

**ON THE OPTIMAL DEPLOYMENT OF DEEP
LEARNING NEURAL NETWORKS ON FIELD
PROGRAMMABLE GATE ARRAYS**

BY

AHMED IBRAHIM AHMED SHAWAHNA

A Dissertation Presented to the
FACULTY OF THE COLLEGE OF GRADUATE STUDIES
KING FAHD UNIVERSITY OF PETROLEUM & MINERALS
DHAHRAN, SAUDI ARABIA

In Partial Fulfillment of the
Requirements for the Degree of

DOCTOR OF PHILOSOPHY

In

COMPUTER ENGINEERING

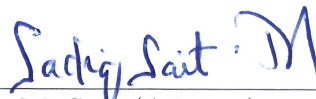
DECEMBER 2022

KING FAHD UNIVERSITY OF PETROLEUM & MINERALS
DHAHRAN 31261, SAUDI ARABIA

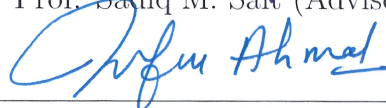
DEANSHIP OF GRADUATE STUDIES

This dissertation, written by **AHMED IBRAHIM AHMED SHAWAHNA** under the direction of his dissertation adviser and approved by his dissertation committee, has been presented to and accepted by the Dean of Graduate Studies, in partial fulfillment of the requirements for the degree of **DOCTOR OF PHILOSOPHY IN COMPUTER ENGINEERING**.

Dissertation Committee



Prof. Sadiq M. Sait (Adviser)



Dr. Irfan Ahmad (Co-Adviser)



Prof. Aiman H. El-Maleh (Member)



Prof. Tarek R. Sheltami (Member)



Dr. Moataz A. Ahmed (Member)



Prof. Aiman H. El-Maleh
Department Chairman



Prof. Suliman S. Al-Homidan
Dean of Graduate Studies

February 9, 2023

Date



©Ahmed Ibrahim Ahmed Shawahna
2022

To my loved ones,

*I dedicate this dissertation to my dear parents Ibrahim and Samira,
my beloved wife Shadan, my amazing child Ghaith, my wonderful
brothers Sohaib and Mohammed, and my beautiful sisters,
Wasan, Hanin, and Weam.*

Thank you for supporting, inspiring, and motivating me along the way.

*Without your love and sacrifice, I would not have been able to
complete this work.*

ACKNOWLEDGMENTS

Alhamdulillah, all the praise and thanks be to Allah Subhanahu Wa Ta'ala, who has provided guidance, opportunity, and knowledge to complete this dissertation. It is a matter of great pleasure and satisfaction for me to write this last part of my Ph.D. dissertation. Although I had my moments of ups and downs during this arduous journey, I consider myself exceedingly fortunate to have some amazing people to keep my morale high.

First and foremost, I am grateful to Prof. Sadiq M. Sait for giving me the opportunity to pursue my Ph.D. research under his supervision. I would like to thank him for his patient guidance, encouragement, and advice during my research and study at King Fahd University of Petroleum and Minerals (KFUPM). His willingness to offer me so much of his time and intellect is the main reason for this dissertation's completion. It was a true privilege and honor to share his exceptional knowledge and profound insights. Also, I would like to thank him for giving me the wonderful opportunity to work under his leadership at the Research Institute as a part-time research engineer sponsored by the Center for Communications and Information Technology Research. I would like to express my sincere thanks to him for creating a friendly environment, encouraging various social events, promoting an open-door policy, and emphasizing coffee breaks and brainstorming sessions.

My special gratitude goes to my co-advisors, Prof. Aiman H. El-Maleh and Dr. Irfan Ahmad, for taking the time to guide me through all the work that is this dissertation. I absolutely appreciate their guidance and energetic support. Without their patience, encouragement, and passion for the subject, I could not have hoped to master the topics covered here. I would also like to acknowledge Prof. Aiman H. El-Maleh for his suggestions on the selection of research topics. In particular, I would like to thank him for sparking my interest in FPGA-based DNN accelerators and narrowing down my research to low-precision CNN quantization. Additionally, a huge thank you to Prof. Aiman H. El-Maleh and Dr. Irfan Ahmad for their valuable and significant contributions to the research work published in this dissertation.

I would like to thank my dissertation committee members, Prof. Aiman H. El-Maleh, Prof. Tarek R. Sheltami, and Dr. Moataz A. Ahmed, for generously offering their time, encouragement, perceptive comments, and useful feedback that significantly improved this work.

I am overwhelmed with gratitude to Dr. Motaz Abdulrahman Alfarraj and the SDAIA-KFUPM Joint Research Center for Artificial Intelligence for providing high-performance computing facilities and access to the center's labs to work on this research. I am also thankful to Eng. Saleh AlSaleh for his technical assistance, interesting discussion on the research problems, and invaluable help and support.

As I am writing this dissertation, which marks the end of my career at KFUPM, I consider myself very fortunate to have spent all these times with the College of Computing and Mathematics. It has been not just somewhere I simply go to learn, but a truly welcoming place that holds some of the most memorable moments I have had

at college. I would like to thank all my computer engineering professors, who have made the study experience at KFUPM a great one and were always willing to lend an ear and a hand. Special thanks to Dr. Yahya E. Osais, who introduced me to the Department of Computer Engineering and inspired me during my years at KFUPM.

At this important juncture of my life, I am sure that nobody in the world will be as happy as my dear parents, Ibrahim and Samira. I thank them for their wholehearted love and encouragement. They have sacrificed all their comforts to see me successful. I can never thank them enough. I am also thankful to my wonderful brothers Sohaib and Mohammed, and my beautiful sisters Wasan, Hanin, and Weam, for their prayers and support throughout this work as well as throughout my life.

My special thanks, gratitude, appreciation, and admiration go to my beloved wife, Shadan, and my amazing child, Ghaith. Shadan, I am very grateful to you for all the love, support, and willingness to go the extra mile in helping me achieve my goals. I fully appreciate your encouragement, understanding, and cooperation, especially during the busy times of my dissertation work. I have no words to express my appreciation toward you. Without you, I would not have had the peace of mind to work. Ghaith, thank you so much for bringing joy and happiness to my life. Your hugs and smiles imparted the right energy to maintain momentum in my research.

Finally, I would like to extend my sincere thanks to my dear friends, who have encouraged me to take up this challenge, which has turned out to be one of the most rewarding experiences I have ever had. In particular, I would like to mention Dr. Khaled J. Assi, Dr. Ibrahim A. Nemer, Eng. Ali Al Omour, Eng. Shadi Abu Samra, and Eng. Saif Marwan for being great friends in all thick and thin. They are wonderful

persons who are always ready to help, even when help is not needed. Also, a special thank you to all my friends and family on campus who have motivated me, allowed me to vent to them on hard days in the lab, and always supported me in everything I wanted to do. Thank you all!

TABLE OF CONTENTS

ACKNOWLEDGMENTS	vi
LIST OF TABLES	xiv
LIST OF FIGURES	xvi
LIST OF ABBREVIATIONS	xx
ABSTRACT (ENGLISH)	xxvi
ABSTRACT (ARABIC)	xxix
CHAPTER 1 INTRODUCTION	1
1.1 Deep Learning Neural Networks	2
1.1.1 Applications	3
1.1.2 Revolution	4
1.1.3 The Need for Hardware Acceleration	5
1.2 Background and Terminology	7
1.2.1 Convolutional Neural Networks	7
1.2.2 Field Programmable Gate Arrays	13
1.3 Motivation	15
1.4 Objectives and Scope	20
1.5 Summary	22
CHAPTER 2 LITERATURE REVIEW	23
2.1 FPGA-based Accelerators of DNNs	24

2.2	Metaheuristics in the Design of CNNs	88
2.2.1	Structure Optimization	88
2.2.2	Weight and Bias Values Optimization	89
2.2.3	Design Variables Optimization	90
2.3	Optimization Mechanisms Employed in FPGA-based DNN Accelerators	92
2.4	Recommendations for Enhancing the Effectiveness of Utilizing FPGAs in Accelerating CNNs	99
2.5	Summary	100

CHAPTER 3 OPTIMIZATION OF FPGA-BASED CNN ACCELERATORS USING METAHEURISTICS

102

3.1	Introduction	104
3.2	Related Work	113
3.3	CNN Accelerator Design on FPGA Platforms	118
3.3.1	Optimizing CLP Computation and Memory Access	118
3.3.2	Multi-CLP Accelerator Design	122
3.3.3	Design Cost and Performance	126
3.4	Design Space Exploration	135
3.4.1	Problem Statement and Formulation	135
3.4.2	Optimization of Multi-CLP Designs	139
3.5	Experiments and Results	144
3.5.1	Experimental Settings	145
3.5.2	Results and Discussion	147
3.6	Conclusion	160

CHAPTER 4 POST-TRAINING QUANTIZATION FOR THE DESIGN OF MIXED LOW-PRECISION DNNs WITH DYNAMIC FIXED-POINT REPRESENTATION

162

4.1	Introduction	164
4.2	Background and Terminology	171
4.2.1	Dynamic Fixed-Point Representation	171

4.2.2	Quantization Approaches	173
4.3	Related Work	175
4.3.1	Quantization-Aware Training	176
4.3.2	Post-Training Quantization	178
4.3.3	Mixed-Precision Quantization	180
4.4	Design of Mixed Low-Precision DNNs for Integer-Only Deployment . .	184
4.4.1	Integer-Only Inference	186
4.4.2	Quantization Methodology	193
4.4.3	Quantization Parameters Optimization	198
4.4.4	Bit-Precision Level Reduction	205
4.5	Experiments and Results	214
4.5.1	Experimental Settings	215
4.5.2	Analysis of Rounding Strategies used to Determine the Initial Quantization Parameters	217
4.5.3	Analysis of Optimization Strategies for Quantization Parameters	221
4.5.4	Analysis of Data-Structures Sensitivity to Quantization	226
4.5.5	Mixed Low-Precision Quantization	229
4.6	Conclusion	246

CHAPTER 5 HARDWARE-DRIVEN TRAINING-AWARE QUANTIZATION FOR ENABLING EFFICIENT DNN INFERENCE ON EDGE DEVICES WITH MIXED LOW-PRECISION INTEGER-ONLY ARITHMETIC 248

5.1	Introduction	250
5.2	Background and Terminology	262
5.2.1	CNN-based Image Classification	262
5.2.2	Knowledge Distillation	266
5.3	Related Work	269
5.4	Integer-Only Inference	272
5.4.1	Quantization Methodology	272

5.4.2	Gradient Formulas for Learning Quantization Parameters and Model Parameters	279
5.4.3	Batch Normalization and Pooling Quantization	282
5.5	Learning the Optimal Layer-Wise Quantization Method	284
5.6	Designing Memory-Driven Mixed Low-Precision DNNs	288
5.7	Improving the Convergence of Low-Precision Quantized DNNs with Adaptive Learning Rate	293
5.8	Results and Discussions	296
5.8.1	Experimental Settings	296
5.8.2	Ablation Study	298
5.8.3	Accuracy on Specific Memory Constraints for Weights and Ac- tivities	300
5.8.4	Comparison with Prior Work	308
5.9	Conclusion	310

CHAPTER 6 CONCLUSION AND FUTURE RESEARCH DIREC-	312
TIONS	

6.1	Conclusion	312
6.2	Future Research Directions	317

REFERENCES	319
-------------------	------------

VITAE	365
--------------	------------

LIST OF TABLES

1.1	CNN layers for VGG models	13
2.1	Implementation and performance summary of FPGA-based accelerators of DNNs	86
2.2	Optimization mechanisms employed for FPGA-based acceleration of DNNs	96
3.1	An example of assigning five CONV layers to five CLPs	137
3.2	FPGA boards hardware resources	146
3.3	Single-CLP and Multi-CLP accelerators for AlexNet on 485T FPGA using FP32 data	149
3.4	Single-CLP and Multi-CLP accelerators for AlexNet on 690T FPGA using FP32 data	151
3.5	AlexNet accelerators performance on 485T and 690T FPGAs using FP32 data	152
3.6	Single-CLP and Multi-CLP accelerators for SqueezeNet 1.1 on 485T and 690T FPGAs using FxP16 data	154
3.7	SA-based Single-/Multi-CLP accelerators for VGG-16 on 690T FPGA using FxP16 data	156
3.8	SA-based Single-/Multi-CLP accelerators for GoogLeNet on 690T FPGA using FxP16	157
3.9	Implementation and performance summary of Single-CLP and Multi-CLP accelerators optimized for AlexNet	159

4.1	Improvement in power consumption and area requirement when using 8-bit fixed-point multiply-accumulate operations	166
4.2	Homogeneous data-structure quantization using different optimization methods for designing fixed-point VGG-16 networks	222
4.3	Quantization results of AlexNet, VGG-16, and ResNet-18 on ImageNet validation dataset	233
4.4	Implementation results for ResNet-18 on Artix-7	240
4.5	Quantization results comparison with prior work for AlexNet, VGG-16, and ResNet-18 on ImageNet validation dataset	243
5.1	MobileNet-V2 overall architecture	267
5.2	The validation accuracy after each design/optimization step	304
5.3	The validation accuracy for the top 10 inaccurately classified categories	308
5.4	The validation accuracy and latency of low-precision MobileNet-V2 models designed by our framework and PACT and HAQ methods . . .	309

LIST OF FIGURES

1.1	ImageNet competition results	5
1.2	An illustration of AlexNet architecture	12
1.3	FPGA basic structure	14
1.4	Convolution loops unrolling	17
2.1	2D CONV module of 3×3 kernel	25
2.2	MAPLE processing core architecture	27
2.3	MAPLE smart memory block	28
2.4	The architecture of DC-CNN	29
2.5	The architecture of neuFlow	31
2.6	Zhang et al. accelerator architecture	35
2.7	Top-level architecture of Microsoft CNN accelerator	37
2.8	The PE module and convolver complex design of Qiu et al. embedded accelerator architecture	39
2.9	SDF graph partitioning	45
2.10	ALAMO overall acceleration modules	47
2.11	Parallel framework exploiting four levels of parallelism	49
2.12	The architecture of MVTU PE	54
2.13	Transforming CONV to matrix-multiplication	54
2.14	CONV acceleration architecture and dataflow	57
2.15	Weights reloading	59
2.16	Overall DL architecture	60
2.17	Winograd-based CNN accelerator	63

2.18	The compute unit with a 2D BRAM-to-PE interconnection and the 2D dispatcher	65
2.19	Line buffer design	66
2.20	Operation of CLPs on CNN with three CONV layers	67
2.21	Systolic array architecture for CNN	70
2.22	Execution flowchart of ResNets layers	72
2.23	DLAU accelerator architecture	73
2.24	Overall architecture and dataflow	76
2.25	CONV reconfigurable computing module	77
2.26	Intel’s DL accelerator	79
2.27	Design flow from CNN model to hardware acceleration	81
2.28	Angel-Eye overall architecture and the structure of a single PE	82
2.29	CONV acceleration architecture and dataflow using data router	84
3.1	The input/output feature maps to/from each layer(s) of AlexNet architecture	105
3.2	An illustration of the convolution layer	106
3.3	The operation of Single-CLP accelerator and Multi-CLP accelerator on CNN with three CONV layers	110
3.4	CLP computation and ping-pong buffer structure	122
3.5	An example of a Single-CLP design	124
3.6	An illustration of roofline model basis	133
3.7	Cost function optimization progress	147
4.1	Generalized format for dynamic fixed-point numbers	172
4.2	Components and workflow of FxP-QNet	185
4.3	Statistics on size of input data-structures for each layer of AlexNet, VGG-16, and ResNet-18 architectures	192
4.4	An overview of FxP-QNet low-precision calculations	197
4.5	Distribution of FC-6 weights in VGG-16 model before and after 3-bit fixed-point quantization	199

4.6	The different types of basic residual blocks used in ResNet-18	217
4.7	The performance of AlexNet, VGG-16, and ResNet-18 architectures after quantization using the initial QPs	219
4.8	Weight distributions of VGG-16 layers that changed their clipping thresh- olds by non-zero integer amount	225
4.9	Data-structures robustness to quantization for AlexNet, VGG-16, and ResNet-18 models	227
4.10	Quantization wordlengths allocated to the data-structures of each layer for AlexNet, VGG-16, and ResNet-18 architectures	231
4.11	The requirement for read-write memory and read-only memory, in MB, for AlexNet, VGG-16, and ResNet-18 architectures	237
4.12	The architecture of the proposed hardware accelerator and FxP-QLayer	239
5.1	The bell-shaped distribution in VGG-16 and ResNet-101 layers	256
5.2	Hardware-aware automated ML framework for mixed low-precision quan- tization	259
5.3	Knowledge distillation from teacher to student (quantized) model . . .	268
5.4	An overview of the low-precision computation of the CONV and FC layers	275
5.5	The distribution of the original FP32 weights and the 4-bit quantized weights of the last point-wise CONV layer in MobileNet-V2	277
5.6	Generalized format for quantized activations and weights	278
5.7	The gradient of the quantizer output with respect to the transformation scalar	280
5.8	An overview of the low-precision computation of the BatchNorm layer .	283
5.9	An illustration of the stochastic network used to learn the optimal quan- tization method	286
5.10	The validation top-1 accuracy for a 2MB MobileNet-V2 model when formulating the clip operation to perform weight gradient clipping and weight gradient pass-through	299

5.11	The validation top-1 accuracy for the low-precision MobileNet-V2 models that satisfy the memory requirements of 512KB for activations and 2MB for weights	301
5.12	The loss function during the exploration for the optimal bit-precision level and quantizer assignment that satisfy the target memory requirement of 2.5MB	302
5.13	The mount of memory reduction during the exploration for the optimal bit-precision level and quantizer assignment that satisfy the target memory requirement of 2.5MB	302
5.14	The validation accuracy during the exploration for the optimal bit-precision level and quantizer assignment that satisfy the target memory requirement of 2.5MB	303
5.15	The memory size for activations and model parameters during the exploration for the optimal bit-precision level and quantizer assignment that satisfy the target memory requirement of 2.5MB	303
5.16	Wordlength allocated to weights and input activations for each CONV layer of MobileNet-V2 architecture	305
5.17	Statistics on size of weights and input activations for each CONV layer of MobileNet-V2 architecture in log-domain	305
5.18	Statistics on MAC operations performed by each CONV layer of MobileNet-V2 architecture in log-domain	305
5.19	Statistics on validation top-1 accuracy for each class in ImageNet dataset on MobileNet-V2 architecture	307
5.20	Statistics on validation top-5 accuracy for each class in ImageNet dataset on MobileNet-V2 architecture	307

LIST OF ABBREVIATIONS

AC	aspiration criteria
AI	artificial intelligence
AL	aspiration level
ALM	adaptive logic module
ASIC	application specific integrated circuit
BatchNorm	batch normalization
BNN	binary neural network
BPB	block random-access memories per bank
BRAM	block random-access memory
BRAM18K	18 kilo bit block random-access memory
BW	bandwidth
C-Block	computational block
CLB	configurable logic block
CLP	convolutional layer processor
CMOS	complementary metal-oxide semiconductor
CNN	convolutional neural network

CNP	convolutional network processor
CONV	convolution
CPU	central processing unit
CTC	computation to communication
DAG	directed acyclic graph
DC-CNN	dynamically configurable convolutional neural network
DDR	double data rate
DL	deep learning
DLAU	deep learning acceleration unit
DMA	direct memory access
DNN	deep learning neural network
domQ	quantization domain
DRAM	dynamic random-access memory
DSL	domain-specific language
DSP	digital signal processor
Eltwise	element-wise
FC	fully-connected
FF	flip-flop
FFT	fast Fourier transform
FIFO	first in first out
FLOP	floating-point operation

FM	feature map
FP16	16-bit floating-point
FP32	32-bit floating-point
FPGA	field programmable gate array
fps	frames per second
FxP	fixed-point
FxP16	16-bit fixed-point
FxP-QLayer	fixed-point quantization layer
FxP-QNet	fixed-point quantizer of deep neural networks
GA	genetic algorithm
GB	giga byte
GDDR	graphics double data rate
GFLOP	giga floating-point operation
GHz	gigahertz
GLVQ	generalized learning vector quantization
GMAC	giga multiply-and-accumulate
GOP	giga operation
GPU	graphic processing unit
HDL	hardware description language
HLS	high-level synthesis
IFM	input feature map

ILSVRC	ImageNet large scale visual recognition challenge
IMM	input-major mapping
IoT	Internet of things
IP	intellectual property
KB	kilo byte
Kb	kilo bit
KL	Kullback-Leibler
LCL	least compact layer
LUT	look-up table
M20K	20 kilo bit random-access memory
MAC	multiply-and-accumulate
MB	mega byte
MCL	most compact layer
MFP	memory footprint
MHz	megahertz
ML	machine learning
MSE	mean square error
MUX	multiplexer
MVM	matrix-vector multiplication
MVTU	matrix-vector-threshold unit
nm	nanometer

NN	neural network
nn-X	neural network next
Norm	normalization
NP	non-deterministic polynomial-time
OFM	output feature map
PE	processing element
pJ	petajoule
POOL	pooling
PT	processing tile
QP	quantization parameter
RAM	random-access memory
ReLU	rectified linear unit
RL	reinforcement learning
RNN	recurrent neural network
RO	read-only
RTL	register transfer level
RW	read-write
SA	simulated annealing
SDF	synchronous dataflow
SDFG	synchronous dataflow graph
SDRAM	synchronous dynamic random-access memory

SGD	stochastic gradient descent
SIMD	single-instruction stream multiple-data streams
SoC	system-on-chip
SSI	supervised semantic indexing
SVD	singular value decomposition
SVM	support vector machine
TL	tabu list
TOP	tera operation
TS	tabu search
TWN	ternary weight network
UAV	unmanned aerial vehicle
VGG	visual geometry group
VIP	virtual image processor
WMM	weight-major mapping

DISSERTATION ABSTRACT

NAME: Ahmed Ibrahim Ahmed Shawahna

TITLE OF STUDY: On the Optimal Deployment of Deep Learning Neural Networks on Field Programmable Gate Arrays

MAJOR FIELD: Computer Engineering

DATE OF DEGREE: December 2022

Deep learning neural networks (DNNs) have demonstrated their effectiveness in a wide range of computer vision tasks, with state-of-the-art results obtained through complex and deep structures that require intensive computation and memory. In the past, graphic processing units (GPUs) enabled these breakthroughs because of their greater computational speed. Nowadays, field programmable gate arrays (FPGAs) have seen a surge of interest in accelerating DNN inference. This is due to their ability to create custom designs with different levels of parallelism. Furthermore, FPGAs provide better performance per watt than other computing technologies such as GPUs, which is a critical requirement for DNN applications on battery-powered unmanned aerial vehicles and Internet of things devices. However, without careful implementation of today's complex DNN models, the design may not fit the target FPGA due to limited

logic resources. Additionally, DNN-based systems have to provide minimum latency overhead and high throughput such that making the right decision happens in time.

In this dissertation, we review recent existing techniques for accelerating DNNs on FPGAs and provide recommendations for future directions that will simplify the use of FPGA-based accelerators and enhance their performance. Then, we provide a couple of works for addressing the requirements for efficient implementations of convolutional neural networks (CNNs). These works optimize CNN implementation on FPGA platforms in terms of throughput, latency, energy efficiency, and power consumption. In particular, we propose and investigate three works, (i) a CNN accelerator and an accompanying automated design methodology that employs metaheuristics for optimal partitioning of available FPGA resources to design high-throughput multiple convolutional layer processors (CLPs), (ii) a framework, referred to as FxP-QNet, to efficiently quantize the weights and activations of pre-trained CNN-based models to low-precision numbers in fixed-point representation, and (iii) a novel end-to-end memory-driven methodology to enable the deployment of CNNs on resource-constrained edge devices while maintaining model's accuracy.

The focus of the first work is on the development of a Multi-CLP acceleration framework with parameterized Verilog HDL modules. The presented optimization tool adopts simulated annealing (SA) and tabu search (TS) metaheuristic algorithms to find the number of CLPs required, their respective hardware configurations, and the assignment of convolutional layers to CLPs that achieve the best system performance on a given target FPGA device. We illustrate that the implemented SA-/TS-based Multi-CLP achieves $1.31\times - 2.37\times$ higher throughput than state-of-the-art Single-/Multi-

CLP approaches in accelerating AlexNet, SqueezeNet 1.1, VGG-16, and GoogLeNet architectures on Xilinx VC707 and VC709 FPGA boards.

The second work, on the other hand, demonstrates the effectiveness of FxP-QNet in achieving the accuracy-compression trade-off with significant improvements in hardware cost, power efficiency, and operating frequency. This is mainly due to FxP-QNet’s replacement of expensive floating-point operations with faster, more hardware-efficient integer operations. In particular, the FxP-QNet-quantized AlexNet, VGG-16, and ResNet-18 reduce the overall memory requirements of their full-precision counterparts by $7.16\times$, $10.36\times$, and $6.44\times$, respectively, with less than 0.95%, 0.95%, and 1.99% accuracy drop. Additionally, the FxP-QNet-quantized ResNet-18 implemented on Xilinx Artix-7 FPGA uses 76%, 67%, and 98% fewer look-up tables, flip-flops, and digital signal processors, respectively, and consumes 455 mW less than those designed using the conventional 8-bit quantization scheme, even though the operating frequency of FxP-QNet-quantized ResNet-18 model is $1.35\times$ higher.

In the third work, we illustrate that the defined optimized gradient formulas to automate the learning of wordlengths and quantization methods through stochastic gradient descent successfully enable the deployment of high-accurate and low-latency MobileNet models that are perfectly tailored to the target hardware architecture on Xilinx Zynq-7020 FPGA edge device. Specifically, the presented framework design a customized integer-only MobileNet-V2 with a 1.53MB model and 0.97MB of activations memory space while achieving top-1 validation accuracy of 72.3% on the ImageNet dataset, improving the top-1 accuracy of previously published implementation by 4.3%.

ملخص الرسالة

الاسم:

أحمد ابراهيم أحمد شواهنة

عنوان الدراسة:

نحو النشر الأمثل للشبكات العصبية للتعلم العميق على مصفوفات البوابات المنطقية القابلة للبرمجة

التخصص:

هندسة الحاسب الآلي

تاريخ الدرجة العلمية: كانون أول 2022

أثبتت الشبكات العصبية للتعلم العميق فعاليتها في مجموعة واسعة من مهام الرؤية الحاسوبية، مع أحدث النتائج التي تم الحصول عليها من خلال الهياكل المعقدة والعميقة التي تتطلب عمليات حسابية مكثفة وذاكرة كبيرة. في الماضي، مكنت وحدات معالجة الرسومات هذه المتطلبات الهائلة بسبب سرعتها الحسابية الكبيرة جداً. في الوقت الحاضر، شهدت مصفوفات البوابات المنطقية القابلة للبرمجة زيادة في الاهتمام من أجل تسريع استدلال الشبكات العصبية للتعلم العميق. ويرجع ذلك إلى قدرتها على إنشاء تصاميم مخصصة بمستويات مختلفة من التوازي. علاوة على ذلك، توفر مصفوفات البوابات المنطقية القابلة للبرمجة أداء أفضل لكل واط مقارنة بتقنيات الحوسبة الأخرى مثل وحدات معالجة الرسومات، وهو مطلب ضروري لتطبيقات الشبكات العصبية للتعلم العميق على الطائرات بدون طيار وأجهزة إنترنت الأشياء التي تعمل بالبطاريات. ولكن، بدون التنفيذ الدقيق لنماذج الشبكات العصبية للتعلم العميق المعقدة اليوم، قد لا يتناسب التصميم مع مصفوفة البوابات المنطقية القابلة للبرمجة المستهدفة بسبب الموارد المنطقية المحدودة. بالإضافة إلى ذلك، يجب أن توفر الأنظمة المستندة إلى الشبكات العصبية للتعلم العميق الحد الأدنى من وقت الاستجابة والإنتاجية العالية بحيث يحدث اتخاذ القرار الصحيح في الوقت المناسب.

في هذه الأطروحة، نستعرض التقنيات الحالية الحديثة لتسريع الشبكات العصبية للتعلم العميق على مصفوفات البوابات المنطقية القابلة للبرمجة ونقدم توصيات للاتجاهات المستقبلية التي من شأنها تبسيط استخدام المسرعات القائمة على مصفوفات البوابات المنطقية القابلة للبرمجة وتعزيز أدائها. بعد ذلك، نقدم عدة أعمال لتلبية متطلبات التنفيذ الفعال للشبكات العصبية التلافيفية. تهدف هذه الأعمال إلى تحسين تنفيذ الشبكات

العصبية التلافيفية على مصفوفات البوابات المنطقية القابلة للبرمجة من حيث الإنتاجية، وقت الإستجابة، كفاءة الطاقة، واستهلاك الطاقة. على وجه الخصوص، نقترح ونحقق في ثلاثة أعمال، (1) مسرع الشبكات العصبية التلافيفية ومنهجية تصميم آلية مصاحبة تستخدم الأدلة العليا للتقسيم الأمثل لموارد مصفوفة البوابات المنطقية القابلة للبرمجة المتاحة لتصميم عدة معالجات عالية الإنتاجية للطبقات التلافيفية، (2) إطار عمل لتكميم هياكل بيانات الشبكات العصبية التلافيفية المدربة مسبقاً بكفاءة إلى أرقام منخفضة الدقة في تمثيل النقطة الثابتة، و (3) منهجية جديدة تعتمد على متطلبات الذاكرة لتمكين نشر الشبكات العصبية التلافيفية على أجهزة حوسبة الحافة محدودة الموارد مع الحفاظ على دقة النموذج.

ينصب تركيز العمل الأول على تطوير إطار تسريع متعدد المعالجات للطبقات التلافيفية. تعتمد أداة التحسين المقدمة عدة خوارزميات للعثور على العدد الأمثل لمعالجات الطبقات التلافيفية المطلوبة، تكوينات الأجهزة الخاصة بها، وتعيين الطبقات التلافيفية للمعالجات التي تحقق الأداء الأمثل للنظام على مصفوفة البوابات المنطقية القابلة للبرمجة المستهدفة. نظهر أن عملنا المطور القائم على استخدام عدة معالجات مصممة باستخدام الخوارزميات يحقق إنتاجية أعلى بمقدار $1.31 - 2.37$ مرة أكثر من أحدث مناهج معالجات الطبقات التلافيفية الأحادية/المتعددة في تسريع أربع شبكات عصبية شائعة الاستخدام على لوحين من مصفوفات البوابات المنطقية القابلة للبرمجة المستخدمة بكثرة في الأعمال الأدبية السابقة.

من ناحية أخرى، يوضح العمل الثاني فعالية أداة التكميم المقترحة في تحقيق المفاضلة بين الدقة والضغط مع تحسينات كبيرة في تكلفة الأجهزة وكفاءة الطاقة وتعدد التشغيل. ويرجع ذلك أساساً إلى استبدال الأداة المقترحة لعمليات الفاصلة العائمة باهظة الثمن بعمليات أعداد صحيحة أسرع وأكثر كفاءة في استخدام الأجهزة. على وجه الخصوص، تعمل ثلاث شبكات عصبية مكتملة شائعة الاستخدام على تقليل متطلبات الذاكرة الإجمالية لنظيراتها كاملة الدقة بمقدار 7.16 و 10.36 و 6.44 مرة، مع انخفاض في الدقة بمقدار أقل من 0.95% و 0.95% و 1.99%. بالإضافة إلى ذلك، تستخدم إحدى الشبكات العصبية المكتملة 76% و 67% و 98% أقل من جداول البحث والدوائر القلابة ومعالجات الإشارات الرقمية، على التوالي، عند تنفيذها على مصفوفة البوابات الرقمية القابلة للبرمجة وتستهلك 455 ميلي واط أقل من تلك المصممة باستخدام نظام التكميم التقليدي 8 بت، على الرغم من أن تردد التشغيل لنموذج الشبكة العصبية المكتملة أعلى بمقدار 1.35 مرة.

في العمل الثالث، نوضح أن صيغ النزول المحسنة لأتمتة تعلم أطوال البت وطرق التكميم من خلال خوارزمية النزول التدريجي تمكن بنجاح من نشر نماذج عالية الدقة ومنخفضة زمن الإستجابة المصممة بشكل مثالي لبنية الأجهزة على مصفوفة البوابات المنطقية القابلة للبرمجة المستهدفة. على وجه التحديد، يصمم إطار العمل المقترح شبكة عصبية تلافيفية تنفذ عمليات حسابية على أعداد صحيحة فقط مع نموذج بحجم 1.53 ميجابايت و 0.97 ميجابايت من مساحة ذاكرة التنشيطات مع تحقيق دقة بنسبة 72.3%، مما يحسن الدقة للتنفيذ المنشور مسبقاً بنسبة 4.3%.

CHAPTER 1

INTRODUCTION

Due to recent advances in digital technologies, and availability of credible data, an area of artificial intelligence (AI), deep learning (DL), has emerged, and has demonstrated its ability and effectiveness in solving complex learning problems not possible before. In particular, convolutional neural networks (CNNs) have demonstrated their effectiveness in image detection and recognition applications. However, CNNs require intensive central processing unit (CPU) operations and memory bandwidth that make general CPUs fail to achieve desired performance levels. Consequently, hardware accelerators that use graphic processing units (GPUs), field programmable gate arrays (FPGAs), and application specific integrated circuits (ASICs) have been employed to improve the throughput of CNNs. Especially, FPGAs have been recently adopted for accelerating the inference of deep learning neural networks (DNNs) due to their ability to maximize parallelism at a reasonable price with high energy efficiency. Additionally, they facilitate the implementation of customized memory hierarchies and hardware architectures with bit-level parallelism, which are critical for efficient data transfers and processing.

In this chapter, we provide an overview on DNNs and their need for hardware accelerators to improve inference throughput to provide the desired level of performance for applications. Then, the chapter presents general background information about CNNs, their key operations, and some well-known CNN architectures. In addition, it introduces the basic structure of FPGAs and highlights the features enabling them to accelerate computationally intensive applications. Finally, we discuss the challenges of implementing CNNs on FPGAs and how this research work overcomes these challenges to design a scalable and efficient FPGA-based accelerator of CNNs.

1.1 Deep Learning Neural Networks

In recent years, due to the availability of massive amounts of credible data (big data: text, video, audio, etc.), and tremendous advances in the area of digital electronics technologies that provide immense computing power, there has been a revival in the area of AI, particularly in the area of DL [1]–[3], a sub-field of machine learning (ML). The field of DL emerged in 2006 after a long pause in the research area of artificial neural networks, usually simply called neural networks (NNs) [4]. A key aspect in DL is that the networks and/or their weight parameters are not designed by human beings. Instead, they are learned from data using a general purpose learning procedure [5], [6]. While ML uses algorithms to parse and learn from data, to make informed decisions, DL structures algorithms in layers to create a NN that can learn, and similar to human intelligence, can make accurate decisions on its own [7]. Therefore, instead of designing algorithms by hand, systems can be built and trained to implement concepts in a way

similar to what comes naturally to humans, and with accuracy sometimes exceeding human-level performance [8], [9].

In DL, each layer is designed to detect features at different levels. A layer transforms the representation at one level (starting from input data which maybe images, text, or audio) to a representation at a higher, slightly more abstract level [10]. For example, in image recognition, where input initially comes in the form of pixels, the first layer uses filters to detect low level features such as edges and curves. The output of the first layer becomes input to the second layer which produces higher level features, for example semi-circles, and squares [11]. The next layer assembles the output of the previous layer to parts of familiar objects, and a subsequent layer detects the objects. As we go through more layers, the network yields feature maps (FMs) that represent more and more complex features. The deeper you go into the network, the filters begin to be more responsive to a larger region of the pixel space. Higher level layers amplify aspects of the received inputs that are important for discrimination and suppress irrelevant variations.

1.1.1 Applications

With the now widely used DNNs [12], [13], it is now possible to solve problems in domains where knowledge is not easily expressed explicitly and implicit information is stored in the raw data. Solutions to multifarious problems in the domain of sciences, business, etc., have been possible that were not conceivable for several years, in spite of best attempts by the AI community. This has been primarily possible due to the excellent ability of DL in discovering intricate structures in high-dimensional data. Ex-

amples include character recognition [14], gesture recognition [15], speech recognition (e.g., in Google Now, Siri, or click-through prediction on an advertisement) [16]–[18], document processing [19]–[21], natural language processing [22], [23], video classification [24], image classification [25]–[30], face detection and recognition [31], [32], robot navigation [33]–[35], real-time multiple object tracking [36], financial forecasting [37], and medical diagnosis systems [38]–[40], to name a few.

Other recent areas of applications include automated driving (e.g., learning to detect stop signs, traffic lights, pedestrians, etc.), aerospace and defense (e.g., identify objects from satellites and identify safe or unsafe zones), medical research (e.g., in identification of cancer cells), industrial automation (e.g., to improve worker safety by detecting when people or objects are within an unsafe distance of machines), and electronics (used in automated hearing, speech translation, etc.) [9], [41]–[44].

1.1.2 Revolution

The success of DNNs grew to prominence in 2012 when Krizhevsky et al. [26] utilized CNNs [45], [46] to win the annual olympics of computer vision, ImageNet large scale visual recognition challenge (ILSVRC) [28]. Using AlexNet CNN-based model, they achieved an astounding improvement as the image classification error dropped from 26% (in 2011) to 15%. ImageNet is a standard benchmark dataset used to evaluate the performance of image classification algorithms. It consists of millions of different images distributed over a thousand object classes.

CNNs have achieved even better accuracy in classification and various computer vision tasks. The classification accuracy in ILSVRC improved to 88.8% [47], 93.3% [29],

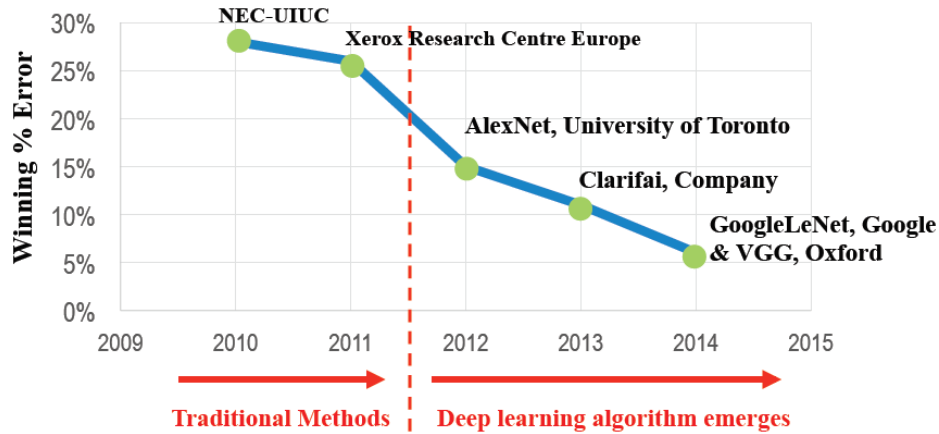


Figure 1.1: ImageNet competition results [51].

and 96.4% [48] in the 2013, 2014, and 2015 competitions, respectively. Thus, CNNs are considered as one of the most influential innovations in the field of computer vision [49]. Figure 1.1 shows the accuracy loss for the winners of ImageNet competitions before and after the emergence of DL algorithms. Thereafter, large host companies started using CNNs at the core of their services. Google, Microsoft, Facebook, Amazon, Pinterest, and Instagram are currently using CNNs for their photo search, Bing’s image feeds, automatic tagging algorithms, product recommendations, home feed personalization, and for their search infrastructure, respectively [11]. However, the classic use-case of CNNs is for image and speech processing [50].

1.1.3 The Need for Hardware Acceleration

To provide more accurate object detection and recognition results, for example in applications such as robots and auto-piloted cars, the size of the CNN needs to be increased by adding more NN layers [26]. However, evolving more and new type of NN layers results in more complex CNN structures as well as high depth CNN models.

Thus, billions of operations and millions of parameters, as well as substantial computing resources are required to train and evaluate the resultant large-scale CNN [29], [52], [53]. This makes real-time recognition/detection using general-purpose processors infeasible. Consequently, hardware accelerators such as ASIC, FPGA, and GPU have been employed to improve the throughput of the CNN. In practice, CNNs are trained off-line using the back-propagation process [54]. Then, the off-line trained CNNs model are used to perform the prediction task using the feed-forward process [55]. Therefore, the speed of feed-forward process is what matters.

GPUs are the most widely used hardware accelerators for speeding up both training and inference processes in CNNs [56]. This is due to their high memory bandwidth and throughput as they are highly efficient in floating-point vector-/matrix-based operations [57]–[59]. Furthermore, GPUs gain their performance from their ability to process a large image batch in parallel. However, for some applications like a video stream, input images should be processed frame by frame as the latency of the result of each frame is critical to the application’s performance. For some tracking algorithms, the result of one frame affects the process of the next frame [60]. On the other hand, the high performance of GPU accelerators comes at the expense of their high power consumption. Therefore, their use in CNN-based applications implemented as a cloud service on large servers or in battery operated devices becomes a challenge. Nurvitadhi et al. [61] recently evaluated emerging DNN algorithms on latest generations of GPUs (i.e., NVIDIA Titan X Pascal) and FPGAs (i.e., Intel Arria 10 GX 1150 and Intel Stratix 10 2800). The experimental results show that current trends in DNNs favor FPGA platforms as they offer higher power efficiency (a.k.a., performance per Watt).

FPGA and ASIC hardware accelerators have relatively limited memory, I/O bandwidths, and computing resources compared with GPU-based accelerators. However, they can achieve at least moderate performance with lower power consumption [62]. The throughput of ASIC design can be improved by customizing memory hierarchy and assigning dedicated resources [63]. However, the development cycle, cost, and flexibility are not satisfactory in ASIC-based acceleration of DNNs [64], [65]. As an alternative, FPGA-based accelerators are currently in use to provide high throughput at a reasonable price with low power consumption and reconfigurability [66], [67]. The availability of high-level synthesis (HLS) tools, using C or C++, from FPGA vendors lowers the programming hurdle and shortens the development time of FPGA-based hardware accelerators [68]–[70].

1.2 Background and Terminology

This section provides an overview of the key operations and terminologies used in CNNs and provides examples of well-known CNN architectures. In addition, it illustrates the basic structure of FPGAs and how DL methods can benefit from the capabilities of FPGAs.

1.2.1 Convolutional Neural Networks

A typical CNN is a multi-layered feed-forward NN with a pipeline-like architecture. Specifically, each layer performs a well-known computation on the outputs of the previous layer to generate the inputs for the next layer. In general, CNNs have two

types of inputs; the data to be tested and the model parameters. Images, audio files, and recorded videos are examples of the input data to be evaluated using CNNs. On the other hand, the weight parameters are the data generated from training the CNN on a dataset containing similar inputs to the one being tested. In this subsection, we describe the key operations involved in the construction of CNNs. Additionally, we list in this subsection some of the well-known CNN architectures.

1.2.1.1 Basic Operations in CNNs

The key operations involved in the construction of CNNs are convolution, activation functions, normalization, pooling, and inner product. Details of these operations are presented next.

A Convolution

The convolution (*CONV*) operation can be thought of as the production of a matrix smaller in size than the original input matrix (a set of N FMs) by sliding a small window (called filter, feature identifier, or kernel) of size $K \times K \times N$ over the input FMs to produce the neurons of an output FM [71]. The filter is an array of numbers called weights or parameters. These weights are computed during the training phase. As the filter slides over the input FMs, it multiplies the values in the filter with the input FMs values, that is, it first performs element-wise multiplication, and then sums the products, to produce a single number.

This operation, starting from the top left corner of the FMs, is repeated by moving the window S strides at a time, first in the right direction, until the end of the FMs is

reached, and then proceeding downwards until the FMs is completely scanned and all the elements of the FMs are covered. The sliding of the window and performing the multiply-and-accumulate (MAC) operation is known by the verb *convolving*, hence the noun *convolution* [11], [72].

Typically, the *CONV* layer employs M filters to extract the unique features from the input FMs, each of which generates one output FM. That is, the m -th filter produces the m -th output FM. Normally, the dimension of the kernel is very small, K is less than or equal to 11. The number of *CONV* kernels, their size, and the window stride are defined by the CNN designers. In summary, the *CONV* operation comprises four main levels of loops; the output FMs loop, the loop across the input FMs, the loop along the dimensions of the input FMs (scan operation), and the kernel window dimensions loop (MAC operation).

B Activation Functions

Activation functions in NNs is similar to action potential in animal cells such as neurons. A neuron is said to fire if it emits an action potential. Activation functions help learn and model complex data relations. A popularly used activation function is the *sigmoid* function which can be expressed as

$$f(x) = \frac{1}{1 + e^{-x}} \quad (1.1)$$

where x represents the input neuron and if it is a sufficiently large positive number, the *sigmoid* function approximates to unity. For sufficiently large negative values of

x , the *sigmoid* function is close to 0. Another popular activation function is

$$f(x) = \tanh(x) \tag{1.2}$$

The above standard *sigmoid* and *tanh* non-linear functions require long training time [26]. A commonly used activation function in CNNs is the rectified linear unit (*ReLU*) which is defined as

$$f(x) = \max(0, x) \tag{1.3}$$

The *ReLU* activation function is known to converge faster in training, and has lesser computational complexity than standard *sigmoid* and *tanh* functions [73], [74]. In addition, it does not require input normalization to prevent it from saturating [75].

C Normalization

In real life, a phenomenon called lateral inhibition appears, which refers to the capacity of an excited neuron to subdue its neighbors, thereby creating a contrast in that area. In CNNs, to accomplish this, local response normalization or simply normalization (*Norm*) is used. It detects high frequency features with a large response. If we normalize around the local neighborhood of the excited neuron, it becomes even more sensitive as compared to its neighbors. At the same time, it will dampen the responses that are uniformly large in any given local neighborhood. If all the values are large, then normalizing those values will diminish all of them. So, basically it performs some kind of inhibition and boosts the neurons with relatively larger activations. Normalization can be done within the same feature or across neighboring features

by a factor that depends on the neighboring neurons. Expressions to compute the response normalized activity can be found in [26], [73].

D Pooling

The Pooling (*POOL*), also known as subsampling, is employed to progressively reduce the spatial size of the representation, thereby reducing the amount of parameters and computation in the network. Moreover, it extracts rotational and positional invariant features, and suppresses noise. The *POOL* layers are periodically inserted in between successive *CONV* layers. They operate independently on every depth slice of the input and resize it spatially using the *MAX* operation. The most common form is a layer with a window of size 2×2 applied where the *MAX* operation would be taking a maximum over 4 samples thereby discarding 75 percent of the activations [76]. In addition to the popular *MAX-POOL*, the pooling units in some CNNs are also used to perform other functions, such as *AVG* and *MIN* operations [73].

E Fully-Connected

A common form of a CNN architecture comprises stacks of a few *CONV* and *ReLU* layers, followed by layers for *POOL*, and this pattern is repeated until the image has merged spatially to a small size. This is followed by one or more fully-connected (*FC*) layers, also known as inner-product layers, whose neurons have full connections to all activations in the previous layer, hence the name. The neuron values are calculated by the MAC operation on the input activations and the connection weights. The last *FC* layer is the classification layer and it holds the output such as the class scores [73].

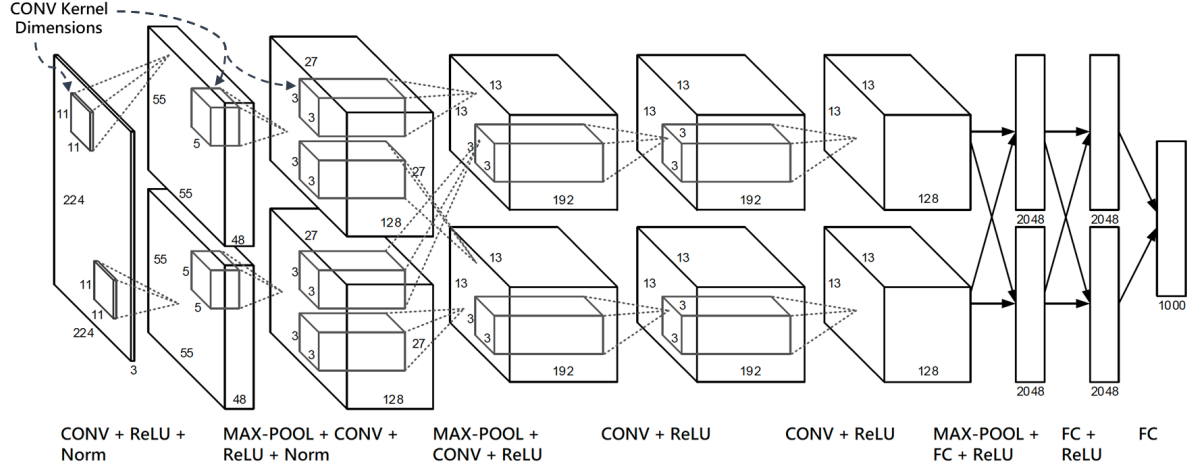


Figure 1.2: An illustration of AlexNet architecture. The input/output FMs to/from each layer(s) are shown as matrices with their respective dimensions and the operations performed are indicated below.

1.2.1.2 Examples of CNN Architectures

The following list describes the details of some well-known CNN architectures.

