference

1 M. N. Islam, R. Shrestha, and S. R. Chowdhury, "Low-Complexity classification technique and hardware-efficient classify-unit architecture for CNN accelerator," ***IEEE International Conference on VLSI Design and Embedded Systems (VLSID)***, pp. 210-215, 2024, India (Kolkata).

2 M. N. Islam, R. Shrestha, and S. R. Chowdhury, "A New Hardware-Efficient VLSI-Architecture of GoogLeNet CNN-Model Based Hardware Accelerator for Edge Computing Applications," ***IEEE Computer Society Annual Symposium on VLSI (ISVLSI)***, pp. 414-417, 2022, Cyprus (Nicosia).

Under Review

1 M. N. Islam, R. Shrestha, and S. R. Chowdhury, “A New Hardware-Efficient and High-Throughput Architecture of Unified-CNN Accelerator for Training and Inference,” (Under Review.)