- **AlexNet (2012)** is a CNN consisting of 5 *CONV* layers, interspersed by 2 *Norm* layers, as well as 3 *FC* layers [26]. Each *CONV* layer performs the activation function using *ReLU*. In addition, 3 *POOL* layers are employed with the first, second, and last *CONV* layers. The architecture of AlexNet is shown in Figure 1.2. AlexNet won the 2012 ImageNet challenge by classifying 224×224 input color images to 1,000 different output classes.
- **VGG (2014)** is a CNN model similar to AlexNet in terms of the number of *FC* layers. However, it consists of 5 groups of *CONV* layers [27], [74]. The exact number of *CONV* layers in each group depends on the version of the visual geometry group (VGG) model. Table 1.1 shows the number of *CONV* and *FC* layers for the most commonly used VGG models.

Table 1.1: CNN layers for VGG models.

Layers	VGG-11	VGG-16	VGG-19
<i>CONV</i> (Group 1)	1	2	2
<i>CONV</i> (Group 2)	1	2	2
<i>CONV</i> (Group 3)	2	3	4
<i>CONV</i> (Group 4)	2	3	4
<i>CONV</i> (Group 5)	2	3	4
<i>FC</i>	3	3	3
Total	11	16	19

- **ResNets (2016)** are deep residual networks with extremely irregular and complex structures compared to AlexNet and VGG CNN-based models [48], [77], [78]. This is due to having more types of layers, where non-adjacent layers incorporate shortcuts to compute the residual functions, as well as having highly deep structures. Unlike AlexNet and VGG models where the layers are connected in sequence, the interconnections in ResNet layers are in the form of a directed acyclic graph (DAG). ResNet-18 and ResNet-101 are widely used CNNs, especially for image classification. ResNet-18/101 structure contains 21/104 *CONV* (most of them are followed by batch normalization (*BatchNorm*) and *ReLU* layers), 1/1 *MAX-POOL*, 1/1 *AVG-POOL*, 1/1 *FC*, and, 8/33 element-wise (*Eltwise*) addition layers, respectively.

1.2.2 Field Programmable Gate Arrays

FPGAs are off-the-shelf programmable devices that provide a flexible platform for implementing custom hardware functionality at a low development cost. They consist mainly of a set of programmable logic cells, called configurable logic blocks (CLBs), a

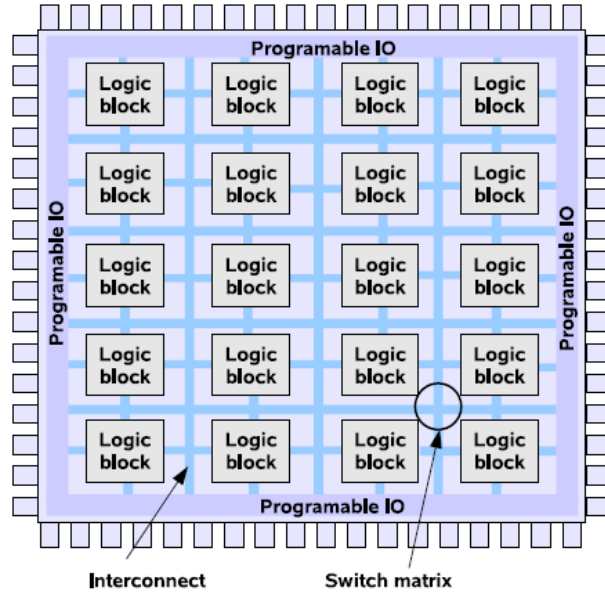


Figure 1.3: FPGA basic structure [79].

programmable interconnection network, and a set of programmable input and output cells around the device [79]. In addition, they have a rich set of embedded components such as digital signal processor (DSP) blocks which are used to perform arithmetic-intensive operations like MAC, block random-access memories (BRAMs), look-up tables (LUTs), flip-flops (FFs), clock management unit, high speed I/O links, and others. Figure 1.3 shows a basic structure of an FPGA.

FPGAs are widely considered as accelerators for computationally-intensive applications as they enable models with highly flexible fine-grained parallelism and associative operations such as broadcast and collective response [80]. In [81], [82], FPGA computing models used for applications acceleration are presented, including data streaming, associative computing, highly parallel memory access, use of standard hardware structures such as first in first out (FIFO) buffers, stacks and priority queues, and functional parallelism.

FPGAs have the advantage of maximizing performance per Watt of power consumption, reducing costs for large scale operations [83]. This makes them an excellent choice as accelerators for battery operated devices and in cloud services on large servers. FPGAs have recently been widely used for DL acceleration given the flexibility in implementing architectures with large degree of parallelism resulting in high execution speeds [84]. The feed-forward nature of DL algorithms makes FPGAs offer a clear advantage as they can create customized hardware circuits that are deeply pipelined and inherently multi-threaded [83]. FPGAs have also the capability of partial dynamic configuration, which allows part of the FPGA to be reconfigured while the rest is being used. This could be of potential benefit to DL methods where the next layer could be reconfigured while the current layer is being used.

1.3 Motivation

In recent years, CNNs have achieved great success in a large number of applications and have been among the most powerful and widely used techniques in computer vision. While achieving state-of-the-art results, CNN-based methods demand much more computations and memory resources compared with traditional methods. As the name implies, *CONVs* are the dominant layers in CNNs and their operations often constitute more than 90% of the total computation time [26], [27], [48], [85]–[87]. Precisely, each *CONV* layer performs $M \times R \times C \times N \times K^2$ MAC operations. Here, N is the number of input FMs, M is the number of output FMs, K is the kernel window size, R is the height of output FMs, and C is the width of output FMs.

The intensive computational requirements of CNNs limit their implementation on general CPUs. For instance, AlexNet and VGG-16 models require about 0.72 giga floating-point operations (GFLOPs) and 15.3 GFLOPs to process a single RGB image of size 224×224 pixels, respectively [88]. However, CNN-based systems such as self-driving cars have to provide minimum latency overhead as well as high throughput such that making the right decision happen in time. On the other hand, the computational efficiency of GPUs and FPGAs make them excellent platforms for accelerating CNNs. In particular, the critical need for lower power consumption in today’s applications, due to CNN deployment on battery-powered devices such as unmanned aerial vehicles (UAVs) and Internet of things (IoT) [89], [90], makes FPGAs more attractive for CNN acceleration. This is because of the high power efficiency, also known as performance per Watt, that FPGAs offer. Furthermore, FPGA-based accelerators have the ability to deliver the requirements of CNN-based applications at a reasonable price [91].

Therefore, many attempts have been made to speed up *CONV* operations by designing a hardware accelerator known as a convolutional layer processor (CLP) [55], [73], [92]–[94]. The CLP maximizes the parallelism of *CONV* computation using loop unrolling technique [95], which replicates the hardware resources of MAC unit. Figure 1.4 illustrates unrolling the four main loops of *CONV* operation. However, different *CONV* layers, even those in the same CNN model, vary considerably in their configurations (N , M , K , S , R , and C). Moreover, different CNN layers exhibit vastly different amounts of intra-output and inter-output parallelism [96]. Intra-output parallelism parallelizes the computation of a single output FM since it is the sum of kernel

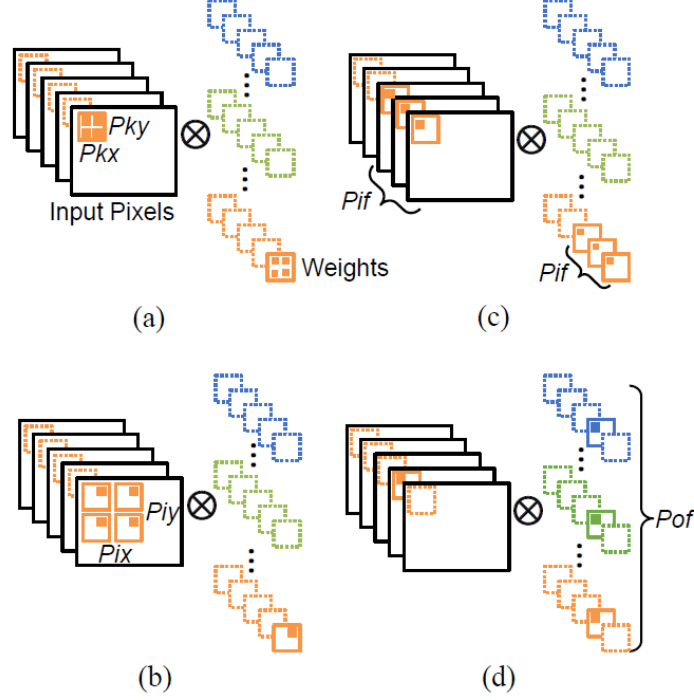


Figure 1.4: Convolution loops unrolling [97]: (a) Unrolling *Loop-1*; (b) Unrolling *Loop-2*; (c) Unrolling *Loop-3*; (d) Unrolling *Loop-4*, where, P_{kx} , P_{ky} , P_{ix} , P_{iy} , P_{if} , and P_{of} are loop unrolling design variables for the kernel window width, kernel window height, input FM width, input FM height, number of input FMs, and the number of output FMs, respectively.

convolutions of all input FMs. On the other hand, inter-output parallelism is based on computing multiple output FMs in parallel. Thus, the developed CNN accelerator must be designed carefully to meet the varying requirements of different layers and needs to be flexible to maximize the performance for each CNN layer.

Without careful implementation of CNNs and maximizing resource sharing, the implementation may not fit on FPGAs due to limited logic resources. The growing complexity of CNN architectures makes optimizing the use of computational resources available on the target FPGA device to deliver the optimal performance more challenging. This is because of the exponential increase in the design variables that must be considered when implementing CNN accelerator as CNN's complexity increases. It

is also expected that future CNN models will get more complex with larger number of layers as the amount of training data continues to grow and the problems to be solved get more complex.

Designing an FPGA-based hardware accelerator of CNNs has further challenges including the requirement of a significant amount of storage and large external memory bandwidth [98]. For example, AlexNet has over 60 million model parameters which need 240MB of memory for storing the weights using 32-bit floating-point representation [73]. The problem exacerbates with more complex models such as VGG which has 16 *CONV* layers with 138 million weight parameters. This large amount of on-chip storage is not supported by existing commercial FPGAs. Thus, weights and activations have to be stored on external memory and transferred to FPGA’s on-chip memories during computation on need basis. The significant amount of storage and external memory bandwidth required become a performance bottleneck. The CLP’s computation throughput needs to well match the memory bandwidth provided by an FPGA platform. Otherwise, the designed CLP will not be able to achieve the best performance due to under-utilization of either logic resource or memory bandwidth.

The authors in [99], [100] proved that the highest energy expense results from accessing the off-chip dynamic random-access memory (DRAM) for data movement. In other words, the energy cost of the increased memory accesses and data movement due to the large number of CNN operations often exceeds the energy cost of computation [64], [86]. Under 45nm CMOS technology, the energy consumption of a 32-bit floating-point adder, a 32-bit on-chip random-access memory (RAM) access, and a 32-bit off-chip DRAM access are 0.9pJ, 5pJ, and 640pJ, respectively. One can

note that DRAM access is 3 orders of magnitude of an add operation [101]. Since DNNs do not fit in on-chip storage and hence require more costly DRAM accesses, the power consumption just for DRAM accesses exceeds the power envelope of typical edge devices. Thus, CNN accelerators need to carefully consider data reuse to achieve efficient architecture in terms of time and power consumption.

Low-precision representation of CNN data-structures would bring great benefits to specialized DL hardware especially when expensive floating-point operations can be avoided and replaced by more efficient fixed-point operations [102], [103]. In particular, lowering the bit-precision level for weights and activations reduces external memory accesses, I/O bandwidth requirements, and inference latency. This leads to better energy efficiency especially when on-chip memory is exploited to improve data reuse.

Additionally, reducing the bit-precision level lessens the logic resources required to implement the MAC operation, which increases the number of processing units that can be designed on the device. In this way, the DL hardware accelerator will be able to process more samples per unit time, allowing a reduction in the number of accelerators required on-site, and thus reducing the overall deployment cost. However, the rigorous quantization leads to a severe accuracy drop. As such, quantization opens a large hyper-parameter space at bit-precision levels, the exploration of which is a major challenge. The potential impact of this topic in academia and the commercial world is one of the major motives for conducting this research.

In this dissertation, we provide a couple of works for addressing the requirements for efficient implementations of CNNs. These works optimize CNN implementation on FPGA platforms in terms of throughput, latency, energy efficiency, and power

consumption. Particularly, (i) a CNN accelerator and an accompanying automated design methodology that employs metaheuristics for optimal partitioning of available FPGA resources to design a high-throughput Multi-CLP accelerator is discussed in Chapter 3, (ii) a framework to efficiently quantize the data-structures of pre-trained CNN-based models to low-precision numbers in fixed-point representation is discussed in Chapter 4, and (iii) a novel end-to-end methodology to enable deployment of DNNs on resource-constrained edge-devices while maintaining model’s accuracy is discussed in Chapter 5.

1.4 Objectives and Scope

The main objectives of the dissertation are to investigate mechanisms for addressing the requirements for efficient implementation of CNNs on FPGA platforms. This includes reusing computational resources and efficient storing of partial results in internal memories to address hardware resource limitations, reordering *CONV* operations to minimize off-chip data transfers, optimizing *CONV* parallelism parameters to maximize computational resource usage, and reducing bit-precision level for weights and activations to significantly reduce external memory accesses, internal buffer sizes, and logic resources required for computations. Thus, the objectives can be listed as follows:

- Conduct a comprehensive survey on recent existing techniques and architectures for implementing CNNs on FPGAs that discusses key features employed by the various techniques to enhance the performance and provides recommendations

for future directions that will simplify the use of FPGA-based accelerators [91].

- Develop a CNN accelerator and an accompanying automated design methodology that employs metaheuristics to optimize the design of CNNs by partitioning available FPGA resources to design a Multi-CLP accelerator. This includes (i) designing a CLP that optimizes the implementation of *CONV* layers by applying loop transformation techniques, (ii) developing an analytical and empirical model to accurately predict the hardware resources usage and performance for a candidate multiple CLPs design given its parameters, and (iii) proposing a metaheuristic-based optimization framework that employs the estimation models to efficiently explore the design space and finds the optimal number of CLPs required and their respective configurations to achieve optimal performance on a given target FPGA device [104].
- Develop a framework that flexibly designs mixed low-precision CNNs for integer-arithmetic-only deployment on hardware devices while still preserving accuracy at a high rate, thus providing unique computational and hardware efficiencies without causing noticeable degradation in application performance [105].
- Develop a novel end-to-end methodology for enabling the deployment of DNNs on resource-limited devices while maintaining model’s accuracy. The framework aims at learning the bit-precision level of every weight and activation data-structure that fulfill the memory constraints of a particular target device [to be submitted].

1.5 Summary

In this chapter, we provided an overview on DNNs and their need for hardware accelerators to improve inference throughput to provide the desired level of performance for applications. Next, the chapter presented general background information about CNNs, their key operations, and some well-known CNN architectures. In addition, it introduced the basic structure of FPGAs and highlighted the features that enable them to accelerate computation-intensive applications and how DL methods can benefit from the capabilities of FPGAs. Finally, the chapter discussed the challenges of implementing CNNs on FPGAs and how this research work overcomes these challenges to design a scalable and efficient FPGA-based accelerator of CNNs.

CHAPTER 2

LITERATURE REVIEW

In this chapter, we review recent existing techniques for accelerating deep learning neural networks (DNNs) on field programmable gate arrays (FPGAs). We highlight the key features employed by the various techniques for improving the acceleration performance. Additionally, we describe the use of metaheuristics in the design and optimization of convolutional neural network (CNN) implementation. In general, the hardware accelerators focus on designing specific modules and architectures that ensure data reuse, enhance data locality, and accelerate convolution (*CONV*) layer operations based on performing needed operations in parallel. Furthermore, we summarize existing design approaches for accelerating DNNs and provide recommendations for future directions that will simplify the use of FPGA-based accelerators and enhance their performance. The techniques investigated in this chapter represent the recent trends in FPGA-based accelerators of DNNs. Thus, this review is expected to direct the future advances on efficient hardware accelerators and to be useful for deep learning (DL) researchers.

2.1 FPGA-based Accelerators of DNNs

FPGA implementations of DNNs appeared in the mid-1990's when Cloutier et al. [106] designed the virtual image processor (VIP) on Altera EPF81500 FPGA. The VIP is a single-instruction stream multiple-data streams (SIMD) multiprocessor architecture with a 2D torus connection topology of processing elements (PEs). VIP improves the performance through the use of low-accuracy arithmetic to avoid implementing full-fledged multipliers. Fortunately, recent digital signal processor (DSP)-oriented FPGAs include large numbers of multiply-and-accumulate (MAC) units which allow for extremely fast and low-power CNN implementations. Thereafter, FPGA implementations of DNNs have mainly focused on accelerating the computational engine through optimizing *CONV* layer operations. Several studies in the literature [107]–[113] have reported FPGA-based implementations of *CONV* operation.

Farabet et al. [114] presented an FPGA implementation of CNN that uses one dedicated hardware convolver and a soft-processor for data processing and controlling, respectively. The proposed implementation is referred to as convolutional network processor (CNP). The CNP exploits the parallelism of *CONV* layers to accelerate the computational engine of CNNs while fully utilizing the large number of DSPs, the MAC hardware units on FPGA. The proposed architecture consists of Virtex-4 SX35 FPGA platform and external memory. The authors designed a dedicated hardware interface with the external memory to allow 8 simultaneous read/write accesses transparently. In addition, they used first in first out (FIFO) buffers between the FPGA and the external memory chip in both directions to guarantee the steadiness

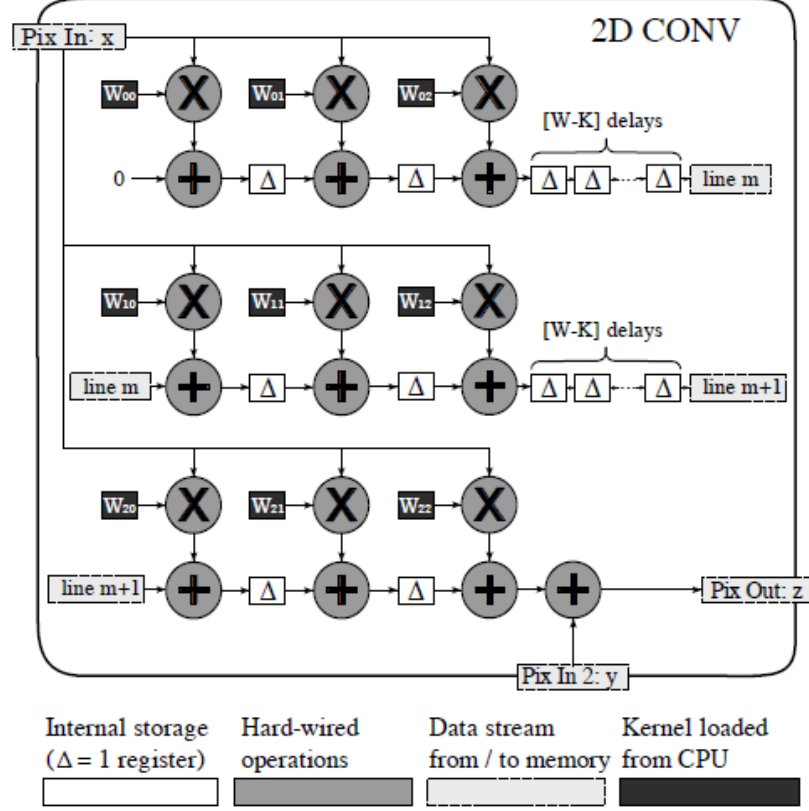


Figure 2.1: 2D *CONV* module of 3×3 kernel [114].

of dataflow.

The vector arithmetic and logic unit in CNP implements 2D *CONV*, pooling (*POOL*), and non-linear activation function operations of CNNs. The implementation of 2D *CONV* with kernel of size 3 (i.e., $K = 3$) is shown in Figure 2.1, where x is the feature map (FM) data received from input FM (IFM), y is the partial result to be accumulated with the current result, z is the result to the output FM (OFM), W_{ij} is the weight value in the *CONV* kernel, and W is the width of the input image. It can be seen that the proposed *CONV* module accomplishes K^2 MAC operations simultaneously in each clock cycle. CNP represents FMs and weights using 16-bit (Q8.8) fixed-point format. The proposed accelerator has been implemented for a face

detection system with LeNet-5 architecture [115]. It utilized 90% and 28% of the general logic and multipliers, respectively. In addition, CNP consumed less than 15 Watts of power.

Sankaradas et al. [116] proposed a massively parallel coprocessor to accelerate CNNs using Virtex-5 LX330T FPGA platform. The proposed accelerator mainly focused on optimizing computation engine by employing the parallelism within *CONV* kernel and FMs. The coprocessor can be considered as parallel clusters of vector PEs where each cluster is designed using 2D convolvers, adders, sub-samplers, and look-up tables (LUTs). Each vector PE consists of MAC and programmable register units to hold kernel weights and FM data. To hold the massive intermediate data of CNNs, the authors employed a dedicated off-chip memory (4 DDR2 memory banks) with a large bandwidth on the coprocessor card. Moreover, the proposed accelerator uses a low-precision data representation feature with memory packing to further improve the memory bandwidth as well as the throughput. 20-bit and 16-bit fixed-point representations were utilized for kernel weights and FMs, respectively.

Sankaradas et al. examined their architecture on CNN with 4 *CONV* layers and without any fully-connected (*FC*) layer for a face recognition application. The results show that the proposed coprocessor is $6\times$ faster than a software implementation on a 2.2 GHz AMD Opteron processor with less than 11 Watts of power dissipation. However, the proposed accelerator cannot be used to accelerate full CNNs as it uses few *CONV* layers without any *FC* layer. A full CNN model consists of both *CONV* and *FC* layers. Thus, an efficient CNN accelerator for real-life applications is needed to consider both. Similar approaches to the work of Sankardas et al. [116] are presented

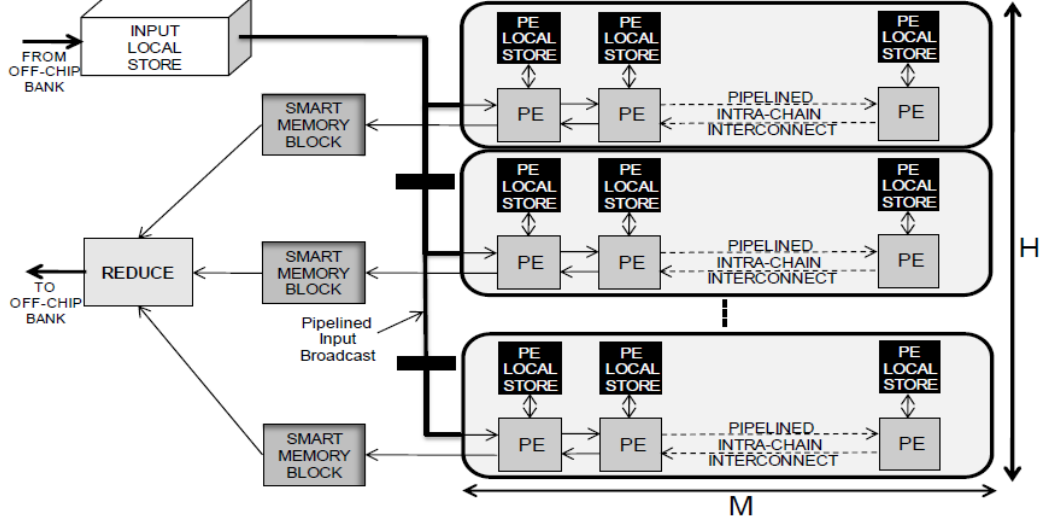


Figure 2.2: MAPLE processing core architecture [119].

in [117], [118] to accelerate support vector machines (SVMs).

MAPLE [119] is a programmable FPGA prototype system presented to accelerate both learning and classification tasks in applications with unstructured large amount of data. The authors analyzed five workload domains to help in designing MAPLE. These workloads are SVM [120], supervised semantic indexing (SSI) [121], K-means [122], generalized learning vector quantization (GLVQ) [123], and CNNs [124]. They found that their computations can be structured as parallel streams of vector or matrix operations. Thus, they designed MAPLE as a 2D grid of vector PEs as shown in Figure 2.2. To efficiently perform matrix multiplication, they allocate a private local storage to each PE which is used to store a column, or part of it, from the multiplier matrix. In this way, matrix multiplication is accomplished by streaming the multiplicand matrix rows through the PEs where each PE performs a MAC operation. The PEs are organized in clusters, where each group is served by a separate memory bank of the banked off-chip memories, which create independent streams for processor-memory

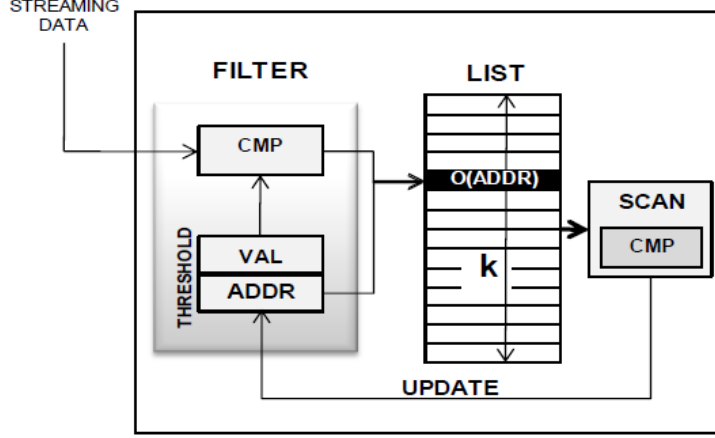


Figure 2.3: MAPLE smart memory block [119].

computation.

Moreover, MAPLE uses on-chip smart memory blocks to process the large intermediate data on-the-fly using in-memory processing. Figure 2.3 shows the architecture of the smart memory block. To illustrate the idea of on-the-fly in-memory processing, let's consider finding the maximum K elements. The filter compares the input data with the threshold value (VAL). If the input value is greater than VAL, it updates the list by replacing VAL at address ADDR with the input value. Then, the scanner (SCAN) searches for the new minimum value in the list and updates the threshold VAL and ADDR accordingly. It should be mentioned here that the employment of in-memory processing reduced the off-chip memory traffic by $1.64\times$, $25.7\times$, and $76\times$ for SSI, K-means, and CNN workloads, respectively. MAPLE prototype has been implemented on Virtex-5 SX240T platform running at 125 MHz. The experimental results for face and digit recognition CNNs [125]–[127] show that MAPLE is 50% faster than that for 1.3 GHz NVIDIA Tesla C870 graphic processing unit (GPU) implementation.

Chakradhar et al. [96] proposed a dynamically configurable CNN (DC-CNN) ar-

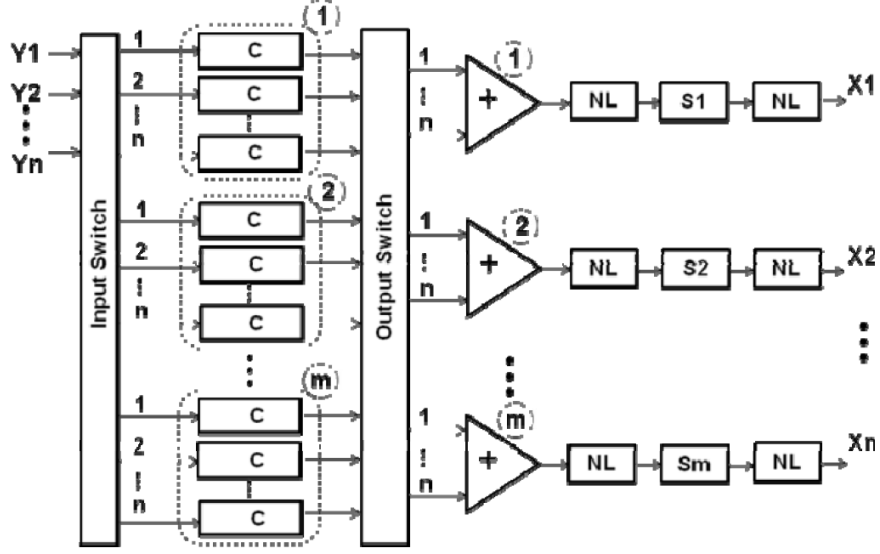


Figure 2.4: The architecture of DC-CNN [96].

chitecture on FPGA. The proposed system consists of three main components; a coprocessor, a DC-CNN processing core, and 3-bank memory subsystem. The coprocessor is designed such that it automatically configures the software and the hardware elements to fully exploit the parallelism at the workload level. The DC-CNN processing core is responsible for executing CNN applications and its architecture is shown in Figure 2.4. It consists of m computational elements (each with n 2D convolvers as well as sub-sampling (S) and non-linearity (NL) pipelined units), m adders (each with n inputs), and input/output switches. The internal structure of the switches vector encloses $m \times n$ selectors which are used to help in exploring the entire design space and to provide the configurability function across different layers of CNN model. To determine the best (m, n) feasible combination for each layer, the system analyzes the workload using integer factorization techniques because it is considered fast for small numbers [128], [129]. Dynamic programming is also used to quickly prune infeasible combinations.

Chakradhar et al. compared the proposed DC-CNN architecture, considering 20 2D convolvers as well as a memory subsystem of 128-bit port width, with a 1.35 GHz NVIDIA’s GPU implementation. The results show that DC-CNN achieved $4.0\times$, $4.4\times$, $5.4\times$, $6.0\times$, and $6.5\times$ speedup for face recognition [125], face detection [127], mobile robot vision [114], video surveillance [96], and automotive safety [96] workloads, respectively. It is worth mentioning that DC-CNN is the first architecture that achieves a performance suitable for real-time processing for video streaming as it processes up to 30 frames per second (fps). In addition, DC-CNN is more energy-efficient than the GPU implementation as it consumes 14 Watts, while more than 150 Watts are consumed by the GPU. On the other hand, the authors modeled their architecture on a CNN with 3 *CONV* layers only without any *FC* layer which makes it unsuitable for today’s other real-life applications.

A second-generation of CNP [114] architecture has been proposed in [130] by designing a stream processor system. The proposed design replaces the dedicated hardware convolver in CNP with multiple parallel vector processing units, named as ALUs, laid out in a 2D grid. Each ALU is composed of four local routers, one global router, and a streaming operator. The local routers are used to stream data to/from the neighbors. Streaming data to and from global data line is done through the global router. The streaming operators in the ALU are fully pipelined to produce a result per clock cycle as described in [114] with the use of Q8.8 coding to represent FMs and weights. The proposed system also uses a multi-port direct memory access (DMA) streaming engine to allow individual streams of data to operate seamlessly within processing blocks. The results show that the proposed stream processor system can run

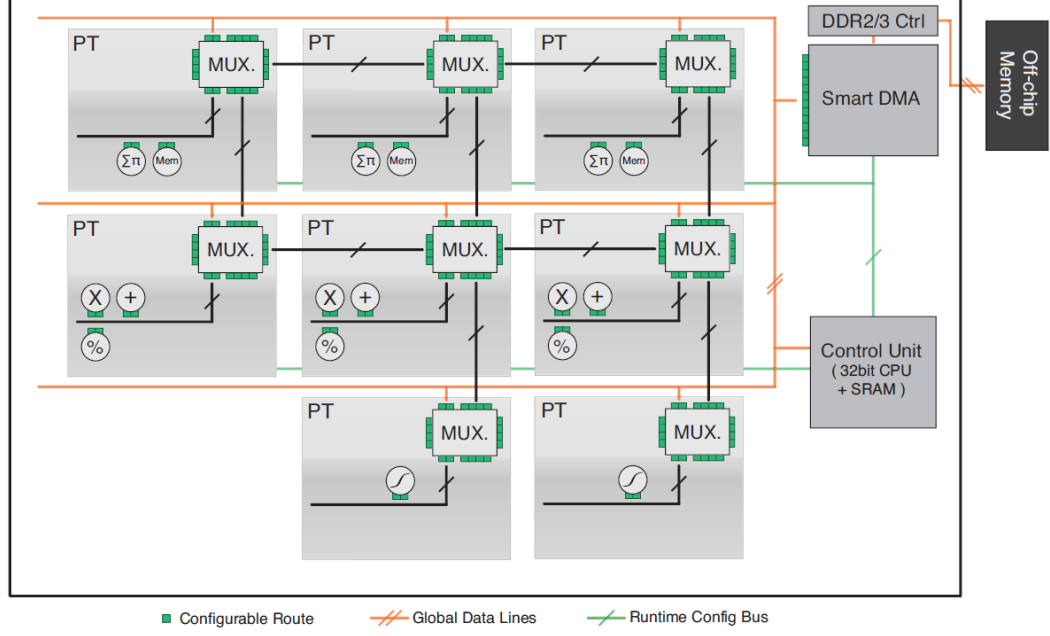


Figure 2.5: The architecture of neuFlow [131].

small CNNs at up to 30 fps while consuming about 15 Watts.

An improved version of CNP architectures given in [114], [130] was presented in [131] and referred to as neuFlow. Particularly, neuFlow has replaced the 2D grid of ALUs in [130] with a 2D grid of processing tiles (PTs). The proposed architecture contains a 2D grid of PTs, a control unit, and a smart DMA module, as shown in Figure 2.5. Each PT consists of local operators and a routing multiplexer (MUX). The top three PTs have been implemented to perform MAC operation. Thus, they can be used to perform 2D *CONV*, simple dot-products, and spatial *POOL*. General-purpose operations, such as dividing and squaring, have been implemented at the middle three PTs. Therefore, the middle row of neuFlow can be used for normalization (*Norm*). Finally, neuFlow’s bottom PTs row implements non-linear operations. Moreover, each operator employed input and output FIFOs to stall its pipeline when required. On the other hand, PT’s MUX is used to connect its local operators with the neighboring

PT’s streaming operators and off-chip memory instead of the used local routers and global router discussed in [130].

NeuFlow uses a dataflow compiler, named luaFlow, to translate a high-level flow-graph representation of CNNs in Torch5 [132] into hardware description language (HDL) scripts with different levels of parallelism. In addition, luaFlow produces a binary code configuration file and holds it in the embedded control unit. Thereafter, the control unit configures the 2D grid of PTs (connections and streaming operator) and the DMA ports through run-time configuration buses. A smart memory module has been designed to support multiple asynchronous accesses of off-chip memory through its reconfigurable ports. By targeting the larger Xilinx Virtex-6 VLX240T FPGA, neuFlow achieved 147 GOPs at 10 Watts for street scene parsing CNN in [133] with the use of 16 bits to represent FMs and weights.

Peemen et al. [134] utilized the flexible off-chip memory hierarchy method to design a configurable memory-centric accelerator template for a variety of models of CNNs. This accelerator exploits data reuse in complex access patterns to reduce off-chip memory communication, which minimizes the bandwidth requirements. The memory-centric accelerator maximizes the efficiency of on-chip memories for better data locality using loop transformation (to optimize the tiling parameters) and block random-access memory (BRAM)-based multi-bank on-chip buffers [135]. At the same time, it minimizes the size of FPGA on-chip memories to optimize energy and area usage, which are key requirements for embedded platforms.

The memory-centric accelerator uses a SIMD cluster of MAC PEs with flexible reuse buffers to accelerate the *CONV* layer. The acceleration template has been

implemented on Virtex-6 FPGAs. In addition, a MicroBlaze processor has been utilized to configure and communicate with the accelerator via FIFO-based fast simplex link. The proposed accelerator has been analyzed for a CNN vision task of size 2.74 GMAC and the results show that the memory-centric accelerator is $11\times$ faster than the standard implementation of similar FPGA resources.

Neural network next (nn-X) [136] is a real-time system-on-chip (SoC) computing system for DNNs on mobile devices. The architecture of nn-X consists of a host processor, a coprocessor, and external memory. The coprocessor accelerates the learning networks by parallelizing their operations throughout arrays of configurable PEs referred to as collections. Each collection contains one *CONV* engine, one *POOL* module, and one non-linear operator. The *CONV* engine accelerates the *CONV* operation by fully pipelining the incoming data with the use of cache memories. The collections are able to communicate with one another using the collection route component to achieve cascaded pipelining, which results in reducing accesses to external memory. The data transfer between the collections and the external memory is accomplished throughout the coprocessor full-duplex memory router, which provides independent data streams. The nn-X has been prototyped on Xilinx ZC706 which contains Zynq XC7Z045, two ARM Cortex-A9 processors, and 1 GB DDR3 memory. Eight collections have been employed to achieve large parallelism. The results for face recognition model in [137] show that nn-X is $115\times$ faster than the two embedded ARM processors.

Zhang et al. [55] proposed a roofline-based model to accelerate CNNs on FPGAs. The roofline model is an intuitive visual performance model used to relate the attainable performance to the peak performance that can be provided by the hardware

platform and the off-chip memory traffic [138]. The focus in their work is primarily on accelerating the *CONV* layers as it consumes more than 90% of the computational time during the prediction process [85]. In doing so, the authors optimized both the computation operations and the memory access operations in *CONV* layers. They considered a CNN application composed of five *CONV* layers that won the ImageNet competition in 2012 [26]. The proposed accelerator uses polyhedral-based data dependence analysis [139] to fully utilize all FPGA computational resources through loop unrolling, loop pipelining, and loop tile size enumeration. Note that loop unrolling maximizes the parallel computation of *CONV* MAC operations. On the other hand, local memory promotion and loop transformation are used to reduce redundant communication operations and to maximize the data sharing/reuse, respectively.

Subsequently, the roofline performance model is used to identify the optimal design from all possible solutions in the design space. Specifically, Zhang et al. model all possible legal designs delivered from the polyhedral analysis in roofline to find the optimal unrolling factor $\langle T_m, T_n \rangle$ for every *CONV* layer, where T_m and T_n are the tile size for the OFMs and IFMs, respectively. However, designing a CNN accelerator with different unrolling factors to each *CONV* layer is challenging. Therefore, the proposed architecture enumerates all possible valid designs to find uniform cross-layer unrolling factors. Thereafter, the hardware accelerator is implemented based on the cross-layer optimal unrolling factors.

The proposed accelerator composed of a computational engine and memory subsystem is depicted in Figure 2.6. The computation engine is designed as T_m duplicated tree-shaped poly structures with T_n inputs from the IFMs, T_n inputs from the weights,

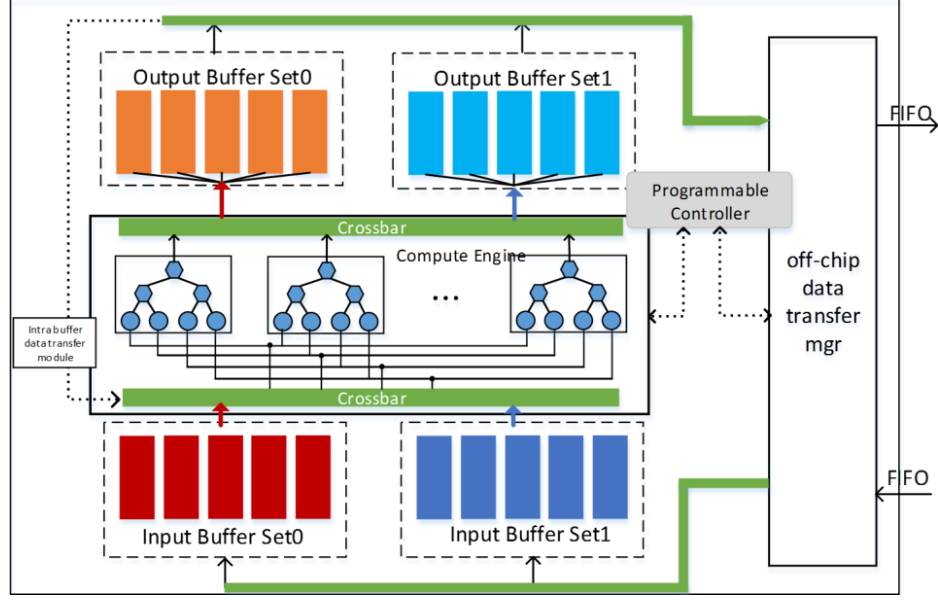


Figure 2.6: Zhang et al. [55] accelerator architecture.

and one input from the bias. On the other hand, the memory sub-system is implemented as four sets of on-chip buffers; two sets to store the IFMs and weights, each with T_n buffer banks, and two buffer sets of T_m independent banks for storing the OFMs. To overlap data transfer with computation, on-chip buffers are operated in a ping-pong manner. In addition, two independent channels are implemented for load and off-load operations to increase the bandwidth utilization. Moreover, MicroBlaze processor is used to send configuration parameters and commands for the accelerator over AXI4-lite bus. The CNN accelerator communicates with external data transfer engines through FIFO interfaces, where the data transfer engines are used to access DDR3 DRAM through AXI4 bus.

The accelerator is designed using Vivado 2013.4 high-level synthesis tool and implemented on Xilinx VC707 FPGA board clocked at 100 MHz. The experimental results depict that the proposed implementation achieves a peak performance of 61.62

GFLOPs as well as a $17.42\times$ speedup over the software implementation on Intel Xeon central processing unit (CPU) E5-2430 at 2.20 GHz with 15 MB cache and 16 threads. In addition to this, the results show that the proposed FPGA architecture is $24.6\times$ more energy-efficient than the software implementation as the total power consumption is only 18.6 Watts. The proposed implementation has some limitations such as designing the accelerator with new cross-layer unrolling factors for different architectures of CNNs. Furthermore, using the CNN accelerator with uniform unrolling factors might be sub-optimal for some *CONV* layers, which affects the overall performance.

In 2014, Microsoft research team of Catapult project integrated FPGA boards into data center applications to successfully achieve $2\times$ speedup for Bing Ranking (the large-scale search engine) [67]. A year later, Ovtcharov et al. [140] at Microsoft Research utilized Catapult hardware infrastructure, a dual-socket Xeon server equipped with Stratix-V GSMD5 FPGA, to design a specialized hardware for accelerating the forward propagation of deep CNNs in a power-constrained data center.

The top-level architecture of the proposed CNN accelerator is shown in Figure 2.7. Multi-banked input buffer and kernel weight buffer are used to provide an efficient buffering scheme of FMs and weights, respectively. To minimize the off-chip memory traffic, a specialized network on-chip has been designed to re-distribute the OFMs on the multi-banked input buffer instead of transferring them to the external memory. The 3D *CONV* operations (such as the dot-product) and other CNN operations are independently performed using spatially distributed scalable vectors of PEs. The controller engine is responsible for streaming and data delivery of multi-banked input buffer and kernel weight buffer data to each of the PE vectors. In addition, it supports

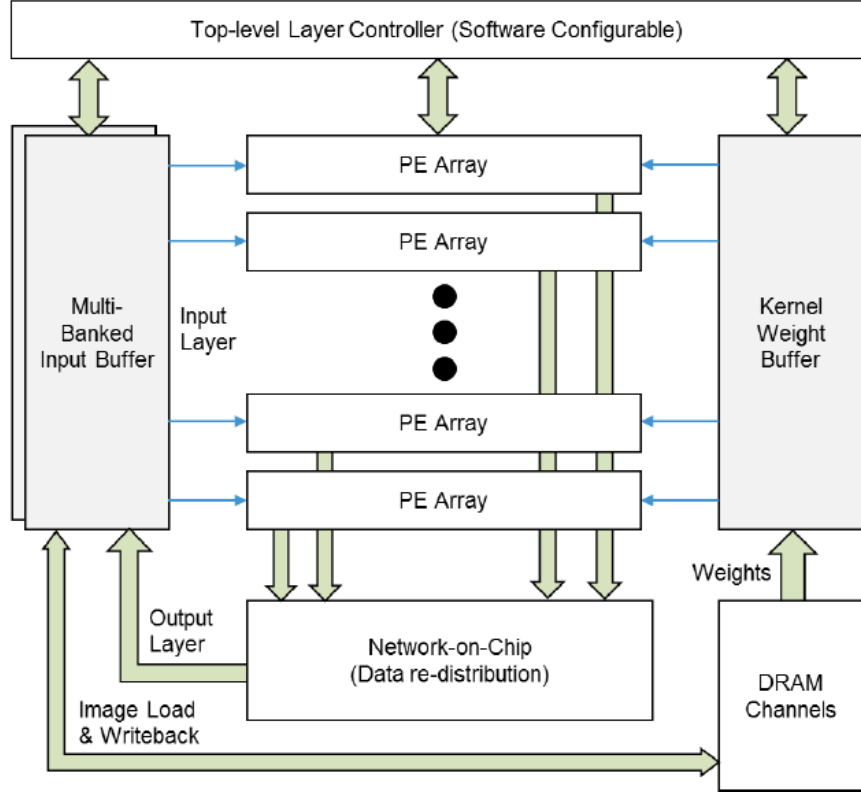


Figure 2.7: Top-level architecture of Microsoft CNN accelerator [140].

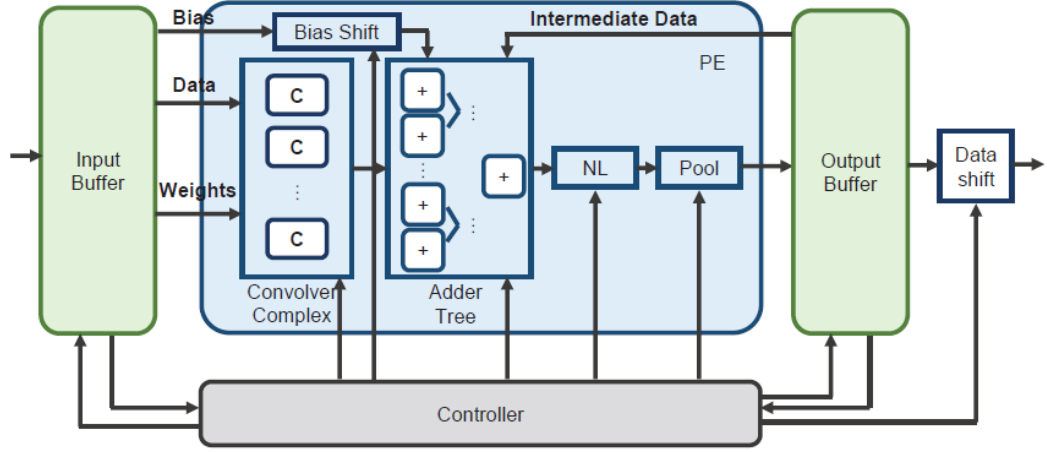
configuring multiple CNN layers at run-time. The results show that the proposed design is able to classify 134 images/sec, while consuming about 25 Watts, for AlexNet model on ImageNet-1K dataset [26], which is $3\times$ better than the published throughput results for the Roofline-based FPGA Accelerator [55]. Ovtcharov et al. mentioned that using top-of-the-line FPGAs such as Arria 10 GX 1150 improves the throughput to around 233 images/sec.

Qiu et al. [88] proposed an FPGA design to accelerate CNNs for a large-scale image classification challenge on embedded systems. The focus was on accelerating both *CONV* and *FC* layers, since they are considered as the most computational-centric and the most memory-centric operations in CNNs, respectively. The proposed accelerator reduces the resource consumption using specific design of convolver hardware module.

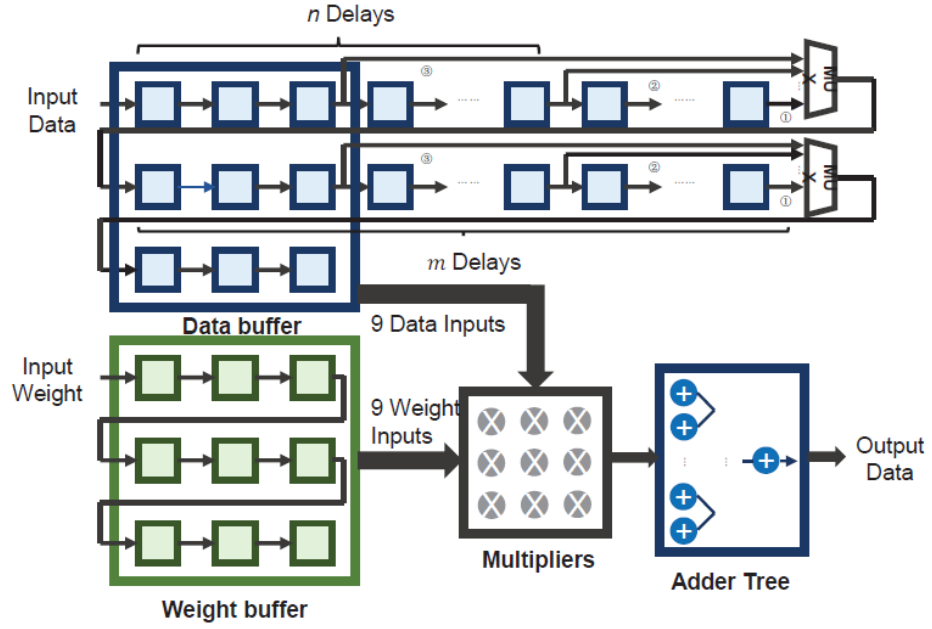
In addition, the authors applied singular value decomposition (SVD) to the weight matrix of *FC* layer in order to reduce memory footprint at this layer [141].

To further reduce memory footprint and bandwidth requirement of CNN, they proposed a dynamic-precision data quantization flow component. This component is responsible for finding the optimal fractional length for weights in each layer as well as the optimal fractional length for FMs in adjacent layers, while achieving minimal accuracy loss. Then, it converts the 32-bit floating-point (FP32) weights and FMs into fixed-point numbers. In addition, Qiu et al. proposed a data arrangement scheme that maximizes the burst length of each transaction to the external memory to accelerate *CONV* and *FC* layers, as well as to avoid unnecessary access latency. Note that maximizing the DRAM burst length raises up the effective DRAM bandwidth [55], [142].

The proposed architecture consists of a processing system (CPU) and a programmable logic (FPGA). CNN computations are performed through special design of PE modules in FPGA. The main modules in the PE are convolver complex, *MAX-POOL*, non-linearity, data shift, bias shift, and adder tree, as shown in Figure 2.8a. The convolver complex is designed as a classical line buffer [143], as illustrated in Figure 2.8b, to achieve *CONV* operations as well as to compute *FC* layer multiplication of matrix-vector. The *POOL* layer implemented in the *MAX-POOL* module is used to output the maximum value in the input data stream with a window of size 2. The activation function of CNN is applied to the input data stream using the non-linearity module. The adder tree accumulates the partial sums generated from the convolvers. Finally, data shift and bias shift modules are responsible for



(a) PE module.



(b) Convolver complex.

Figure 2.8: The (a) PE module, and (b) Convolver complex design of Qiu et al. [88] embedded accelerator architecture.

accomplishing dynamic quantization.

The proposed embedded FPGA platform has been implemented using VGG-16-SVD network with 16-bit fixed-point (Fxp16) numbers on Zynq XC7Z045 platform. The results demonstrate that applying SVD technique reduces memory footprint of *FC* layer by 85.8% with a compression rate of 7.04% while introducing an accuracy loss of only 0.04%. Finally, the overall performance of the proposed accelerator reported is 136.97 GOPs under 150 MHz working frequency with the top-5 accuracy of 86.66% and a total power consumption of 9.63 Watts.

DeepBurning [144] is an FPGA-based neural network (NN) design automation tool. It allows for building learning accelerators for specific NN with optimized performance and custom design parameters configuration using a pre-constructed register transfer level (RTL) module library. The RTL library holds the hardware descriptive scripts for NN reconfigurable components as well as their configuration scripts. In addition, it contains other RTL building blocks for logical and arithmetic operations such as the connection box (used to exchange data between NN layers as well as to approximate the division operation) and approximate LUT (used to simplify a function or operation to allow it to be mapped into hardware).

In order to design an optimized hardware, DeepBurning compresses the passed NN model to the greatest extent using temporal and spatial folding which helps also in satisfying the resource constraints and minimizing the required hardware modules. DeepBurning not only generates the hardware description for NN scripts, but also analyzes the complex access pattern and data locality using an integrated compiler to generate a run-time control flow which provides energy-efficient, and, better data reuse

implementation. In addition, the DeepBurning compiler investigates the accelerator on-chip memory size and throughput to properly tile and partition the NN weights and feature data layouts. Moreover, DeepBurning uses the address flow component to automatically fetch and store off-chip memory and on-chip memory data. Compared to [55], considering AlexNet model and the high-budget resources-constrained DeepBurning on a Zynq-7045 device operating at 100 MHz, the results show that DeepBurning is $1.13\times$ slower but $1.45\times$ more energy-efficient.

In [73], Suda et al. proposed an OpenCL-based optimization framework to accelerate large-scale CNN models. They found that the number of performed *CONV* MAC operations in parallel (N_{CONV}), SIMD vectorization factor (S_{CONV}), *Norm* loop unrolling factor (N_{Norm}), the number of parallel *POOL* outputs (N_{POOL}), and the number of parallel *FC* MAC operations (N_{FC}) are the key variables that determine the parallelism of the design. Subsequently, they analytically and empirically modeled the execution time for each layer as a function of the above mentioned variables. Then, the genetic algorithm was used to explore the design space for finding the optimal combination of the key design variables considering the resources constraints.

Suda et al. implemented the scalable *CONV* block in a similar fashion to that in [126] as a matrix multiplication by flattening and on-the-fly rearrangement of the feature data. The OpenCL software has been utilized in their work due to its parallel programming model as well as its ability to integrate the compiled RTL design with external memory interfacing intellectual properties (IPs) [145], which uses memory coalescing technique with complex load and store units. In addition, it has optimized matrix multiplication and CPU-FPGA communication libraries [146], [147].

The framework is used on both VGG-16 and AlexNet CNN models which are implemented on P395-D8 [148] and DE5-Net [149] FPGA boards with fixed-point operations according to their precision study. They compared the proposed implementation with 3.3 GHz core i5-4590 CPU implementation that uses Caffe tool [58] with ATLAS [150] optimized library for matrix/vector operations. The results show that the OpenCL optimized framework on P395-D8 achieved $5.5\times$ (117.8 GOPs) and $9.5\times$ (72.4 GOPs) speedups for VGG-16 and AlexNet models, respectively. On the other hand, DE5-Net FPGA achieved less throughput speedup than the P395-D8 ($2.2\times$ (47.5 GOPs) for VGG-16, and $4.2\times$ (31.8 GOPs) for AlexNet) as it has $7.67\times$ less DSPs than what is available on P395-D8.

Zhang et al. [142], [151] analyzed the transformation of *CONV* and *FC* layers to regular matrix multiplication presented in prior work [88]. For VGG-16 model, they found that such transformation necessitates up to $25\times$ duplication of IFMs. To address this problem and improve the bandwidth utilization, they designed a uniformed matrix multiplication kernel that uses either input-major mapping (IMM) or weight-major mapping (WMM) techniques while computing *FC* layer. In IMM, the designed kernel batches a group of different IFMs together, and then performs the matrix multiplication. IMM technique improves the data reuse of *FC* weights. On the other hand, the designed kernel with WMM technique makes use of the fact that the weight matrix is much larger than the IFM matrix in *FC* layers. In particular, it loads IFM matrix to a weight buffer and loads weight matrix to IFM buffer. Subsequently, a regular matrix multiplication is performed on these matrices. As a result, WMM may allow for a higher data reuse than IMM, especially for IFMs that can be

reused multiple times considering the limited hardware resources.

For the above, the roofline model was applied to identify the optimal mapping technique under different batch sizes and data precisions. The results demonstrate that WMM is better than IMM in term of data reuse and bandwidth utilization, especially in small batch sizes which is required for real-time inference. Hence, the same matrix multiplication kernel is utilized for the computation of both *CONV* and *FC* layers, but with the use of IMM in *CONV* layer and WMM in *FC* layer. Based on this, Zhang et al. proposed a software/hardware co-design library, which they named Caffeine, to accelerate CNNs on FPGAs.

With an easy-to-use developed tool, Caffeine aids in automatically choosing the best hardware parameters, using the model files from Caffe and FPGA device specifications obtained from the user. Caffeine FPGA engine uses a high-level synthesis (HLS)-based systolic-like architecture to implement matrix multiplication kernel. It allows changing parameters such as number of PEs, precision, and FM size. Caffeine further maximizes the FPGA computing capability by optimizing multi-level data parallelism discussed in [55] and pipeline parallelism using polyhedral-based optimization framework given in [152]. Caffeine framework also handles the weights and biases reorganization in off-chip DRAM to maximize the underlying memory bandwidth utilization. In addition, the double-buffering technique is employed to prefetch the next data tile for each PE. Caffeine has been evaluated by implementing AlexNet and VGG-16 CNNs on Ultrascale KU060 (20nm and 200 MHz) and on Virtex-7 690T (28nm and 150 MHz) considering different precisions. The VGG-16 implementation with FxP16 on Ultrascale KU060 and Virtex-7 690T provided $43.5\times$ and $65\times$ overall

throughput enhancement, respectively, compared to implementation on a two-socket server, each with a 6-core Intel CPU (E5-2609 at 1.9 GHz).

A special case of dataflow, referred to as synchronous dataflow (SDF) [153], is a paradigm of computation that allows for representing a computing system as a streaming problem. In this way, the SDF model can represent the hardware implementation of CNNs using linear algebra and directed SDF graph (SDFG). Each node of SDFG represents a hardware building block that can immediately start its computation as soon as the data are available through its input arcs. Such representation of CNN model offers a fast design space exploration. Venieris and Bouganis [154] employed SDF model to optimize the mapping of CNNs onto FPGAs based on HLS.

In particular, the proposed `fpgaConvNet` framework in [154] takes as input a high-level script programmed by DL expert describing the CNN model, along with specifications of the targeted FPGA platform. Thereafter, it parses the input script through a developed domain-specific language (DSL) processor to model the CNN in the form of a directed acyclic graph (DAG) where each node corresponds to a CNN layer. Then, the DAG-based CNN is transformed into an SDFG representation and modeled as a topology matrix. The topology matrix contains the number of incoming parallel streams, the width of each data stream, and the production or consumption rates at each node. In addition, the DSL processor extracts information about the platform-specific resource constraints.

Unlike other attempts, instead of exploring the design space for the optimal parameters of loop unrolling and tiling, `fpgaConvNet` explores the design space of the topology matrix components while considering the resource constraints. In doing so,

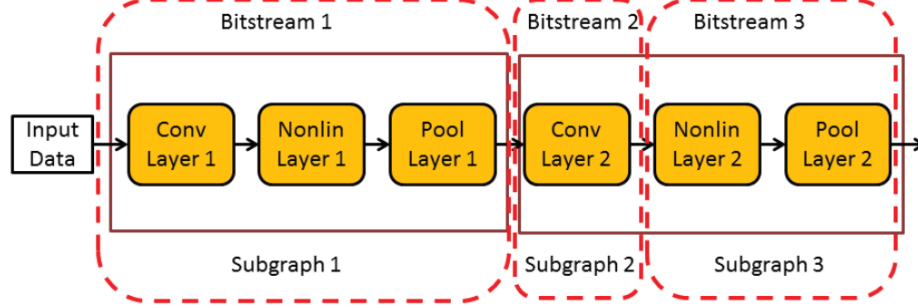


Figure 2.9: SDF graph partitioning [154].

fpgaConvNet performs graph partitioning, coarse-grained folding, and fine-grained folding. The graph partitioning splits the original SDFG into subgraphs and each subgraph is then mapped to a distinct bitstream as shown in Figure 2.9. Note that the proposed multi-bitstream architecture might have multiple convolutional layer processors (CLPs), as in the provided example. This way, on-chip RAM is used for intermediate results and data reuse within the subgraph, while accesses of off-chip memory is minimized and limited for input and output streams of the subgraph. However, this scheme adds reconfiguration penalty due to the need for reconfiguring the FPGA when the data flows between adjacent subgraphs. To amortize this overhead, several input data streams are processed in a pipelined manner.

Thereafter, each bitstream architecture is optimized using coarse-grained folding and fine-grained folding. In coarse-grain folding, *CONV*, *POOL*, non-linear, and other major operations of each layer are unrolled to provide the highest possible throughput by having several parallel units of each operation. The fine-grain folding controls the unrolling and pipelining of the dot-product operations inside *CONV* and average *POOL* units. Instead of fully unrolling the implementation of dot-product which produces a 1 dot-product per cycle, with the use of a high number of multipliers and

adders, fpgaConvNet uses a smaller number of MAC units and schedules the execution of different operations using time-multiplexing.

A trade-off between the performance and the required hardware resources can be achieved by changing the unroll factor and the degree of multiplexing. Therefore, fpgaConvNet employed simulated annealing [155] to find the optimal partitioning points and folding factors. Finally, fpgaConvNet uses optimal components to derive the configuration of PEs and buffers, and generates a synthesizable Vivado HLS hardware design. fpgaConvNet framework has been evaluated by mapping LeNet-5 and scene labelling [156] small CNN models with Q8.8 fixed-point representation onto a Zynq-7000 XC7Z020 FPGA platform working at 100 MHz. In mapping LeNet-5, fpgaConvNet achieves up to $1.62\times$ the performance density of CNP [114]. Compared to Tegra K1 GPU implementation of scene labelling CNN, fpgaConvNet surpasses Tegra K1's power efficiency by $1.05\times$.

Ma et al. [87] proposed a Python-based modularized RTL compiler to accelerate CNNs by employing loop unrolling [55], [95] for *CONV* layer operations. A detailed review article of this work has been recently published and referred to as ALAMO [157]. The proposed compiler integrates both the RTL finer level optimization and the flexibility of HLS to generate efficient Verilog parameterized RTL scripts for application specific integrated circuit (ASIC) or FPGA platform under the available number of parallel computing resources (i.e., the number of multipliers (N_m)). If N_m is greater than the number of IFMs (N_{if}), the proposed compiler fully unrolls *Loop-3* (N_{if} , refer to subsection A for more details) while it partially unrolls *Loop-4* (N_{of}) to exploit the data reuse of shared features among N_m/N_{if} OFMs. Otherwise, it partially unrolls

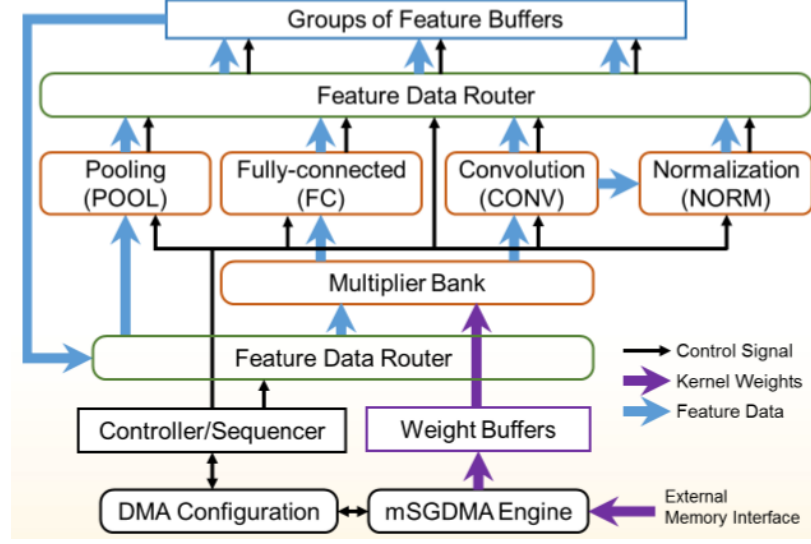


Figure 2.10: ALAMO overall acceleration modules [87].

Loop-3 which results in N_{if}/N_m repeated sliding of kernel window. On the other hand, *Loop-2* ($X \times Y$) is serially computed after *Loop-1* (K) to minimize the number of partial sums.

The overall modules of the proposed CNN accelerator are shown in Figure 2.10. The controller is responsible for directing and ensuring in-order computation of CNN modules for each layer. The data routers oversee the selection of data read and data write of two adjacent modules as well as the assignment of buffer outputs to shared or pool multipliers of the multiplier bank. The feature buffers hold the FMs using on-chip RAMs. The weight buffers are used to ensure the availability of *CONV* and *FC* layers' weights before their computation as well as to overlap the transfer of *FC* layer weights with its computation.

The *CONV* module consists of control logic, groups of adder trees, and rectified linear unit (*ReLU*) components. The control logic component parametrizes the loop unrolling factors based on the configuration of each layer (N_{if} , N_{of} , X , Y , and K). The

N_m/N_{if} adder groups are used to accumulate the results of the N_{if} parallel multipliers. Moreover, the adder trees can be shared by layers with identical N_{if} to be as one single module. The *ReLU* component checks the input sign bit to either output zero or the data itself. On the other hand, the *POOL* module contains accumulators and comparators to perform *AVG* and *MAX/MIN* operations, respectively. The *Norm* module maintains the required components to perform the square, non-linear, and multiplication operations. Finally, the *FC* module shares the multiplier bank module with the *CONV* module to perform the matrix-vector multiplication (MVM).

ALAMO architecture permits the output pixels to be only stored in the feature buffers, which makes ALAMO suitable for CNNs with only small intermediate data volumes. The proposed RTL compiler has been tested by accelerating two CNN models; AlexNet and NiN [158]. The generated parameterized RTL scripts for AlexNet and NiN are synthesized using Altera Quartus synthesis tool and implemented on DE5-Net FPGA board. The experimental results for AlexNet model are compared with the results for OpenCL-based design [73] as both use the same FPGA board with similar hardware resources for AlexNet. ALAMO achieved $1.9\times$ and $1.3\times$ improvement for throughput and power consumption, respectively. Moreover, the overall throughput of NiN model is $1.03\times$ better than that of AlexNet. This is because NiN has more *CONV* layers and many of them have the same N_{if} .

Liu et al. [159] proposed a parallel framework for FPGA-based CNN accelerators that exploits four levels of parallelism; task level, layer level, loop level, and operator level. Task-level parallelism involves executing multiple image prediction tasks simultaneously. Layer-level parallelism exploits pipelining across layers to enable par-

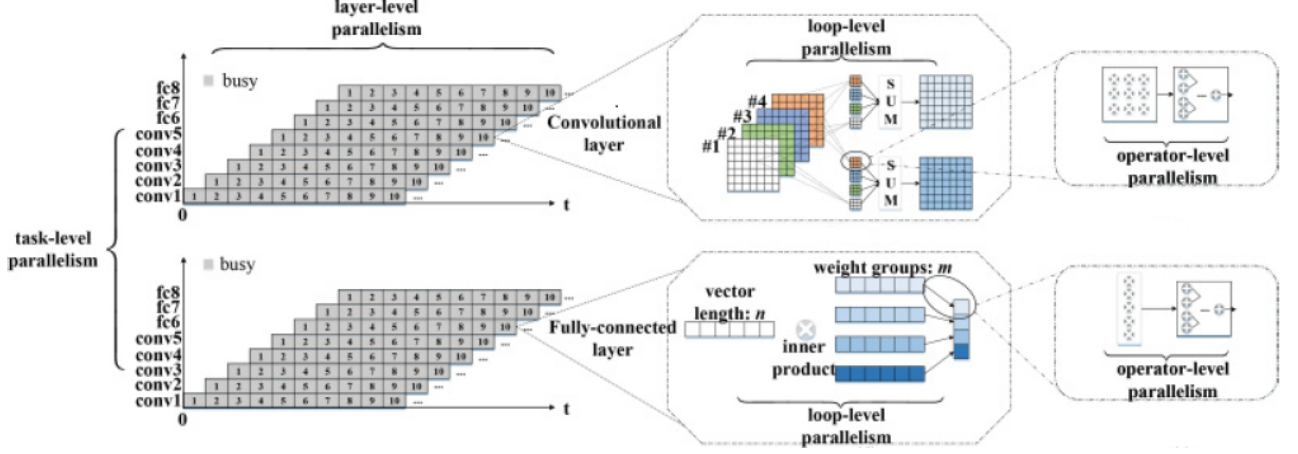


Figure 2.11: Parallel framework exploiting four levels of parallelism [159].

allel execution of all layers with different images. Loop-level parallelism utilizes loop unrolling in performing convolutions and this can be achieved either through intra-output or inter-output parallelism. Finally, operator-level parallelism is achieved by parallelising the $k \times k$ MAC operations needed for *CONV* operation in *CONV* layers or the n MAC operations needed for inner-product computation in *FC* layers. Figure 2.11 shows the parallel framework exploiting these four levels of parallelism.

Liu et al. have used FxP16 format for representing IFMs and OFMs data. However, they have used 32 bits for intermediate results which get truncated to 16 bits. In addition, they have used 8 bits for representing kernel weights. They have presented a systematic methodology for design space exploration to find the optimal solution that maximizes the throughput of an FPGA-based accelerator under given FPGA constraints such as on-chip memory, computational resources, external memory bandwidth, and clock frequency.

The proposed technique has been evaluated by implementing three CNN accelerators on the VC709 board for LeNet, AlexNet, and VGG-S. It has achieved a throughput

of 424.7 GOPs, 445.6 GOPs, and 473.4 GOPs for LeNet, AlexNet, and VGG-S accelerators, respectively. In addition, the performance has been compared with MatConvNet tool running the CNN models on Intel Core i7-4790K CPU (4.0 GHz) and NVIDIA GTX-770 GPU (1,536 CUDA cores, 2 GB GDDR5, 224.3 GB/s memory bandwidth). Compared to the CPU implementations, the accelerators for LeNet, AlexNet, and VGG-S achieved $14.84\times$, $6.96\times$, and $4.79\times$ improvement in performance, respectively, and $51.84\times$, $24.69\times$, and $16.46\times$ improvement in power efficiency, respectively. Compared to the GPU implementations, the accelerators achieved better performance in the small-scale network LeNet ($3.17\times$), comparable performance in the medium-scale network AlexNet ($0.96\times$), and worse performance in the large-scale network VGG-S ($0.56\times$). However, the accelerators achieved higher power efficiency than the GPU implementations in all three networks with $28.3\times$ for LeNet, $8.7\times$ for AlexNet and $4.98\times$ for VGG-S.

FP-DNN [160] is an end-to-end framework that automatically generates optimized FPGA-based implementations of DNNs using an RTL-HLS hybrid library. FP-DNN compiler, programed using C++ and OpenCL, takes TensorFlow symbolic descriptions [161] of DNNs, and then performs model inference through the use of model mapper, software generator, and hardware generator modules. The model mapper extracts the topological structure and layers configurations of DNN model from the TensorFlow descriptions and generates an execution graph for the target model. The execution graph shows layer-by-layer operations and read/write data transactions.

FP-DNN compiler allocates off-chip DRAM data buffers to store intermediate data, weights, and model parameters and configurations. The model mapper maximizes the

storage resource reuse through minimizing the number of required physical buffers. Specifically, it formulates the data reuse problem as a graph coloring problem [162], and then the left-edge algorithm is applied to generate kernel configuration and kernel schedule. Subsequently, the software generator uses the kernel schedule to generate a host C++ program which initializes the model, manages the data buffers, and schedules the kernel execution. On the other hand, the hardware generator uses the kernel configuration and the execution graph to generate the FPGA hardware codes by instantiating the corresponding optimized templates from an expandable RTL-HLS hybrid library. Each template is comprised of Verilog-based computational engine and OpenCL-based control logic engine.

The architecture of the proposed FPGA-based accelerator consists of matrix multiplication and data arranger modules. Matrix multiplication module is a hand-written Verilog code that is designed and optimized based on the hardware constraints of Altera Stratix-V GSMD5 FPGA. It applies tiling and ping-pong double buffers techniques to improve the throughput. On the other hand, data arranger is an OpenCL-based module that is responsible for mapping the computational part of a layer to matrix multiplication as well as performing data communication with off-chip memory and matrix multiplication module.

Mapping DNNs computational operations to matrix multiplication has been widely applied in prior studies [73], [119], [163]. FP-DNN maps *FC* layer to matrix multiplication by batching input vectors together. Before model deployment, FMs and weights are rearranged in DRAM using the channel-major scheme to optimize the communication between the accelerator and off-chip DRAM. On the other hand, both

floating-point and fixed-point representations have been supported for implementation, and they can be adjusted by the user.

The proposed RTL-HLS hybrid framework has been evaluated by accelerating VGG-19, LSTM-LM [164], ResNet-152 DNNs on Stratix-V GSMD5 FPGA. Note that this is the first work that implements ResNet-152 on FPGA. The experimental results demonstrated that the speedup of FP-DNN for FxP16 implementations are about $1.9\times$ - $3.06\times$ compared with the server that includes 2 processors each with 8-core Intel Xeon E5-2650v2 at 2.6 GHz.

In line with the current trends towards compressed NNs, with dramatically reduced weights and activations bit-width using 1-bit or 2-bit quantization [102], [165]–[168], Umuroglu et al. [169] conducted a set of experiments to estimate the trade-off between the network size and precision using the roofline model. They found that binary NNs (BNNs) [168] require 2 to 11 times more operations and parameters than an 8-bit fixed-point CNN to achieve a comparable accuracy on MNIST dataset [124]. However, the performance of BNN is found to be $16\times$ faster than the fixed-point network.

Subsequently, Umuroglu et al. proposed a framework, referred to as FINN [169], that maps a trained BNN onto FPGA. FINN generates a synthesizable C++ network description of a flexible heterogeneous streaming architecture. The architecture consists of pipelined compute engines that communicate via on-chip data streams. Each BNN layer has been implemented using dedicated compute engines with 1-bit values for weights and FMs; +1 and -1 are used to represent a set bit and unset bit, respectively.

Umuroglu et al. have optimized accumulation, batch normalization (*BatchNorm*),

activation, and *POOL* operations of BNNs. In particular, the accumulation of a binary dot-product has been implemented as a counter of set bits (popcount operation). The popcount-accumulate reduces the number of required LUTs and flip-flops (FFs) by a half, compared to the implementation of signed-accumulation. BNN *BatchNorm* and activation operations have been simplified and implemented together as unsigned comparison with a threshold τ_k , +1 is produced when the input value is greater than or equals to τ_k , and -1 otherwise. The value of τ_k is computed during run-time. Such an implementation of *BatchNorm*-activation operations requires much smaller number of LUTs, without the need for DSPs and FFs, compared to regular implementation of *BatchNorm*-activation. On the other hand, *MAX-POOL*, *AVG-POOL*, and *MIN-POOL* have been effectively implemented with Boolean OR-operator, Boolean majority function, and Boolean AND-operator, respectively.

The accelerator architecture is composed of building blocks from the FINN hardware library. The matrix-vector-threshold unit (MVTU) is the core computational building block as matrix-vector operations followed by thresholding form the majority of BNN operations. The design of MVTU consists of an input buffer, an array of P parallel PEs each with S SIMD lanes, and an output buffer. BNN weight matrix is distributed across the PEs and stored locally in on-chip memory. Subsequently, the input images are streamed through the MVTU and multiplied with the weight matrix. Particularly, the PE computes the dot-product between an input vector and a row of weight matrix, each of S bits wide, using an XNOR gate, as shown in Figure 2.12. Then, it compares the number of set bits to a threshold and produces a 1-bit output value as previously discussed.

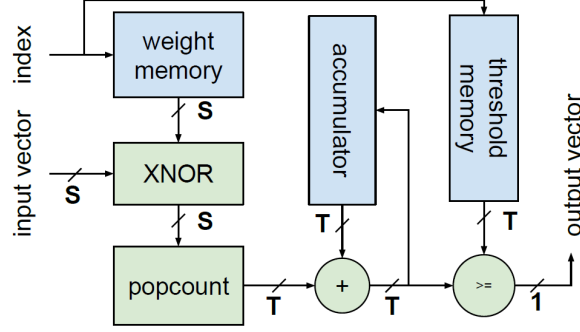


Figure 2.12: The architecture of MVTU PE [169].

Umuroglu et al. [169] implemented the *CONV* layer using a sliding window unit and an MVTU, where *CONV* operation is transformed to matrix-multiplication operation. The sliding window unit generates the image matrix to MVTU by moving the sliding window over the IFMs, while the filter matrix is generated by packing the weights from the *CONV* filters as shown in Figure 2.13. In order to meet the user throughput requirement, MVTU is folded (time-multiplexed) by controlling the values of P and S . Folding of MVM decides partitioning of the matrix across PEs.

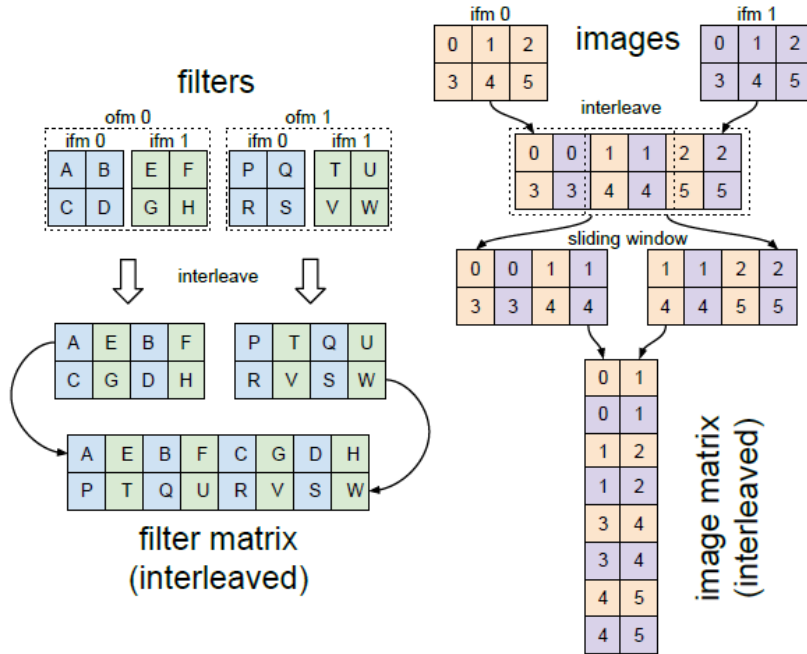


Figure 2.13: Transforming *CONV* to matrix-multiplication [169].

Every row of matrix tile is mapped to a distinct PE and every column of PE buffer is mapped to a distinct SIMD lane. In this way, the required number of cycles to compute one MVM (total fold) is obtained as $(X \times Y)/(P \times S)$, where X and Y are the dimensions of the matrix. The folding factors of BNN layers have been determined such that every BNN layer takes nearly the same number of cycles.

To evaluate FINN, Umuroglu et al. implemented CNV topology on Xilinx Zynq-7000 board at 200 MHz to accelerate BNNs inference on CIFAR-10 [170]. The CNV contains three repetitions of two 3×3 *CONVs* and 2×2 *MAX-POOL* layers. Its topology is inspired by VGG-16 and BinaryNet [168]. Although CNV accepts images with 24-bits/pixel as an input and produces a 10-element vector of 16-bit values, 2-bits are used for representing intermediate results while 1-bit is used for representing *CONV* and *FC* weights. Experimental results demonstrated that the proposed design provides high performance (2.5 TOPs) while incurring low-energy consumption (11.7 Watts). FINN outperforms the design by Ovtcharov et al. [140] by over $13.8\times$ for throughput.

In [97], loop optimization techniques [55], [95] have been employed in FPGA to design a customized CNN accelerator through speeding up *CONV* layer operations. Firstly, an in-depth analysis is provided to numerically characterize loop unrolling, loop tiling, and loop interchange optimization techniques. In doing so, 8 *CONV* dimensions parameters (N^*), 8 loop unrolling design variables (P^*), and 8 loop tiling design variables (T^*) have been used with a constraint, as for a specific loop level, $1 \leq P^* \leq T^* \leq N^*$. Note that unrolling *Loop-1* and *Loop-3* requires $Pkx \times Pky$ and Pif multipliers, respectively, an adder tree with fan-in of $Pkx \times Pky$ and Pif , respectively, and an accumulator. On the other hand, unrolling *Loop-2* requires $Pix \times Piy$

parallel units of MAC to reuse the same weight for $Pix \times Piy$ times, while the input feature pixel can be reused by Pof times when unrolling *Loop-4* with the use of Pof parallel MAC units. Thus, $Pkx \times Pky \times Pif \times Pix \times Piy \times Pof$ multipliers are required. Please refer to Figure 1.4 for more details on *CONV* loops levels and their parameters. In loop tile optimization, the authors have numerically set the lower bound on the required size of the input pixel buffer, the weight buffer, and output pixel buffer that ensures reading each input feature pixel and weight from the off-chip memory only once. On the other hand, loop interchange technique has a great impact on the times of memory access as well as the number of partial sums since it determines the order of computing *CONV* loops.

Secondly, MA et al. [97] have provided a quantitative analysis of the design variables to minimize each of computing latency, partial sum storage, on-chip buffer access, and off-chip DRAM access. Subsequently, MATLAB scripts are used to randomly sample a subset of the solution space to find the optimal design configurations. This is due to the large solution space, more than 7.2×10^{13} possible configurations for loop tiling variables of OFM width (Pox) and height (Poy) alone. According to the randomly sampling results for VGG-16 on Arria 10 GX 1150 FPGA, uniform unrolling factors for *CONV* layers are used with $Pix = Pox = Piy = Poy = 14$ and $Pof = 16$ for *Loop-2* and *Loop-4*, respectively, to reuse IFMs and weights. On the other hand, *Loop-1* and *Loop-3* are serially computed to prevent the movement of the partial sums between the MAC units and consume them as soon as possible since both *Loop-1* and *Loop-3* need to be finished in order to obtain one final output pixel. More importantly, the order of loops computation has been found to be as follows. *Loop-1* is computed

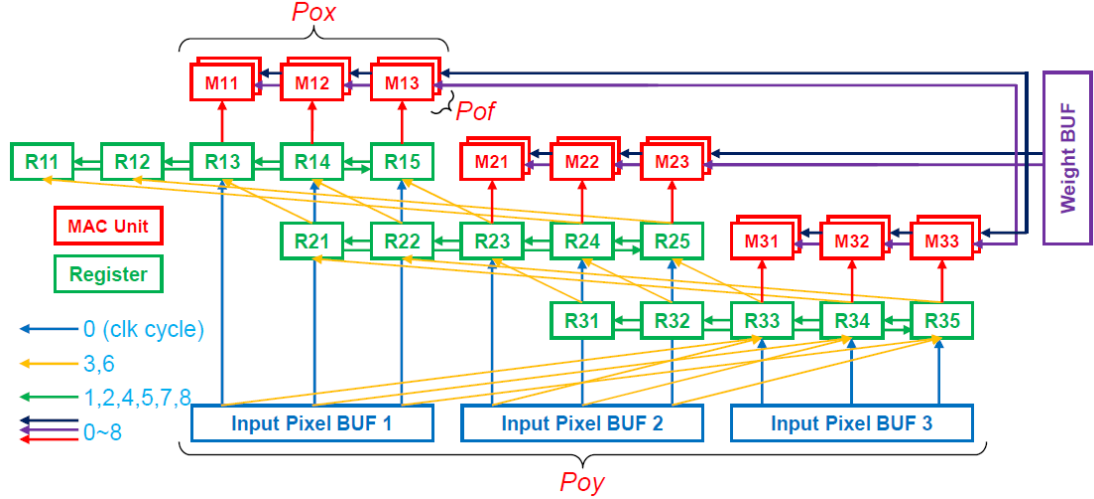


Figure 2.14: *CONV* acceleration architecture and dataflow [97].

first, then comes *Loop-3*, and finally *Loop-2* and *Loop-4* are computed in any order.

Finally, a customized *CONV* accelerator module with efficient dataflow has been designed based on the previous results and used for all VGG-16 *CONV* layers. The *CONV* accelerator consists of $3,136$ ($P_{ix} \times P_{iy} \times P_{of}$) independent MAC units and 14 (P_{of}) input pixel buffers. Figure 2.14 shows an example of the designed *CONV* accelerator when P_{ix} , P_{ox} , P_{iy} , P_{oy} , and P_{of} are all equal to 3. The input pixels are shifted after fetching them out of the input pixel buffers. Subsequently, they can be reused among the input register arrays. Then, the input pixels are fed into the associated MAC units. The figure also shows that the input pixels and weights are shared by P_{of} and $P_{ix} \times P_{iy}$ MAC units, respectively.

The overall CNN acceleration system mainly consists of two SDRAM banks that hold the input feature pixels and weights, two modular Scatter-Gather DMA engines to facilitate the simultaneous read/write from/to the SDRAMs, and a controller to govern the sequential computation of layers as well as the iterations of the four

CONV loops. On the other hand, dual weight buffers have been used to increase the throughput of *FC* layer through overlapping the inner-product computation with off-chip communication. The acceleration system has been written as parametrized Verilog scripts. The experimental results show that the proposed accelerator has a throughput of 645.25 GOPs, which is more than $3.2\times$ enhancement compared to prior VGG-16 FPGA-based implementations [73], [88].

Venieris and Bouganis [171] further extended fpgaConvNet framework [154] to allow for optimizing either throughput or latency depending on the size of the workload. For large workloads, weights reloading transformation has been introduced to efficiently design latency-critical CNNs on FPGA. In contrast with fpgaConvNet, where a distinct architecture is designed for each subgraph, the weights reloading transformation allows for generating a single flexible architecture, named as the reference architecture and derived using pattern matching, to execute the workloads of all subgraphs by transitioning to different modes. Upon the execution of a new subgraph, the subgraph’s weights are read into the on-chip memory and the MUXs are configured to form the appropriate datapath. Figure 2.15 demonstrates how weights reloading is applied. The authors have mentioned that the required time for transferring subgraph’s weights is much smaller than the average time for full FPGA reconfiguration, $272.7\times$ less when loading 4.5 MB of weights for a VGG-16 layer on Zynq XC7Z045.

In the situation discussed above, due to limited on-chip memory capacity, it might not be possible to load all weights required for a single *CONV* layer. To handle this, Venieris and Bouganis introduced an IFMs folding factor (f_{in}) with each *CONV* layer. A *CONV* layer ($CONV_i$) is partitioned into f_{in_i} subgraphs in which each

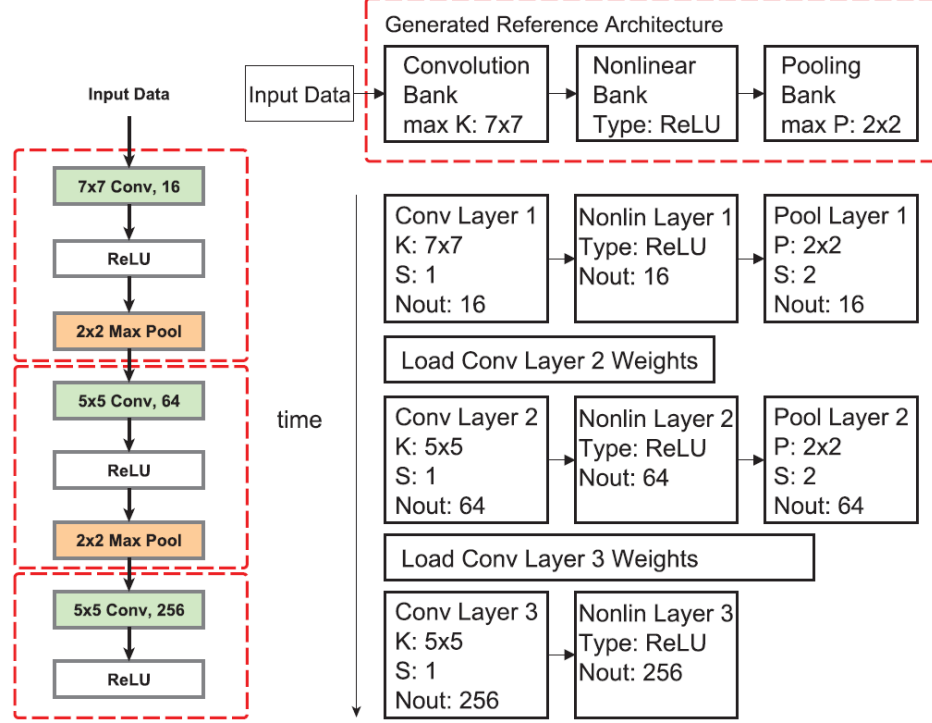


Figure 2.15: Weights reloading [171].

subgraph executes a fraction of $CONV_i$ to produce a fraction of the OFMs. The proposed latency-driven methodology has been evaluated by implementing AlexNet and VGG-16 with FxP16 precision on Zynq XC7Z045 at 125 MHz. The experimental results showed $1.49\times$ and $0.65\times$ higher $CONV$ throughput than DeepBurning [144] and the embedded FPGA accelerator in [88] for AlexNet and VGG-16 implementations, respectively.

Lavin and Gray [172] demonstrated that CNN algorithms with small filters can be efficiently derived using Winograd algorithm [173] and fast Fourier transform (FFT) algorithm [174] due to their advantages in improving resource efficiency and reducing arithmetic complexity. Winograd computation involves a mix of element-wise and general-purpose matrix multiplication, where some of the matrices need to be transformed. In particular, Winograd algorithm exploits the structure similarity among

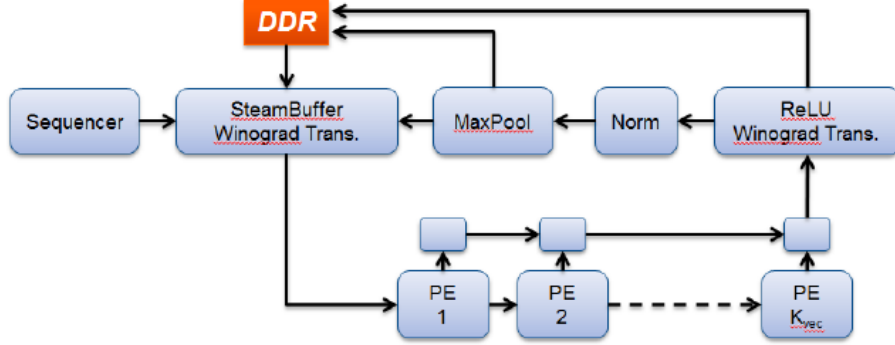


Figure 2.16: Overall DL architecture in [176].

$n \times n$ tiled IFM pixels given a filter of size $r \times r$ to generate $m \times m$ tiled pixels of the OFM, where m represents the stride between Winograd tiles ($m = n - r + 1$), while minimizing the number of required *CONV* multiplications from $m^2 r^2$ for conventional *CONV* algorithm to n^2 . In another work, Zhang et al. [175] implemented FFT algorithm for CNN on FPGA platform. However, their proposed implementation shows little reduction of computation complexity with small filters such as 3×3 .

Aydonat et al. [176] presented a DL architecture based on OpenCL. Their proposed architecture reduces the external memory bandwidth requirements by an order-of-magnitude for both the *CONV* and *FC* layers. This is achieved by caching all intermediate FMs on-chip in stream buffers. For *FC* layers, image batching is used where a batch of images are processed together through the *FC* layers. The approach utilizes the Winograd transformation to reduce the MAC operations, which could reduce the number of needed operations by about 50%. In addition, it uses 16-bit floating-point (FP16) operations with shared exponents, which significantly reduces the needed computational resources.

The overall DL architecture is shown in Figure 2.16. Each PE consists of dot-product units, accumulators, and caches, for performing dot-products for *CONV* and *FC* layers. Caches are used for storing filter weights. To avoid idle computation cycles, double-buffering is used such that filter weights for the next *CONV* layer are prefetched onto the caches while filter weights are loaded from the caches for a particular *CONV* layer. Stream buffers store feature data and stream it to PEs. Each stream buffer is double-buffered similar to filter caches. Images are loaded from the DDR and are stored in stream buffers before the first *CONV* layer starts execution.

During a *CONV* layer execution, while feature data for a *CONV* layer is being streamed into the PEs, the outputs of convolutions are simultaneously stored in the buffers. The StreamBuffer unit applies the Winograd transformations to features, and streams the transformed features to the first PE which are forwarded through all the PEs via the daisy-chained input connections between them. The *ReLU* unit receives the outputs of the PEs via daisy-chained output connections. Then, the *Norm* unit receives the outputs of the *ReLU* unit and applies the *Norm* formula across the FMs. The *POOL* unit receives the outputs of the *Norm* unit and computes the maximum value in a window. The output of the *POOL* unit is stored back in the stream buffer for further processing, if more *CONV* layers are to follow. Otherwise, the outputs of the *POOL* unit are stored in external memory.

For the *FC* layers, features data are stored on PEs caches while filter weights are stored in stream buffers. For the first *FC* layer, features data are read back from external memory and loaded onto the PE caches. The *ReLU* output is sent directly to DDR, without applying *Norm* or *POOL*. The sequencer generates the control signals

to control the operation of the various blocks in DL architecture according to the topology of the executed CNN. Executing a different CNN requires just changing the sequencer configuration.

The DL architecture has been evaluated by implementing AlexNet on Intel’s Arria 10 dev kit which contains a A10-1150 device (20nm) using a 96 batch size for the *FC* layers. It achieved a performance of 1020 images/s. In addition, it achieved $8.4\times$ more GFLOPs than the latest Ultrascale (KU 20nm) result reported in [151], which uses a 32 batch size for the *FC* layers, and $19\times$ more GFLOPs than the latest Stratix V result reported in [73]. Furthermore, it has achieved energy efficiency at 23 images/s/W, which is similar to what is achieved with the best publicly known implementation of AlexNet on NVIDIA Titan X GPU.

Unlike DL architecture [176] where a 1D Winograd algorithm was employed to reduce arithmetic complexity, Lu et al. [177] implemented a novel FPGA architecture with a two-dimensional Winograd algorithm [173] to accelerate *CONV* computation of CNNs. The overall architecture consists of line buffer structure and Winograd PE engine, as shown in Figure 2.17. Particularly, $n + m$ input lines and m output lines of on-chip buffers are used to effectively reuse FM data among different tiles. While Winograd PE engine reads the first n input lines to perform Winograd computation, the next m input lines load pixels from off-chip memory using FIFOs to overlap the data transfer and computation. Thereafter, the input lines are rotated in a circular fashion to make the next n input lines ready.

On the other hand, Winograd PE engine composed of 4 pipelined stages performs transformation, element-wise matrix multiplication, additional transformation, and

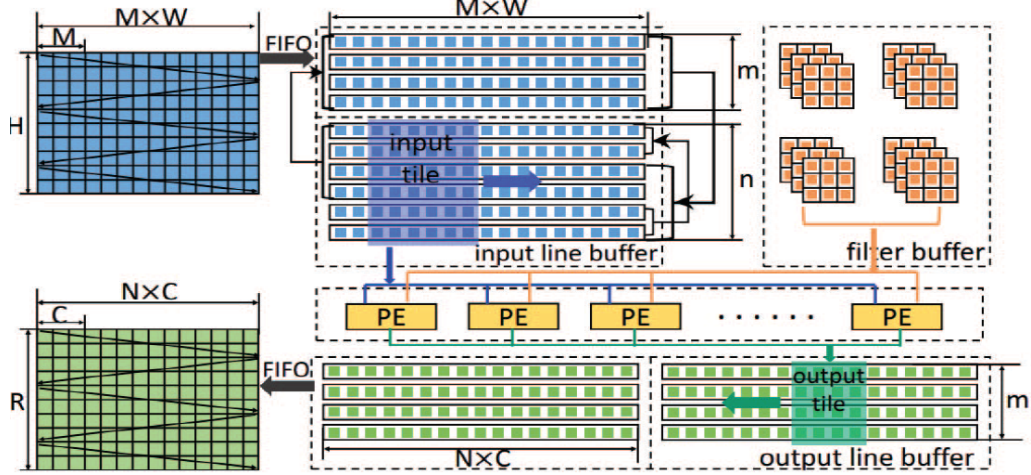


Figure 2.17: Winograd-based CNN accelerator [177], where, m is the size of the IFM tile, n is the size of the OFM tile, M is the number of input channels, N is the number of output channels, W is the maximal width of all IFMs, C is the width of the OFMs.

accumulation of output tiles, respectively. A vector of PEs is employed to achieve parallelism through unrolling *Loop-4* (*Pof*) and *Loop-3* (*Pif*) similar to that in [55]. To implement *FC* layer, the proposed accelerator uses the input line buffers to hold *FC* weights while input neurons are stored on the filter buffers. Then, Winograd PE engine is reused to implement *FC* operation but with bypassing the transformation stages. Moreover, a batch (N_{batch}) of IFMs are assembled and processed together in order to improve the memory bandwidth. An analytical model has been proposed for a fast design space exploration of optimal design parameters (n , Pof , Pif , N_{batch}) constrained by FPGA configuration with a FxP16 representation for FMs and filters.

The proposed accelerator has been evaluated by implementing AlexNet and VGG-16 on Xilinx ZCU102 FPGA. AlexNet *CONV* layers have 3 different filters. Conventional *CONV* algorithm has been applied to the first *CONV* layer as it has a filter of size 11×11 while a uniform filter of size 3×3 for Winograd algorithm has been used to implement the rest of the layers. The design parameters are found to be equal to

(6, 4, 8, 128) and (6, 4, 16, 128) for AlexNet and VGG-16, respectively. The experimental results demonstrated that the proposed Winograd-based CNN accelerator has an average performance of 854.6 GOPs and 2940.7 GOPs for AlexNet and VGG-16, respectively, with power consumption of 23.6 Watts for both.

The proposed accelerator has also been evaluated on Xilinx ZC706 platform where the design parameters are found to be as (6, 2, 8, 32) and (7, 4, 4, 32) for AlexNet and VGG-16, respectively. The experimental results demonstrated that Winograd-based CNN accelerator has an average performance of 201.4 GOPs and 679.6 GOPs for AlexNet and VGG-16, respectively, with power consumption of 23.6 Watts for both. Compared to the implementation of VGG-16 on NVIDIA Titan X with the latest CuDNN 5.1, Titan X gives better performance than Xilinx ZC706 but the implementation on Xilinx ZC706 achieves $1.73\times$ higher energy efficiency.

Zhang et al. [178] presented an OpenCL-based architecture for accelerating CNNs on FPGA. They also proposed an analytical performance model to identify the bottleneck in OpenCL-based acceleration of VGG-19 model on modern FPGA platforms such as Altera Arria 10 GX 1150. Based on roofline mode analysis, it is shown that the bandwidth requirement of VGG-19 workload is higher than what is provided by the FPGA board. Thus, they identified on-chip memory bandwidth as the key performance bottleneck. In addition, they observed that exploited data-level parallelism in the existing Altera OpenCL library [179] leads to wasteful replication of on-chip memory (BRAM). This is due to connecting each PE with a dedicated BRAM port.

Therefore, a Verilog-based accelerator kernel has been designed and warped to an OpenCL IP in order to optimally balance on-chip memory bandwidth with workload

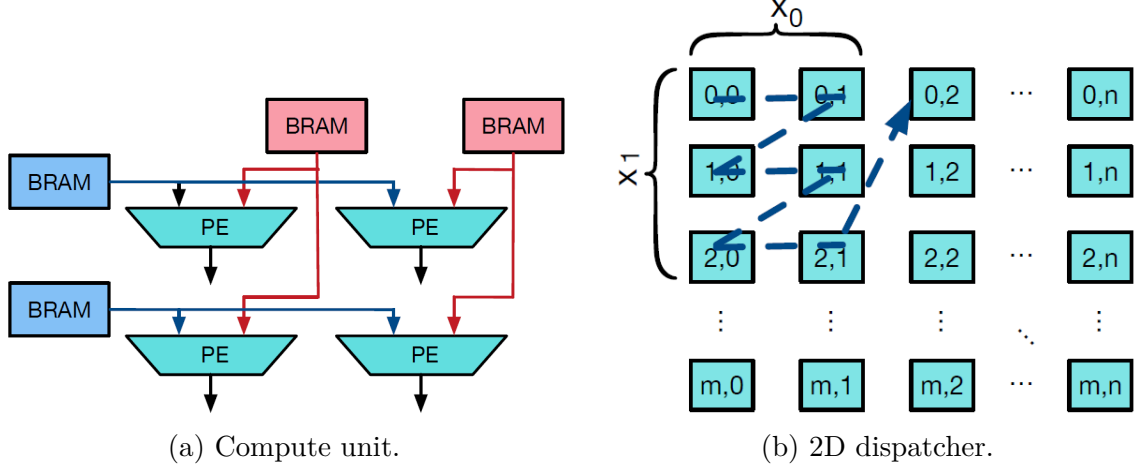


Figure 2.18: The (a) compute unit with a 2D BRAM-to-PE interconnection, and (b) 2D dispatcher, where, X_0 is the column size of kernel buffer as well as the row size of the input feature buffer, and X_1 is the row size of kernel buffer [178].

computational throughput and off-chip memory accesses. In particular, the proposed kernel consists of a compute subsystem, a local memory subsystem, and a 2D dispatcher. The compute subsystem is organized hierarchically into compute units and PEs. At PE level, Zhang et al. have designed a 2D multi-cast interconnection between BRAMs (32-bit data width) and PEs to improve the efficiency of on-chip BRAM usage by sharing the data of one BRAM port with several PEs as shown in Figure 2.18a.

The compute unit has been designed as a 2D PE array of size 16×16 to match the computational bandwidth with the maximum streaming bandwidth (512-bit data bus) provided by off-chip memory. The 2D dispatcher divides the work items into work groups each of size (X_0, X_1) as shown in Figure 2.18b. Thereafter, it adaptively schedules the work items within each work group to the compute units starting with the lowest dimension to balance the memory bandwidth with capacity. The 2D dispatcher is also responsible for host/device memory data transfers. In addition, Zhang et al. have limited the maximum fan-out for registers to 100 in order to guarantee a

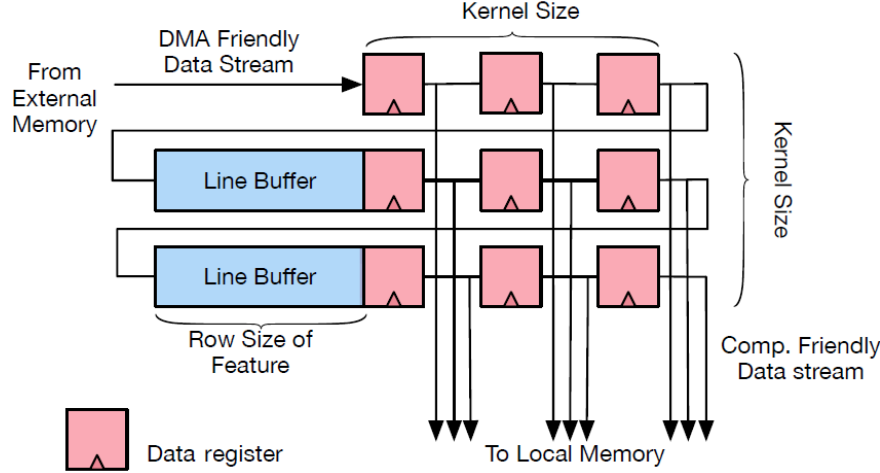


Figure 2.19: Line buffer design [178].

higher frequency.

The *CONV* layer has been implemented as a matrix multiplication by flattening and rearranging the data using line buffer [143], as shown in Figure 2.19, in a similar fashion to that in [73]. The line buffer converts continuous address stream from external memory into a stream conducive for *CONV* operation to substantially reduce the bandwidth requirement of off-chip memory. To implement *FC* layer, the proposed accelerator uses one column of PEs in the compute unit. The proposed implementation has achieved 866 GOPs and 1790 GOPs with the use of FP32 and FxP16, respectively, under 370 MHz and 385 MHz working frequencies, respectively.

All previously discussed FPGA-based CNN accelerators, except the ones discussed in [154], [159], have employed a single CLP to maximize the aggregate throughput of performed consecutive *CONV* operations. However, Shen et al. [93] noted that using a single globally-optimized CLP design for the computation of *CONV* layers of radically different configurations and dimensions leads to sub-optimal performance and insufficient utilization of FPGA resources. Figure 2.20a demonstrates the use of a

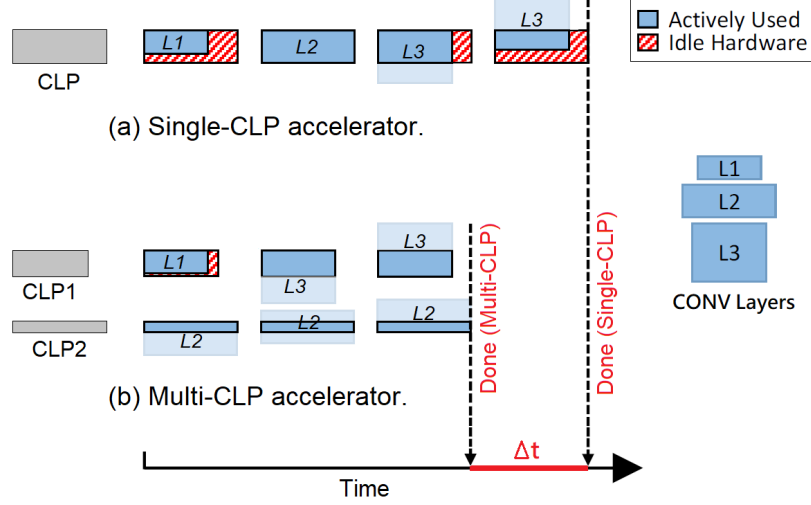


Figure 2.20: Operation of CLPs on CNN with three *CONV* layers [93].

single CLP to iteratively process $L1$, $L2$, and $L3$ *CONV* layers where the dimensions of the hardware (CLP , $CLP1$, and $CLP2$) and the layers are represented by the size and shape of the boxes. It is clear that computing $L1$ and portions of $L3$ leaves FPGA resources unutilized as their dimensions are smaller than the dimension of the used CLP. Note that processing a *CONV* layer with a dimension bigger than the dimension of CLP, such as $L3$, requires the repeated use of CLP to process the different portions.

Shen et al. have also followed the methodology in [55] to derive an optimal single CLP, through finding the optimal unrolling factor $\langle T_m, T_n \rangle$, for implementing SqueezeNet [180] and AlexNet on Virtex-7 690T FPGA with FP32 and FxP16 arithmetic units, respectively. They found that one quarter of DSP slices of SqueezeNet's CLP remain unused. Even more worse utilization has been observed for AlexNet. The optimal single CLP has not utilized, on average, more than one quarter of the arithmetic unit resources.

On the other hand, they also noted that using one CLP for each stage of *CONV* layer in a fashion similar to that in [92] is not efficient due to three reasons. First, it re-

duces the on-chip BRAM buffer size of each CLP which minimizes overall data locality. Second, such one-to-one mapping of *CONV* layers and CLPs requires orchestrating many off-chip memory accesses which incurs latency and bandwidth overheads. Third, the overall control overhead scales with the number of CLPs which leaves insufficient resources for the computation of CNN.

To address the above inefficiencies, Shen et al. [93] proposed a multi-CLP accelerator system for CNNs where the available PFGA hardware resources are partitioned across multiple smaller CLPs. Each CLP is tailored with a dimension that closely matches the dimensions of a subset of *CONV* layers. Thereafter, these specialized CLPs are used to concurrently operate on a batch of images to achieve a higher overall throughput, as shown in Figure 2.20b, where the same hardware in Figure 2.20a is partitioned into two parallel CLPs; *CLP1* and *CLP2*.

Shen et al. [93] developed an optimization search algorithm that uses dynamic programming to find optimal designs. For given configurations of CNN model (i.e., *CONV* layers descriptions) and resource constraints of the targeted FPGA platform (i.e., number of DSP slices, 18 Kb BRAM (BRAM18K) units, and off-chip memory bandwidth), it derives the optimal number of CLPs (along with their $\langle T_m, T_n \rangle$ dimensions) as well as the optimal mapping between *CONV* layers and CLPs that maximize the performance. The assignment of CNN layers to CLPs is static, where each CNN layer is mapped and bounded to a particular CLP. Subsequently, CNN layers are pipelined to their CLP, as shown in Figure 2.20b, where *L1* and *L3* are pipelined to *CLP1* while *L2* is repeatedly processed on *CLP2* with very little idle hardware which improves the performance compared to single CLP approach. Moreover, the

optimization algorithm also finds the optimal partition of on-chip BRAM resources of each CLP that minimizes the overall off-chip memory accesses. Note that the optimal dimension of each CLP is found based on the work in [55].

Subsequently, C++ (HLS) templates are parameterized to design CLPs and to form a complete implementation of CNN. A standard AXI crossbar is used to interconnect the independent CLPs. The ping-pong double-buffering technique is also used for IFMs, OFMs, and weights to allow for transferring data while computation is in progress. The experimental results of implementing AlexNet with FP32 data using multi-CLP accelerator on Virtex-7 485T and 690T FPGAs at 100 MHz demonstrate $1.31\times$ and $1.54\times$ higher throughput than the state-of-the-art single CLP design in [55], respectively. For the more recent SqueezeNet network, the proposed multi-CLP accelerator results in speedup of $1.9\times$ and $2.3\times$ on Virtex-7 485T and 690T FPGAs at 170 MHz with FxP16 data, respectively.

Wei et al. [181] presented a systolic architecture for automatically implementing a given CNN on FPGA based on OpenCL description, maximizing clock frequency and resource utilization. The proposed systolic architecture is shown in Figure 2.21. Each PE shifts the data of the weights and IFMs horizontally and vertically to the neighboring PEs in each cycle. The 2D structure of PEs is designed to match the FPGA 2D layout structure to reduce routing complexity and achieve timing constraints.

The technique first finds a feasible mapping for the given CNN to the systolic array to guarantee that proper data is available at specific locations in the PE array at every cycle. Then, the size of PE array (dimensions) is determined which has an impact on the required number of DSPs, the clock frequency, and the DSPs efficiency. Finally, the

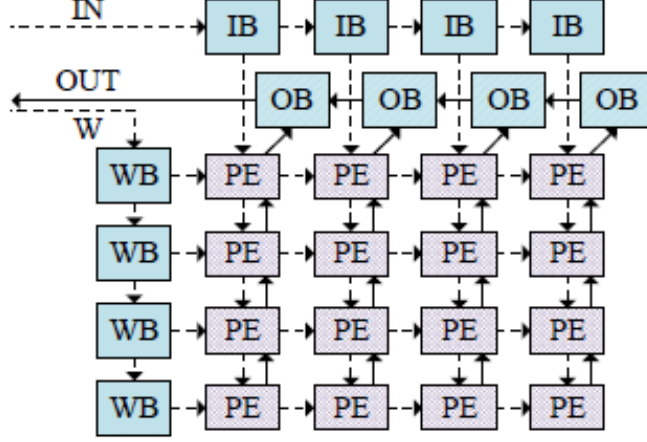


Figure 2.21: Systolic array architecture for CNN [181].

data reuse strategy is determined by choosing proper tiling sizes. Evaluation results for VGG16 on Intel’s Arria 10 GT 1150 board show that the technique achieves up to 1,171 GOPs with a clock frequency of 231.85 MHz when using 8-bit fixed-point weights and 16-bit fixed-point FMs.

In another recent research work, Ma et al. [182] generalized the previously proposed accelerator in [97] to efficiently accelerate ResNet-50 and ResNet-152 on Arria 10 GX 1150 FPGA. In doing so, they designed flexible and scalable *CONV*, *ReLU*, *BatchNorm*, *scale*, *POOL*, *FC*, and element-wise (*Eltwise*) primitives. In addition, local control logic and registers have been used with each primitive to control their computation order and to hold their configurations, respectively. By doing so, ResNets primitives can be efficiently reused for different parameters of each layer.

For ResNets scalable *CONV* primitive, there are four size configurations (kernel, stride); $(3 \times 3, 1)$, $(1 \times 1, 1)$, $(1 \times 1, 2)$, and $(7 \times 7, 2)$. Therefore, a similar architecture and dataflow to that shown in Figure 2.14 has been used for *CONV* but with the use of two sets of register arrays; with shifting between the registers (which is shown in

Figure 2.14, *Set-1*), and without shifting between the registers (*Set-2*). The *CONV* primitive with 3×3 kernel and stride of 1 uses *Set-1* register array, while *Set-2* is used with $(1 \times 1, 1)$, $(1 \times 1, 2)$, and $(7 \times 7, 2)$ configurations. In *CONV* primitive with *Set-2*, the input pixels are fed from the input pixel buffers into the corresponding registers without shifting, and then to MAC units. The skipped input pixels in $(1 \times 1, 2)$ configuration are not stored to the input pixel buffers. On the other hand, the $(7 \times 7, 2)$ configuration of the kernel and stride sizes is retained as the $(1 \times 1, 1)$ case while transferring repeated input pixels into the input pixel buffers and rearranging their storage patterns. The *CONV* primitive also takes care of zero-paddings for different (kernel, stride) size configurations.

The loop unrolling and tiling techniques in [97] have also been employed to accelerate *CONV* primitive with a uniform mapping of PEs to all ResNets *CONV* layers. However, designing efficient CNN modules is not enough, as the memory accesses and data movements between these modules must also be minimized. Therefore, Ma et al. have designed a layer-by-layer computation flow. The global control logic is responsible for governing the sequential operations of primitives and their dataflow through predefined and preloaded layered-based execution flowchart, as shown in Figure 2.22. In addition, it has been modeled to reconfigure ResNet primitives according to the parameters of each layer during runtime. For instance, it maps a particular number of PEs to *CONV* layer based on loop unrolling parameters as well as it controls the selection of register array type (*Set-1* or *Set-2*) based on *CONV* parameters.

On the other hand, a custom DMA manager has been designed to control the operations of DMA. Note that the DMA is responsible for transferring the FMs and weights

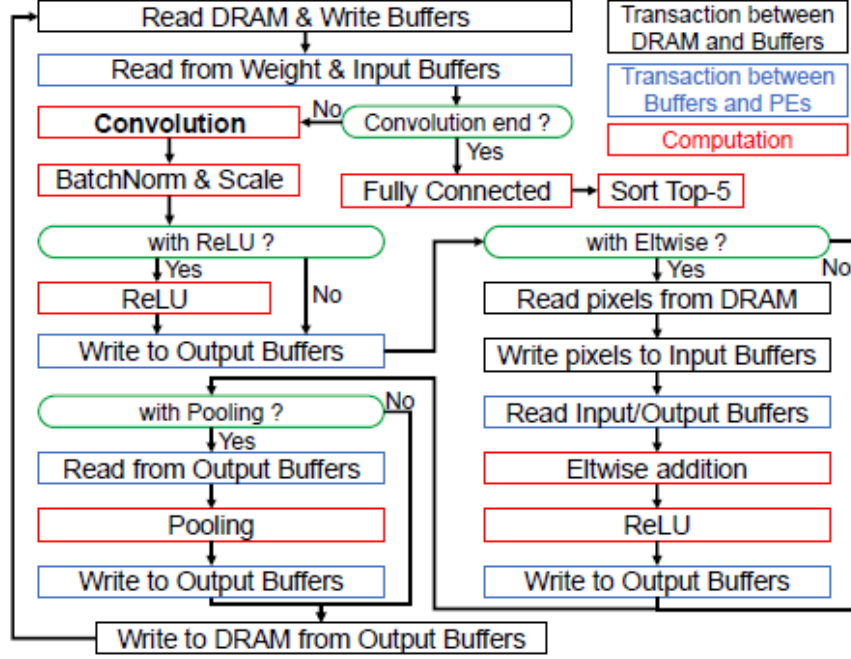


Figure 2.22: Execution flowchart of ResNets layers [182].

between off-chip memory and on-chip buffers. Unlike ALAMO architecture [157] where the output pixels are only stored in on-chip buffers, this work as well as the work discussed in [97] store the output pixels in off-chip memory with the use of loop tiling technique in order to have a flexible architecture that can process large-scale CNNs. The dual weight buffers technique has not been used in this work due to the current trend in CNNs where either the size of FC weights has been significantly reduced (2 million parameters in ResNet compared with 123.6 million parameters in VGG) or the FC layers are completely removed such as in NiN. The experimental results demonstrated that the achieved throughput for ResNet-50 and ResNet-152 are 285.1 GOPs and 315.5 GOPs, respectively. Finally, Ma et al. mentioned that higher throughput can be achieved using batch computing [92].

Wang et al. [183] proposed a scalable design on FPGA for accelerating DL algo-

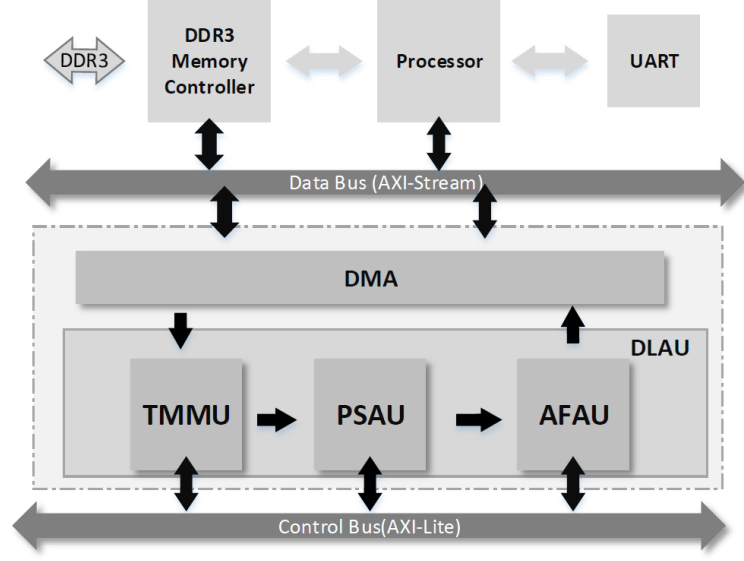


Figure 2.23: DLAU accelerator architecture [183].

rithms. In order to provide a scalable architecture and support various DL applications, the proposed architecture utilizes the tiling technique in which the large-scale input data is partitioned into small subsets. The size of the tile is configured to leverage the trade-off between the hardware cost and the speedup. Moreover, the authors explored hot spots profiling to determine the computational parts that need to be accelerated to improve the performance. The experimental results illustrated that matrix multiplication and activation functions are the key operations in DL algorithms as they consume about 98.6% and 1.1% of the overall execution time, respectively. Thus, the proposed accelerator is responsible for speeding up both matrix multiplication and activation function computations.

The main components of the proposed architecture are the embedded processor, the DDR3 memory controller, the DMA module, and the deep learning acceleration unit (DLAU), as shown in Figure 2.23. The embedded processor utilizes the JTAG-UART to communicate with the acceleration unit [184]. The DLAU accesses the

DDR3 memory to read the tiled input data and to write the results back through the DMA module during the execution. The DLAU utilizes three fully pipelined processing units to improve the throughput, while minimizing the memory transfer operations. These units are tiled matrix multiplication unit (TMMU), partial sum accumulation unit (PSAU), and activation function acceleration unit (AFAU).

The TMMU is responsible for multiplication and generating the partial sums. To optimize the performance, TMMU is structured as a pipelined binary adder tree. Moreover, it uses two sets of registers alternately to overlap the computation with the communication, one group is used for the computation, while in parallel, the other group is loaded with the next node data every clock cycle. On the other hand, PSAU is responsible for accumulating the partial sums generated from TMMU. Finally, AFAU implements the sigmoid function using piecewise linear interpolation to speed up the computation with negligible accuracy loss. Since the processing units in DLAU might have inconsistent throughput rates, each unit has input FIFO buffer and output FIFO buffer to prevent data loss.

Wang et al. implemented the proposed architecture on Xilinx Zynq Zedboard with ARM Cortex-A9 processors clocked at 667 MHz. In addition, they used the MNIST dataset as a benchmark considering the network size as 64×64 , 128×128 , and 256×256 . The experimental results demonstrated that the speedup of DLAU is up to $36.1 \times$ compared with the Intel Core2 processors at 256×256 network size. In addition, the results depict that the proposed architecture is quite energy-efficient as the total power consumption was only 234 mW.

In [185], a generalized end-to-end acceleration system of the previously proposed

accelerators in [87], [97], [157], [182] has been developed to support diverse CNN models. In doing so, a user-friendly interface and an RTL-level compiler have been proposed to automatically generate customized FPGA designs. The authors have developed an expandable optimized RTL-based library containing the most commonly used CNN operations. These operations have been coded in Verilog and designed based on the quantitative analysis and optimization strategies discussed in [97]. The compiler generates a DAG-based structure for the used CNN model and then compiles it with RTL modules in the library. The proposed compiler allows the user to input the high-level information of the used CNN model (previously designed on Caffe framework [58]) as well as the design variables (i.e., loop unrolling and loop tiling variables) with the resource constraints of the targeted FPGA platform. Such utility facilitates the exploration of the best trade-off between the resource usage and the performance.

Unlike the architecture in [157] where individual *CONV* module is assigned to each *CONV* layer, the scalable RTL computing module proposed in this work is reused by all CNN layers of the same type for different CNNs as shown in Figure 2.24. Note that it is not necessary to have all these modules in the architecture. For instance, the RTL compiler will not compile or synthesize *Eltwise* and combined *BatchNorm* with scale modules for VGG-16 model which greatly saves the hardware resources.

On the other hand, MA et al. [185] categorized CNN layers into key layers (i.e., *CONV*, *POOL*, and *FC* layers) and affiliated layers (e.g., *ReLU*, *BatchNorm*, *Eltwise*, and all other layers). They have also defined layer combos, where each combo is composed of a key layer and several affiliated layers. Layer combos are sequentially executed according to their order in the DAG. Moreover, the layer combo is also

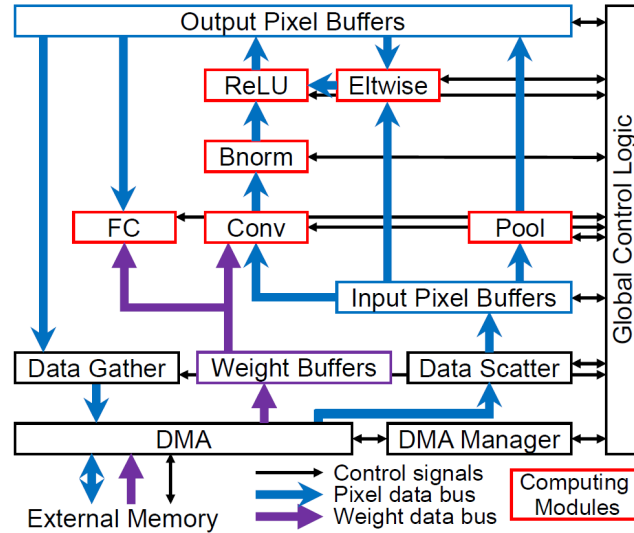


Figure 2.24: Overall architecture and dataflow [185].

divided into several sequential tiles. The computation of each combo tile starts by reading its input pixels from off-chip DRAM and ends by writing back its output pixels to off-chip DRAM. The global control logic, inter-tile control logic, and intra-tile control logic are responsible for governing the sequential operations of layer combos and reconfiguring the modules, combo tiles, and tile layers (key and affiliated layers), respectively, through predefined flexible execution schedule similar to that in [182].

MA et al. have also employed special storage pattern of both input pixels and weights on off-chip memory before the acceleration process to maximize data reuse and minimize of data communication. The architecture of *CONV* module is designed based on the acceleration strategies in [97], [182] but with a different organization of MAC units as shown in Figure 2.25. The MAC units of *CONV* module have been organized into $P_{iy} \times P_{of}$ independent MAC blocks, with each MAC block containing P_{ix} MAC units to further minimize the buffer read operations and the partial sums movements. Moreover, such organization enables to handle varying (kernel, stride)

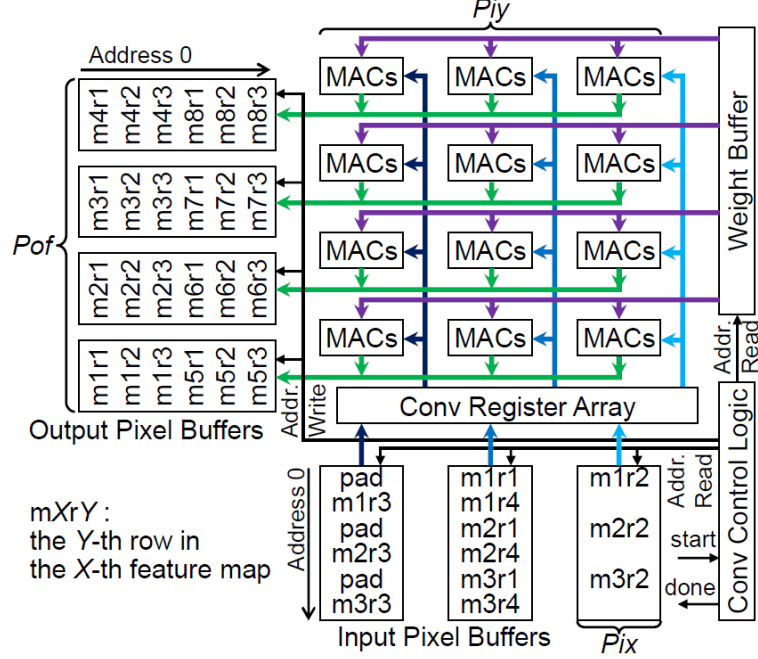


Figure 2.25: *CONV* reconfigurable computing module [185].

sizes configurations through generating different variants of *CONV* register arrays during the compilation.

Experimental results demonstrated that the achieved throughput on Intel Stratix V GXA7 for NiN, VGG-16, ResNet-50, and ResNet-152 are 282.67 GOPs, 352.24 GOPs, 250.75 GOPs, and 278.67 GOPs, respectively. On the other hand, the achieved throughput on Intel Arria 10 GX 1150 was 587.63 GOPs, 720.15 GOPs, 619.13 GOPs, and 710.30 GOPs for NiN, VGG-16, ResNet-50, and ResNet-152, respectively. More than $2\times$ throughput improvements have been achieved on Intel Arria 10 GX 1150 since it has $1.8\times$ and $5.9\times$ more logic elements and DSPs than the Intel Stratix V GXA7, respectively, which allows for larger loop unrolling variables.

Recently, the programmable solutions group at Intel has developed an FPGA software-programmable and run-time reconfigurable overlay for DL inference [186]. The developed overlay is referred to as the DL accelerator. For the hardware side of

Intel’s DL accelerator, the team has partitioned the configurable parameters into run-time and compile-time parameters. The run-time parameters allow for easy and quick use of different NN frameworks, while the compile-time parameters provide a tunable architecture for performance. Intel’s DL accelerator uses a lightweight very long instruction word (VLIW) network, an 8-bit unidirectional ring network, to support the control and reprogramming logic.

Compared with typical overlays, Intel’s DL accelerator comes with only $\sim 1\%$ overhead while other typical overlays tend to always come with larger overheads [187]. The reprogramming of Intel’s DL accelerator overlay allows for consecutive runs of multiple NNs in a single application run [188] without the need for reconfiguring and recompiling the FPGA. Figure 2.26 shows that a 1D array of PEs is used to perform *CONV*, multiplication, or any other matrix operations. Each PE contains a double-buffered filter cache allowing for pre-loading of next filters while computing. The stream buffer employed the double-buffering mechanism as well to store the inputs and the intermediate data on-chip.

To have flexible NN architecture, Intel’s DL accelerator employs an Xbar interconnect that connects all the core functions required. Thus, DL functions can be easily added to the overlay through the Xbar by picking them from a suite of pre-optimized functions of the select frameworks that Intel’s DL accelerator uses. The width adaptation module has been used to control the throughput of the function. In addition, Intel’s DL accelerator supports vectorization across the input width (Q_VEC), input height (P_VEC), input depth (C_VEC), output depth (K_VEC), filter width (S_VEC), and other dimensions as depicted in Figure 2.26. Abdelfattah et al. [186]

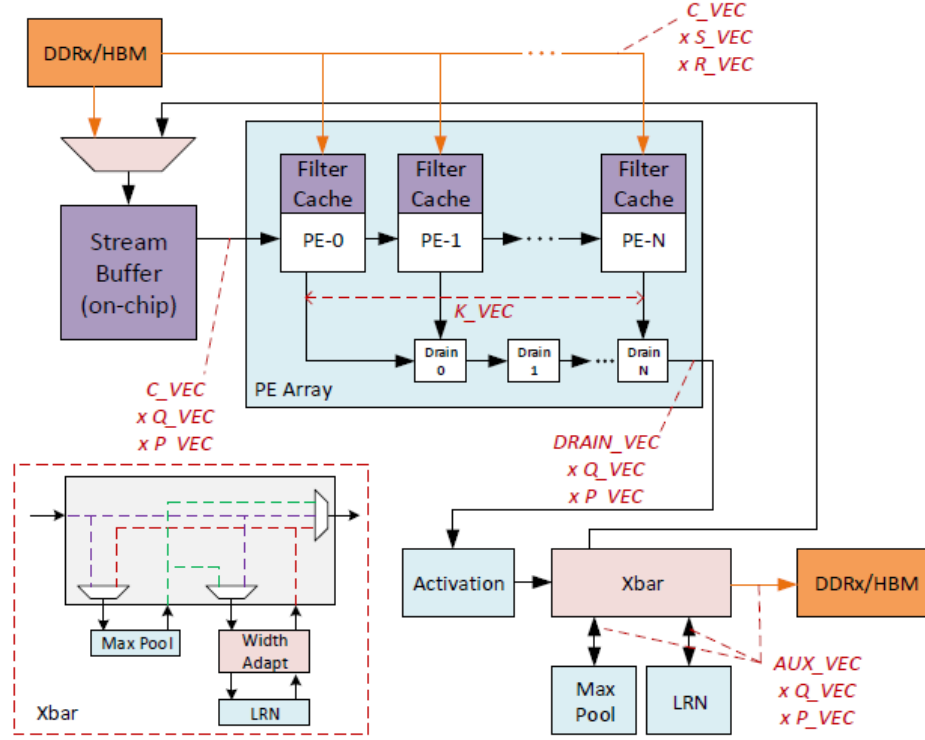


Figure 2.26: Intel's DL accelerator [186].

mention that vectorization depends on the layers' dimensions of the considered framework. However, they did not provide a systematic way for finding the optimal balance for the number of PEs used and the size of the caches. For efficient use of resources, Intel's DL accelerator maps *AVG-POOL* and *FC* primitives to convolutions in order to avoid having under-utilized (over time) dedicated auxiliary functions.

For the software side of Intel's DL accelerator, the proposed accelerator uses a graph compiler to map a NN architecture to the overlay for maximizing the hardware efficiency through slicing, allocation, and scheduling. In the slicing pass, the graph compiler breaks down the architecture into subgraph (a chain of functions) in such a way that they fit within the computing and storage resources of the overlay. A single *CONV* layer followed by a *POOL* layer is an example of CNN subgraph. The graph

compiler optimizes the external memory spill-points by group slicing technique. The group slicing allows several sequential convolutions, for instance, of a single slice to be computed before moving onto the next slice while using the whole stream buffer. During the allocation pass, the graph compiler optimizes the use of a custom developed filter caches and stream buffer by managing the read and write from the stream buffer for each slice. Moreover, it assigns an external memory address when the stream buffer is not big enough to hold the slice data.

Finally, Intel’s DL accelerator compiler schedules the execution of subgraphs using cost-based (the ratio of the output size to the effective input size) priority queue. Abdelfattah et al. utilized the software-programmable and run-time reconfigurable overlay to optimize the software and hardware implementation of GoogleNet [29] and ResNet [48] CNNs. The benchmark results on an Arria 10 GX 1150 FPGA demonstrated that Intel’s DL accelerator has a throughput of 900 fps on GoogLeNet. The team pointed that multi-FPGA deployment [189] might be used to further improve the throughput of Intel’s DL accelerator.

Kaiyuan et al. [60] proposed a complete design flow, referred to as Angel-Eye, for mapping CNNs onto FPGA. It includes a data quantization strategy, a parameterizable and run-time configurable hardware architecture to support various CNNs, FPGA platforms, and a compiler to map a given CNN onto the hardware architecture. It adopts the approach of using a flexible hardware architecture and maps different CNNs onto it by changing the software. The proposed design flow from CNN model to hardware acceleration is shown in Figure 2.27.

Due to the large dynamic range of data across different layers, the best radix

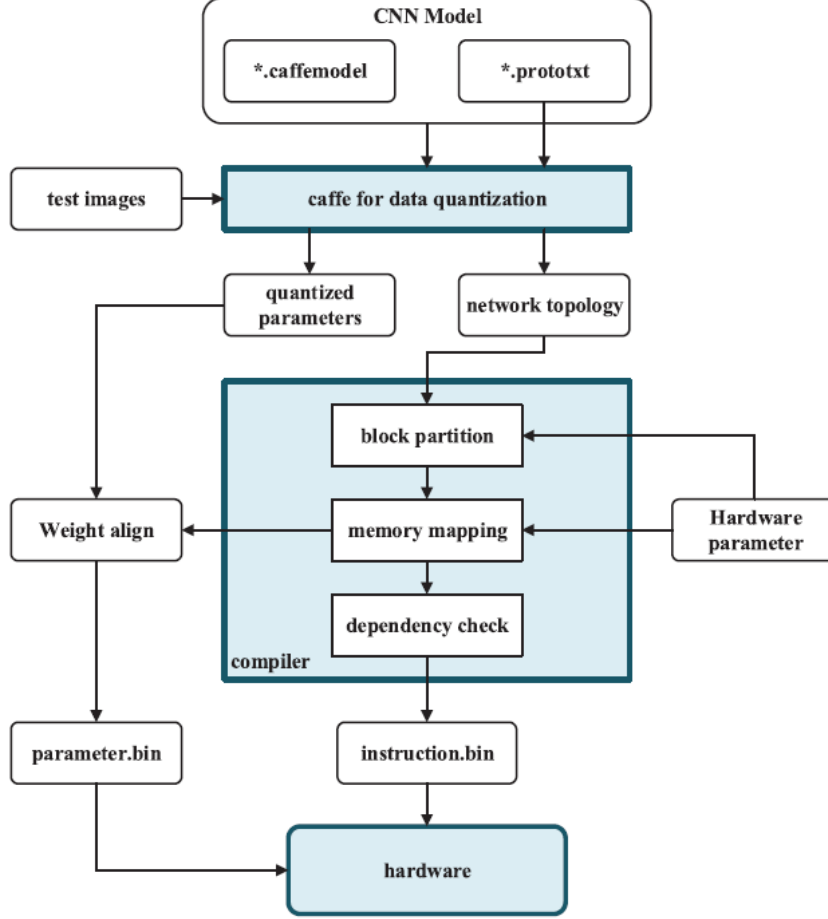
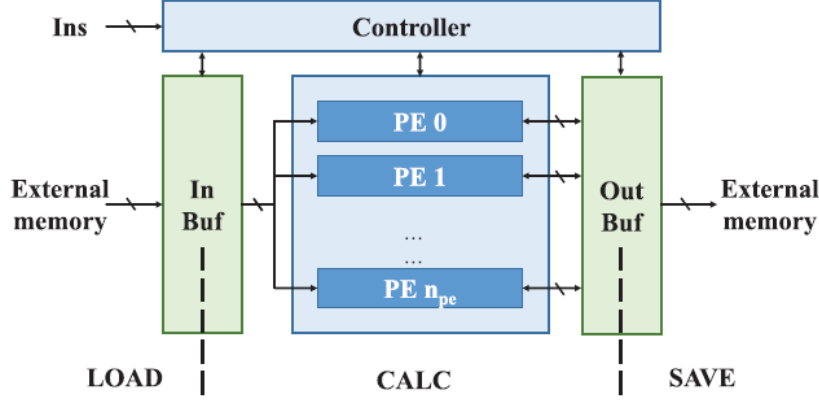


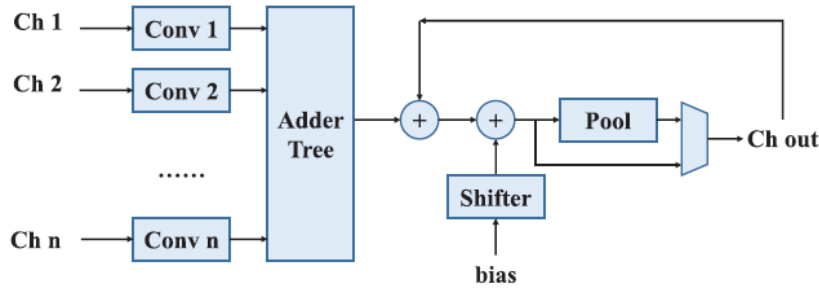
Figure 2.27: Design flow from CNN model to hardware acceleration [60].

point is found for each layer for a given bit-width. They demonstrated that their strategy can simplify state-of-the-art CNNs to 8-bit fixed-point format with negligible accuracy loss. Although 8 bits are used for representing FMs and weights, 24 bits are used for representing intermediate data, which is then aligned and quantized to 8 bits. Figure 2.28 shows the overall architecture of Angel-Eye and the structure of a single PE. The architecture is designed for supporting an instruction interface that supports three types of instructions; LOAD, SAVE, and CALC.

The overall architecture is divided into four main components; PE array, on-chip buffer, external memory, and controller. The PE array implements the *CONV* oper-



(a) Overall architecture.



(b) PE structure.

Figure 2.28: Angel-Eye (a) overall architecture, and (b) structure of a single PE [60].

ations in CNN and supports kernel level parallelism, and input and output channel parallelisms. It uses a 3×3 *CONV* kernel, as this size is most popular in state-of-the-art CNN models. However, larger kernel sizes are supported based on the use of multiple 3×3 kernels. The use of on-chip buffers allows data I/O and *CONV* calculation to be done in parallel. The controller is responsible for receiving the instructions and issuing them to the other components. The compiler maps the CNN descriptor to the set of instructions that will be executed by the hardware. It follows basic scheduling rules to fully utilize data localization in CNN and reduce data I/O.

The block partition step partitions the calculation of one layer to fit each block into the hardware. The memory mapping step allocates memory for communication

between the host CPU and the CNN accelerator. Based on the block partition result, on-chip memory is allocated for the IFM and OFM blocks and for the *CONV* weight and bias values. The dependency check step checks data dependency among instructions and sets appropriate instruction flags to maximize parallelism between *CONV* calculation and data I/O. Based on experimental results, it is shown that the 8-bit implementation of Angel-Eye on XC7Z020 achieves up to $16\times$ higher energy efficiency than NVIDIA TK1 and $10\times$ higher than NVIDIA TX1. In addition, the 16-bit implementation of Angel-Eye on XC7Z045 is $6\times$ faster and $5\times$ higher in power efficiency than peer FPGA implementation on the same platform [136].

In [97] and [185], a special register array architecture has been designed to rearrange buffers data and direct them into PEs for the purpose of implementing *CONV* module that supports specific stride and zero-padding settings. Although the designed *CONV* module is not generalized for any (kernel, stride) size configurations, it is composed of complex wire routing and control logic as shown in Figure 2.14. To have flexibility in directing the dataflow of *CONV* pixels, Ma et al. [190] replaced the register array architecture in [185] with a data router as shown in Figure 2.29.

The data router is a scalable set of data bus from buffer to PE (BUF2PE). The BUF2PE data bus consists of simple register arrays with FIFOs in between to form a line buffer similar to that in [143]. The register array uses the FIFO to pass its input pixels to the adjacent registers. Each BUF2PE data bus has different data movements within its register arrays to implement specific stride and kernel size settings. Unlike the register array architecture in [97] where the west zero-paddings are handled by changing the storage pattern within the input pixel buffer, the BUF2PE handles such

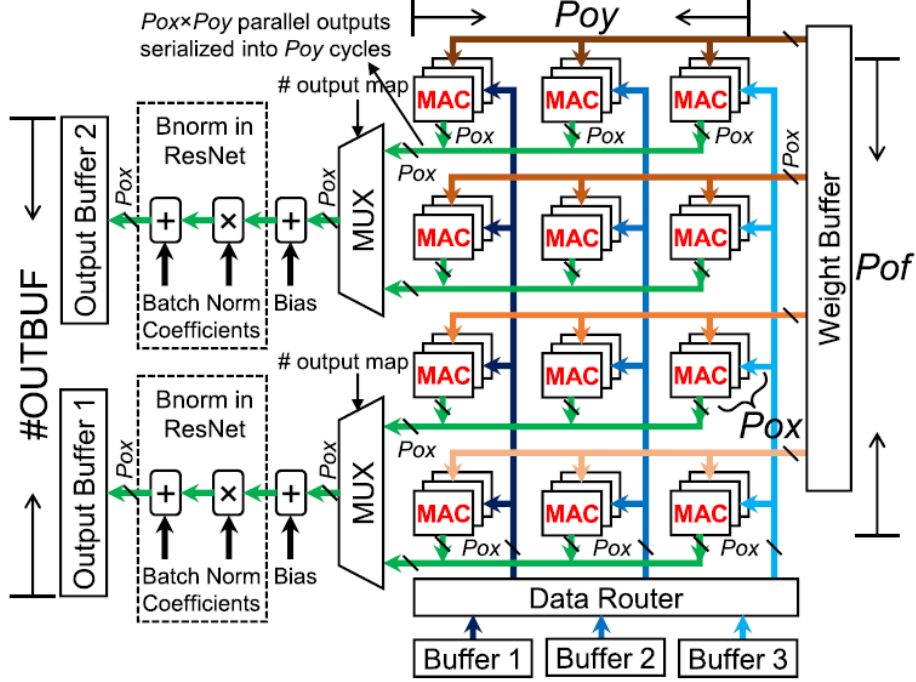


Figure 2.29: *CONV* acceleration architecture and dataflow using data router [190], where, $P_{ix} = P_{ox}$, and $P_{iy} = P_{oy}$.

kind of paddings by shifting the connection between the register arrays and the input pixel buffers to simplify the data transferring from off-chip memory to on-chip buffers. However, there is still a need for adjusting the storage pattern within the input buffers in order to handle other zero-paddings.

The global control logic is responsible for selecting the suitable BUF2PE data bus from the data router as well as the suitable storage pattern within the input buffers based on the (kernel, stride) size configuration of *CONV* layer. The *CONV* module has also been optimized by reducing the required number of parallel adders that add the partial sums with biases as well as the number of parallel multipliers and adders needed to perform *BatchNorm* operation by serializing the parallel outputs using multipliers. In addition, FxP16 has been used to represent both weights and FMs, while dynamically adjusting the decimal point in different layers to fully utilize the

existing data width [60]. The proposed compiler in [185] has been used to configure the parameterized Verilog scripts of the overall CNN acceleration system. Experimental results show throughput degradation on both Intel Arria 10 GX 1150 and Intel Stratix V GXA7 in comparison to the results in [185].

In Table 2.1, we summarize the reviewed FPGA-based DNNs acceleration techniques. For each technique, we list the DL model used, the number of needed operations per image, the FPGA platform used to implement the technique, the precision used for the weights and FMs, the clock frequency achieved, the type of LUT for the used platform, the design entry for describing the modeled DNN, the number of resources available by the used platform in terms of BRAMs/M20Ks, LUTs/ALMs, FFs, and DSPs, the percentage of each resource utilization, the performance in GOPs, and the power efficiency (GOPs/Watt).

Table 2.1: Implementation and performance summary of FPGA-based accelerators of DNNs.

Technique	Key Features	DL Model	Image Ops (GOPs)	Platform	Precision	Frequency (MHz)	LUT Type	Design Entry	Resources				Resources Utilization				Performance (GOPs)	Power Efficiency (GOPs/Watt)
									BRAMs/ M20Ks	LUTs/ ALMs	FFs	DSPs	BRAMs/ M20Ks	LUTs/ ALMs	FFs	DSPs		
CNN [114]	2D CONV modules, Memory interface with 8 simultaneous accesses	LeNet-5	0.52	Virtex-4 SX35	16-bit fixed-point	200	4-input	Lush	192	30,720	30,720	192	N/A	90%	90%	28%	5.25	0.35
CONV Coprocessor Accelerator [116]	Parallel clusters of 2D convolver units, data quantization, off-chip memory banks	4 CONV layers	1.06	Virtex-5 LX330T	16-bit fixed-point	115	6-input	C	324	207,360	207,360	192	0.93%	17%	19.05%	55.73%	6.74	0.61
MAPLE [119]	In-memory processing, banked off-chip memories, 2D array of vector PEs	4 CONV layers	1.06	Virtex-5 SX240T	fixed-point	125	6-input	C++	516	149,760	149,760	1,056	N/A				7	N/A
DC-CNN [96]	Integer factorization to determine the best config for each layer, input and output switches	3 CONV layers	0.52	Virtex-5 SX240T	48-bit fixed-point	120	6-input	RTL	516	149,760	149,760	1,056	N/A				16	1.14
NeuFlow [131]	Multiple full-custom processing tiles (PTs), Pipelining, Fast streaming memory interface	4 CONV layers in [133]	N/A	Virtex-6 VLX240T	16-bit fixed-point	200	6-input	HDL	416	150,720	301,440	768	N/A				147	14.7
Memory-Centric Accelerator [134]	Flexible off-chip memory hierarchy, Data reuse, Loop transformation	4 CONV layers	5.48	Virtex-6 VLX240T	fixed-point	150	6-input	C	416	150,720	301,440	768	45.5%	1.1%	N/A	6%	17	N/A
mn-X [136]	Cascaded pipelining, Multiple stream processing	2 CONV layers	0.55	Zynq XC7Z045	16-bit fixed-point	142	4-input	Lua	545	218,600	437,200	900	N/A				23.18	2.9
Roofline-based FPGA Accelerator [55]	Roofline-based model, Loop transformation, Loop tiling/unrolling	AlexNet	1.33	Virtex-7 VX485T	32-bit float-point	100	4-input	C	2,060	303,600	607,200	2,800	50%	61.3%	33.87%	80%	61.62	3.31
Microsoft Specialized CNN Accelerator [140]	Multi-banked buffers, Network on-chip for re-distribution of output data, Software configurable	AlexNet	1.33	Spartix-V GSDM5	32-bit float-point	100	4-input	C	2,014	172,600	690,000	1,590	N/A				178.63	7.15
Embedded FPGA Accelerator [88]	Data quantization and arrangement, SVD	VGG-16	30.76	Zynq XC7Z045	16-bit fixed-point	150	4-input	RTL	545	218,600	437,200	900	86.7%	83.5%	29.2%	80.2%	136.97	14.22
DeepBurning [144]	NN model compression, Compiler-based library, Automatic partitioning/tiling, Function Approx.	AlexNet	1.46	Zynq XC7Z045	16-bit fixed-point	100	4-input	RTL	545	218,600	437,200	900	N/A	17.3%	7.65%	16%	73	6.43
OpenCL-based FPGA Accelerator [73]	Design space exploration for all CNN layers, Genetic algorithm, Altera OpenCL SDK	AlexNet ^(a) VGG-16 ^(b)	1.46 30.9	Spartix-V ⁽¹⁾ GX47 Spartix-V ⁽²⁾ GSD8	(8-16)-bit fixed-point	120	7-input	OpenCL	2,560 2,567	234,720 292,400	938,880 1,050,000	256 1,963	37% 52%	47% 46%	N/A N/A	100% 37%	31.8 ^(a) 47.5 ^(b) 72.4 ^(a) 117.8 ^(b)	1.23 N/A 3.79 N/A
Caffeine [142], [151]	Systemic array architecture, Loop unrolling, Double buffering, Pipelining, Roofline model	AlexNet ^(a) VGG-16 ^(b)	1.46 31.1	Virtex-7 ⁽¹⁾ VX690T Xilinx ⁽²⁾ K1000	16-bit ^(a) fixed-point 8-bit ^(b) fixed-point	150 200	6-input	C++	2,940 2,160	433,200 331,680	866,400 663,360	3,600 2,760	42% 36%	81% 31%	36% 11%	78% 38%	35.4 ^(a) 266 ^(2a) 1171.7 ^(b) 165 ^(2a)	13.62 10.64 45.07 N/A
fpvgaConvNet [154]	Data-path optimization, Synchronous dataflow, Partitioning and folding, Design space exploration	LeNet-5 Scene Labelling [156]	0.0038 7.6528	Zynq XC7Z020	16-bit fixed-point	100	4-input	HLS	140	53,200	106,400	200	4.4%	18.2%	13%	1.2%	0.48	N/A
Throughput-Optimized FPGA Accelerator [159]	Four-levels of parallelism, Memory-cache mechanism, Design space exploration	LeNet AlexNet VGG-S	0.04 1.46 5.29	Virtex-7 VX690T	(8-16)-bit fixed-point	100	6-input	N/A	1,470	433,200	866,400	3,600	32.4% 60.5% 88.8%	53.8% 47.7% 51.7%	35.5% 37.3% 34.5%	80.8% 79.8% 81.9%	424.7 445.6 473.4	16.85 17.97 18.49
FP-DNN [160]	RTL-HLS hybrid compiler, Hand-written matrix multiply, Quantization, Tiling and double buffers	VGG-19 ResNet-152	39.26 22.62	Spartix-V GSDM5	16-bit fixed-point	150	4-input	C++ and OpenCL	2,014	172,600	690,000	1,590	48%	27%	N/A	66%	364.36 226.47	14.57 9.06
FINN [169]	Pipelining, Automatic partitioning and tiling, Approximate arithmetic	CNN	0.113	Zynq XC7Z045	(1-2)-bit precision	200	4-input	C++	545	218,600	437,200	900	34.1%	21.2%	N/A	N/A	2,465.5	210.7

Table 2.1: Implementation and performance summary of FPGA-based accelerators of DNNs (continued).

Technique	Key Features	DL Model	Image Ops (GOPs)	Platform	Precision	Frequency (MHz)	LUT Type	Design Entry	Resources			Resources Utilization			Performance (GOPs)	Power Efficiency (GOPs/Watt)		
									BRAMs/ M20Ks	LUTs/ ALMs	FFs	DSPs						
Customized CONV Loops Accelerator [97]	Numerical analysis of CONV loops opt., Solution space exploration, Dataflow opt.	VGG-16	30.95	Arria 10 GX 1150	(8-16)-bit float-point	150	8-input	Verilog	2,713	427,200	1,708,800	1,518	70%	38%	N/A	100%	645.25	30.44
		AlexNet	1.3315	Zynq XC7Z045	16-bit fixed-point	125	4-input	HLS	545	218,600	437,200	900	N/A	N/A	N/A	N/A	161.98	N/A
Latency-Driven Design for FPGA-based CNNs [171]	Synchronous dataflow, Weights reloading and SDF transformations, Design space exploration	VGG-16	30.72														123.12	
DL Architecture [176]	Winograd transformations, Double stream buffers, PE's double cache buffers, Daisy-chained PE's	AlexNet	1.46	Arria 10 GX 1150	Half-precision FP16 with shared exponent	303	8-input	OpenCL	2,713	427,200	1,708,800	1,518	92%	58%	40%	97%	1.382	30.71
Winograd-based CNN Accelerator [177]	Winograd algorithm, Loop unrolling, Double buffers, Batching, Line buffers, Design space exploration	AlexNet ^(a)	1.33	Zynq ⁽¹⁾ ZCU102	16-bit fixed-point	200	6-input	C	912	274,080	548,160	2,520			N/A		854.6 ^(a1)	36.2
		VGG-16 ^(b)	30.76	Xilinx ⁽²⁾ ZC706	16-bit fixed-point	167	4-input		545	218,600	437,200	900					294.0 ^(b1)	124.6
OpenCL-based Architecture for Accelerating CNNs [178]	2D BRAM-to-PE interconnection, 2D dispatcher, Roofline model, OpenCL	VGG-16	30.76	Arria 10 GX 1150	32-bit float-point 16-bit fixed-point	370	8-input	OpenCL	2,713	427,200	1,708,800	1,518	46.1%	N/A	N/A	87%	866	20.75
		AlexNet	1.33	Virtex-7 ⁽¹⁾ VX485T	32-bit float-point	100	4-input	C++	2,060	303,600	607,200	2,800	39.4%	58.3%	44.6%	87.3%	85.2 ^(a1)	11.2
Multi-CLP Accelerator for CNNs [93]	Dynamic programming, Double-buffering	SqueezeNet ^(b)	0.77	Virtex-7 ⁽²⁾ VX690T	16-bit fixed-point	170	6-input		2,940	433,200	866,400	3,600	37.7%	N/A	N/A	97.1%	708.3 ^(b1)	N/A
Automated Systolic Array Architecture for CNN [181]	2D systolic array architecture, Roofline model, Automation flow Design space exploration	AlexNet ^(a)	1.4	Arria 10 GX 1150	32-bit ⁽¹⁾ float-point (8-16)-bit ⁽²⁾ fixed-point	239.62 ^(a1)	8-input	OpenCL	2,713	427,200	1,708,800	1,518	87%	82%	N/A	85%	360.4 ^(a1)	20.75
		VGG-16 ^(b)	31.1										90.5%	83%	N/A	88.3%	460.5 ^(b1)	N/A
End-to-End Scalable FPGA Accelerator [182]	Flexible and scalable ResNet modules, CONV loops opt., Dataflow opt., Controlled execution flow of ResNets layers	ResNet-50	7.74	Arria 10 GX 1150	16-bit fixed-point	150	8-input	Verilog	2,713	427,200	1,708,800	1,518	61.5%	73%	N/A	98.8%	1,171.3 ^(b2)	N/A
		ResNet-152	22.62										80%	30%	N/A	69%	285.07	N/A
DLAU [183]	Pipelined processing units, Tiling, FIFO buffers	DNN	N/A	Zynq XC7Z020	48-bit float-point	200	4-input	N/A	280	53,200	106,400	220	12.5%	68.4%	26.6%	75.9%	N/A	N/A
An Automatic RTL Compiler for High-Throughput Deep CNNs [185]	Library-based RTL compiler, Flexible and scalable CNN modules, Layer combo computation, CONV loops and dataflow opt., Controlled execution flow of CNN layers	NIN ^(a)	2.2	Stratix-V ⁽¹⁾ GXA7	16-bit fixed-point	150	7-input	Verilog	2,560	234,720	938,880	256	59%	96%	N/A	100%	282.67 ^(a1)	
		VGG-16 ^(b)	30.95										86%	90%	N/A	100%	352.24 ^(b1)	
		ResNet-50 ^(c)	7.74	Arria 10 ⁽²⁾ GX 1150	16-bit fixed-point	200	8-input						76%	74%	N/A	100%	250.75 ^(c1)	
		ResNet-152 ^(d)	22.62										93%	77%	N/A	100%	278.67 ^(d1)	
ALAMO [87], [157]	Modularized RTL compiler, Loop unrolling, Loop tiling	AlexNet	1.46	Stratix-V ⁽¹⁾ GXA7	(8-16)-bit fixed-point	100	7-input	Verilog	2,560	234,720	938,880	256	56%	37%	N/A	100%	587.63 ^(a2)	N/A
Angel-Eye [177]	Quantization, Parallel PE's, Compiler, On-chip buffers	NIN	2.2	Arria 10 ⁽²⁾ GX 1150	16-bit fixed-point	200	8-input		2,713	427,200	1,708,800	1,518	82%	30%	N/A	100%	720.15 ^(b2)	
		VGG-16	30.76	Zynq XC7Z020	8-bit fixed-point 16-bit fixed-point	214	4-input	N/A	140	53,200	106,400	220	71%	51%	N/A	100%	619.13 ^(c2)	
Optimizing the CONV Operation to Accelerate DNNs on FPGA [190]	Scalable set of data buses (BUF2PE), Optimized CONV module for different (kernel, stride) size configurations, Flexible and scalable CNN modules, CONV loops and dataflow opt.	NIN ^(a)	2.2	Stratix-V ⁽¹⁾ GXA7	16-bit fixed-point	150	7-input	Verilog	2,560	234,720	938,880	256	59%	97%	N/A	100%	278.2 ^(a1)	
		VGG-16 ^(b)	30.95										86%	93%	N/A	100%	348.8 ^(b1)	
		ResNet-50 ^(c)	7.74	Arria 10 ⁽²⁾ GX 1150	16-bit fixed-point	200	8-input						76%	75%	N/A	100%	243.3 ^(c1)	
		ResNet-152 ^(d)	22.62										93%	78%	N/A	100%	276.6 ^(d1)	

2.2 Metaheuristics in the Design of CNNs

Currently, CNN structures are designed based on human expertise. For a given application, this consists of determining the number of *CONV* layers, number of *FC* layers, sizes of FMs in each layer, along with other operators. Recent research has demonstrated that a large number of weights in *FC* layers could be eliminated with minimal impact on accuracy. In addition, although the suggested CNN structures by experts perform well for various applications, the question arises whether the suggested structures could be optimized for performance with minimal impact on accuracy. Since the designed CNN has a significant impact on the complexity of its implementation, we review in this section some approaches attempting to optimize the design of CNNs using metaheuristics. NP-hard combinatorial optimization problems [191] appear in the design of CNNs. Some examples of areas include design of CNN structures, selection of weights and bias values to improve accuracy, and determination of optimal values of variables to reduce run-time. Below, we briefly touch upon some existing literature in these areas.

2.2.1 Structure Optimization

In the design of CNNs, the number of possible network structures increases exponentially with the number of layers. Xie and Yuille used genetic algorithm algorithm in learning deep network structures [192]. The objective was to find the best CNN struc-

ture that would minimize the error rate. The cost function was the CNN accuracy. They proposed an elegant encoding of chromosome using a fixed length binary string to represent each network structure. A CNN string represents only the *CONV* layers.

In each generation, using standard genetic operations, new individuals are generated and weak ones eliminated. The quality of an individual was assessed by its recognition accuracy which is obtained via the time consuming operation of training the network, and evaluating it on a validation set. Two small data sets were used (MNIST and CIFAR-10) to run the genetic implementation via which they demonstrated the discovery of new structures.

2.2.2 Weight and Bias Values Optimization

An attempt to train CNNs using metaheuristics (that is, determine weights and bias values) is presented in [193]. The objective again was to improve accuracy and minimize the estimated error. The authors experiment with three metaheuristic algorithms, namely; simulated annealing, differential evolution, and harmony search. The algorithms compute the values of weights and bias in the last layer. These values are used as the solution vector denoted by x which is to be optimized. The move comprised adding a small random value of Δx to perturb the state. The cost function y is modeled as

$$y = \frac{1}{2} \sqrt{\frac{\sum_{i=1}^N (o_i - u_i)^2}{N}} \quad (2.1)$$

where, o_i and u_i are the expected output and the real output for i -th input sample, respectively, and N is the number of used samples. The stopping criterion is when the

iteration count is reached or when the cost function goes below a pre-specified value.

2.2.3 Design Variables Optimization

Suda et al. [73] presented a systematic methodology for design space exploration with the objective of maximizing the throughput of an OpenCL-based FPGA accelerator for a given CNN model. FPGA resource constraints such as on-chip memory, registers, computational resources, and external memory bandwidth are considered. The optimization problem comprises finding the best combination of N_{CONV} , S_{CONV} , N_{Norm} , N_{POOL} , and N_{FC} variables, where

- N_{CONV} is the number of performed *CONV* MAC operations in parallel,
- S_{CONV} is the factor by which computational resources are vectorized to execute in a SIMD fashion,
- N_{Norm} represents the number of *Norm* operations performed in a single cycle,
- N_{POOL} is the number of parallel outputs of the *POOL* layer in a single cycle to achieve acceleration, and,
- N_{FC} is the number of parallel MAC operations performed in a single work-item within the *FC* layer.

The objective function to be minimized is the run-time (RT), and is given by

$$\sum_{\ell=0}^L RT_{\ell} [N_{CONV}, S_{CONV}, N_{Norm}, N_{POOL}, N_{FC}] \quad (2.2)$$

subject to DSP slices, logic resources, and memory constraints, where L represents the total number of CNN layers including the repeated layers, and RT_ℓ is the execution time of the ℓ -th layer. The *CONV* layer run-time is analytically modeled as a function of the design variables as

$$RT_{CONV_\ell} = \frac{\# \text{ of Ops in } CONV_\ell}{N_{CONV} \times S_{CONV} \times Frequency} \quad (2.3)$$

As for the other layers, that is, *Norm*, *POOL*, and *FC*, the following general model is proposed

$$RT_{Layer_\ell} = \frac{\# \text{ of Ops in } Layer_\ell}{Layer_\ell \text{ Unroll factor} \times Frequency} \quad (2.4)$$

The above analytical models are later validated by performing full synthesis at selective points and running them on the FPGA accelerator. Clearly, in order to determine the best values of the discussed design variables, exhaustive search is infeasible especially if the number of variables and or FPGA resources is large. We have to resort to iterative non-deterministic heuristics [191] such as simulated annealing, simulated evolution, tabu search, genetic algorithm, particle swarm optimization, cuckoo search, or any of the modern metaheuristics, to efficiently traverse the search space to find acceptable solutions.

The proposed methodology in [73] employing genetic algorithm was demonstrated by optimizing the implementation of two representative CNNs, AlexNet and VGG, on two Altera Stratix-V FPGA platforms, DE5-Net and P395-D8 boards, both of which have different hardware resources. Peak performance is achieved for both, the *CONV* operations and the entire CNN network.

One major issue related to the use of non-deterministic iterative heuristics in the design of NNs and CNNs is the large amount of memory required to store the state of solution and the amount of time taken to determine the cost of the solution, be it accuracy/error estimation, run-time, or any other objective. Reasonable estimation techniques and analytical formulations are required to efficiently traverse the design space in search of efficient solutions.

2.3 Optimization Mechanisms Employed in FPGA-based DNN Accelerators

In this section, we highlight the key features discussed in the acceleration of DNNs implemented on FPGAs. All reviewed techniques are centered around accelerating *CONV* operation as it consumes around 90% of the computational time. This is achieved by utilizing parallel MAC operations bounded by resource limitations. In addition, careful design of data access patterns are targeted to minimize the memory bandwidth requirements utilizing internal memory structures and maximizing data reuse. This is crucial in the acceleration process due to the large memory data that needs to be accessed including FMs and weights.

To minimize the memory footprint and to achieve effective utilization of resources, some techniques optimize the number of bits used to represent the FMs and weights with minimal impact on accuracy. This is combined with the optimized selection of the number of fraction bits used for each layer. Other techniques optimize the number of used weights in the *FC* layers as they are memory-intensive. Coprocessors are also

employed to automatically configure both the software and the hardware elements to fully exploit parallelism [96].

To optimize parallelization of *CONV* operations, several approaches have been attempted. Work load analysis has been tried to determine computations that can be structured as parallel streams [119]. The roofline model based accelerator uses polyhedral-based data dependence analysis to find the optimal unrolling factor for every *CONV* layer [138], and to fully utilize all FPGA computational resources through loop pipelining. To optimize performance, tiled matrix multiplication is structured as a pipelined binary adder tree for performing multiplication and generating partial sums [184]. An optimization framework has been proposed by Suda et al. who identified the key variables of the design [73] and optimize them to maximize parallelism.

To reduce computational complexity of *CONV* layers and improve resource efficiency, a number of approaches such as [172], [176], [177] utilized Winograd transformation in performing *CONV* operations as this reduces the computational complexity by around 50%. On the other hand, to maximize throughput, several techniques such as [93], [154], [159] have used multiple CLPs instead of using a single CLP that is optimized for all *CONV* layers. This pipelines the operation of the multiple CLPs achieving layer-level parallelism which maximizes resource utilization and enhances performance in comparison to using a single CLP.

Since the computational requirement of *FC* layers is significantly less than that of *CONV* layers, to improve performance, and maximize resource utilization, a number of techniques such as [142], [151], [176], [177] create batches by grouping different IFMs and processing them together in *FC* layers. Complex access patterns and data locality

are used in DeepBurning tool [144] for better data reuse. In [183], the authors explored hot spots profiling to determine the computational parts that need to be accelerated to improve the performance. Acceleration is accomplished by reducing the memory bandwidth requirements. Techniques proposed exploit data reuse to reduce off-chip memory communications. Loop transformations have also been used by reducing tiling parameters to improve data locality, and to reduce redundant communication operations to maximize the data sharing/reuse.

Efficient buffering, where the weight buffers are used to ensure the availability of *CONV* and *FC* layers' weights before their computation, as well as to overlap the transfer of *FC* layer weights with its computation, helps in improving performance [87], [157]. In the Catapult project, FPGA boards were integrated into data center applications and achieved speedup. Microsoft Research's Catapult utilized multi-banked input buffer and kernel weight buffer to provide an efficient buffering scheme of FMs and weights, respectively. To minimize the off-chip memory traffic, a specialized network on-chip was designed to re-distribute the output FMs on the multi-banked input buffer instead of transferring them to the external memory [140].

To further reduce memory footprint and bandwidth requirement, optimal fractional length for weights and FMs in each layer are used. SVD has also been applied to the weight matrix of *FC* layer in order to reduce memory footprint at this layer [88]. Tiling techniques have been proposed where large-scale input data is partitioned into small subsets or tiles whose size is configured to leverage the trade-off between the hardware cost and the speedup [183].

Automation tools have been developed that automatically build NNs with op-

timized performance [144]. They employ pre-constructed RTL module library that holds hardware (including logical and arithmetic operations) and configuration scripts. DeepBurning, for example, generates the hardware description for NN scripts. Another modularized RTL compiler, ALAMO, integrates both the RTL finer level optimization and the flexibility of HLS to generate efficient Verilog parameterized RTL scripts for ASIC or FPGA platform under the available number of parallel computing resources (i.e., the number of multipliers) [87], [157]. Acceleration is achieved by employing loop unrolling technique for *CONV* layer operations. Some of the reviewed techniques also help minimize the size of FPGA on-chip memories to optimize energy and area usage [134], [135]. In Table 2.2, we list the optimization mechanisms utilized by each of the reviewed techniques to maximize performance and throughput of FPGA-based DNNs.

Table 2.2: Optimization mechanisms employed for FPGA-based acceleration of DNNs.

Optimization Mechanism	VIP [106]	CNP [114]	<i>CONV</i> Coprocessor Accelerator [116]	MAPLE [119]	DC-CNN [96]	NeuFlow [131]	Memory-Centric Accelerator [134]	mn-X [136]	Roofline-based FPGA Accelerator [55]	Embedded FPGA Accelerator [88]	DeepBurning [144]
Loop Unrolling									×		
Loop Tiling	×					×	×		×	×	×
Loop Interchange							×				
Pipelining	×	×	×	×	×	×	×	×	×		
Input Batching											
<i>FC</i> Layer Batching											
Multi-CLPs											
Binarized CNN											
Fixed-Point Precision	×	×	×	×	×	×	×	×		×	×
Per-Layer Quantization										×	
Singular Value Decomposition										×	
Prefetching					×	×					
Rearranging Memory Data										×	×
In-Memory Processing				×							
Line Buffer										×	
Double Buffering									×		
Padding Optimizations											
Winograd Algorithm											
Approximating Non-Linear AF		×	×			×		×			×
Eliminating <i>FC</i> Layer			×		×			×	×		
Roofline Model											
Polyhedral Optimization									×		
Dynamic Programming					×						
Graph Coloring											
Graph Partitioning											
Pattern Matching											

Table 2.2: Optimization mechanisms employed for FPGA-based acceleration of DNNs (continued).

Optimization Mechanism	OpenCL-based FPGA Accelerator [73]	Caffeine [142], [151]	fpgaConvNet [154]	ALAMO [87], [157]	Throughput-Optimized FPGA Accelerator [159]	FP-DNN [160]	FINN [169]	Customized CONV Loop Accelerator [97]	Latency-Driven Design for FPGA-based CNNs [171]	DL Architecture [176]
Loop Unrolling	×	×	×	×	×			×	×	
Loop Tiling		×				×	×	×		×
Loop Interchange								×		
Pipelining	×	×	×	×	×		×		×	×
Input Batchting		×								
FC Layer Batching		×				×				×
Multi-CLPs			×		×					
Binarized CNN							×			
Fixed-Point Precision	×	×	×	×	×	×		×	×	
Per-Layer Quantization										
Singular Value Decomposition										
Prefetching		×		×						×
Rearranging Memory Data	×	×								
In-Memory Processing										
Line Buffer										
Double Buffering		×				×		×		×
Padding Optimizations										
Winograd Algorithm										×
Approximating Non-Linear AF	×		×				×		×	
Eliminating FC Layer										
Roofline Model		×					×			
Polyhedral Optimization		×								
Dynamic Programming										
Graph Coloring						×				
Graph Partitioning			×						×	
Pattern Matching									×	

Table 2.2: Optimization mechanisms employed for FPGA-based acceleration of DNNs (continued).

Optimization Mechanism	Wingrad-based CNN Accelerator [177]	OpenCL-based Architecture for CNNs [178]	Multi-CLP Accelerator for CNNs [93]	Automated Systolic Array Architecture for CNN [181]	End-to-End Scalable FPGA Accelerator [182]	DLAU [183]	An Automatic RTL Compiler for High-Throughput Deep CNNs [185]	Intel's DL Accelerator [186]	Angel-Eye [60]	Optimizing the CONV Op. to Accelerate DNNs on FPGA [190]
Loop Unrolling	×		×		×		×			×
Loop Tiling	×		×	×	×	×	×			×
Loop Interchange										×
Pipelining	×		×	×		×				
Input Batching			×							
FC Layer Batching	×									
Multi-CLPs			×						×	
Binarized CNN										
Fixed-Point Precision	×	×	×	×	×		×		×	×
Per-Layer Quantization									×	×
Singular Value Decomposition										
Prefetching	×							×		
Rearranging Memory Data		×			×					
In-Memory Processing										
Line Buffer	×	×								×
Double Buffering	×		×	×		×		×		
Padding Optimizations					×					×
Winograd Algorithm	×									
Approximating Non-Linear AF						×				
Eliminating FC Layer										
Roofline Model		×								
Polyhedral Optimization				×						
Dynamic Programming			×							
Graph Coloring								×		
Graph Partitioning										
Pattern Matching										

2.4 Recommendations for Enhancing the Effectiveness of Utilizing FPGAs in Accelerating CNNs

To enhance utilization of FPGAs in CNNs acceleration and to maximize their effectiveness, we recommend the development of a framework that includes a user-friendly interface that allows the user to easily specify the CNN model to be accelerated. This includes specifying the CNN model parameters in terms of number of *CONV* layers and their sizes, and number of *FC* layers along with other intermediate operations. The specified CNN model weights will be read from a file. In addition, the user should have the option of specifying the FPGA platform that will be used for implementing the CNN accelerator and the maximum tolerable error, along with the selection of a library from a set of applications to be used for model optimization and evaluation.

The framework then should perform optimizations to find the minimum number of bits that need to be used for representing the weights and FMs and the number of fraction bits to be used for each layer. In addition, optimization of *FC* layers is performed to minimize the memory requirements. All such optimizations are carried out bounded by the maximum error specified by the user for the specified application library. The framework should be designed based on the development of a scalable hardware architecture that works for any given FPGA platform and achieves higher speedup with the availability of higher resources.

Based on the available resources, specified by the FPGA platform, the tool will

perform optimizations to maximize parallelism and data reuse, given the resource limitations. The tool will then automatically generate the CNN model that will fit on the given FPGA platform and will allow the user to evaluate the performance based on the chosen application library. This will allow the user to evaluate the performance gains while evaluating different FPGA platforms with different resources. The tool should have the option to generate performance measures based on different performance metrics as selected by the user such as number of frames processed per second or number of operations performed per second. In addition, the tool will report other design metrics such as resource utilization, memory sizes and bandwidth, and power dissipation.

Furthermore, it is desired to have the option for the user to specify the desired performance for a given CNN model and have the tool perform necessary analysis and evaluation and recommend to the user candidate FPGA platforms for achieving the desired performance levels. This will require the development of reasonably accurate analytical models that will estimate the needed resources for achieving the desired performance. The user can then choose the recommended FPGA platform and perform complete evaluation to verify that the desired performance levels are met.

2.5 Summary

In this chapter, we reviewed recent developments in the area of acceleration of DNNs and, in particular, CNNs on FPGAs. The chapter provided a brief overview of DL techniques highlighting their importance, key operations, and applications. Special

emphasis is given on CNNs as they have wide applications in the area of image detection and recognition and require both CPU and memory intensive operations that can be effectively accelerated utilizing FPGA inherent ability to maximize parallelism of operations.

While the chapter briefly touches upon the acceleration techniques for DL algorithms and CNNs from both software and hardware perspectives, the core of this chapter has been the review of recent techniques employed in the acceleration of CNNs on FPGAs. A thorough up-to-date review is provided that illustrates the employment of various possibilities and techniques such as exploitation of parallelism utilizing loop tiling and loop unrolling, effective use of internal memory to maximize data reuse, operation pipelining, and effective use of data sizes to minimize memory footprint, and, to optimize FPGA resource utilization.

The chapter also presented the use of tools for generating RTL scripts that not only help in automating the design process, but also help in exploring the design space and suggesting efficient hardware. The chapter discusses the use of analytics such as: (i) work load analysis in determining the computations that can be parallelized, (ii) optimal loop unrolling factors, (iii) determining access patterns to improve data locality, etc. In addition, a brief review of the use of non-deterministic heuristics in solving NP-hard combinatorial optimization problems in the design and implementation of CNNs has been presented. Finally, the chapter summarized the key features employed by the various FPGA-based CNN acceleration techniques and provided recommendations for enhancing the effectiveness of utilizing FPGAs in CNNs acceleration.

CHAPTER 3

OPTIMIZATION OF FPGA-BASED CNN ACCELERATORS USING METAHEURISTICS

In recent years, convolutional neural networks (CNNs) have demonstrated their ability to solve problems in many fields and with accuracy that was not possible before. However, this comes with extensive computational requirements, which made general central processing units (CPUs) unable to deliver the desired real-time performance. At the same time, field programmable gate arrays (FPGAs) have seen a surge of interest in accelerating CNN inference. This is due to their ability to create custom designs with different levels of parallelism. Furthermore, FPGAs provide better performance per Watt compared to other computing technologies such as graphic processing units

(GPUs). The current trend in FPGA-based CNN accelerators is to implement multiple convolutional layer processors (CLPs), each of which is tailored for a subset of layers. However, the growing complexity of CNN architectures makes optimizing the resources available on the target FPGA device to deliver the optimal performance more challenging. This is because of the exponential increase in the design variables that must be considered when implementing a Multi-CLP accelerator as CNN’s complexity increases.

In this chapter, we present a CNN accelerator and an accompanying automated design methodology that employs metaheuristics for partitioning available FPGA resources to design a Multi-CLP accelerator. Specifically, the proposed design tool adopts simulated annealing (SA) and tabu search (TS) algorithms to find the number of CLPs required and their respective configurations to achieve optimal performance on a given target FPGA device. Here, the focus is on the key specifications and hardware resources, including digital signal processors (DSPs), block random-access memories (BRAMs), and off-chip memory bandwidth (BW). Experimental results and comparisons using four well-known benchmark CNNs are presented demonstrating that the proposed acceleration framework is both encouraging and promising. The SA-/TS-based Multi-CLP achieves $1.31\times - 2.37\times$ higher throughput than the state-of-the-art Single-/Multi-CLP approaches in accelerating AlexNet, SqueezeNet 1.1, VGG-16, and GoogLeNet architectures on the Xilinx VC707 and VC709 FPGA boards.

3.1 Introduction

CNN is a powerful method used for processing data with predefined grid-like topology, such as 1-D time series in speech recognition [194], 2-D image retrieval in face detection and recognition [195], [196], and in intelligent transportation systems [197]–[199], to name a few. The CNN proved its effectiveness when it was used to win the ImageNet challenge, an annual competition for visual object recognition, in 2012, by dropping the classification error record from 26% to 15%, which was a significant improvement at the time [26]. Since then, more studies and applications have emerged to enhance CNN application in different fields [91], [200], [201].

In general, CNNs consist of an input layer, an output layer, and multiple intermediate hidden layers. The key operations involved in the construction of CNN layers include convolution (*CONV*), pooling (*POOL*), and inner product. The *CONV* layer extracts the unique features from the input image. Specifically, the first *CONV* layer in the architecture captures low-level features, such as edges, colors, gradients, orientations, etc., while subsequent *CONV* layers adapt to extract higher-level features, resulting in a wholesome understanding of the processed image.

On the other hand, the *POOL* layer reduces the dimensionality of the processed data, extracts rotational and positional invariant features, and suppresses noise by applying the *max*, *avg*, or *min* functions. Last, but not least, the inner product layer, also known as the dense or the fully-connected (*FC*) layer, is a layer whose neurons are connected to all the neurons of its preceding layer, hence the name. It is worth noting that the last *FC* layer in the architecture is also referred to as the classification layer

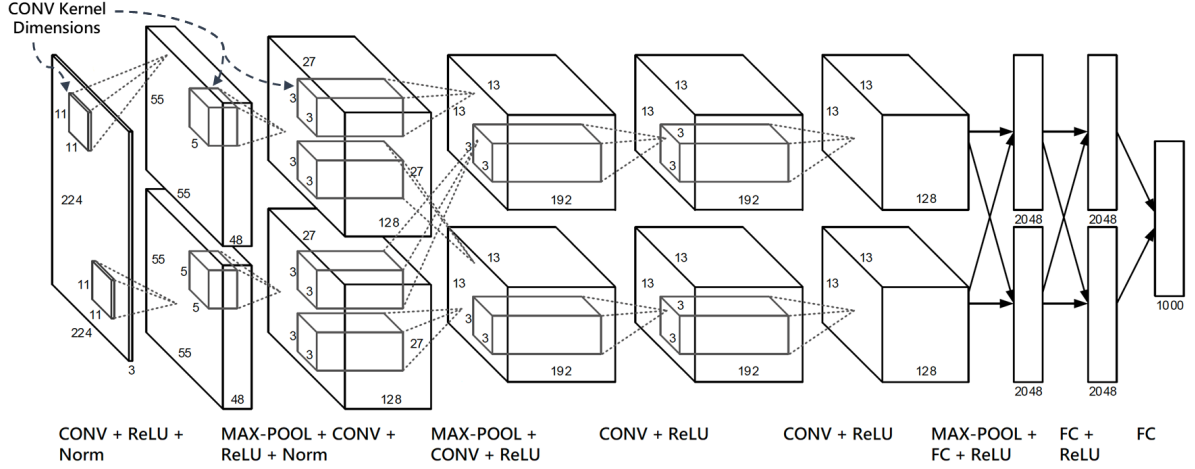


Figure 3.1: An illustration of AlexNet architecture. The input/output feature maps to/from each layer(s) are shown as matrices with their respective dimensions and the operations performed are indicated below.

because it is responsible for decision-making, such as the class score. Here, we must emphasize that there are other layers that can also be used in constructing CNNs such as normalization (*Norm*) and rectified linear unit (*ReLU*). The *Norm* layer is used to smooth the data, whereas the *ReLU* activation layer helps in learning and modeling complex data.

CNN layers are represented by a set of matrices that reflect the main elements and the relationship between them according to the network structure. Specifically, each layer receives the output matrix of the previous layer as input, which is a set of 2-D arrays named feature maps (FMs). Then, it performs its operation to produce the FMs for the subsequent layer. Figure 3.1 shows a well-known CNN architecture called AlexNet [26]. It consists of 5 *CONV* layers each of which is followed by a *ReLU* activation function, interspersed by 2 *Norm* layers, 3 *MAX-POOL* layers, and concluded by 3 *FC* layers.

In CNNs, as the name implies, *CONV* layers are the most critical ones, and their

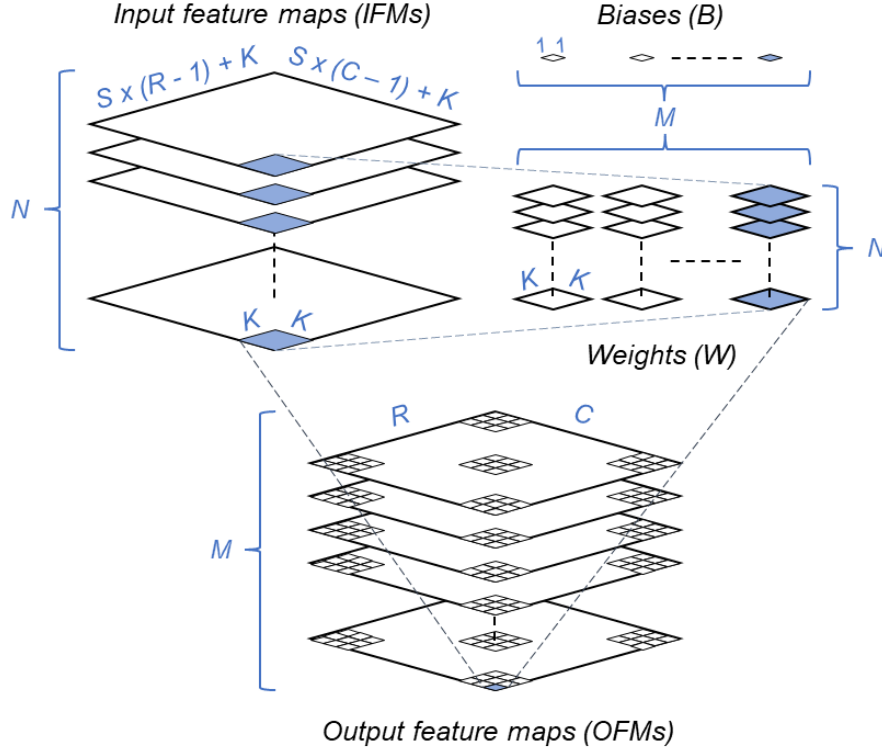


Figure 3.2: An illustration of the convolution layer. The blue feature in the last OFM is computed by taking the dot-product of the blue weights with the blue highlighted portion of the IFMs and then adding the blue shaded bias value to the result.

operations constitute over 90% of the total computation time [85]. The *CONV* layer receives a set of N FMs from the previous layer. Then, it convolves the input FMs (IFMs) with a set of M small kernels, also known as filters, each of size $K \times K$ and containing N channels. This is done using a shifting window that slides over the IFMs with a stride of size S . The kernel is a matrix of numbers called weight parameters whose values are learned during the training phase by the backpropagation algorithm. Each kernel is used to compute one specific output FM (OFM), that is, the m -th kernel produces the m -th OFM as demonstrated in Figure 3.2. Thus, the number of OFMs equals the number of *CONV* layer kernels. Additionally, each kernel has one bias (B) term. Each bias is added to every element in its corresponding OFM

to produce the final OFMs, which form the IFMs for the next layer. To determine the value of the neuron at position (r, c) of m -th OFM, the following computation is performed

$$\mathbf{OFM}[m][r][c] = \mathbf{B}[m] + \sum_{n=0}^{N-1} \sum_{i=0}^{K-1} \sum_{j=0}^{K-1} \mathbf{IFM}[n][S \times r + i][S \times c + j] \times \mathbf{W}[m][n][i][j] \quad (3.1)$$

Here, $\mathbf{B}[m]$ represents the bias parameter of the m -th kernel, $m \in [M]$, $\mathbf{W}[m][n][i][j]$ denotes the weight parameter at position (i, j) in the n -th channel of the m -th kernel, $n \in [N]$, i and $j \in [K]$, $\mathbf{IFM}[n][x][y]$ indicates the neuron at position (x, y) of the n -th IFM, and S is the stride size. By applying Equation (3.1) to compute all neurons of OFMs, we can see that the main structure of *CONV* layer consists of 6 nested loops as shown in Algorithm 1. One can also note that these loops contain a massive number of the multiply-and-accumulate (MAC) operations. Precisely, each layer performs $M \times R \times C \times N \times K^2$ MAC operation. Overall, AlexNet requires about 1.46 giga operations (GOPs) to process a single RGB image of size 224×224 pixels.

The intensive computational requirements of CNNs limit their real-time applications on general CPUs. On the other hand, the computational efficiency of GPUs and FPGAs make them excellent platforms for accelerating CNNs. However, the critical need for lower power consumption in today's applications, due to CNN deployment on battery-powered devices such as drones and Internet of things (IoT) [89], [90], makes FPGAs more suitable for CNN acceleration [91]. This is because of the high power efficiency, also known as performance per Watt, that FPGAs offer.

Algorithm 1: Convolution Algorithm.

Input: The input feature maps (**IFM**), weight (**W**) and bias (**B**) parameters, number of input feature maps (N), number of output feature maps (M), size of each output feature map, rows (R) and columns (C), kernel size (K), and window stride size (S).

Output: The output feature maps (**OFM**).

Procedure *CONV* (**IFM**, **W**, **B**, N , M , R , C , K , S):

```
1  for  $m \leftarrow 0$  to  $M - 1$  do
2      for  $r \leftarrow 0$  to  $R - 1$  do
3          for  $c \leftarrow 0$  to  $C - 1$  do
4              OFM $[m][r][c] \leftarrow \mathbf{B}[m]$  ▷ Set output neuron
5              for  $n \leftarrow 0$  to  $N - 1$  do
6                  for  $i \leftarrow 0$  to  $K - 1$  do
7                      for  $j \leftarrow 0$  to  $K - 1$  do
8                           $\mathcal{P} \leftarrow \mathbf{W}[m][n][i][j] \times \mathbf{IFM}[n][S \times r + i][S \times c + j]$ 
9                          OFM $[m][r][c] \leftarrow \mathbf{OFM}[m][r][c] + \mathcal{P}$ 
10                     end
11                 end
12             end
13         end
14     end
15 end
16 return OFM
```

With regards to model parameters, each standard *CONV* layer learns $M \times N \times K^2$ weight parameters. For example, AlexNet's first *CONV* layer receives 3 IFMs, uses 11×11 kernels, and produces 96 OFM, resulting in 34,848 parameters. Overall, AlexNet has over 60 million parameters which need about 240 MB of memory space. The memory requirements for FMs and model parameters exceed what commercially available FPGAs can provide in on-chip memory. Thus, they must be stored in off-chip memory and transferred to on-chip memory during computation on need basis. The significant amount of storage and external memory bandwidth required become a performance bottleneck. Therefore, many attempts have been made to maximize the throughput of CNN applications by designing a hardware accelerator known as a

CLP [55], [73], [92]–[94].

The CLP optimizes the implementation of *CONV* layers by applying loop transformations to Algorithm 1. Specifically, the loop unrolling technique is used to maximize the parallelism of *CONV* computations by replicating the hardware resources of MAC unit. Furthermore, loop tiling and local memory promotion techniques are adopted to reduce off-chip memory accesses and maximize data sharing and reuse. Here, we must emphasize that the CLP design is parametrized by the number of IFMs/OFMs to/from *CONV* layers as well as their dimensions. However, different *CONV* layers, even those in the same architecture, vary considerably in their configurations (N , M , R , C , K , and S).

To overcome this problem, three main design schemes have been proposed in the literature. In the first approach, such as in [92], a CLP is modeled for each *CONV* layer. In this way, a CNN of L *CONV* layers is accelerated using L CLPs that process L independent images in a pipelined fashion. Even though this approach optimizes the computations of each *CONV* layer, it suffers from non-negligible latency and bandwidth overheads. This is due to the need for orchestrating the off-chip memory accesses for a large number of CLPs. Additionally, dividing the limited on-chip memory among many CLPs reduces the overall data locality. Last, but not least, implementing a dedicated controller for each CLP leaves them with insufficient resources for computation.

The second approach, on the other hand, designs a single, unified CLP based on the optimal parameters that achieve the lowest overall latency [55], [73]. The globally optimized CLP is then used to iteratively process *CONV* layers, one layer at a time.

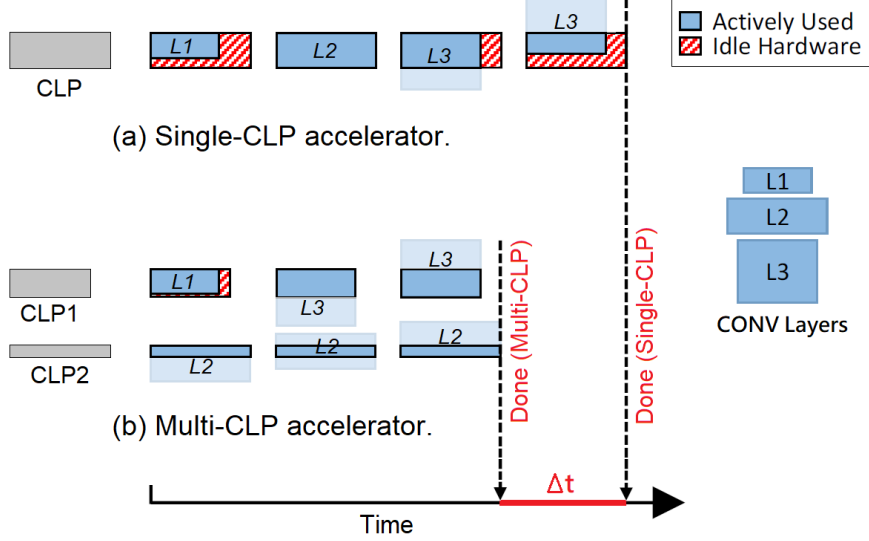


Figure 3.3: The operation of (a) Single-CLP accelerator and (b) Multi-CLP accelerator on CNN with three *CONV* layers. The dimensions of the hardware (*CLP*, *CLP1*, and *CLP2*) and the *CONV* layers (*L1*, *L2*, and *L3*) are represented by the size and shape of the boxes.

However, a CLP that provides the best performance across all layers is not necessarily optimal in utilizing its hardware resources. This is due to the radically varying *CONV* configurations. Considering the three *CONV* layers shown in Figure 3.3 as an example, one can note that the Single-CLP accelerator in Figure 3.3a is underutilized while processing *L1* and the portions of *L3*.

To alleviate this issue, the third approach implements multiple CLPs and optimally distributes the hardware resources among them [93], [94]. In Figure 3.3b, the same hardware resources used to implement the Single-CLP accelerator in Figure 3.3a are partitioned into two CLPs (*CLP1* and *CLP2*). Thereafter, by mapping *L1* and *L3* to *CLP1* and *L2* to *CLP2*, the Multi-CLP design achieves Δt reduction in the overall execution time compared to the Single-CLP design.

The problem now is, how to determine the appropriate number of CLPs to use, on what basis to assign a *CONV* layer to a CLP, how many hardware resources each

CLP can utilize, and how to determine the appropriate tiling factors to assign to each *CONV* layer? This intractable design space makes finding near-optimal configurations by exhaustive search algorithms almost impossible. Thus, intelligent techniques are needed to efficiently explore the design space for the optimal configurations of Multi-CLP design that improve the throughput of CNN applications.

Metaheuristics algorithms have proved their capabilities for finding optimal solutions for several NP-hard problems, with very efficient performance [202]. However, very few studies exist in the current literature that have proposed and exploited the use of metaheuristics algorithms in optimizing CNNs. Specifically, the standard genetic algorithm was employed in [73] to explore the design space for an optimal Single-CLP accelerator. On the other hand, the authors in [203] adopted the simulated annealing algorithm to improve the training of the LeNet-5 architecture.

In this chapter, we present a systematic methodology for the optimization of the throughput of an FPGA-based accelerator with Multi-CLP implementation. Specifically, we present an analytical and empirical design scheme for accurate modeling of cost, in terms of hardware resources used, and performance for a Multi-CLP design given its parameters. Then, we employ simulated annealing and tabu search metaheuristic algorithms to search for the optimal Multi-CLP design, constrained by the given CNN architecture and target FPGA specifications. As a case study, we implemented CNN accelerators for four well-known benchmark architectures, namely, AlexNet [26], SqueezeNet 1.1 [180], VGG-16 [27], and GoogLeNet [29] on VC707 and VC709 FPGA boards and compared them with previous approaches. Our implementation achieves a performance of 113.92 GFLOPs under 100 MHz working frequency.

The key contributions of this work are summarized as follows:

- Designing a CNN accelerator with loop tiling, loop unrolling, and loop pipelining techniques.
- Modeling CNN implementation and identifying the critical optimization variables for multiple CLPs design on FPGA platforms.
- Clarifying the complexity of finding the optimal configurations for multiple CLPs accelerator, i.e., defining the problem statement and the resultant intractable design space.
- Introducing a metric to quantify the dimensional mismatches between CLP dimensions and *CONV* layer dimensions, and use it to improve CNN throughput.
- Presenting computational performance and resource usage models for cost estimation of a candidate multiple CLPs design.
- Proposing a metaheuristic-based optimization framework that employs the estimation models to efficiently explore the design space using simulated annealing and tabu search algorithms, and find the optimal multiple CLPs design for CNNs on FPGAs based on performance and resources constraints.
- Validating the solutions obtained and comparing them with related works.

The rest of the chapter is organized as follows. Section 3.2 reviews previous work related to accelerating CNNs using FPGAs. In Section 3.3, we discuss the optimization techniques applied to achieve efficient implementation of CNNs on FPGA platforms.

Furthermore, we present an analytical and empirical model to accurately predict the performance and hardware resources required for a given design variant. Section 3.4 presents the methodology used for exploring the design space to achieve an efficient implementation of Multi-CLP hardware accelerator for *CONV* layers. The results of our experiments are presented and discussed in Section 3.5, and the chapter is concluded in Section 3.6.

3.2 Related Work

In this section, we review the existing approaches targeting the optimization of *CONV* implementation on FPGA platforms. Although CNN structures perform well for their intended applications, they have the potential to be further optimized with minimal impact on accuracy. For example, the values of FMs and weight parameters are originally represented as 32-bit floating-point (FP32) numbers. However, it has been demonstrated that fewer bits can be used to represent these values without a noticeable accuracy drop [105]. This reduces the hardware requirements for CNN implementation as well as reduces the inference latency.

The authors in [204] proposed an accelerator for LeNet-5 architecture to perform handwritten digits classification. The proposed strategy is based on three major aspects; loop parallelization to utilize resources, fixed-point data optimization to find the minimum number of bits that maintains accuracy level, and finally implementing MAC approximate units through logic blocks such as look-up tables (LUTs) and flip-flops (FFs) rather than using high-precision DSPs. With these optimizations, the

authors achieve less memory usage and reduced network latency. However, the solution is problem specific and deals with a relatively small CNN. Large CNNs would require advanced techniques to achieve the full potential of FPGA resources.

In [55], the authors adopted the polyhedral-based data dependence analysis [139] to optimize the computations and memory access operations in *CONV* layers. Specifically, they proposed an analytical design scheme to estimate the computational performance of a given *CONV* design. The loop transformations are employed to derive all possible *CONV* designs. The goal behind employing these techniques is to fully utilize the hardware resources provided by the target FPGA platform for effective acceleration. After enumerating all candidate solutions, the roofline performance model [138] is used to identify the optimal design for each layer.

Here, we must emphasize that different *CONV* layers have different structures, and therefore varying optimal loop unrolling and loop tiling factors. Considering AlexNet architecture discussed in Figure 3.1, the optimal unrolling factors $\langle T_n, T_m \rangle$ for the second and third *CONV* layers are $\langle 24, 20 \rangle$ and $\langle 5, 96 \rangle$, respectively, where T_n and T_m are the unrolling factors for the IFMs and OFMs, respectively. Hence, designing a CLP to accelerate these differently structured layers requires complex hardware implementation to reconfigure the computational units and their interconnects.

To overcome this issue, the authors in [55] designed the CLP based on the uniform cross-layer unrolling factors. More precisely, they found the unified unrolling factors $\langle T'_n, T'_m \rangle$ that maximize the overall performance. Accordingly, the computational engine of the CLP is implemented as T'_m duplicated tree-shaped poly structures. These structures receive identical T'_n values from the IFMs and each structure receives T'_n

weights from one of the T'_m kernels. Each structure multiplies its data using T'_n digital multipliers. Then, an adder tree is used to accumulate the multipliers outputs as well as the previous partial result. Furthermore, they used the double-buffering technique to overlap the off-chip memory and CLP buffers data transfers with computation.

Suda et al. [73] proposed an OpenCL-based framework for accelerating CNN inference. Focusing on *CONV* layers, the authors reformulated the *CONV* operation into a matrix multiplication operation. Specifically, they organized *CONV* kernels as a 2-D matrix of size $M \times (N \times K \times K)$. Similarly, the IFMs were flattened and rearranged as a 2-D matrix of $(N \times K \times K)$ rows and $(R \times C)$ columns. In this way, the output of *CONV* layer is calculated by multiplying these two matrices.

To speed up CNN operations, the authors followed the same strategy in [55] and modeled each type of CNN layer using unified loop unrolling factors. Precisely, they unrolled the output matrix of *CONV* operation in both dimensions by the factor T_{out} . Thus, they designed a CLP consisting of $T_{out} \times T_{out}$ structures, each of which computes an output feature. Accordingly, kernel weights and IFMs matrices are tiled into blocks of size $T_{out} \times T_{out}$. On each iteration, a tile from the kernel weight matrix and a tile from the IFMs matrix are fetched into the on-chip memory. Then, each CLP performs T_{out} MAC operations on a row of kernel weights tile and a column of IFMs tile.

To further improve the throughput of *CONV* layer, the computations on the inputs to CLP structures were also unrolled using the factor T_{in} so as to perform T_{in} MAC operations, out of the T_{out} , in parallel. To find the optimal unrolling configuration, the authors model the execution time of each layer as a function of the unrolling factors. Then, the standard genetic algorithm (GA) was used to explore the design space for

the minimum overall execution time considering FPGA resource constraints. Hence, this approach is referred to as GA-based Single-CLP. It is noteworthy that the high-level synthesis tool employed to compile their OpenCL codes to hardware restricted T_{in} to be from the set $\{1, 2, 4, 8, 16\}$ and T_{out} to be integer multiplicative of T_{in} .

The FPGA-based CNN accelerators discussed earlier employed a single globally optimized CLP design to maximize the overall throughput. However, using a CNN accelerator with uniform unrolling factors leads to sub-optimal performance for some *CONV* layers due to their significantly varying dimensions, which affects the overall performance. For example, the cross-layer optimization for AlexNet layers in [55] increases the total execution cycles of the layer-based optimization by 44,442 cycles. Moreover, following the methodology in [55] to derive the optimal Single-CLP design for SqueezeNet 1.1 implementation on Virtex-7 690T FPGA, the results show that the dynamic utilization of CLPs MAC units is less than 77% [93].

To improve the throughput of the Single-CLP design, Shen et al. [93] proposed to partition the available hardware resources between multiple CLPs. In doing so, they introduced a two-step iterative algorithm that searches for candidate partitions in the first step and then optimizes the tiling factors of each *CONV* layer in the second step. Specifically, the algorithm starts with a predefined target performance aiming to find a design with such a performance.

During the first phase, it generates several candidate partitions of computational resources. For each candidate design, the number of CLPs is set to the partition size. Then, the methodology in [55] is adopted to compute $\langle T_n, T_m \rangle$ for each CLP. The last process in this phase distributes CNN layers on the adopted CLPs. To facilitate this

process, the authors proposed to order memory-bounded layers based on their need for computation and communication, whereas computational-bounded layers are ordered based on the difference between the number of IFMs and the number of OFMs.

With such ordering, the authors assume that similar behavioral layers will be neighboring. Thus, they constrain the algorithm to only assign neighboring layers to the same CLP. At the end of the first phase, the performance of candidate designs is evaluated. If no one meets the target performance, the algorithm slightly reduces the target performance and repeats the first phase. When a valid design is found, the algorithm moves on to the second phase and computes $\langle T_r, T_c \rangle$ based on the methodology in [55]. The parameters for the design with the minimum bandwidth requirement are then used to configure a generic CLP template in high-level synthesis.

In this work, we improve the previously discussed works by efficiently partitioning hardware resources between multiple CLPs. In doing so, we employ intelligent meta-heuristics to help explore the intractable design space for multiple CLPs design that is more optimized than that achieved with conventional iterative algorithms. This results in better utilization of FPGA compute resources, improving CNN throughput.

Furthermore, unlike the work in [73] which constraints the unrolling factors to be from a predefined set, this work allows the unrolling factors to be whatever value that yields the highest performance. Last, but not least, the authors in [93] have specified the number of CLPs in multiple CLPs design to be 6 at most. This is because they want to keep the optimization time of their algorithm within an acceptable amount of time. This chapter proposes a metaheuristic-based optimization framework that is efficient in exploring design space and takes less than a minute to provide an optimal

Multi-CLP design on a single CPU.

3.3 CNN Accelerator Design on FPGA Platforms

In this section, we discuss the optimization techniques applied to achieve efficient implementation of CNNs on FPGA platforms. A hardware accelerator referred to as CLP is typically designed to improve *CONV* layer throughput. Here, the focus is on *CONV* layers because they are the most computationally intensive layers. Furthermore, we present an analytical and empirical model to accurately predict the performance and hardware resources required for a given design variant.

3.3.1 Optimizing CLP Computation and Memory Access

The computation of the *CONV* layer on FPGAs requires a memory space to store IFMs, OFMs, and all kernel weights. For instance, the second *CONV* layer in the VGG-16 architecture demands 12.25 MB, 0.14 MB, and 12.25 MB for IFMs, kernel weights, and OFMs, respectively. However, FPGA platforms usually contain limited on-chip memory. For example, Xilinx Virtex-7 485T FPGA can store a maximum of 4.52 MB of data on-chip. Thus, we employ the loop tiling technique to overcome this issue. Initially, all the data required for calculation is stored in the external memory. Then, the *CONV* operation is iteratively performed on a small portion of data, called a tile or block. Each tile is loaded and cached in on-chip buffers before being fed into the computational engine of the CLP.

However, loop tiling opens several design challenges that affect CLP performance.

Improper tiling may degrade the efficiency of data reuse and the parallelism of data processing. Therefore, deciding whether to tile a loop and with what factor plays an important role in the performance that can be achieved. Typically, *CONV* kernels are small in size, $K \leq 11$. Hence, the loops over the kernel dimensions, the loop iterators i and j in Algorithm 1, are not tiled.

On the other hand, the loops over the N IFMs, the M OFMs, and the dimensions, rows (R) and columns (C), of each OFM are tiled with the factors T_n , T_m , T_r , and T_c , respectively. In other words, each of these loops is transformed into two loops; an outer loop that iterates over the tiles, and an inner loop that iterates over the elements in each tile. Note that we denote the loop iterators over the tile elements of N , M , R , and C as nt , mt , rt , and ct , respectively.

Additionally, the *CONV* algorithm is a good candidate for parallelism. Thus, we adopt the loop unrolling technique to speed up *CONV* operations. Specifically, loop unrolling utilizes the available FPGA computation resources to maximize the parallelism of the MAC units. Even though loop unrolling considerably improves CLP throughput, it imposes complex connection topologies and affects CLP operating frequency. To mitigate these shortcomings, we only unroll IFMs and OFMs loops based on their tiling factors.

More precisely, the loops indicated by the iterators nt and mt are unrolled with the factors T_n and T_m , respectively. In this way, all their operations are performed in parallel. To avoid loop-carried dependence, we transformed unrolled loops to the innermost level. Moreover, the operations of the unrolled loops are fully-pipelined to improve system throughput. The *CONV* algorithm after optimization for loop tiling,

Algorithm 2: Optimized Convolution Algorithm.

Input: The input feature maps (**IFM**), weight (**W**) and bias (**B**) parameters, number of input feature maps (N) and their tiling factor (T_n), number of output feature maps (M) and their tiling factor (T_m), size of each output feature map, rows (R) and columns (C), and their corresponding tiling factors, T_r and T_c , kernel size (K), and window stride size (S).

Output: The output feature maps (**OFM**).

Procedure *CONV*(**IFM**, **W**, **B**, N , T_n , M , T_m , R , T_r , C , T_c , K , S):

```
1  for  $r \leftarrow 0$  to  $R - 1$  by  $T_r$  do
2    for  $c \leftarrow 0$  to  $C - 1$  by  $T_c$  do
3      for  $m \leftarrow 0$  to  $M - 1$  by  $T_m$  do
4        OFM_BUF  $\leftarrow$  Broadcast(B[ $m : m + T_m$ ],  $T_r$ ,  $T_c$ )
5        for  $n \leftarrow 0$  to  $N - 1$  by  $T_n$  do
6          IFM_BUF  $\leftarrow$  IFM[ $n : n + T_n$ ][ $S \times r : S \times (r + T_r - 1) + K$ ][ $S \times c : S \times (c + T_c - 1) + K$ ]
7          W_BUF  $\leftarrow$  W[ $m : m + T_m$ ][ $n : n + T_n$ ][ $0 : K$ ][ $0 : K$ ]
8          for  $i \leftarrow 0$  to  $K - 1$  do
9            for  $j \leftarrow 0$  to  $K - 1$  do
10             for  $rt \leftarrow 0$  to  $\min(T_r - 1, R - r - 1)$  do
11               for  $ct \leftarrow 0$  to  $\min(T_c - 1, C - c - 1)$  do
12                 for all  $mt \leftarrow 0$  to  $T_m - 1$  in parallel do  $\triangleright$  Unroll
13                   for all  $nt \leftarrow 0$  to  $T_n - 1$  in parallel do  $\triangleright$  Unroll
14                      $\mathcal{P} \leftarrow$  IFM_BUF[ $nt$ ][ $S \times rt + i$ ][ $S \times ct + j$ ]  $\times$ 
15                       W_BUF[ $mt$ ][ $nt$ ][ $i$ ][ $j$ ]
16                     OFM_BUF[ $mt$ ][ $rt$ ][ $ct$ ]  $\leftarrow$  OFM_BUF[ $mt$ ][ $rt$ ][ $ct$ ] +  $\mathcal{P}$ 
17                   end
18                 end
19               end
20             end
21           end
22         end
23         OFM[ $m : m + T_m$ ][ $r : r + T_r$ ][ $c : c + T_c$ ]  $\leftarrow$  OFM_BUF
24       end
25     end
26   end
27   return OFM
```

loop unrolling, and loop pipelining is demonstrated in Algorithm 2. Note that the nested loops are reordered to maximize data reuse, thus reducing off-chip memory accesses. Determining the optimal value of tiling factors for each CNN architecture will be discussed later in Section 3.4.

The optimized *CONV* algorithm can be divided into two stages; the memory data transfer stage, and the data computation stage. In the data transfer phase, highlighted in red and blue, the IFMs on-chip buffer (IFM_BUF) and weight parameters on-chip buffer (W_BUF) are filled with a block of IFMs and a block of kernel weights, respectively. Later, the OFM data block contained in the OFMs on-chip buffer (OFM_BUF)

is copied back to external memory as shown in line 23. Note that loop tiling factors control the size of these buffers, and thus the amount of data that is moved for each buffer refill or write-out.

On the other hand, the operations of the computation phase, shaded in yellow, are unrolled based on T_n and T_m tiling factors. Thus, loop tiling factors also control how CLP computational engine is constructed. Specifically, the computational engine is modeled as T_m duplicated MAC tree tiles. Each of which receives T_n different weights but they all share the same T_n input features. Each MAC tree tile multiplies its data using T_n digital multipliers. Finally, an adder tree is used to accumulate the output of the multipliers with the previous partial result.

To alleviate, or even prevent, blocking of CLP computations due to external memory data transfer, CLP adopts the double-buffering scheme for all on-chip buffers similar to [55], [93]. In other words, each on-chip buffer can be logically considered as two independent sets operating in a ping-pong manner as illustrated in Figure 3.4. In the first compute stage, the computational engine processes the features and weights from the input buffer set 0 (IFM_BUF0 and W_BUF0). During the same time, the features and weights required for the second compute stage are copied from external memory to input buffer set 1 (IFM_BUF1 and W_BUF1).

The next compute stage does the same but uses the opposite input buffers. That is why it is called the ping-pong buffer structure. With regards to the resulting OFMs, the output features from the first $\lfloor N / T_n \rfloor$ compute stages are stored in the output buffer set 0 (OFM_BUF0). During these stages, the content of the OFM_BUF1, which are the results of the previous $\lfloor N / T_n \rfloor$ compute stages, is transferred to off-

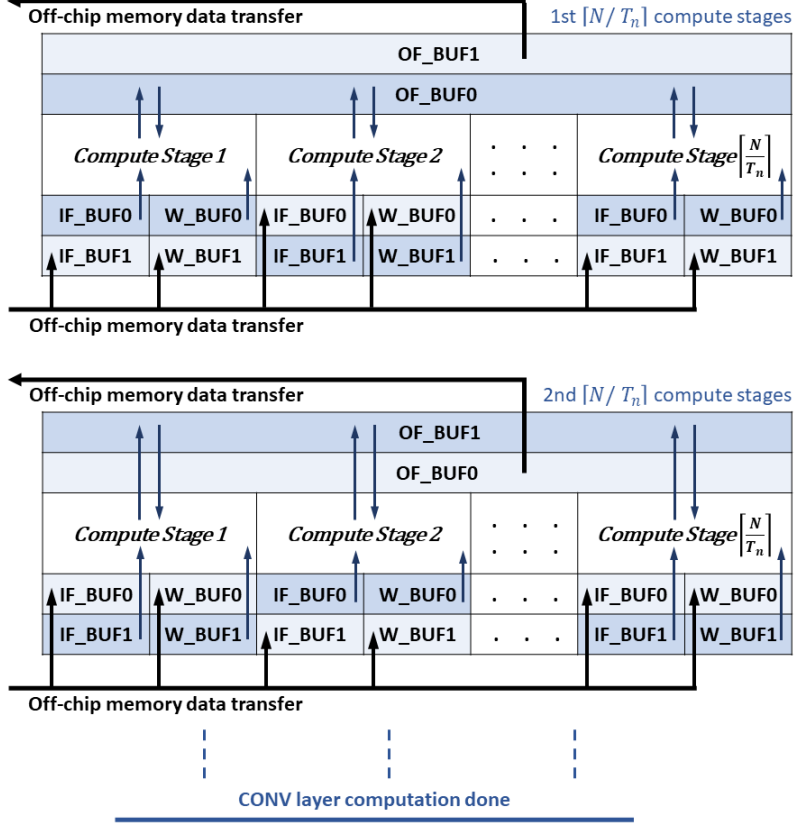


Figure 3.4: CLP computation and ping-pong buffer structure.

chip memory. Note that the CLP repeats this entire process several times to cover all the computations of *CONV* layer. As evident from Figure 3.4, double-buffering causes data transfer time to overlap with computation.

3.3.2 Multi-CLP Accelerator Design

In the previous section, we discussed an optimized CLP design that processes *CONV* layers of a CNN architecture iteratively one after another. Given that CNN layers have radically varying dimensions, designing a single CLP to process all layers may be ideal for a particular *CONV* layer, or some *CONV* layers, but causes under-utilization of computational resources in others, affecting overall performance. This is because

of the mismatch between the CLP dimensions (T_n and T_m) and the *CONV* layers dimensions (N and M).

Considering AlexNet layers illustrated in Figure 3.1 as a motivating example, we found that, on average, 78% of the computational resources were idle, or doing useless work, while processing the first *CONV* layer. Note that we followed the methodology in [55] to find the cross-layer optimized T_n and T_m . The utilization of g -th CLP computational resources while it is processing *CONV* layer ℓ can be quantified by

$$\lambda_\ell^{(g)} = \frac{\left(N_\ell / T_n^{(g)}\right) \times \left(M_\ell / T_m^{(g)}\right)}{\left\lceil N_\ell / T_n^{(g)} \right\rceil \times \left\lceil M_\ell / T_m^{(g)} \right\rceil} \quad (3.2)$$

To illustrate the computation resource utilization problem, Figure 3.5 shows a simple example of a CLP designed with factors $\langle T_n, T_m, T_r, T_c \rangle$ equal to $\langle 2, 3, 2, 2 \rangle$. The CLP is used to process a *CONV* layer whose configurations $\langle N, M, R, C, K, S \rangle$ are $\langle 3, 4, 5, 5, 2, 1 \rangle$. One can notice that N and M are not a perfect multiple of T_n and T_m , respectively. In particular, T_n needs two iterations to cover N . The first iteration covers the first two IFMs, leaving one IFM for the second iteration. The same goes for T_m where one OFM is left for the second iteration.

During the first cycle, the CLP works on complete tiles, therefore all of the T_m MAC tree tiles are in full use. On the other hand, the CLP uses a partially filled tile of input features in cycle 185, during which only T_m (out of $T_n \times T_m$) multipliers and adders do useful calculations. To make matters worse, during the last cycle, cycle 400, which computes the output feature **OFM**[3][4][4], neither IFM_BUF nor W_BUF is completely full. Thus, only half of the resources of the first MAC tree tile are used.

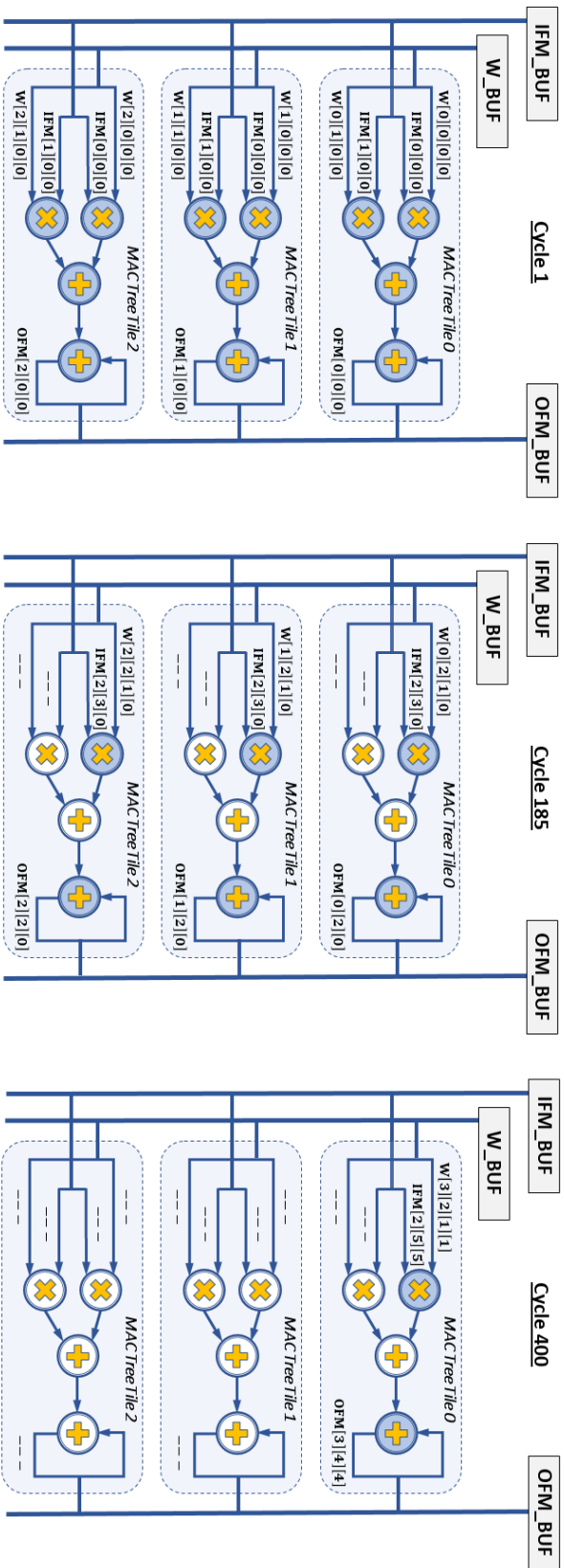


Figure 3.5: An example of a Single-CLP design with $\langle T_n, T_m, T_c \rangle$ factors set to $\langle 2, 3, 2, 2 \rangle$. The CLP processes a *CONV* layer with $\langle N, M, R, C, K, S \rangle$ configurations equal to $\langle 3, 4, 5, 5, 2, 1 \rangle$.

The combined effect of T_n and T_m dimensional mismatches leads to utilizing only 50% of the computational resources based on Equation (3.2).

To overcome the issue of under-utilization, and thus improve CNN throughput, the CLP can be designed with reconfigurable unrolling factors to make it work well with different layers. However, such a design requires the construction of complex hardware structures, which leaves the CLP with insufficient resources for computation. Instead, we partition the available hardware resources among multiple small, specialized CLPs. Specifically, the Multi-CLP design adopts G CLPs to process CNN layers, where $G \in [1, L]$ and L is the number of *CONV* layers. Each *CONV* layer is bound to a single CLP. In other words, a fixed set of layers, referred to as $\mathfrak{L}_g$, is assigned to g -th CLP, $\mathfrak{L}_g \subseteq [L]$. Later, each CLP sequentially processes its assigned *CONV* layers.

In this context, we define an *episode* as a single pass through CLP layers. To avoid intra-*episode* data dependencies, the OFMs produced from *CONV* layer ℓ during the i -th *episode* are not used in *episode* i . Instead, they are used as input for *CONV* layer $(\ell + 1)$ in *episode* $(i + 1)$. Therefore, CLPs need to synchronize before starting a new *episode*. The advantage of Multi-CLP accelerator comes from various dimensions supported that can accommodate *CONV* layers of varying dimensions. Furthermore, by applying data pipelines to the CLP level as well, it becomes possible to work concurrently on L independent images. Since a single *CONV* layer is processed for a given input image in an *episode*, processing an image requires L *episodes*. Note that this nature of back-to-back processing allows data transfer for one layer to be overlapped with computation for another.

The problem now is, how to determine the appropriate number of CLPs to use, on

what basis to assign a *CONV* layer to a CLP, and how many hardware resources each CLP can utilize? Here, we must emphasize that this research aims to optimize CNN throughput. To achieve that, each *CONV* layer must be assigned to the CLP that most closely matches its dimensions. Therefore, we need to look for a *CONV-CLP* assignment that maximizes resource utilization. Additionally, CNN throughput is constrained by the CLP that takes the longest time to finish the *episode*. Thus, the *episode* time for that CLP should be as short as possible. Furthermore, the size of on-chip buffers is inversely related to the required off-chip bandwidth. Therefore, we need to ensure that the Multi-CLP design uses sufficient on-chip buffers to minimize duplicate data transfers.

To fulfill the aforementioned requirements, in Section 3.3.3, we present an analytical and empirical model to help estimate the performance and cost, in terms of hardware resources, for a candidate Multi-CLP design. Then, in Section 3.4, we introduce an optimization framework that employs the proposed model as well as metaheuristics to explore the intractable design space for the optimal Multi-CLP design, constrained by the given CNN architecture and target FPGA specifications.

3.3.3 Design Cost and Performance

The Multi-CLP design for a CNN is characterized by (i) the number of CLPs, (ii) the assignment of *CONV* layers to CLPs, (iii) the unrolling factors $\langle T_n, T_m \rangle$ of each CLP, and (iv) the tiling parameters $\langle T_r, T_c \rangle$ of each *CONV* layer. Considering that synthesis, placement, and routing of a candidate Multi-CLP design may take several minutes or even hours, one can note that it is infeasible to perform these processes at each design

point for selecting the candidate with the highest performance. Hence, this section presents an analytical and empirical design scheme for accurate modeling of cost, in terms of hardware resources used, and performance for a Multi-CLP design given its parameters. Here, the focus is on the key specifications and hardware resources of the target FPGA platform, including DSPs, BRAMs, and off-chip memory BW.

3.3.3.1 Computational Performance

To evaluate the efficiency of a Multi-CLP design, the speed of each CLP, in terms of computation cycles, must be considered. Based on Algorithm 2, the number of cycles needed to process *CONV* layer ℓ assigned to g -th CLP is modeled as follows. Along the rows and columns dimensions of the OFMs, the algorithm iterates over the tiles and the elements in each tile. Thus, it runs for $R_\ell \times C_\ell$ cycles to process loop iterators r , c , rt , and ct . On the other hand, IFMs and OFMs are unrolled using the factors $T_n^{(g)}$ and $T_m^{(g)}$, respectively. Therefore, processing loop iterators n , m , nt , and mt need $\left\lceil N_\ell / T_n^{(g)} \right\rceil \times \left\lceil M_\ell / T_m^{(g)} \right\rceil$ cycles. Note that the two innermost loops, highlighted in yellow, only take 1 cycle because they are unrolled. Finally, loop iterators i and j are neither tiled nor unrolled, and thus, require $K_\ell \times K_\ell$ cycles. Altogether, the number of cycles required to process *CONV* layer ℓ on g -th CLP is calculated as in [93] by

$$\text{Comp_Cycle}_\ell^{(g)} = \left\lceil \frac{N_\ell}{T_n^{(g)}} \right\rceil \times \left\lceil \frac{M_\ell}{T_m^{(g)}} \right\rceil \times R_\ell \times C_\ell \times K_\ell^2 \quad (3.3)$$

Accordingly, the computational performance of the g -th CLP design in processing *CONV* layer ℓ is defined as the total number of operations, i.e., multiply and accu-

multimate operations, required by the CLP to process the layer over the total number of cycles needed to do so as follows

$$\begin{aligned} \text{Comp_Perf}_\ell^{(g)} &= \frac{2 \times N_\ell \times M_\ell \times R_\ell \times C_\ell \times K_\ell^2}{\text{Comp_Cycle}_\ell^{(g)}} \\ &= \frac{2 \times N_\ell \times M_\ell}{\left\lceil N_\ell / T_n^{(g)} \right\rceil \times \left\lceil M_\ell / T_m^{(g)} \right\rceil} \end{aligned} \quad (3.4)$$

3.3.3.2 DSP Slice Usage

The use of DSP slices in each CLP is dominated by the T_m MAC tree tiles that work in parallel to improve computational throughput. Each MAC tree tile consists of T_n parallel multipliers and an adder tree. Specifically, each multiplier performs multiplication on an input activation feature and an input kernel weight. Then, the resultant products are accumulated with the old partial result using a binary adder tree consisting of T_n adders. Thus, CLP computational engine contains $T_n \times T_m$ multipliers and adders.

It is noteworthy that the number of CLP slices required depends on the operation type and data representation. To estimate how many DSP slices are needed for a given CLP design, we empirically measure the number of DSPs used to implement the multiplier and adder for each representation in the design space and use a lookup table model to calculate the total usage. Based upon this, the number of DSP slices required to implement the computational engine of g -th CLP is determined by

$$\text{DSP_Usage}^{(g)} = \text{DSP}(\mathcal{Q}_{IFM}, \mathcal{Q}_W) \times T_n^{(g)} \times T_m^{(g)} \quad (3.5)$$

where $\mathcal{Q}_{IFM}$ and $\mathcal{Q}_W$ indicate whether input features and weights, respectively, are in single-precision floating-point (FP32) representation or quantized to low-precision representation such as 16-bit fixed-point (FxP16) format, and the function $\text{DSP}(\mathcal{Q}_{IFM}, \mathcal{Q}_W)$ returns the number of DSP slices required to implement a single multiplier and a single adder using the specified format. For instance, the FP32 adder and multiplier comprise 2 and 3 DSP slices, respectively [205]. That is, $\text{DSP}(\text{FP32}, \text{FP32})$ equals 5. On the other hand, the FxP16 adder and multiplier can be implemented on the same DSP slice, and therefore $\text{DSP}(\text{FxP16}, \text{FxP16})$ equals 1. For a Multi-CLP design consisting of G CLPs, the number of DSP slices required is the sum of the DSPs used by all CLPs

$$\text{DSP_Usage} = \sum_{g=0}^{G-1} \text{DSP_Usage}^{(g)} \quad (3.6)$$

3.3.3.3 BRAM Usage

Each CLP in Multi-CLP design requires three on-chip buffers, namely, IFM_BUF, W_BUF, and OFM_BUF as discussed in Section 3.3.1. These buffers are typically constructed using BRAMs. To accurately model BRAM usage in each CLP, the following must be considered; (i) BRAM capacity and configuration capability, (ii) data precision, (iii) buffer scheme (single/double buffering), (iv) number of banks needed per buffer, (v) number of read/write ports required for each bank, (vi) CLP unrolling factors (T_n, T_m) , and (vii) tiling parameters (T_r, T_c) and configurations (K, S) for each *CONV* layer assigned to the CLP.

In this work, we employ the double-buffering scheme for all buffers to overlap off-chip data transfer with computation. Furthermore, we adopt the memory banking

method to allow each buffer to provide the required number of input/output channels. Accordingly, the bank size must be large enough to support the two most demanding successive tiles. If considering the g -th CLP that processes *CONV* layers in subset $\mathfrak{L}_g$, the above requirement on minimum bank depth is translated as

$$\text{Min_Depth}_d^{(g)} = \max \left(\text{MFP}(d, \ell) + \text{MFP}(d, \ell + 1), \right. \\ \left. 2 \times \text{MFP}(d, \ell) : \forall \ell \in \mathfrak{L}_g \right) \quad (3.7)$$

where the function $\text{MFP}(d, \ell)$ returns the memory footprint, in terms of data size, for layer ℓ tiles on each bank in the buffer type d , $d \in \{IFM, W, OFM\}$, and $\mathfrak{L}_g$ is a set of all *CONV* layers assigned to g -th CLP, $\mathfrak{L}_g \subseteq [L]$. The number of BRAMs per bank (BPB) is calculated by

$$\text{BPB}_d^{(g)} = \left\lceil \frac{\text{Min_Depth}_d^{(g)}}{\text{ADDR}(\mathcal{Q}_d)} \right\rceil \quad (3.8)$$

Here, the $\text{ADDR}(\mathcal{Q}_d)$ is a function that returns the appropriate BRAM depth, i.e., the number of addresses, to store data represented with the bit-width specified by $\mathcal{Q}_d$. The BRAM model considered in this chapter is based on the RAMB18E1 primitive provided by Xilinx 7 series FPGAs [206]. This memory can be configured (*depth* $\times$ *width*) as a 512×36 , 1024×18 , 2048×9 , 4096×4 , 8192×2 , or 16384×1 . Thus, $\text{ADDR}(\text{FP32})$ is equal to 512, while $\text{ADDR}(\text{FxP16})$ is equal to 1024. Thus, the total number of BRAMs required to construct the buffer type d of g -th CLP is calculated by

$$\text{BRAM_Usage}_d^{(g)} = \text{BPB}_d^{(g)} \times \mathcal{B}_d^{(g)} \quad (3.9)$$

where $\mathcal{B}_d^{(g)}$ represents the number of memory banks needed to design the buffer type d of g -th CLP. Details of the specification of BRAMs and the number of banks needed for each buffer type are presented next.

IFM_BUF: The CLP computational engine requires T_n channels from IFM_BUF. Therefore, this buffer is designed using T_n banks, i.e., $\mathcal{B}_{IFM}^{(g)} = T_n^{(g)}$. When processing *CONV* layer ℓ , its footprint on each of these banks is

$$\text{MFP}(IFM, \ell) = (K_\ell + S_\ell \times (T_{r\ell} - 1)) \times (K_\ell + S_\ell \times (T_{c\ell} - 1)) \quad (3.10)$$

Note that each of IFM_BUF BRAMs must provide two independent ports; a read-only port for supplying CLP computational engine and a write-only port for loading off-chip memory data. Therefore, the type of these BRAMs is set to simple dual-port mode [206].

W_BUF: The weights buffer is similar to IFMs buffer in terms of operating mode. But, unlike IFM_BUF which is shared among all MAC tree tiles, the digital multipliers of MAC tree tiles use different kernel weights, and thus require independent memory banks. Accordingly, the W_BUF is organized into $T_n \times T_m$ banks, i.e., $\mathcal{B}_W^{(g)} = T_n^{(g)} \times T_m^{(g)}$. The memory footprint of *CONV* layer ℓ on each of these banks is

$$\text{MFP}(W, \ell) = K_\ell \times K_\ell \quad (3.11)$$

OFM_BUF: In each computation cycle, the old output feature is loaded and accumulated with the result of MAC tree tile. After that, the new output feature is

stored back in the OFMs buffer. As we note, the accumulator needs a read port and a write port. However, these ports use the same address for both operations. Additionally, the resulting output features from the previous stage must be transferred to off-chip memory. Therefore, the OFM_BUF uses true dual-port BRAMs in write-first mode [206]. One can also note that OFM_BUF must be organized into T_m banks. That is, $\mathcal{B}_{OFM}^{(g)}$ equals $T_m^{(g)}$. Thus, the memory footprint of *CONV* layer ℓ on each of these banks is calculated as

$$\text{MFP}(\text{OFM}, \ell) = T_{r\ell} \times T_{c\ell} \quad (3.12)$$

Together, the total number of BRAM resources required to construct the on-chip buffers of the Multi-CLP design is calculated as

$$\text{BRAM_Usage} = \sum_{g=0}^{G-1} \sum_{d \in \{IFM, W, OFM\}} \text{BRAM_Usage}_d^{(g)} \quad (3.13)$$

3.3.3.4 Bandwidth Usage

The computational performance discussed in Section 3.3.3.1 represents the maximum number of operations that can be performed with the available computational resources per cycle. Thus, it is also known as the computational roof. However, one critical problem is that CLP computation may be blocked by data transfer for memory-bound layers. Consequently, such layers cannot achieve the best performance supported by computational resources. To find the attainable computational performance in such a case, we adopt the roofline model [138].

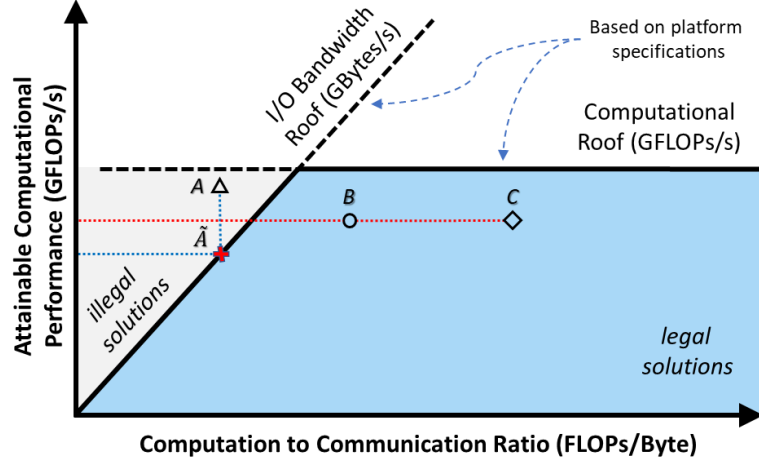


Figure 3.6: An illustration of roofline model basis. As we can see, implementation A is memory-bounded, i.e., system throughput is constrained by the communication with off-chip memory. Therefore, the attainable performance for implementation A is the computational performance obtained from implementation $\tilde{A}$. On the other hand, implementations B and C are computation-bounded, and thus system throughput can only be improved by using additional computational resources. One can also note that design C has better data reuse compared to design B , and thus it is more preferable than design B .

The roofline model is used to provide performance estimates for a candidate implementation on a particular system. It relates the attainable performance to the peak performance the system can achieve and the off-chip DRAM traffic as illustrated in Figure 3.6. The I/O bandwidth roof determines the maximum computational performance supported by the memory system for a given computation to communication (CTC) ratio, which is calculated as $CTC \times BW$. Thus, the attainable performance for $CONV$ layer ℓ when it is processed by g -th CLP is given by

$$\text{Perf}_{\ell}^{(g)} = \min\left(\text{Comp_Perf}_{\ell}^{(g)}, CTC_{\ell}^{(g)} \times BW\right) \quad (3.14)$$

where $\text{Comp_Perf}_{\ell}^{(g)}$ is the computational roof indicated in Equation (3.4), the BW

refers to the off-chip memory bandwidth, and the $CTC_\ell^{(g)}$ describes the off-chip memory traffic that the CLP needs during *CONV* layer processing. The CTC can be calculated by dividing the total number of operations by the total amount of off-chip data accessed in a similar way to [55] as follows

$$CTC_\ell^{(g)} = \frac{2 \times N_\ell \times M_\ell \times R_\ell \times C_\ell \times K_\ell^2}{\sum_{d \in \{IFM, W, OFM\}} \alpha_{d, \ell}^{(g)} \times \mathcal{B}_d^{(g)} \times \text{MFP}(d, \ell)} \quad (3.15)$$

Here, $\mathcal{B}_d^{(g)}$ denotes the number of memory banks in the buffer type d of g -th CLP, $\text{MFP}(d, \ell)$ refers to the memory footprint for *CONV* layer ℓ tiles on these memory banks as mentioned in Equations (3.10), (3.11), and (3.12), and $\alpha_{d, \ell}^{(g)}$ indicates the number of off-chip memory accesses needed for buffer type d during the computation of *CONV* layer ℓ by g -th CLP, such that

$$\begin{aligned} \alpha_{IFM, \ell}^{(g)} &= \alpha_{W, \ell}^{(g)} = \frac{N_\ell}{T_n^{(g)}} \times \frac{M_\ell}{T_m^{(g)}} \times \frac{R_\ell}{T_{r\ell}} \times \frac{C_\ell}{T_{c\ell}} \\ \alpha_{OFM, \ell}^{(g)} &= \frac{M_\ell}{T_m^{(g)}} \times \frac{R_\ell}{T_{r\ell}} \times \frac{C_\ell}{T_{c\ell}} \end{aligned} \quad (3.16)$$

To estimate the bandwidth required to support the maximum computation speed, we focus on the peak bandwidth that a CLP needs. Specifically, the minimum bandwidth required to achieve the peak computational performance for *CONV* layer ℓ which is processed by g -th CLP is determined by

$$\text{Min_BW}_\ell^{(g)} = \frac{\text{Comp_Perf}_\ell^{(g)}}{CTC_\ell^{(g)}} \quad (3.17)$$

For a memory-bound layer, the speed of g -th CLP design in processing *CONV* layer ℓ is defined by the data transfer cycles instead of the computation cycles mentioned in Equation (3.3). Thus, the arithmetic utilization of Multi-CLP design processes a CNN with L *CONV* layers using G CLPs is given by

$$\text{Utilization} = \frac{\sum_{g \in [G]} \sum_{\ell \in \mathfrak{L}_g} \lambda_{\ell}^{(g)} \times \text{Cycle}_{\ell}^{(g)}}{G \times \max\left(\text{Cycle}^{(g)} : \forall g \in [G]\right)} \quad (3.18)$$

where $\lambda_{\ell}^{(g)}$ is the utilization ratio of g -th CLP computational resources while it is processing *CONV* layer ℓ that is computed as mentioned in Equation (3.2).

3.4 Design Space Exploration

In this section, we discuss the difficulty of determining the value of design variables for multiple CLPs used to accelerate CNNs on FPGAs. Then, we present our proposed approach to finding the optimal Multi-CLP design for a given CNN architecture and resource budget.

3.4.1 Problem Statement and Formulation

The implementation of a CNN architecture on an FPGA platform using multiple CLPs while optimizing their performance is a non-trivial task. This is because it requires optimizing the utilization of both processing units and memory units to minimize the execution cycles. However, the number of cycles required depends on the number of CLPs used, the assignment of *CONV* layers to CLPs, the loop unrolling factors

$\langle T_n, T_m \rangle$ for each CLP, and the tiling parameters $\langle T_r, T_c \rangle$ for each *CONV* layer, which imposes an intractable design space. Furthermore, we must ensure that the peak memory bandwidth and resources utilized for a given candidate design comply with the given hardware constraints.

Consequently, determining the best values of the design variables through exhaustive search is infeasible, especially when the number of variables is large and/or FPGA resources are limited. Additionally, the number of possible assignments for *CONV* layers to CLPs is exponential in the number of layers. Considering the unrolling factors $\langle T_n, T_m \rangle$ as a motivating example, the search space of a given CNN structure with L *CONV* layers would consist of $L^L \times \max(N_\ell, \forall \ell \in [L])^L \times \max(M_\ell, \forall \ell \in [L])^L$ possible solutions, where N_ℓ and M_ℓ are the number of IFMs and OFMs for *CONV* layer ℓ , respectively.

For such a computationally hard problem, there is no existing deterministic algorithm that can find the optimal solution. Ad-hoc heuristic solutions have been employed by Shen et al. [93] to find good solutions. However, iterative non-deterministic heuristics can be employed to efficiently traverse the search space as they have been proven to find excellent solutions to such hard problems [191]. In this work, we employ simulated annealing (SA) and tabu search (TS) algorithms to find the optimal Multi-CLP design for a given CNN structure. The SA and TS are adopted due to their efficiency in finding high-quality solutions to such kinds of problems, simplicity of implementation, ability to provide a globally optimal solution in a reasonable time, the ease of tuning their parameters, and providing a balance between the exploration of solution space and the exploration of information obtained [207].

Table 3.1: An example of assigning five *CONV* layers to five CLPs with initial random values to T_n , T_m , T_r , and T_c .

Config	T_n	T_m	L_1	L_2	L_3	L_4	L_5	# DSPs
CLP_1	3	5	0	0	0	1	1	75
CLP_2	4	1	0	1	0	0	0	20
CLP_3	1	1	1	0	1	0	0	5
CLP_4	7	2	0	0	0	0	0	-
CLP_5	3	6	0	0	0	0	0	-
T_r			3	5	2	4	1	
T_c			3	3	3	4	2	

The space of solutions comprises finding optimal values for the following four elements; (i) the number of CLPs to be used, (ii) the assignment of *CONV* layers to CLPs, (iii) the values of T_n and T_m for each CLP, (iv) the values of T_r and T_c for each *CONV* layer. The solution representation is composed of $L + 2$ rows as shown in Table 3.1. Each of the first L rows represents a single CLP, while the last two rows represent the T_r and T_c values of each *CONV* layer. On the other hand, CLP rows contain $L + 2$ columns. The first two columns store the current values of T_n and T_m , while the remaining L columns indicate the assignment of layers to CLPs by storing 0 (unassigned) or 1 (assigned). Here, we must emphasize that a column representing a layer assignment must have a single 1, while a row can have multiple 1s.

Initially, the search begins with a random assignment of *CONV* layers to CLPs, and random T_n and T_m values for each CLP provided they do not violate the hardware resource constraints. The candidate solution represented in Table 3.1 shows that the total number of DSPs is 100, which can be calculated based on Equation (3.6) assuming features and weights are in FP32 representation. One can notice that CLP_4 and CLP_5 do not require any DSP since no layer is assigned to them. Note that in

the case where more than one layer is assigned to a CLP, the CLP processes these layers sequentially one after another. Thus, the number of cycles required by a CLP is the sum of the number of cycles needed to process all its layers.

The proposed optimization algorithm explores the design space by generating neighboring solutions from the current one, aiming to reach the optimal candidate. In our formulation, a neighboring solution is generated by performing a single move from the current solution. A move or step is defined as a single change in the current solution either by changing the layer assignment to a different CLP, or by changing one of the T_n , T_m , T_r , or T_c parameters for a CLP and *CONV* layer. An optimal solution is defined as a Multi-CLP design that requires the least number of cycles to process all layers. The number of cycles required for the candidate design $\mathcal{S}$ is calculated as

$$\text{Cycle}(\mathcal{S}) = \max \left(\sum_{\ell \in \mathfrak{L}_g} \text{Cycle}_\ell^{(g)}, \forall g \in [G] \right) \quad (3.19)$$

where $\text{Cycle}_\ell^{(g)}$ is the number of cycles required to process *CONV* layer ℓ in g -th CLP, G is the number of CLPs in the candidate solution $\mathcal{S}$, and $\mathfrak{L}_g$ is a set of all *CONV* layers assigned to g -th CLP, $\mathfrak{L}_g \subseteq [L]$. Note that the number of cycles for memory-bounded layers is defined by the data transfer cycles, while the number of cycles for computational-bounded layers is defined by the computational roof as discussed in Section 3.3.3. Based on the discussion above, the optimization problem can be

summarized as

$$\begin{aligned}
& \underset{\mathcal{S}}{\text{minimize}} \quad \text{Cycle}(\mathcal{S}) \\
& \text{subject to} \quad \text{DSP_Usage}_{\mathcal{S}} \leq \text{DSP}_{max} \\
& \quad \quad \quad \text{BRAM_Usage}_{\mathcal{S}} \leq \text{BRAM}_{max}
\end{aligned} \tag{3.20}$$

Here, the $\text{DSP_Usage}_{\mathcal{S}}$ and $\text{BRAM_Usage}_{\mathcal{S}}$ are the total number of DSP and BRAM resources required for the solution $\mathcal{S}$, which are computed as discussed in Equations (3.6) and (3.13), respectively, the DSP_{max} and BRAM_{max} are the number of DSPs and BRAMs available in the target FPGA platform. For each candidate solution, the memory optimization to find the values of T_r and T_c for each *CONV* layer is based on minimizing the peak memory bandwidth required as discussed in Section 3.3.3.4. The parameters T_r and T_c in turn would set the buffer sizes for each CLP, and thus they must comply with the available on-chip memory budget as demonstrated in Section 3.3.3.3.

3.4.2 Optimization of Multi-CLP Designs

In the section, we describe the employment of simulated annealing and tabu search metaheuristic algorithms in optimizing the implementation of FPGA-based CNN accelerators based on multiple CLPs.

3.4.2.1 Simulated Annealing-based Multi-CLP

The SA is an effective optimization tool based on inspiration derived from annealing of metals [208], [209]. Its power lies in its simplicity of implementation, and the

Algorithm 3: Simulated Annealing Algorithm.

Input: The initial solution (S_0), initial temperature (T_0), cooling rate (α), annealing temperature constant (β), total allowed time ($MaxTime$), and time until next update (M).

Output: The best admissible solution (S^*).

Procedure SA($S_0, T_0, \alpha, \beta, MaxTime, M$):

```
1   $S \leftarrow S_0$ 
2   $T \leftarrow T_0$ 
3   $Time \leftarrow 0$ 
4  repeat
5       $S \leftarrow \text{Metropolis}(S, T, M)$ 
6       $Time \leftarrow Time + M$ 
7       $T \leftarrow \alpha \times T$ 
8       $M \leftarrow \beta \times M$ 
9  until  $Time \geq MaxTime$ 
10  $S^* \leftarrow S$ 
11 return  $S^*$ 

Procedure Metropolis( $S, T, M$ ):
12 repeat
13      $S_{new} \leftarrow \text{Neighbor}(S)$  ▷ Do a single move
14      $\Delta h \leftarrow \text{Cost}(S_{new}) - \text{Cost}(S)$  ▷ Using Eq. (3.19)
15     if ( $\Delta h < 0$ ) or ( $random < e^{-\Delta h/T}$ ) then
16          $S \leftarrow S_{new}$  ▷ Accept the solution
17     end
18      $M \leftarrow M - 1$ 
19 until  $M = 0$ 
20 return  $S$ 
```

ease of tuning its parameters. During search, the moves that cause a decrease in the cost function, also known as good moves, are always accepted. On the other hand, the moves that lead to higher cost, also known as bad moves or uphill moves, are sometimes accepted with a probability that depends on a parameter called temperature (T). Initially, that is when T is high, the search is almost random. As T decreases, the probability of accepting bad moves decreases and the search begins to become greedy. At zero temperature, the search becomes totally greedy, and only good moves are accepted.

The basic structure of SA algorithm is shown in Algorithm 3, the core of which

is the **Metropolis** procedure. The **Metropolis** procedure simulates the annealing process at a given temperature value T [210]. It receives the current temperature T and the current solution $\mathcal{S}$ as input and improves it. It is also provided with the value M , which is a multiplication factor that increases the amount of time annealing is applied at a given temperature T [211]. The temperature is initialized to a value T_0 (explained below), and is slowly decreased using the parameter α . As temperature is lowered, the amount of time spent in annealing at a given temperature is gradually increased, using the parameter β , $\beta > 1$. The time spent in each call to **Metropolis** procedure is stored in variable *Time*. SA algorithm stops when the time limit is reached, i.e., $Time \geq MaxTime$.

The **Metropolis** procedure generates a local new solution $\mathcal{S}_{new}$ from a given solution $\mathcal{S}$ using the procedure **Neighbor**. Two perturbation schemes are used to generate a new neighbor out of the current solution; either (i) by randomly changing the assignment of a single *CONV* layer to CLPs, or (ii) by mutating T_n or T_m of a randomly selected CLP. These two perturbations are performed probabilistically, by making changes to T_n or T_m with 80% probability while making changes of *CONV* layer assignment to CLPs with 20% probability. The rationale for this is that because of the observation that optimizing the values of T_n and T_m for CLPs has greater effect on the solution cost. Additionally, during later iterations, changing *CONV* layer assignment to CLPs results in more invalid solutions. This is because adding new CLPs would require additional DSPs which may exceed the DSP constraint.

The function **Cost**($\mathcal{S}$) returns the cost of a given solution $\mathcal{S}$, which represents the number of cycles required to complete the execution for the given solution repre-

sentation as mentioned in Equation (3.19). The new solution is accepted if its cost, $\text{Cost}(\mathcal{S}_{new})$, is better than the cost of the current solution $\mathcal{S}$. The new solution is also accepted by **Metropolis** probabilistically when its cost is higher than the cost of the current solution. This is done by generating a random number in the range 0 to 1. If the generated number is smaller than $e^{-\Delta h/T}$, then the inferior solution is accepted, resulting in hill-climbing, i.e., transition from a low-cost solution to a high-cost solution, where e is the Euler's number, and Δh is the change in cost.

3.4.2.2 Tabu Search-based Multi-CLP

The TS is an another non-deterministic algorithm that also starts from an initial feasible solution and generates a list of candidate solutions, that forms what is known as the candidate list (V^*), by making neighborhood moves, and then selecting from this list the solution that is best among all candidates in the current iteration [212]–[215]. To avoid move reversals, that is recycling back to previously visited solutions, a device called tabu list (TL) is used that stores some attribute(s) of these moves. The TL has a given size and can be viewed as a window or queue on accepted moves. The moves stored in TL are not allowed as they may undo previous moves returning back to the same solution.

The tabu status is overridden when certain criteria, known as aspiration criteria (AC), are satisfied. The AC temporarily override the tabu status if the move causes the cost to be less than the aspiration level (AL), a value defined based on the AC adopted. For instance, the best cost criterion defines the AL as the cost of the best admissible solution. Accordingly, if the tabu move leads to a solution whose cost is less

Algorithm 4: Tabu Search Algorithm.

Input: A set of feasible solutions (Ω), candidate list size (N), tabu list size (M), aspiration criteria (AC), total allowed time ($MaxTime$).

Output: The best admissible solution ($\mathcal{S}^*$).

Procedure TS($\Omega, N, M, AC, MaxTime$):

```
1  Start with an initial feasible solution  $\mathcal{S}, \mathcal{S} \in \Omega$ 
2  Initialize best admissible solution and current time
    $\mathcal{S}^* \leftarrow \mathcal{S}$ 
    $Time \leftarrow 0$ 
3  Initialize tabu list and aspiration level
    $\mathbf{TL} \leftarrow \{\}$ 
    $\mathbf{AL} \leftarrow \text{Cost}(\mathcal{S}^*)$   $\triangleright$  Assuming  $AC$  is "best cost"
4  repeat
5      Generate neighborhood solutions of  $\mathcal{S}, \mathbb{N}(\mathcal{S})$ 
6      Select  $N$  neighbor solutions and add them to candidate list
        $\mathbf{V}^* \leftarrow \text{Sample}(\mathbb{N}(\mathcal{S}), N)$ 
7      Choose best solution in candidate list
        $\mathcal{S}^* \leftarrow \text{Best}(\mathbf{V}^*)$ 
8      if  $\text{Move}(\mathcal{S}, \mathcal{S}^*) \notin \mathbf{TL}$  or  $\text{Cost}(\mathcal{S}^*) < \mathbf{AL}$  then
9          Update tabu list
            $\mathbf{TL} \leftarrow \mathbf{TL}[1 : M] + \{\text{Move}(\mathcal{S}^*, \mathcal{S})\}$ 
10          $\mathcal{S} \leftarrow \mathcal{S}^*$   $\triangleright$  Accept the solution
11         if  $\text{Cost}(\mathcal{S}) < \mathbf{AL}$  then
12              $\mathbf{AL} \leftarrow \text{Cost}(\mathcal{S})$   $\triangleright$  Update aspiration level
13         end
14          $Time \leftarrow Time + 1$ 
15     end
16 until  $Time = MaxTime$ 
17 return  $\mathcal{S}^*$ 
```

than $\mathbf{AL}$, which is an indication that there is no cycling back to a previously visited solution, then the AC override the tabu status and accept the move. Additionally, the $\mathbf{AL}$ is updated to the cost of the new solution.

An algorithmic description of TS metaheuristic is shown in Algorithm 4. The procedure starts from a feasible solution, a neighborhood $\mathbb{N}(\mathcal{S})$ is defined for each solution $\mathcal{S}$. Among all the possible neighborhood solutions, N neighbor solutions are randomly selected and added to the candidate list using the procedure $\text{Sample}(\mathbb{N}(\mathcal{S}), N)$. Note that the number of possible neighborhood solutions is typically much larger than the

candidate list size, i.e., $|V^*| = N \ll |\aleph(\mathcal{S})|$. From these N neighborhood solutions in the candidate list, the best solution is chosen for consideration as the next solution $\mathcal{S}^*$. The selected solution could have a cost higher than the cost of the current solution resulting in hill-climbing.

Two perturbation schemes are used to generate neighborhood solutions; either (i) by randomly changing the assignment of a single *CONV* layer to CLPs, or (ii) by mutating T_n or T_m of a randomly selected CLP. For each of the perturbation schemes, a separate TL is maintained. The attributes stored in TLs are related to the perturbation scheme. In the case where a perturbation consists of changing the assignment of a *CONV* layer to CLPs, the *CONV* layer ID and the CLP ID to which it is assigned are stored. For the case of changing parameter T_n or T_m , the attributes saved are the CLP ID and the new value of the parameter. These two perturbations are done with the same probabilities as in the case of SA, i.e., 80% for changing T_n or T_m , and 20% for changing *CONV* layer assignment to CLPs.

3.5 Experiments and Results

In this section, we describe our experimental settings. Then, we employ the proposed optimization framework to design multiple CLPs, Multi-CLP in short, to accelerate four widely used typical CNNs on FPGAs. Next, we discuss the results and findings. Additionally, we compare the Multi-CLP accelerator designed using the presented metaheuristic-based framework with the state-of-the-art Single-CLP and Multi-CLP design frameworks.

3.5.1 Experimental Settings

To demonstrate the versatility of metaheuristics in designing Multi-CLP accelerators for CNNs, we conduct extensive experiments on the ImageNet large scale visual recognition challenge (ILSVRC) dataset [28], which is known as one of the most popular image classification benchmarks. The ILSVRC dataset contains 50 thousand images for validation, all of which are natural high-resolution images. Each image is annotated as one of 1,000 classes. To meet the configurations required for input data to CNNs, images are resized and then center cropped before being used as input.

For those experiments, we use our metaheuristic-based optimization technique discussed in Section 3.4 to design Multi-CLP accelerators for four benchmark architectures, namely, AlexNet [26], SqueezeNet 1.1 [180], VGG-16 [27], and GoogLeNet [29], as they are well-known in the field of image classification. The number of *CONV* layers in AlexNet, VGG-16, SqueezeNet 1.1, and GoogLeNet architectures is 10, 13, 26, and 57, respectively. To investigate the applicability of the proposed framework in designing Multi-CLP accelerators for CNNs with different bit-precision levels, we consider two formats; 32-bit floating-point (FP32) and 16-bit fixed-point (Fxp16).

The Multi-CLP design is implemented as a parameterized Verilog hardware description language (HDL). The design is synthesized and implemented using Xilinx Vivado Design Suite (v2020). Our application targets the VC707 and VC709 FPGA boards, both boards operate at a frequency of 100 MHz but each has different hardware resources. Table 3.2 summarizes the hardware specifications of these two boards in terms of FPGA platform, DSPs, 18 Kb BRAM (BRAM18K), LUTs, FFs, and

Table 3.2: FPGA boards hardware resources.

Specification	VC707 [216]	VC709 [217]
Platform	Virtex-7 VX485T	Virtex-7 VX690T
DSP Slices	2,800	3,600
BRAM18K	2,060	2,940
LUTs	303,600	433,200
FFs	607,200	866,400
External Memory	1GB DDR3 SODIMM	2 × 4GB DDR3 SODIMM

external memory.

The optimization problem described in Equation (3.20) employs the computational performance and resource utilization models discussed in Section 3.3.3 to find the highest-throughput Multi-CLP design. The maximum number of DSPs (DSP_{max}) and BRAMs (BRAM_{max}) available for the optimization process is set to 80% of the boards resources as in [93]. This is due to the need for some hardware resources to implement, for example, the soft-core processor and controllers. Accordingly, the DSP_{max} and BRAM_{max} are set to 2,240 and 1,648 for the VC707 board, respectively, while maximum of 2,880 DSP and 2,352 BRAM are allowed for the optimization procedure when working on the VC709 board.

With regards to the optimization procedure, the SA metaheuristic algorithm needs to start from a high temperature T , which is set in our experiments to 25,000. The parameter α for updating the temperature is set to the value 0.99. On the other hand, the constant β which controls the time spent at each annealing temperature is set to 1.005. The stopping criterion is set to 1,000 iterations. For TS metaheuristic algorithm, the size of the candidate list used is 20, as it has been found based on

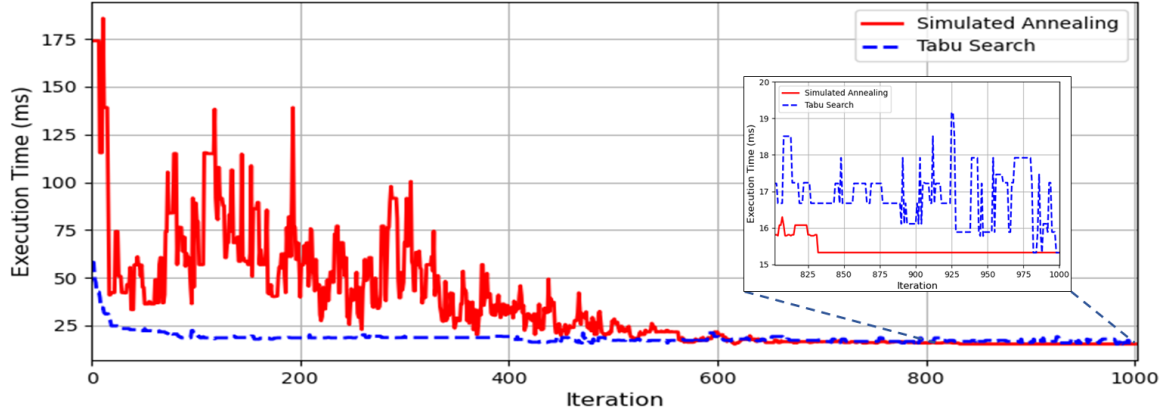


Figure 3.7: Cost function optimization progress for a run of 1,000 iterations for AlexNet on 485T FPGA using simulated annealing and tabu search metaheuristic algorithms.

experimental analysis that increasing this number does not make significant improvements while decreasing it has a negative effect on the quality of the obtained solutions. Based on the empirical analysis, the size of the tabu list chosen was 7, as this gave the best results, while the stopping criterion is set to 1,000 iterations. For each performed experiment, we have reported the best results obtained from 10 runs.

3.5.2 Results and Discussion

In the first experiment, we study the effectiveness of the proposed optimization framework in finding the optimal configurations that maximize the throughput of the Multi-CLP design. Hence, this experiment uses the computational performance and resource usage models as well as the maximum resources available on the FPGA to find the solution with the minimum execution cycles.

The behavior of a single run of SA algorithm for AlexNet CNN acceleration using multiple CLPs on 485T FPGA is shown in Figure 3.7. The figure shows the variation

in the execution time, which is defined as the total number of cycles divided by the operating frequency. The number of cycles is the cost function, illustrated in Equation (3.19), that SA algorithm tries to optimize for 1,000 iterations when running with the parameters mentioned in the previous section. We refer to this approach as SA-based Multi-CLP. As can be seen, since the algorithm starts with a high temperature, the fluctuation in costs is large, which gets reduced over time as the temperature is reduced and better solutions are discovered with minimal cost variations toward the end. Final solution configurations can be found in Table 3.3.

Similarly, the behavior of a single run of TS algorithm for the same CNN and the same FPGA board is also illustrated. This approach to designing a Multi-CLP based on TS algorithm is indicated by the TS-based Multi-CLP. As can be seen from Figure 3.7, the cost reduces very quickly in the first few iterations as there are several neighboring candidates to choose from in each iteration, and then, it reduces at a slower rate in subsequent iterations with hill-climbing behavior, until reaching the final solution shown in Table 3.3.

From the figure, we can deduce the effectiveness of these algorithms in finding the optimal solution to the Multi-CLP design problem. Both algorithms were able to efficiently explore the intractable design space and find Multi-CLP design solutions with approximately the same low level of execution time, but each has a different final solution. It is noteworthy that the optimization framework is very fast, it explores the search space in less than a minute on a general-purpose processor.

Next, we use the SA-based Multi-CLP and TS-based Multi-CLP approaches to accelerate AlexNet architecture on 485T FPGA. The obtained designs are then com-

Table 3.3: Single-CLP and Multi-CLP accelerators for AlexNet on 485T FPGA using FP32 data.

Technique	CLP No.	Unrolling Factors		Layer Mapping	Cycles	Execution Time (ms)	DSPs	BRAMs
		T_n	T_m					
Single-CLP [55]	CLP_1	7	64	$1a, 1b$	732×10^3	20.06	2,240	618
				$2a, 2b$	510×10^3			
				$3a, 3b$	338×10^3			
				$4a, 4b$	256×10^3			
				$5a, 5b$	170×10^3			
Multi-CLP [93]	CLP_1	3	24	$1a, 1b$	$1,464 \times 10^3$	15.58	2,240	731
	CLP_2	8	19	$2a, 2b$	$1,531 \times 10^3$			
	CLP_3	1	96	$3a, 3b$	$1,558 \times 10^3$			
	CLP_4	2	64	$4a, 4b$	876×10^3			
				$5a, 5b$	584×10^3			
SA-based Multi-CLP (ours)	CLP_1	3	24	$1a$	732×10^3	15.31	2,240	644
				$4a$	779×10^3			
	CLP_2	3	24	$1b$	732×10^3			
				$4b$	779×10^3			
	CLP_3	16	11	$2a, 2b$	$1,312 \times 10^3$			
				$5a$	219×10^3			
	CLP_4	16	8	$3a, 3b$	$1,168 \times 10^3$			
				$5b$	292×10^3			
TS-based Multi-CLP (ours)	CLP_1	5	24	$3a$	633×10^3	15.32	2,240	648
				$4a$	475×10^3			
				$5a$	356×10^3			
	CLP_2	3	12	$1a$	$1,464 \times 10^3$			
				$1b$	$1,464 \times 10^3$			
	CLP_3	3	12	$2a, 2b$	875×10^3			
				$3b$	292×10^3			
				$4b$	219×10^3			
				$5b$	146×10^3			

pared with the state-of-the-art Single-CLP [55] design and Multi-CLP [93] design, as shown in Table 3.3. The table shows the number of CLPs adopted by each technique and the parallelism factors for each CLP. Additionally, it shows the assignment of *CONV* layers to CLPs. The number of cycles each CLP spends in processing its layers is also provided. Note that when layer assignment column contains more than one layer in a row, the cycle count provided is the total cycles needed to process all of those layers.

The table also shows the execution time of an image’s *CONV* layer, which represents the time interval in which the system can receive a new image for processing as discussed in Section 3.3.2. The execution time is calculated by dividing design cycles count, illustrated in Equation (3.19), over design operating frequency. Specifically, the cycle count for each CLP is the number of cycles required to execute all of its assigned layers. On the other hand, CLPs in Multi-CLP design work concurrently. Therefore, the overall cycle count for such a design is the maximum cycle count for its CLPs. The last two columns in Table 3.3 present the number of DSP slices and BRAMs utilized for each technique.

As evidenced by the results, the number of cycles required in the case of a single CLP implementation is almost $1.3\times$ more than that of the multiple CLPs implementations. The results also show that SA-based Multi-CLP and TS-based Multi-CLP were able to find implementation configurations with a fewer number of cycles (less execution time) compared to Multi-CLP [93] even though all designs use the same number of CLPs and DSP slices. This demonstrates the effectiveness of the proposed SA-based Multi-CLP and TS-based Multi-CLP in balancing the computational re-

Table 3.4: Single-CLP and Multi-CLP accelerators for AlexNet on 690T FPGA using FP32 data.

Technique	CLP No.	Unrolling Factors		Layer Mapping	Cycles	Execution Time (ms)	DSPs	BRAMs
		T_n	T_m					
Single-CLP [55]	CLP_1	9	64	$1a, 1b$	732×10^3	17.69	2,880	758
				$2a, 2b$	437×10^3			
				$3a, 3b$	265×10^3			
				$4a, 4b$	201×10^3			
				$5a, 5b$	134×10^3			
Multi-CLP [93]	CLP_1	1	64	$5a, 5b$	$1,168 \times 10^3$	11.68	2,880	1,238
	CLP_2	1	96	$4a, 4b$	$1,168 \times 10^3$			
	CLP_3	2	64	$3a, 3b$	$1,168 \times 10^3$			
	CLP_4	1	48	$1a$	$1,098 \times 10^3$			
	CLP_5	1	48	$1b$	$1,098 \times 10^3$			
	CLP_6	3	64	$2a, 2b$	$1,166 \times 10^3$			
SA-based Multi-CLP (ours)	CLP_1	3	16	$1a$	$1,098 \times 10^3$	11.68	2,880	1,344
	CLP_2	3	16	$1b$	$1,098 \times 10^3$			
	CLP_3	12	8	$2a$	$1,166 \times 10^3$			
	CLP_4	6	16	$2b$	$1,166 \times 10^3$			
	CLP_5	16	16	$3a, 3b$	584×10^3			
				$4a, 4b$	438×10^3			
				$5a$	146×10^3			
	CLP_6	8	4	$5b$	$1,168 \times 10^3$			
TS-based Multi-CLP (ours)	CLP_1	6	32	$2a$	583×10^3	11.81	2,880	1,348
				$3b$	392×10^3			
				$5a$	195×10^3			
	CLP_2	3	48	$1a$	366×10^3			
				$3a$	523×10^3			
				$5b$	292×10^3			
	CLP_3	3	16	$1b$	$1,098 \times 10^3$			
	CLP_4	6	32	$2b$	583×10^3			
				$4a, 4b$	584×10^3			

Table 3.5: AlexNet accelerators performance on 485T and 690T FPGAs using FP32 data.

Technique	Virtex-7 VX485T				Virtex-7 VX690T			
	BW (GB/s)	Arith. Util.	Thr. (img/s)	Perf. (GFLOPs/s)	BW (GB/s)	Arith. Util.	Thr. (img/s)	Perf. (GFLOPs/s)
Single-CLP [55]	1.40	74.1%	48.85	65.05	1.78	64.0%	55.40	73.77
Multi-CLP [93]	1.38	95.6%	63.98	85.20	1.49	98.1%	85.55	113.92
SA-based Multi-CLP (ours)	1.42	97.4%	65.32	86.98	1.35	98.1%	85.55	113.92
TS-based Multi-CLP (ours)	1.33	95.7%	65.27	86.91	1.30	98.1%	84.67	112.75

sources and workloads assigned to each CLPs such that each CLP can be kept busy most of the time. Although the implementation configurations obtained using SA and TS metaheuristic algorithms show small improvements, the results indicate the importance of using metaheuristics in finding different configurations for a given CNN using a systematic methodology.

Table 3.4 shows similar experiments performed for AlexNet on 690T FPGA. The number of cycles required in the case of a single CLP implementation is about $1.5\times$ more than that of multiple CLPs implementations. For the implementation configuration obtained using SA-based Multi-CLP, we got the same number of cycles as obtained by Multi-CLP [93]. Although both techniques result in designs with same number of cycles and CLPs, these designs have completely different configurations for their adopted CLPs. On the other hand, implementation configuration obtained by TS-based Multi-CLP has used only 4 CLPs and results in a higher execution time.

In Table 3.5, we summarize the results presented in Tables 3.3 and 3.4 to get an insight into the required bandwidth, arithmetic unit utilization, *CONV* layers throughput, and performance. We can see that SA-/TS-based Multi-CLP provides $1.34\times$ and $1.54\times$ more throughput improvement than Single-CLP [55] design on 485T and 690T FPGAs, respectively. This improvement comes from its ability to make better use of the available arithmetic units. Specifically, Single-CLP [55] can provide a useful work to the multipliers and adders only 74% and 64% of the time on 485T and 690T FPGAs, respectively, whereas SA-/TS-based Multi-CLP improves the arithmetic units utilization to about 96% and 98% on 485T and 690T FPGAs, respectively. Note that the arithmetic utilization is computed as discussed in Equation (3.18). Furthermore, SA-/TS-based Multi-CLP improves *CONV* layer throughput of Multi-CLP [93] design by $1.02\times$ on 485T.

On the other hand, there is a trade-off between off-chip memory bandwidth and on-chip buffer size. Using large buffers saves off-chip memory bandwidth, whereas adopting small buffers results in a high bandwidth requirement. Here, we must emphasize that the proposed optimization framework gives a higher cost to bandwidth requirement than that given to buffer size. In other words, when more than one feasible design with the same low number of execution cycles is encountered, the design with minimum bandwidth requirement is chosen as optimal. We have adopted this strategy to minimize power expenses since on-chip memory modules consume less power than off-chip memory modules [218].

The results for single CLP and multiple CLPs accelerators for SqueezeNet 1.1 on 485T and 690T FPGAs with 16-bit fixed-point data are shown in Table 3.6.

Table 3.6: Single-CLP and Multi-CLP accelerators for SqueezeNet 1.1 on 485T and 690T FPGAs using FxP16 data.

Technique	Virtex-7 VX485T					Virtex-7 VX690T				
	CLP No.	Unrolling		Layer Mapping	Cycles	CLP No.	Unrolling		Layer Mapping	Cycles
		T_n	T_m				T_n	T_m		
Single-CLP [55]	CLP_1	32	68	1 – 26	349×10^3	CLP_1	32	87	1 – 26	331×10^3
Multi-CLP [93]	CLP_1	6	16	2, 3, 6, 5	179×10^3	CLP_1	8	16	2, 6, 3, 5	125×10^3
	CLP_2	3	64	1, 8, 9, 12	183×10^3	CLP_2	3	64	1	115×10^3
	CLP_3	4	64	<i>all others</i>	165×10^3	CLP_3	11	32	<i>all others</i>	133×10^3
	CLP_4	8	64	7, 4, 16, 19	176×10^3	CLP_4	8	64	7, 4, 16	145×10^3
	CLP_5	8	128	26, 22, 25, 13	185×10^3	CLP_5	5	256	19, 26, 22, 25	144×10^3
	CLP_6	16	10	10	183×10^3	CLP_6	16	26	13, 10	141×10^3
SA-based Multi-CLP (ours)	CLP_1	2	7	14	176×10^3	CLP_1	64	16	5, 23, 25, 26	139.5×10^3
	CLP_2	16	32	7, 13, 20, 25	179×10^3	CLP_2	3	64	1, 12	132×10^3
	CLP_3	8	16	2, 16, 17	180×10^3	CLP_3	32	8	11, 13	138×10^3
	CLP_4	13	19	5, 10	148×10^3	CLP_4	8	22	2, 16, 24	139×10^3
	CLP_5	43	13	26	181×10^3	CLP_5	8	32	3, 4, 9	138×10^3
	CLP_6	11	4	3, 8	176×10^3	CLP_6	8	32	10, 18, 20	139×10^3
	CLP_7	3	64	1, 6, 23, 15	177×10^3	CLP_7	8	43	19, 22	138×10^3
	CLP_8	16	13	4, 18, 24, 12	181×10^3	CLP_8	8	32	7, 6, 14	138×10^3
	CLP_9	8	26	9, 21, 22	172×10^3	CLP_9	7	8	8, 15	93×10^3
	CLP_{10}	16	8	11, 19	177×10^3	CLP_{10}	14	4	17, 21	129×10^3
TS-based Multi-CLP (ours)	CLP_1	8	26	3, 22, 6	179×10^3	CLP_1	3	66	21, 1	132×10^3
	CLP_2	8	22	7	170×10^3	CLP_2	12	4	5	138×10^3
	CLP_3	11	53	26, 9	182×10^3	CLP_3	16	32	6, 11, 7, 20, 25	141×10^3
	CLP_4	16	10	10	176×10^3	CLP_4	43	1	17	85×10^3
	CLP_5	3	66	1, 14, 21, 23	183×10^3	CLP_5	16	16	2, 4, 14	135×10^3
	CLP_6	16	22	4, 18, 25	175×10^3	CLP_6	16	2	3	100×10^3
	CLP_7	10	8	8, 11, 20	183×10^3	CLP_7	16	32	<i>all others</i>	139×10^3
	CLP_8	11	19	2, 13, 24	183×10^3	CLP_8	8	22	19, 12, 24	133×10^3
	CLP_9	6	32	15, 16, 19	179×10^3	CLP_9	3	41	16	141×10^3
	CLP_{10}	10	8	5, 12, 17	178×10^3	CLP_{10}	8	26	10	141×10^3
						CLP_{11}	47	16	8, 26	140.5×10^3

For SqueezeNet 1.1 accelerators on 485T FPGA, the single CLP implementation requires about $1.9\times$ more cycles than the multiple CLPs implementation. Even more importantly, though, the results show that as the number of *CONV* layers increases, the search space also increases, making it more challenging to obtain an optimal multiple CLPs design using conventional iterative algorithms. Specifically, SA-based Multi-CLP processes images in a 1.04 ms and a 1.43 ms shorter time than Multi-CLP [93] on 485T and 690T FPGAs, respectively. Additionally, although TS-based Multi-CLP produced a solution that requires more cycles than that of SA-based Multi-CLP, it processes images in a shorter time and with fewer DSP slices than Multi-CLP [93].

On the other hand, when comparing the performance of SA-based Multi-CLP in accelerating SqueezeNet 1.1 on VC707 and VC709 boards, we can notice that it used the available resources to the maximum extent to improve the network throughput. Specifically, with a $1.29\times$ increase in computational resources in VC709 compared to VC707, SA-based Multi-CLP increased SqueezeNet 1.1 throughput by $1.30\times$, which demonstrates the scalability of the proposed accelerator.

In Tables 3.7 and 3.8, we show the configurations of single CLP and multiple CLPs accelerators designed for VGG-16 and GoogLeNet on 690T FPGA with 16-bit fixed-point representation. Note that no results have been reported in Single-CLP [55] and Multi-CLP [93] accelerators for these two large CNNs. Therefore, with the sake of brevity, we only present the results for SA-based Single-/Multi-CLP as it turns out from previous experiments that SA algorithm provides more optimized solutions than TS algorithm in this specific problem.

Table 3.7: SA-based Single-/Multi-CLP accelerators for VGG-16 on 690T FPGA using FxP16 data.

Technique	CLP No.	Unrolling		Layer Mapping	Cycles
		T_n	T_m		
SA-based Single-CLP (ours)	CLP_1	44	65	1 – 13	$6,631 \times 10^3$
SA-based Multi-CLP (ours)	CLP_1	8	64	1, 8, 4	$5,955 \times 10^3$
	CLP_2	32	19	3, 7, 11	$5,503 \times 10^3$
	CLP_3	48	2	13	$4,976 \times 10^3$
	CLP_4	64	26	<i>all others</i>	$5,841 \times 10^3$

For VGG-16, SA-based Multi-CLP reduces the total cycles compared to SA-based Single-CLP by about 676×10^3 cycles. Here, we see a significant difference compared to the results from AlexNet, which is due to the variance in the dimensions between *CONV* layers in VGG-16, where, for example, the first *CONV* layer has $\langle N, M \rangle$ as $\langle 3, 64 \rangle$, and those for last *CONV* layer is $\langle 512, 512 \rangle$. Hence, using a single CLP results in a large DSP resource under-utilization during the processing of some layers. For GoogLeNet, it requires around $2\times$ more cycles for the single CLP implementation than that of multiple CLPs. For large CNNs, such as GoogLeNet which has 57 *CONV* layers, it becomes more challenging to find an optimized multiple CLPs configuration without using metaheuristics algorithms.

Additionally, we modeled the behavior of multiple CLPs design in Verilog and synthesized it using Xilinx Vivado Design Suite targeting the Xilinx Virtex-7 485T FPGA. We kept all synthesis properties to their default. Table 3.9 shows a summary of the implementation and performance results for the proposed SA-based Multi-CLP approach and compares them with those reported in Single-CLP [55], GA-based Single-CLP [73],

Table 3.8: SA-based Single-/Multi-CLP accelerators for GoogLeNet on 690T FPGA using FxP16.

Technique	CLP No.	Unrolling		Layer Mapping	Cycles
		T_n	T_m		
SA-based Single-CLP (ours)	CLP_1	45	64	1 – 57	$1,330 \times 10^3$
SA-based Multi-CLP (ours)	CLP_1	51	2	11, 23, 31, 43, 28	620×10^3
	CLP_2	1	21	32	470×10^3
	CLP_3	10	64	3, 19, 53	636×10^3
	CLP_4	78	1	17, 21, 40	571×10^3
	CLP_5	26	2	8, 15	564×10^3
	CLP_6	8	20	36, 16	594×10^3
	CLP_7	4	13	54	635×10^3
	CLP_8	2	40	10, 45	608×10^3
	CLP_9	18	3	37, 38, 51, 56, 57	634×10^3
	CLP_{10}	7	13	18, 34, 50	587×10^3
	CLP_{11}	8	13	30, 47	631×10^3
	CLP_{12}	9	32	12	635×10^3
	CLP_{13}	14	1	27	464×10^3
	CLP_{14}	3	66	1	637×10^3
	CLP_{15}	1	11	49	122×10^3
	CLP_{16}	6	11	13, 39, 44	555×10^3
	CLP_{17}	2	38	35, 26, 48	636×10^3
	CLP_{18}	6	27	25, 42, 55	602×10^3
	CLP_{19}	12	20	9, 14, 24, 46	575×10^3
	CLP_{20}	4	6	5	602×10^3
	CLP_{21}	9	8	29, 33, 7, 52	521×10^3
	CLP_{22}	98	1	2, 20, 41	631×10^3
	CLP_{23}	99	2	4, 6, 22	596×10^3

and Multi-CLP [93] approaches when they are used to accelerate AlexNet architecture. For each technique, the table provides the FPGA platform used for implementation, operating frequency, precision adopted to represent FMs and weights, design entry employed to describe the accelerator, hardware resource usage after placement and routing in terms of DSPs, BRAMs, FFs, and LUTs, and power efficiency (a.k.a., performance per Watt).

When comparing resource usage estimates with resource usage from Vivados implementation report for SA-based Multi-CLP, one can note that the model underestimated the DSP slices count by 53 DSP slices per CLP, on average. The reason is that Vivados utilization report for SA-based Multi-CLP design accounts not only DSPs used in CLPs computational engine, but also those used in the control logic, address calculations, and loop indexing as well. When considering only those DSPs used to implement the computational engines for SA-based Multi-CLP, we found that the estimates match those reported by Vivado utilization report.

On the other hand, the implemented and predicted BRAM counts are perfectly matched. This is because in the implemented parametrized Verilog modules, we instantiate the exact required memory type, mode, and count using Xilinx HDL templates. However, this does not apply to Single-CLP [55], GA-based Single-CLP [73], and Multi-CLP [93] because they use high-level synthesis tools to compile C, OpenCL, and C++ codes into HDL codes, respectively. As can be seen from the results, designing with multiple CLPs makes better use of available resources than a single CLP design. Due to the dimensional mismatch issue in Single-CLP [55] design, this design scheme could not take advantage of more than 82% of DSP slices. One can also note

Table 3.9: Implementation and performance summary of Single-CLP and Multi-CLP accelerators optimized for AlexNet.

Technique	Platform	Frequency (MHz)	Precision	Design Entry	Resources Utilization				Power Efficiency (GOPs/Watt)
					DSPs	BRAMs	FFs	LUTs	
Single-CLP [55]	Virtex-7 VX485T	100	32-bit floating-point	C	2,309 (82%)	689 (34%)	219,815 (36%)	146,325 (48%)	3.31
GA-based Single-CLP [73]	Stratix-V GSD8	120	(8/16)-bit fixed-point	OpenCL	727 (37%)	1,480 (58%)	N/A	119,622 (46%)	3.79
Multi-CLP [93]	Virtex-7 VX485T	100	32-bit floating-point	C++	2,443 (87%)	812 (39%)	270,991 (45%)	176,876 (58%)	11.21
SA-based Multi-CLP (ours)	Virtex-7 VX485T	100	32-bit floating-point	Verilog	2,452 (88%)	644 (31%)	269,053 (44%)	176,449 (58%)	12.08

that the proposed SA-based Multi-CLP increases the usage of FF and LUT resources in Single-CLP [55] by about 5% for each additional CLP. All this increase in resource usage contributes to improving the power efficiency by $3.65\times$.

Compared to GA-based Single-CLP [73], the proposed SA-based Multi-CLP has effectively made use of the available resources resulting in a $3.19\times$ increase in power efficiency even though GA-based Single-CLP [73] uses 8-bit fixed-point weights and 16-bit fixed-point FMs while our accelerator uses 32-bit floating-point data and operates at a lower frequency. This is due to the high-level synthesis tool employed in [73] which constrains T_{in} to be from the set $\{1, 2, 4, 8, 16\}$ and T_{out} to be integer multiplicative of T_{in} , where T_{out} and T_{in} are the unified unrolling factors for the output matrix of *CONV* operation and the input vectors to CLP structures, respectively, which are determined by GA to minimize execution time. In contrast, this work allows unrolling factors to be whatever value yields the highest performance.

On the other hand, the usage of DSP, FF, and LUT resources for Multi-CLP [93]

and the proposed SA-based Multi-CLP is almost consistent. However, the additional 168 BRAMs used in Multi-CLP [93] led to a 0.4 Watt increase in its power consumption, which also caused its power efficiency to be about 0.87 GOPs/Watt lower than that of the proposed SA-based Multi-CLP.

3.6 Conclusion

CNNs have shown great performance in a wide range of machine learning applications. However, they require significant computational power that cannot be met by general purpose processors. Nowadays, FPGAs have been found to deliver excellent performance in accelerating CNNs as they provide the best performance per Watt. Most of the existing FPGA-based accelerators use a single CLP for processing all *CONV* layers. However, the current trend in CNN acceleration is using multiple CLPs to make efficient use of hardware resources.

The multiple CLPs design involves optimizing the number of CLPs used, assignment of *CONV* layers to CLPs, the parallelization factors $\langle T_n, T_m \rangle$ for each CLP, and the tiling parameters $\langle T_r, T_c \rangle$ for each *CONV* layer. This imposes an intractable design space where it is almost impossible to find near-optimal configurations through exhaustive search algorithms. Therefore, in this work, we employed SA and TS algorithms to design CNN accelerators using multiple CLPs on FPGAs targeting the optimization of the number of execution cycles.

Experimental results on accelerating AlexNet, VGG-16, SqueezeNet 1.1, and GoogLeNet architectures demonstrated the effectiveness of the proposed optimization framework

in providing efficient accelerators. The results have also shown the importance of using metaheuristics in finding different configurations for a given CNN using a systematic methodology. Compared to a single CLP design, SA-based Multi-CLP and TS-based Multi-CLP accelerate CNN computations by about $1.31\times - 2.37\times$ when targeting Xilinx Virtex-7 FPGAs. Our implementation achieves a performance of 113.92 GFLOPs under 100 MHz working frequency.

In future work, we consider improving the quality of resource usage model in predicting DSP slices required by taking into account those used in control logic, loop indexing, and address calculation. Additionally, we target the adoption of data quantization technique for further acceleration. Lowering the bit-precision level of activations and weights in turn allows multipliers and accumulators to be built not only using DSPs, but also using slice logic, improving CNN throughput even further.

CHAPTER 4

POST-TRAINING QUANTIZATION FOR THE DESIGN OF MIXED LOW-PRECISION DNNs WITH DYNAMIC FIXED-POINT REPRESENTATION

Deep learning neural networks (DNNs) have demonstrated their effectiveness in a wide range of computer vision tasks, with the state-of-the-art results obtained through complex and deep structures that require intensive computation and memory. In the past, graphic processing units (GPUs) enabled these breakthroughs because of their greater

computational speed. Now-a-days, efficient model inference is crucial for consumer applications on resource-constrained platforms. As a result, there is much interest in the research and development of dedicated deep learning (DL) hardware to improve the throughput and energy efficiency of DNNs. Low-precision representation of DNN data-structures through quantization would bring great benefits to specialized DL hardware especially when expensive floating-point operations can be avoided and replaced by more efficient fixed-point operations. However, the rigorous quantization leads to a severe accuracy drop. As such, quantization opens a large hyper-parameter space at bit-precision levels, the exploration of which is a major challenge. In this chapter, we propose a novel framework referred to as the fixed-point quantizer of deep neural networks (*FxP-QNet*) that flexibly designs a mixed low-precision DNN for integer-arithmetic-only deployment. Specifically, the *FxP-QNet* gradually adapts the quantization level for each data-structure of each layer based on the trade-off between the network accuracy and the low-precision requirements. Additionally, it employs post-training self-distillation and network prediction error statistics to optimize the quantization of floating-point values into fixed-point numbers. Examining *FxP-QNet* on state-of-the-art architectures and the benchmark ImageNet dataset, we empirically demonstrate the effectiveness of *FxP-QNet* in achieving the accuracy-compression trade-off without the need for training. The results show that *FxP-QNet*-quantized AlexNet, VGG-16, and ResNet-18 reduce the overall memory requirements of their full-precision counterparts by $7.16\times$, $10.36\times$, and $6.44\times$ with less than 0.95%, 0.95%, and 1.99% accuracy drop, respectively.

4.1 Introduction

DNNs have achieved state-of-the-art results on computer vision tasks, such as image classification [26], [27], [29], [30], [48], [219], [220], object detection [43], [221], face recognition [32], and semantic segmentation [222]–[224]. Examples of recent fields of DNN applications include robot navigation, industrial automation, automated driving, aerospace and defense [3], [42], [44]. These applications heavily depend on the complex DNN structures as well as on the deep and complicated models that over-fit the distribution of numerous training data. However, this also leads to over-parameterization resulting in a dramatic increase in memory and computational requirements for deployment.

A typical DNN often takes hundreds of mega byte (MB) of memory space. For instance, 45/170 MB for ResNet-18/101 [48], 240 MB for AlexNet [26], and 550 MB for VGG-16 [27], and requires billions of floating-point operations (FLOPs) per image during inference that rely on powerful GPUs, e.g., 0.72 GFLOPs for AlexNet, 1.8 GFLOPs for ResNet-18, and 15.3 GFLOPs for VGG-16. This not only limits the real-time applications of pre-trained DNNs but also challenges the ubiquitous deployment of high-performance DNNs on embedded wearables and Internet of things (IoT) devices such as smartphones and drones. Additionally, with the drive to put DNNs on low-power devices (unlike GPUs) now-a-days, developers and researchers often find themselves constrained by computational capabilities, such as the case when high-precision multiply-and-accumulate (MAC) instructions are not supported for hardware efficiency purposes [225]. Thus, network compression and acceleration are important

issues in DL research and application.

A promising approach to overcome the aforementioned challenges is to reduce the bit-precision level of DNN data-structures by quantization [102]. With the network topology unchanged, quantization has the potential to reduce the model size to only a fraction of the original by utilizing low-precision representation of parameters. An example of quantizing the parameters is to represent them with a lightweight floating-point format that uses a configurable number of bits for the exponent and mantissa segments [226]. Furthermore, the weights and activations could also be quantized to discrete values in the fixed-point representation. Then, the network inference is significantly accelerated by increasing the arithmetic intensity, that is, the ratio of operations performed to each memory byte accessed. Additionally, converting expensive floating-point arithmetic to more effective fixed-point and bit-wise operations gives an additional boost by lowering the inference latency [227].

Besides its benefits on memory and computation, quantization also offers large power savings [64], [86], [228], [229]. Reducing the wordlength of neurons also benefits DNN hardware accelerators such as application specific integrated circuits (ASICs) and field programmable gate arrays (FPGAs) [91], because it means significantly lessening the resources required for calculations such as area and power requirements for multipliers. Table 4.1 demonstrates the improvement in power consumption and area requirement when using 8-bit fixed-point multiply-accumulate units compared to their fixed-/floating-point counterparts in 45nm technology (0.9V) [228]. The improvement in computation, power, and hardware efficiency through network quantization makes the deployment of DNNs on dedicated embedded systems feasible now-a-days. Thus,

Table 4.1: Improvement in power consumption and area requirement when using 8-bit fixed-point multiply-accumulate operations compared to their fixed-point and floating-point counterparts in 45nm technology (0.9V) [228].

Bit-Precision	Multiply		Accumulate	
	Power	Area	Power	Area
32-bit fixed-point	15.5×	12.4×	3.3×	3.8×
16-bit floating-point	5.5×	5.8×	13.3×	37.8×
32-bit floating-point	18.5×	27.5×	30×	116×

quantization has received considerable attention and has become an active research area [230].

However, an extremely low-precision representation of data-structures leads to a severe information loss, referred to as a quantization error. The substantial quantization error, in turn, leads to a significant degradation in accuracy. For instance, the binary neural networks (BNNs) [165], [168], [231]–[235] adopt a rigorous quantization method to reduce the wordlength of activations and weights to only 1 bit. In a similar way, the ternary weight networks (TWNs) [236]–[239] use aggressively quantized weights to benefit from the 16× smaller model size. Accordingly, the BNNs and TWNs perform computations using bit-wise operations, but this comes at the cost of sacrificing model’s accuracy. Even though several techniques have been proposed to alleviate BNNs/TWNs performance drop, including, but not limited to, optimizing the quantization scheme and the training algorithms, there is still a non-negligible accuracy gap between BNNs/TWNs and their full-precision counterparts. Therefore, researchers moved on to design k -bit DNNs, $k > 2$, with the intent of optimizing not only network accuracy but also reducing computational and memory costs [167].

In pursuit of maintaining the overall accuracy for the quantized DNNs, the majority of literature incorporates the quantization with the training process. In this way, the optimization algorithm used to train DNNs, e.g., stochastic gradient descent (SGD), will learn the network parameters that yield the best performance, and thus the accuracy degradation due to quantization is minimized. Hence, this approach is referred to as quantization-aware training. However, training requires a full-size dataset, which is often unavailable in real-world scenarios for reasons such as proprietary and privacy, especially when working on an off-the-shelf pre-trained model from community or industry. Training also requires tight integration of network design, training algorithms, initialization, and hyper-parameters configuration, all of which are not always feasible.

Therefore, many efforts have been made to quantize a pre-trained model using sophisticated methods with the intent of minimizing the perturbation in the distribution of quantized data-structures. Hence, this quantization approach is denoted by post-training quantization. Quantization methods utilizing this approach are easy to use and allow DNNs to be quantized even with limited data. Further, quantized DNNs based on a post-training quantization scheme are the fastest ones to be implemented and deployed. This is because they avoid retraining, a time-consuming process [101], which is usually employed repeatedly in quantization-aware training.

Nevertheless, achieving near state-of-the-art accuracy on the large-scale image classification tasks such as ImageNet [28] with low-precision data-structures and without retraining remains a challenge. Generally, 8-bit is considered to be a limit for post-training quantization of DNNs [240]. This gap has led to extensive research efforts by

several tech giants to improve post-training quantization, such as, but not limited to, NVIDIA [241], Samsung [242], IBM [243], Huawei [244], Hailo Technologies [245], and Intel [246].

Most of the work to date homogeneously quantizes all layers, i.e., the same number of bits are allocated to all the layers. Yet, it has been asserted that DNN layers have a different structure and therefore different properties related to the quantization process [247]. Furthermore, DNN layers have a different arithmetic intensity on hardware. Spurred by that, very recently, there have been a few approaches targeting the use of mixed-precision for representing DNN data-structures to design more efficient quantized DNNs [248]–[251]. This flexibility was originally not supported by chip vendors until recently when some advanced chips were released, including Apple A12 Bionic chip, Nvidia’s Turing GPU architecture, and Imagination neural network accelerator IP, all of these support mixed-precision arithmetic [252]. Besides industry, academia also is working on bit-level flexible hardware design to accelerating DNNs. For instance, BitFusion [253] supports spatial multiplications of 2, 4, 8, and 16 bits, while BISMO [254], on the other hand, adopts a bit-serial multiplier to support scalable precision.

The problem now is, how to determine the wordlength of the data-structures for each layer so that network performance is preserved while minimizing computational and memory costs? Here, it must be emphasized that there is no systematic way to determine the quantization level of different data-structures. In [248], [255], deciding the appropriate bit-precision level is manual and laborious so as to maintain accuracy. Thereafter, automated algorithms are designed that can discover the appro-

appropriate quantization level for each data-structure with accuracy in mind [256], [257].

However, most of these methods rely on performing expensive MAC operations with low-precision integer arithmetic, and then multiplying the outputs with floating-point scalar(s) to improve accuracy by recovering model’s dynamic range. The problem with all of these methods is extra memory and computational requirements for floating-point scaling. Thus, FPGA-/ASIC-based hardware accelerators of DNNs still need to implement expensive floating-point multiplication and accumulation operations to handle activation update. For its significant benefits to device designers and application developers, the flexible design of efficient DNNs with mixed-precision for integer-only deployment has become an interesting research area.

Hence, in this chapter, we review the current status of using quantization for the design of low-precision DNNs. We highlight the implementation details of existing techniques and state their drawbacks. Then, a new automated framework for fixed-point quantization of DNN data-structures referred to as the *FxP-QNet* is proposed. The *FxP-QNet* flexibly designs a mixed low-precision DNN for integer-arithmetic-only deployment from a pre-trained model. The contributions of this chapter can be summarized as below:

1. We introduce *FxP-QNet* to design a mixed low-precision DNN based on the trade-off between network accuracy and low-precision requirements. The *FxP-QNet* employs a two-level clustering approach and the data-structures sensitivity to quantization to gradually reduce the wordlength of each data-structure.
2. A detailed description of a post-training quantization method to represent both

model parameters and activations as dynamic fixed-point numbers to implement highly resource-/energy-efficient FPGA-/ASIC-based hardware accelerators of DNNs.

3. We present a metric to quantify the effect of discretization on DNNs using the post-training self-distillation and network prediction error statistics and utilize it to optimize the quantization of floating-point values.
4. Three typical classification DNNs, including AlexNet, VGG, and ResNet-18 are designed with mixed low-precision data-structures for integer-only deployment based on the proposed *FxP-QNet* framework.
5. Examining the quantized architectures on the benchmark ImageNet dataset, we show that *FxP-QNet*-quantized AlexNet, VGG-16, and ResNet-18 requires 85.99%, 91.62% and 84.96% less memory space for model parameters with less than 0.95%, 0.95%, and 1.99% accuracy drop compared to their full-precision counterparts, respectively.

The rest of the chapter is organized as follows. Section 4.2 provides background information about the dynamic fixed-point representation and highlights its terminology. It also discusses the principally used quantization approaches and the benefits of each on hardware systems. Section 4.3 presents a literature review of the existing DNNs quantization frameworks. In Section 4.4, we discuss the proposed *FxP-QNet* framework and describe its main functional components together with their implementation details. Section 4.5 demonstrates the effectiveness of *FxP-QNet* via extensive

experiments on several typical DNNs architectures. Finally, we conclude and describe the future directions of this chapter in Section 4.6.

4.2 Background and Terminology

This section provides an overview of the basics and terminologies of dynamic fixed-point representation. Additionally, it illustrates the quantization approaches used to map full-precision values to low-precision ones and highlights how hardware systems can benefit from these approaches.

4.2.1 Dynamic Fixed-Point Representation

In fixed-point representation, a real number x is represented as an integer number $\mathbf{x}$ with k bits. An implied binary point is used to configure the integer length α and the fractional length β of the binary word. Here, k is the wordlength, α and $\beta \in \mathbb{Z}$, and $k = \alpha + \beta$. Generally, fixed-point representation is applied on a group basis. In other words, a group of n numbers, each of which with k bits, share a common fixed scaling factor equal to $2^{-\beta}$. That is why it is called *fixed-point*. Thus, each integer number $\mathbf{X}_i$ in the group is interpreted to a rational number $\tilde{X}_i$ expressible as

$$\tilde{X}_i = \frac{\mathbf{X}_i}{2^\beta} = \mathbf{X}_i \gg \beta, \quad \forall i \in [n] \quad (4.1)$$

That is, the value represented $\tilde{X}_i$ is $\mathbf{X}_i$ right-shifted β bit positions. Consequently, the fixed-point representation allows using fractional numbers on low-cost integer

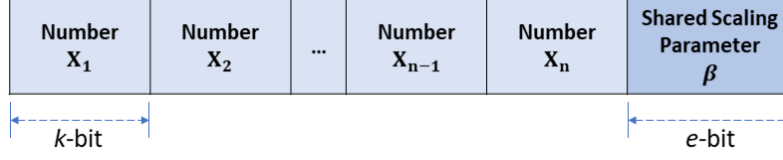


Figure 4.1: Generalized format for dynamic fixed-point numbers.

hardware. Regarding the required memory space, it needs $(n \times k)$ bits for the internal representation of the numbers plus the memory space needed to store the scaling parameter, i.e., e bits. Note that when the group size is large, the associated cost and storage overhead for β is marginal because it is shared across many numbers.

If β has the flexibility to change depending on the real numbers to be represented, the representation is named *dynamic fixed-point* [258]. In this chapter, we adopt the dynamic fixed-point format, depicted in Figure 4.1, to represent the data-structures of DNNs. Therefore, we use a configurable β to increase the flexibility as in the floating-point format but at a lower associated hardware cost because it is shared by a group of numbers. Specifically, given n numbers to be represented, i.e., the weights or the activations of a layer, β is tuned with the aim of minimizing a predefined cost function as will be illustrated in Section 4.4.3.

More precisely, we employ the unsigned dynamic fixed-point format, referred to as $U(\alpha, \beta)$, to represent DNN data-structures that consist of only positive rationals. In contrast, the signed two’s complement dynamic fixed-point format, denoted as $S(\alpha, \beta)$, is used for representing signed DNN data-structures. Here, we must emphasize that α includes the sign bit in the $S(\alpha, \beta)$ format. It is worth noting that the value of β is not limited to the range $[0, k]$ in dynamic fixed-point representation. This is very crucial for representing a group of large magnitude real numbers, or on the other hand, a

group of small proper fractions ($\pm$) with just a few bits. For a better representation in such cases, the representation may end up with a negative configuration for α or β .

For a given dynamic fixed-point format, the *dynamic range* denotes the span of numbers that the format type can represent. Accordingly, a $U(\alpha, \beta)$ format has a dynamic range of $[0, 2^\alpha - 2^{-\beta}]$, whereas the dynamic range of $S(\alpha, \beta)$ format is $[-2^{\alpha-1}, 2^{\alpha-1} - 2^{-\beta}]$. Furthermore, the *resolution* indicates the difference between two consecutive values in the dynamic range, which is equal to $2^{-\beta}$. For example, a $U(3, -1)$ format has a wordlength of 2 bits for representing the integer numbers $\mathbf{X}_i$, a resolution of 2, and $\tilde{X}_i \in \{0, 2, 4, 6\}$. On the other hand, when $\mathbf{X}$ is in $S(-1, 3)$ format, then, $\mathbf{X}_i$ has a wordlength of 2 bits, a resolution of 0.125, and $\tilde{X}_i \in \{-0.25, -0.125, 0, 0.125\}$.

4.2.2 Quantization Approaches

In this section, we briefly discuss the approaches used to map 32-bit floating-point (FP32) values to lower bit-precision and the benefits gained from each. Practitioners mainly use three types of quantization, namely, codebook-based, uniform, and non-uniform quantization approaches. Notwithstanding their differences, all methods aim at having compressed DNN data-structures to improve the memory efficiency and speed up the inference of neural networks.

In codebook-based quantization, a FP32 value x is mapped to a k -bit codebook index i using a pre-created codebook $C = \{(i, v_i) : \forall i \in [0, 2^k), v_i \in \mathbb{R}\}$ such that v_i is the closest value to x [101], [259]. The idea behind this quantization method is to find a smallest possible optimal codebook for each data-structure. To perform a

computation on the quantized data-structure, the codebook is used to retrieve the corresponding FP32 values. Accordingly, the computations in such a quantized DNN are floating-point arithmetic. However, computational and power efficiency increases through low bandwidth overhead and reduced memory requirements. To reduce the hardware cost, a simplified version uses integer codebook values with a single FP32 scalar [260].

On the other hand, the uniform quantization adopts an equal step size between the consecutive values. The linear [167] and fixed-point [242] quantization methods are examples of uniform quantization. These methods map a FP32 value x to a k -bit integer value $\mathbf{x}$ using a scalar ψ such that $\mathbf{x} = \lfloor x/\psi \rfloor$. Here, x/ψ is rounded to the nearest integer in a representation domain $C \subseteq \{0, \pm 1, \dots, \pm 2^k\}$, where C is defined by the format type and the wordlength. Basically, the scalar of the linear quantization method depends on the considered range for the FP32 values. For instance, if the values are clipped/transformed into the range $[-\mu, \mu]$, then the scalar can be defined as $\mu/(2^{k-1} - 1)$, where μ denotes the clipping/transformation parameter. On the other hand, the scalar can be equal to $\mu/(2^k - 1)$ when the considered range is $[0, \mu]$. Unlike linear quantization where the scalar is a FP32 value, the scalar in fixed-point quantization is a rational number, i.e., $\psi = 1/2^\beta$, $\beta \in \mathbb{Z}$. Thus, the idea behind the uniform quantization method is to find the optimal representation domain and scalar.

Accordingly, to perform the MAC operation between the FP32 vectors X and Y in a lower bit-precision, the input vectors are uniformly quantized based on their predefined scalars and representation domains. Here, we refer to the scalars of X and Y as ψ_x and ψ_y , respectively. Thereafter, the MAC operation is performed on the

lower bit-precision representations $\mathbf{X}$ and $\mathbf{Y}$ with integer arithmetic. Then, the result is rescaled to produce the output value z , such that

$$z = (\psi_x \cdot \psi_y) \cdot \text{MAC}(\mathbf{X}, \mathbf{Y}) \quad (4.2)$$

It should be noted that quantization and rescaling are done using floating-point arithmetic in linear quantization, whereas they are just simply bit-shift operations in fixed-point quantization. Finally, the non-uniform quantization adopts unequal step size between the consecutive values. Specifically, it quantizes the FP32 values in their logarithmic representation [261]. Thus, non-uniform quantization method eliminates the need for digital multipliers. However, this comes at the cost of being more complicated during the quantization process and sacrificing model’s accuracy.

4.3 Related Work

Inspired by the effectiveness and efficiency of low-precision operations, there have been a good number of studies on designing k -bit DNNs. These efforts can roughly fall into two categories, quantization-aware training and post-training quantization. In the former approach, quantization is performed during network training so as to compensate for the effect of quantization. The latter approach, on the other hand, takes a pre-trained model and applies a static quantization formula with the intent of minimizing quantization error and maintaining network accuracy. Due to the emerging use of different bit-precision to represent DNN data-structures in different layers, a

subsection is also devoted to mixed-precision quantization.

4.3.1 Quantization-Aware Training

In DoReFa-Net [167], activations are clipped with a bounded activation function and weights are transformed by the tanh function before being linearly quantized in the range $[0, 1]$. A much simpler weight quantization scheme is used in WRPN [262] through clipping the weights to the range $[-1, 1]$. On the other hand, PACT [263] indicates that homogeneous clipping of all activations to 1 is not effective for rigorous quantization. Instead, PACT learns the optimal clipping value for each layer during network training. Later in [264], the idea of parametrized clipping extended to weights as well.

HWGQ-Net [265] and TSQ [266] obtain the scalar for quantizing activations via Lloyd’s algorithm [267] on samples from the Gaussian distribution. Additionally, TSQ prunes activations below a handcrafted threshold to 0. Then, it learns the quantized weights along with their kernel-wise scalars that minimize the mean square error (MSE) between full-precision (FP32) and quantized activations. QIL [268], on the other hand, learns both the clipping interval and the transformation parameter for weights and activations of each layer so as to quantize them linearly in the range $[0, 1]$.

LogQuant [261] quantizes the FP32 values in their logarithmic representation based on manually fine-tuned layer-wise parameters, which is not a trivial process. Similarly, LPBN [269] quantizes activations in their log-space during the batch normalization (*BatchNorm*) process using a manually defined formula for each wordlength. To simplify the log-based quantization, Hubara et al. [233] quantizes the FP32 values to the

index of the most significant bit in their binary representation. In WEQ [259], the authors automate the search for LogQuant parameters that maximize the weighted entropy of the quantized activations during training. On the other hand, they use an iterative search algorithm to find the codebook for the weights of each layer with the highest overall weighted entropy.

BalancedQ [270] enforces frequency balance for the FP32 values mapped to each quantization bin. INQ [271] quantizes the weights of a pre-trained model incrementally. On each iteration, a portion of the full-precision weights are quantized to fixed-point numbers, while the others are fine-tuned to compensate for the drop in accuracy. Apprentice [255] employs the knowledge distillation [272] to improve the accuracy of the quantized network through the teacher network, which usually has a deeper architecture and a larger number of parameters and is trained in full-precision.

SYQ [260] quantizes each weight’s kernel to binary or ternary values using s_k^2 codebooks, each of which with a learned FP32 scalar, where s_k is the kernel size. With regards to activations, it uses 8-bit fixed-point numbers where the fractional length is adjusted while training the network. In LQ-Nets [273], the authors replace the standard basis of a k -bit quantized value with a basis vector that consists of k floating-point scalars. During the training phase, LQ-Nets learns the layer-wise basis vector for k_a -bit activations and the channel-wise basis vector for k_w -bit weights that minimizes the quantization error. Thus, unlike most of the techniques discussed where floating-point arithmetic is used once per MAC operation, SYQ and LQ-Nets use s_k^2 and $(k_a \times k_w)$ floating-point multiplication per MAC operation, respectively.

ALT [274], LSQ [275], and PROFIT [276] defined optimized gradient formulae.

Accordingly, they learn low-precision parameters and quantization scalar of each layer when quantizing FP32 models to desired precision. While PROFIT and IBM’s LSQ use FP32 quantization scalars, ALT restricts them to power-of-2. Furthermore, PROFIT proposed a training method in which sensitive weights, that cause activation instability, are progressively freezed.

4.3.2 Post-Training Quantization

In TWN [236], the authors use a layer-wise non-negative threshold (Δ) and a scalar (α) to design ternary weight networks. More precisely, the TWN maps weights greater than Δ into α and weights less than Δ into $-\alpha$, while other wights are mapped into 0. To increase TWN model capacity, FGQ [277] partitions FP32 weights into groups along the input channels, and then ternarizes each group independently as in TWN but using separate thresholds for positive and negative values. The brute-force method is used in both approaches to find the quantization thresholds and scalars.

Nvidia’s TensorRT [241] adopts the linear quantization by using a clipping parameter that minimizes the Kullback-Leibler (KL) divergence between FP32 and quantized values. In [242], Samsung proposed a channel-wise quantizer for 8-bit fixed-point DNNs. The fractional length is estimated from the n -th moments of the channel-level distribution. In [243], IBM introduced a new format, referred to as FPEC, for representing DNN data-structures. The FPEC includes two or more bits, denoted as compensation bits, which are used to estimate and compensate for the error due to fixed-point convolution. An accuracy-driven algorithm is proposed to reduce FPEC wordlength and adjust compensation bits.

In [244], Huawei indicated that using linear quantization for data-structures with a high MSE may not be optimal. Therefore, a handcrafted threshold (τ) is employed so that data-structures with an MSE larger than τ are quantized by codebook-based quantization, while others are quantized by linear quantization with the maximum absolute value as a clipping parameter. Then, quantization scalars of each layer are refined to minimize the MSE of the resulting activations of that layer. Meller et al. [245] at Hailo Technologies Ltd employ the weight factorization to make DNNs more robust to layer-wise post-training quantization. More precisely, for each weight layer, the output channels are scaled to match their dominant channel, that is, the channel with the largest absolute value, and thus equalizing channels.

KDE-KM [278] creates a quantization codebook for the weights of each layer from the center values of the clusters identified through the k -means clustering algorithm. Huang et al. [279] proposed a mixed quantization framework, denoted as MXQN, for quantizing activations either to fixed-point numbers or quantizing them in their log-space based on the signal-to-quantization-noise ratio. On the other hand, the MXQN quantizes weights to fixed-point values due to their need for fine-grained resolution.

To mitigate the significant accuracy drop due to base-2 logarithmic quantization of weights, SegLog [280] quantizes large magnitude weights in their base- $\sqrt{2}$ log-space. Specifically, SegLog creates a codebook with N base-2 codes and $(2^{(k-1)} - N)$ base- $\sqrt{2}$ codes for each layer, where N is manually tuned to achieve the best accuracy. Additionally, SegLog approximates $\sqrt{2}$ to an expression of power-of-2 integer values. Thus, it implements the multiplication between full-precision activations and quantized weights using shifters and adders.

4.3.3 Mixed-Precision Quantization

DNAS [256], proposed by Facebook and Berkeley AI Research, represents the network as a computational directed acyclic graph (DAG). The nodes of the DAG represent activations while edges represent a parameterized convolution operation on the input node and its weights. Cascading nodes are connected using several edges each of which with a manually configured wordlengths. The SGD is used to select the edges, based on system requirements, as well as to optimize the weights of these edges. Note that DNAS follows PACT approach to quantize weights and activations.

In ACIQ [246], Intel proposed to allocate each data-structure channel with a different wordlength, whereby the average data-structure wordlength is maintained. Thereafter, with the use of a channel-wise clipping parameter, the range is divided into equal sized regions, and the FP32 values within each region are mapped to the midpoint. The optimal wordlength allocation and clipping thresholds are determined analytically to minimize the channel-wise MSE.

ADMM [281] performs an iterative optimization to prune and quantize the weights in a layer-wise manner. The bit-precision of each layer is determined manually and the quantization scalars are defined so as to minimize the overall MSE. On each iteration, after pruning the weights, the ADMM quantizes a portion of FP32 weights that are closest to their discrete bins, and remaining weights are fine-tuned.

ReLeQ [282] and HAQ [283] adopt reinforcement learning to learn the wordlength of each data-structure in a layer-wise manner. Specifically, the ReLeQ uses the predicted bit-precision level to quantize the weights as in WRPN technique, whereas the

HAQ quantizes both weights and activations on each explored wordlength in the same way as TensorRT scheme. Subsequently, they retrain the quantized model to recover the performance, and then move on to reduce the wordlength of the next layer until the predefined constraints are met.

Rusci et al. [284] proposed to gradually reduce the bit-precision of weights and activations of convolution operation to one of the wordlengths in the set $\{8, 4, 2\}$. Specifically, when the memory requirement for pair of input and output activations of a layer exceeds the memory constraint, the wordlength of the output activations is reduced to the next bit-precision level if and only if they have a higher wordlength or memory footprint than the input activations. This process is adopted during the forward and backward passes. On the other hand, the wordlength of the weights with the highest memory footprint and lowest layer’s index is reduced until meeting the model memory constraint. PACT quantization strategy is adopted for both weights and activations. To achieve the integer-only deployment, the authors combined the FP32 quantization scalars for activations and weights with *BatchNorm* layer parameters, and then they converted the resulting parameters to 32-bit fixed-point values.

HAWQ [285] defines the quantization sensitivity of each layer/block as the ratio between the top Hessian eigenvalue and the number of parameters. Then, it uses the measured sensitivity to indicate layer’s prioritization to bit-precision reduction. Thus, HAWQ investigates the bit-precision space to determine the wordlength of each layer so that the assigned bit-precision ratio corresponds somewhat to the measured sensitivity of all layers. After quantization, HAWQ retrains the layers, one after another, in decreasing order of their quantization error and Hessian eigenvalue.

CCQ [257] performs stages of competition and collaboration to gradually adapt weight’s wordlength. The competition stage is carried out to measure the effect of quantizing randomly chosen layers to next bit-precision level on accuracy and memory. Thereafter, the CCQ randomly picks a layer based on the competition measurements and quantizes its weights to next level. On the other hand, the collaboration stage retrains all layers until the accuracy reaches a predefined threshold.

Chu et al. [286] heuristically assign the wordlength of activations and weights of each layer based on the separability of their hierarchical distribution. However, for large datasets such as ImageNet, it is unaffordable to obtain a complete separability matrix. Therefore, they used a random subset of the samples for this purpose. To reach the desired quantized network, the proposed framework utilized DoReFa-Net quantization technique with progressively decreasing wordlength during network training.

DoubleQExt [287] quantizes weights and activations to 8-bit integers using layer-wise FP32 scalar and offset parameters. Thereafter, it quantizes the integer weights again to represent them in power-of-2 form using 5 bits, thus, reducing computational and memory cost. To recover the accuracy after the double quantization process, DoubleQExt iteratively reverts the most informative weights to their 8-bit representation until the recovered accuracy reaches a predefined recovery threshold.

As a summary of the literature, improving the accuracy of quantized DNNs comes at the expense of floating-point computational cost in [165], [167], [234], [236], [237], [241], [244], [255]–[257], [260], [262]–[266], [268], [273], [275], [277]–[279], [281]–[283], [285]–[287]. Specifically, these approaches scale output activations of each layer with

FP32 coefficient(s) to recover the dynamic range, and/or perform batch normalization as well as the operations of first and last layers with FP32 data-structures. Thus, the FPGA-/ASIC-based hardware accelerators of such quantized DNNs still need to implement expensive floating-point units for multiplication and accumulation, and even the much more expensive non-linear quantization arithmetics in [259], [261], [269]. To overcome this issue, this chapter adopts a simpler and a more realistic hardware-aware fixed-point quantization for integer-only deployment of quantized DNNs.

Another notable drawback of the discussed techniques in [257], [271], [276], [282]–[286] is that they usually perform training repeatedly, which is highly inefficient and takes a lot of time to construct the quantized model [101]. Furthermore, training requires a full-size dataset, which is often unavailable in real-world scenarios for reasons such as proprietary and privacy, especially in the case when working on an off-the-shelf pre-trained model from a community or industry for which data is no longer accessible. Additionally, training involves hyper-parameters configuration, initialization, and training methods, which are not always feasible. In this chapter, we tackle these drawbacks by avoiding training and quantizing pre-trained models.

At the same time, we achieve accuracy at par with the full-precision model, a major challenge in the discussed post-training quantization approaches, through an optimized quantization method with mixed-precision assignment. In contrast to methods that determine quantization parameters based on a local optimization of a metric, such as minimizing the KL divergence in [241] and MSE in [244], [246] of each layer/channel, this chapter involves the data-structures of all layers affected by quantization as well as the network outcome in this process. Thus, it insures a global

optimization of quantization parameters.

To overcome the computationally expensive exploration and sorting phases during wordlength reduction in the discussed mixed-precision quantization approaches, this chapter adopts a simple computation-and-memory based 2-level clustering and traversal of data-structures. Additionally, as opposed to the mixed-precision design in [284], this work considers the sensitivity of layers to quantization when decreasing their bit-precision level. This is because we experimentally found it to significantly affect the accuracy of the model. Last, but not least, this chapter improves the computational and power efficiency of the channel-wise mixed-precision design in [246], [284] through adopting layer-wise quantization and allocating less bits, 16 bits or less, to *BatchNorm* parameters and partial results.

4.4 Design of Mixed Low-Precision DNNs for Integer-Only Deployment

In this section, we provide an insight into our *FxP-QNet*. The *FxP-QNet* takes a pre-trained model with the description of network structure and the design constraints. Then, it designs a network structure for integer-only deployment and quantizes model parameters at different bit-precision levels. The *FxP-QNet* main components are the *Preprocessor*, the *Forward Optimizer*, and the *Network Designer* as shown in Figure 4.2. The *Preprocessor* imposes integer-only inference where floating-point data-structures are quantized to integers in dynamic fixed-point representation using the quantization method that will be illustrated in Section 4.4.2. The *Forward Optimizer*

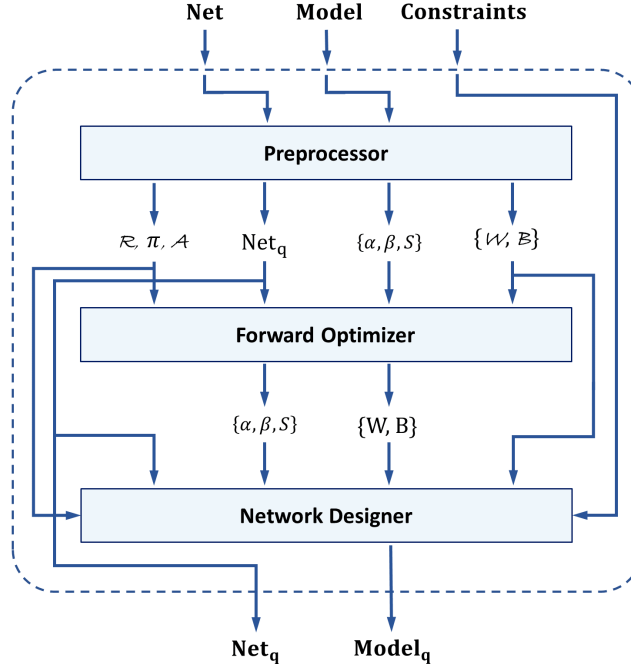


Figure 4.2: Components and workflow of *FxP-QNet*.

employs the optimization framework that will be presented in Section 4.4.3 to improve the quality of the initially designed low-precision DNN. Finally, the *Network Designer* finds the best quantization level for each data-structure to design a mixed low-precision DNN as will be discussed later in Section 4.4.4.

Here, we must emphasize that the dynamic fixed-point quantization is adopted in this chapter because; (i) it allows using fractional numbers on low-cost integer hardware, (ii) it is more hardware-friendly than the non-uniform quantization, (iii) it facilitates the implementation of highly resource-/energy-efficient FPGA-/ASIC-based hardware accelerators, and (iv) its arithmetic is naturally supported in the instruction-set of all general-purpose programmable microcontroller units in edge devices. Details of the *FxP-QNet* components and workflow are presented next.

4.4.1 Integer-Only Inference

In machine learning frameworks, the learnable parameters, e.g., weights (w) and biases (b), as well as the features, i.e., activations (a), of a DNN consisting of L layers are represented with FP32 format. Since not all DNN layers may have learnable parameters, we define the set $\mathfrak{L}_p$ as the layers with learnable parameters, $\mathfrak{L}_p \subseteq [L]$. Formally, we represent the set of all layers activations, weights, and biases as $\mathcal{A} = \{\mathcal{A}_\ell : \ell \in [L]\}$, $\mathcal{W} = \{\mathcal{W}_\ell : \ell \in \mathfrak{L}_p\}$, and $\mathcal{B} = \{\mathcal{B}_\ell : \ell \in \mathfrak{L}_p\}$, respectively.

To avert using floating-point arithmetic, the computations of convolution (*CONV*), fully-connected (*FC*), *BatchNorm*, pooling (*POOL*), rectified linear unit (*ReLU*), Scale, and element-wise (*Eltwise*) layers must be performed on integers. For this purpose, *Fxp-QNet* quantizes the FP32 data-structures required for these operations to integer data-structures either in unsigned dynamic fixed-point representation, $U(\alpha, \beta)$, or in signed 2's complement dynamic fixed-point representation, $S(\alpha, \beta)$. In the last subsection, we show how *preprocessor* specifies the initial quantization parameters, i.e., integer length α , fractional length β , and sign strategy $\mathcal{S}$, for each data-structure.

While model parameters can be quantized offline just before the deployment, quantizing activations requires on-the-fly shifting and clipping within the computational path of the network structure during the forward propagation. To this end, we implemented a layer, referred to as a *fixed-point quantization layer* (*Fxp-QLayer*), to carry out activation quantization. In Section 4.4.1.2, we discuss where the *Fxp-QLayers* are inserted into the computation data path of the network architecture to ensure that the inference is performed using low-precision integer activations. It is note-

worthy that *Fxp-QNet* adopts the layer-wise quantization strategy for both model parameters and activations. On the other hand, the *BatchNorm* formulation hinders integer-only deployment because of its need for floating-point arithmetic. The next subsection explains how *preprocessor* elevates this issue.

4.4.1.1 Batch Normalization Refinement

DNNs are designed with *BatchNorm* layers to accelerate the training phase [288]. *BatchNorm* layer performs two operations, namely, normalizing the inputs to have zero mean and unit variance distributions, and then scaling the normalized inputs to any other distribution through a linear transform function. During network training time, each *BatchNorm* layer learns its scale γ and bias shift δ parameters. Additionally, it computes the global mean μ and variance σ^2 through a running average of the statistics of each mini-batch of training. During network forward pass, *BatchNorm* transform is applied to the input activations $\mathcal{A}_{\ell-1}$ to produce the output activations $\mathcal{A}_{\ell}$ as

$$\mathcal{A}_{\ell} = \text{BatchNorm}(\mathcal{A}_{\ell-1}; \gamma_{\ell}, \delta_{\ell}, \mu_{\ell}, \sigma_{\ell}^2) = \gamma_{\ell} \cdot \frac{\mathcal{A}_{\ell-1} - \mu_{\ell}}{\sqrt{\sigma_{\ell}^2 + \epsilon}} + \delta_{\ell} \quad (4.3)$$

where γ_{ℓ} , δ_{ℓ} , μ_{ℓ} , and σ_{ℓ}^2 are channel-wise FP32 parameters learned for *BatchNorm* layer ℓ , and ϵ is a small constant used for numerical stability. It is worth noting that this kind of formulation hinders integer-only deployment as it contains floating-point arithmetic. In [289], the authors fold *BatchNorm* parameters into *CONV* weights before quantization. However, the folding process itself deteriorated the accuracy. In

this chapter, we simplify and rewrite the *BatchNorm* formula as

$$\text{BatchNorm}(\mathcal{A}_{\ell-1}; \gamma_\ell, \delta_\ell, \mu_\ell, \sigma_\ell^2) = \aleph_{1_\ell} \cdot \mathcal{A}_{\ell-1} + \aleph_{2_\ell} \quad (4.4)$$

where $\aleph_{1_\ell}$ and $\aleph_{2_\ell}$ are new channel-wise FP32 parameters defined as

$$\begin{aligned} \aleph_{1_\ell} &= \frac{\gamma_\ell}{\sqrt{\sigma_\ell^2 + \epsilon}} \\ \aleph_{2_\ell} &= \delta_\ell - \frac{\gamma_\ell \cdot \mu_\ell}{\sqrt{\sigma_\ell^2 + \epsilon}} \end{aligned} \quad (4.5)$$

Accordingly, *preprocessor* modifies the network structure by converting *BatchNorm* layers into much simpler Scale layers, updates the model parameters by assigning to each of these Scale layers its scale $\aleph_{1_\ell}$ and bias shift $\aleph_{2_\ell}$ parameters, and finally removes the parameters related to *BatchNorm* layers. At this point, performing integer-only inference becomes feasible by quantizing the parameters of Scale layers in the same way as quantizing weights and biases of *CONV/FC* layers. Note that, for simplicity, we refer to $\aleph_{1_\ell}$ and $\aleph_{2_\ell}$ as $\mathcal{W}_\ell$ and $\mathcal{B}_\ell$, respectively, during the design of low-precision DNNs.

4.4.1.2 FxP-QLayer: An On-the-Fly Quantization of Activations

The *FxP-QLayer* is responsible for changing the wordlength of input activations to a lower bit-precision level. This is done through a simple on-the-fly shift and add operations without using expensive multipliers, as will be discussed in Section 4.4.2. Here, an important issue is where to inject *FxP-QLayers* into the computation data path of the network architecture. To perform the first computation on low-precision

integers, *Fxp-QLayer* must be inserted at the beginning of the network to quantize input features. In addition, output activations of *CONV*, *FC*, *POOL*, *ReLU*, *Scale*, and *Eltwise* layers must be quantized before being fed to the next layer. However, for efficient inference of DNNs in hardware, some DNN layers can be fused with each other to form what we call a computational block (*C-Block*).

For instance, *ReLU* and *Scale* layers can be fused with their previous layer since they perform feature-wise operations. Thus, *Fxp-QLayer* is also required after each *C-Block* that has a structure such as " $\mathcal{X}$ – *Scale* and/or *ReLU*" or even as simple as " $\mathcal{X}$ ", where $\mathcal{X}$ here can be *CONV*, *FC*, *POOL*, or *Eltwise* layer. Note that unlike *AVG-POOL* operation where a patch of activations is replaced by their average, *MAX-/MIN-POOL* operations replace a patch of activations with one of them, which is the max/min value. Accordingly, there is no need for a *Fxp-QLayer* after *MAX-/MIN-POOL* layer. On the other hand, for *AVG-POOL* layer, the reciprocal of the patch size is handled in the same way as the model parameters.

Last but not least, activations produced by last network layer are used to predict the model outcome and no other calculations are made based on them. So that, there is no need to insert a *Fxp-QLayer* after the last network layer. The *preprocessor* modifies the network structure by injecting *Fxp-QLayers* as discussed above. Thus, DNNs can be viewed as a graph with *Fxp-QLayers* as vertices and *C-Blocks* as edges. Each *C-Block* performs its layers operations in a pipeline fashion, and thereafter *Fxp-QLayer* carries out quantization to its *C-Block* output activations.

4.4.1.3 Initial Design Generation

The *preprocessor* adopts a statistics-driven approach to determine the initial fixed-point representation for a given FP32 data-structure $X_\ell^{(d)}$, $X_\ell^{(d)} \in \{\mathcal{A}_\ell, \mathcal{W}_\ell, \mathcal{B}_\ell\}$, and a target wordlength $k_\ell^{(d)}$. Here, d is the data-structure type, $d \in \{a, w, b\}$, and ℓ is the layer ID, $\ell \in [L]$. Specifically, the smallest FP32 value in the data-structure is used to determine its sign strategy as follows

$$\mathcal{S}_\ell^{(d)} = \mathcal{I} \left(X_{\ell \text{ MAX-}}^{(d)} < 0 \right) \quad (4.6)$$

where $\mathcal{I}(\cdot)$ is the indicator function; it returns 1 if the condition is satisfied and 0 otherwise. Thereafter, the *preprocessor* analyzes the scalar ψ based on the dynamic range, which is determined by wordlength and sign strategy as mentioned in Section 4.2.1, such that

$$\psi_\ell^{(d)} = \begin{cases} \frac{X_{\ell \text{ MAX+}}^{(d)}}{2^{k_\ell^{(d)}} - 1} & , \quad \mathcal{S}_\ell^{(d)} = 0 \\ \min \left(\frac{X_{\ell \text{ MAX-}}^{(d)}}{-2^{k_\ell^{(d)}} - 1}, \frac{X_{\ell \text{ MAX+}}^{(d)}}{2^{k_\ell^{(d)}} - 1} \right) & , \quad \mathcal{S}_\ell^{(d)} \neq 0 \end{cases} \quad (4.7)$$

where $X_{\ell \text{ MAX+}}^{(d)}$ and $X_{\ell \text{ MAX-}}^{(d)}$ are the largest and smallest FP32 values in the data-structure $X_\ell^{(d)}$, respectively. Since the scalar of fixed-point representation equals $2^{-\beta}$, the FP32 scaling parameter can be calculated as

$$\hat{\beta}_\ell^{(d)} = -\log_2 \psi_\ell^{(d)} \quad (4.8)$$

Accordingly, the *preprocessor* initializes the integer length α and fractional length β based on $\hat{\beta}_\ell^{(d)}$, $\mathcal{S}_\ell^{(d)}$, and a given rounding strategy $\mathfrak{R}\mathfrak{s}$. Here, we adopt three rounding

strategies, namely, *conservative* ($\mathfrak{Rsc}$), *neutral* ($\mathfrak{Rsn}$), and *aggressive* ($\mathfrak{Rsa}$) strategies.

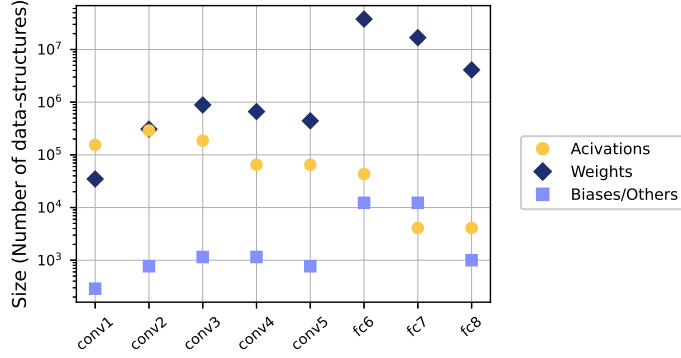
While the *neutral* rounding strategy calculates the fractional length using the *round* function with half is rounded to the nearest even, the *conservative* and *aggressive* rounding strategies use the *floor* and *ceil* functions, respectively, such that

$$\beta_\ell^{(d)} = \begin{cases} \lfloor \hat{\beta}_\ell^{(d)} \rfloor & , \quad \mathfrak{R}\mathfrak{s} \text{ is } \mathfrak{Rsc} \\ \text{round}(\hat{\beta}_\ell^{(d)}) & , \quad \mathfrak{R}\mathfrak{s} \text{ is } \mathfrak{Rsn} \\ \lceil \hat{\beta}_\ell^{(d)} \rceil & , \quad \mathfrak{R}\mathfrak{s} \text{ is } \mathfrak{Rsa} \end{cases} \quad (4.9)$$

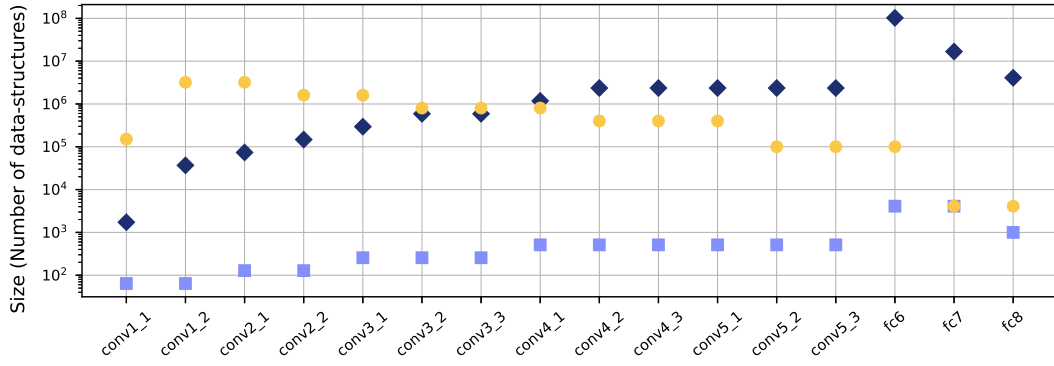
$$\alpha_\ell^{(d)} = k_\ell^{(d)} - \beta_\ell^{(d)} - \mathcal{S}_\ell^{(d)}$$

In Section 4.5.2, we analyze these strategies by demonstrating the performance of a DNN that was quantized using initial quantization parameters derived from Equation (4.9) at different homogeneous wordlengths. Here, an important issue is deciding which initial bit-precision level should be assigned to each data-structure. The experimental results for data-structure sensitivity to quantization, illustrated in Section 4.5.4, show that data-structures in shallow layers are more sensitive to quantization than the ones in deep layers. On the other hand, output activations of the last network layer are used to predict the model outcome. Hence, the computation of first and last network layers should initially use accurate neurons, i.e., neurons with a high bit-precision level, in order to maintain the performance.

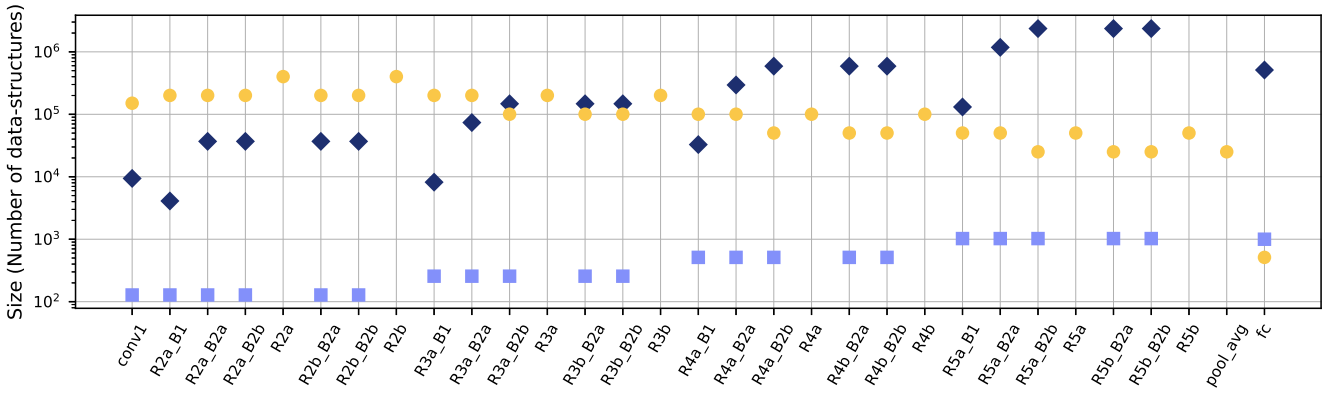
This fact that the first and the last layers should be treated differently has also been observed in various quantization techniques [165], [167], [234], [237], [260], [262], [263], [265], [266], [275], [286], where quantization of these layers is avoided as it may



(a) AlexNet architecture.



(b) VGG-16 architecture.



(c) ResNet-18 architecture.

Figure 4.3: Statistics on size of input data-structures for each layer of (a) AlexNet, (b) VGG-16, and (c) ResNet-18 architectures in log-domain.

have devastating effects on the overall accuracy. However, we found that quantizing these two layers can greatly increase the compression ratio which in turn significantly reduces the computational power required. Thus, the proposed framework starts the bit-precision level of activations and weights of first and last layers with 32 bits, i.e., $k_0^{(a)} = k_1^{(w)} = k_{L-1}^{(a)} = k_L^{(w)} = 32$.

On the other hand, to achieve the accuracy-compression trade-off, the framework sets the wordlength of other activations and weights of *CONV* and *FC* layers to 8 bits. This is because it has been shown that representing these data-structures with 8-bit integers achieves comparable accuracy to a full-precision network without the need for fine-tuning [242]. Other data-structures, such as biases of *CONV* and *FC* layers as well as the parameters of Scale layer, have a relatively small memory footprint in typical DNNs as shown in Figure 4.3. Thus, the framework uses 32 bits to initially represent them.

4.4.2 Quantization Methodology

The quantization method, denoted as $Q(.; ., ., .)$, converts a FP32 data-structure $X_\ell^{(d)}$, $X_\ell^{(d)} \in \{\mathcal{A}_\ell, \mathcal{W}_\ell, \mathcal{B}_\ell\}$, to a low-precision integer data-structure $\mathbf{X}_\ell^{(d)}$, $\mathbf{X}_\ell^{(d)} \in \{\mathbf{A}_\ell, \mathbf{W}_\ell, \mathbf{B}_\ell\}$, either in unsigned dynamic fixed-point representation, $U(\alpha_\ell^{(d)}, \beta_\ell^{(d)})$, or in signed 2's complement dynamic fixed-point representation, $S(\alpha_\ell^{(d)}, \beta_\ell^{(d)})$, depending on the layer-wise quantization parameters $\alpha_\ell^{(d)}$, $\beta_\ell^{(d)}$, and $\mathcal{S}_\ell^{(d)}$, such that

$$\mathbf{X}_\ell^{(d)} = Q\left(X_\ell^{(d)}; \alpha_\ell^{(d)}, \beta_\ell^{(d)}, \mathcal{S}_\ell^{(d)}\right) = \text{clip}\left(\left\lfloor X_\ell^{(d)} \cdot 2^{\beta_\ell^{(d)}} \right\rfloor; \mathbf{Q}_{\min}, \mathbf{Q}_{\max}\right) \quad (4.10)$$

where d denotes the data-structure type, $d \in \{a, w, b\}$, such that $\mathbf{X}^{(a)} \equiv \mathbf{A}$, $\alpha_\ell^{(d)}$ and $\beta_\ell^{(d)}$ refer to the integer length and fractional length, respectively, $\alpha_\ell^{(d)}$ and $\beta_\ell^{(d)} \in \mathbb{Z}$. Thus, each integer number in $\mathbf{X}_\ell^{(d)}$ is represented with $k_\ell^{(d)}$ bits, $k_\ell^{(d)} = \alpha_\ell^{(d)} + \beta_\ell^{(d)}$. Note that the values represented by $\mathbf{X}_\ell^{(d)}$ are calculated as mentioned in Equation (4.1) using the parameter $\beta_\ell^{(d)}$. On the other hand, the sign strategy $\mathcal{S}_\ell^{(d)}$ indicates whether the data-structure type d at layer ℓ consists of unsigned (0) or signed (1) numbers, $\mathcal{S}_\ell^{(d)} \in \{0, 1\}$. The sign strategy is employed in order to avoid losing half of the dynamic range when, for instance, using signed number representation for non-negative data-structures. Additionally, the $\lfloor \bullet \rfloor$ represents the round operation where the half is rounded to the nearest even to prevent bias [289], and

$$\text{clip}(\hat{\mathbf{X}}; \min, \max) = \begin{cases} \min & , \quad \forall \hat{\mathbf{X}}_i < \min \\ \hat{\mathbf{X}}_i & , \quad \min \leq \forall \hat{\mathbf{X}}_i \leq \max \\ \max & , \quad \forall \hat{\mathbf{X}}_i > \max \end{cases} \quad (4.11)$$

Furthermore, the $\mathbf{Q}_{\min}$ and $\mathbf{Q}_{\max}$ represent the smallest and largest quantization levels, respectively, and are computed as

$$\begin{aligned} \mathbf{Q}_{\min} &= -\mathcal{S}_\ell^{(d)} / 2^{\mathcal{S}_\ell^{(d)} - \alpha_\ell^{(d)} - \beta_\ell^{(d)}} \\ \mathbf{Q}_{\max} &= 2^{\alpha_\ell^{(d)} + \beta_\ell^{(d)} - \mathcal{S}_\ell^{(d)}} - 1 \end{aligned} \quad (4.12)$$

For the special case of quantizing a signed data-structure to a very low-precision level, i.e., 1-bit, the quantization method utilizes the binary quantization [165]. More

precisely, it quantizes the elements of the input $X_\ell^{(d)}$ based on their *sign* as follows

$$\mathbf{X}_\ell^{(d)} = Q\left(X_\ell^{(d)}; \alpha_\ell^{(d)}, \beta_\ell^{(d)}, \mathcal{S}_\ell^{(d)}\right) = \begin{cases} -1 & , \quad \forall X_{i_\ell}^{(d)} \in (-\infty, 0) \\ 0 & , \quad \forall X_{i_\ell}^{(d)} \in [0, \infty) \end{cases} \quad (4.13)$$

Then, before performing operations on $\mathbf{X}_\ell^{(d)}$, the values of $\mathbf{X}_\ell^{(d)}$ are decoded such that 0s are mapped into +1. Thus, the values represented by $\mathbf{X}_\ell^{(d)}$ become $\pm 2^{-\beta_\ell^{(d)}}$, eliminating the need for digital multipliers. Here, we must emphasize that *Fxp-QNet* performs a layer-wise quantization. One can also note that, compared to the previous quantization techniques discussed in Section 4.2.2, the proposed quantization method is more computationally efficient as *all* DNN data-structures are quantized into fixed-point numbers, and thus, *all* the performed operations are integer arithmetic, or even bit-wise operations, whereas floating-point arithmetic is not involved at all.

In Figure 4.4, we provide an overview of *Fxp-QNet* low-precision computations. During forward propagation, integer activations and model parameters are used as inputs to a low-precision integer *C-Block* consisting of *CONV*, *Scale*, and *ReLU* layers. A MAC tree of integer arithmetic can be used to convolve an activation window $\mathbf{A}_{i-1,j}$ with a weight kernel $\mathbf{W}_{i,m}$ each of N integers. To avoid the overflow, the partial results of the MAC tree in first and last layers are computed with 32 bits, while on the other hand, MAC tree partial results in other layers are computed with 16 bits. This is because these data-structures have a low bit-precision level during deployment and a limited data range with weights in bell-curve distribution.

Thereafter, the resultant feature from MAC tree needs to be accumulated with the

bias parameter $\mathbf{B}_{i,m}$ to produce the output feature of *CONV* layer. However, these two values may have different scaling parameters. Hence, $\mathbf{B}_{i,m}$ must be left-shifted to have the same scalar as MAC tree result before accumulation. When considering arithmetic operations within MAC tree, one can notice that multiplying two integers having scalars $2^{-\beta_{i-1}^{(a)}}$ and $2^{-\beta_i^{(w)}}$ yields an integer having $2^{-(\beta_{i-1}^{(a)} + \beta_i^{(w)})}$ as its scalar. On the other hand, accumulating integers results in a value that has a scalar equal to that of the integers. Thus, the amount of shifting required to align fractional parts is specified by *Fxp-QNet* as $\delta_i = \beta_{i-1}^{(a)} + \beta_i^{(w)} - \beta_i^{(b)}$, where $\beta_i^{(b)}$ is the scaling parameter of $\mathbf{B}_i$.

Similarly, an integer multiplier and accumulator is used to carry out the computation of *BatchNorm/Scale* layers. Thereafter, the produced feature is compared to 0 and the output value is the larger of them. The output feature from the *C-Block* is then rounded and clipped after a right-shift operation by δ_{i+2} , which is equal to $\beta_{i-1}^{(a)} + \beta_i^{(w)} + \beta_{i+1}^{(w)} - \beta_{i+2}^{(a)}$. Here, we must emphasize that each layer within a *C-Block* hypothetically considers activations produced by *C-Block* as the activations it produced. Hence, it is necessary to note that when mentioning layer ℓ activations, we mean the hypothetical activations of layer ℓ , i.e., activations produced by *C-Block* in which layer ℓ is. Additionally, if a *C-Block* is followed by a *Fxp-QLayer*, then when referring to the activations produced by *C-Block*, we actually mean those produced by *Fxp-QLayer*.

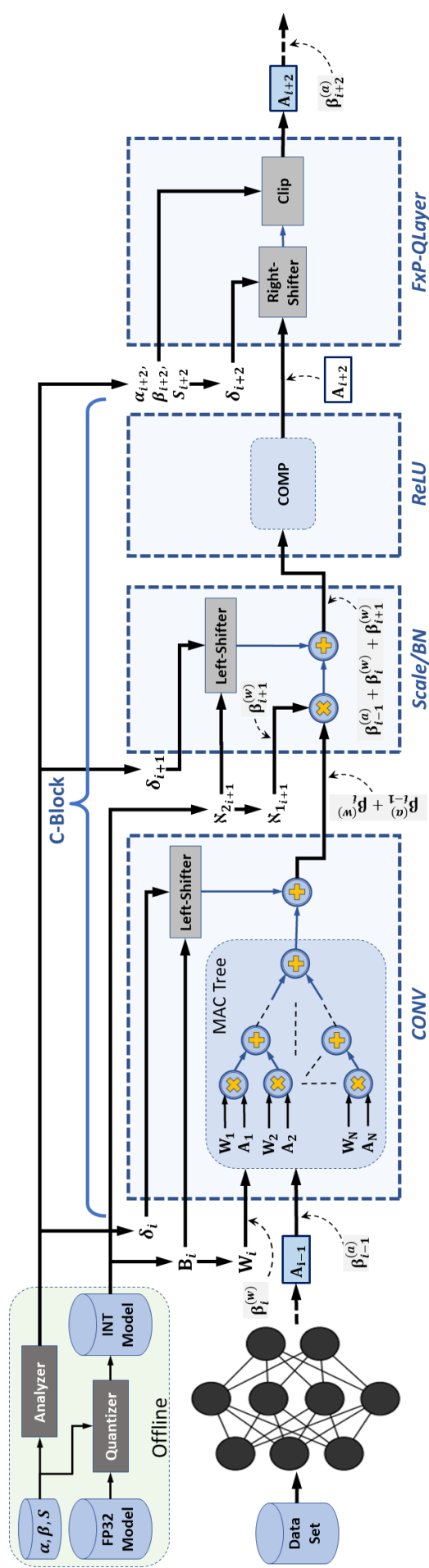
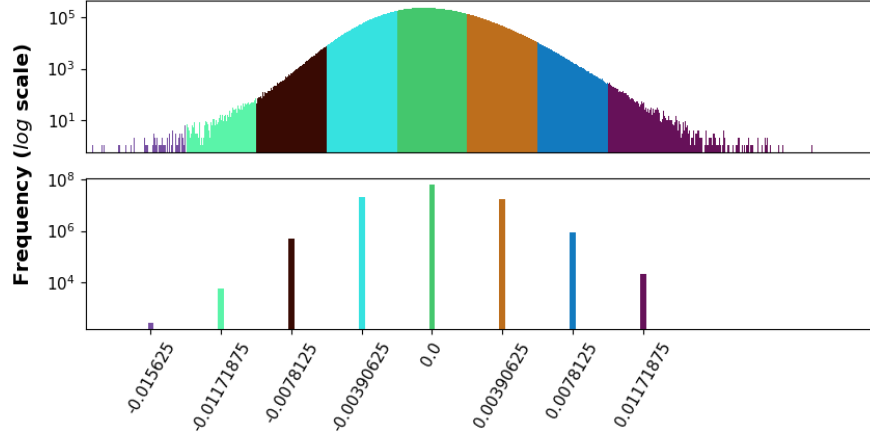


Figure 4.4: An overview of *FxP-QNet* low-precision calculations in a computational block consisting of *CONV*, *Scale*, and *ReLU* layers, as envisioned here. Note that shift, wordlength, and sign strategy are layer-wise parameters consisting of a very small number of bits, and therefore their memory footprint is negligible.

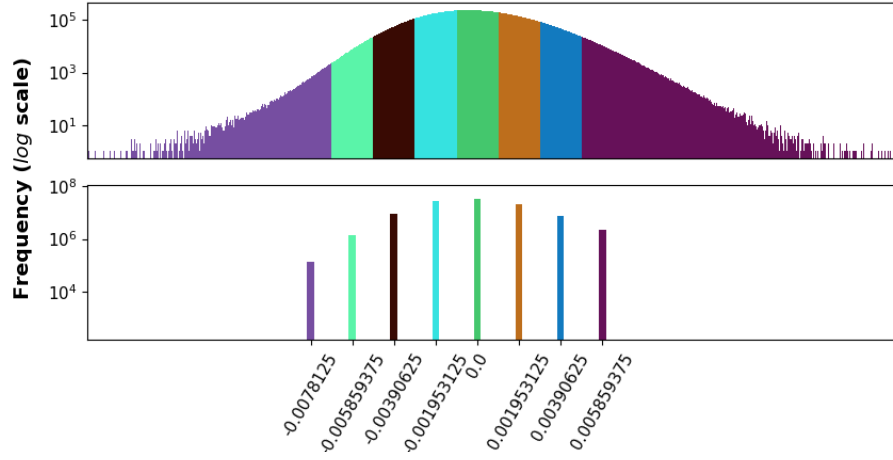
4.4.3 Quantization Parameters Optimization

A k -bit quantized data-structure can have several quantization domains (domQs) based on the selected quantization parameters (QPs), i.e., the integer length α and fractional length β . The domQ refers to the set of possible values that can be represented by the given QPs. Figure 4.5 shows the effect of these parameters on quantizing the weights of the first fully-connected layer in the VGG-16 model [27], i.e., the *FC-6* layer. As we can see, using a large β represents many small weights more finely but at the expense of losing the accuracy in the representation of large weights. Here, we mean the magnitude of the weights, not the sign. One can also note that such data-structures tend to have a bell-curve distribution, which enables them to be quantized effectively with error-minimization-based techniques without the need for retraining [246].

Therefore, an important step during data-structure quantization is to select appropriate QPs that result in the optimal domQ. Prior quantization policies [60], [243], [257] choose the domQ with the highest overall network accuracy. However, it turns out, empirically, that the resulting network is prone to overfitting. Thus, retraining is mandatory to maintain the performance of the quantized network with an accuracy-based selection of domQ. Alternatively, other quantization policies [88], [241], [244], [246], [290] assume that if the perturbation in the distributions of data-structures due to discretization is minimized, then the network accuracy can be maintained. Consequently, we initially considered the optimal QPs as the ones that minimize the



(a) Quantization with $\alpha = -5$ and $\beta = 8$.



(b) Quantization with $\alpha = -6$ and $\beta = 9$.

Figure 4.5: Distribution of FC-6 weights in VGG-16 model before (*top*) and after (*bottom*) 3-bit fixed-point quantization. The two figures in (a) show the quantization of the weights with α and β as -5 and 8 , respectively. While, the two figures in (b) show the quantized weights with α equals -6 and β equals 9 .

data-structure-based quantization error, denoted as $\mathcal{L}$, given by

$$\mathcal{L}(X; \alpha, \beta, \mathcal{S}) = \|X - \tilde{X}\|_2^2 \quad (4.14)$$

Here, $\tilde{X}$ is the low-precision representation of the FP32 X that is computed as mentioned before in Equations (4.1), (4.10), and (4.13), and $\|\bullet\|_2^2$ is the squared $L2$ norm. However, we empirically observed that minimizing the quantization error for each layer and data-structure independently produces a local optimum solution. Such kind of local optimization results in a massive degradation in the overall accuracy. This is because the quantization error propagates to the succeeding layers and magnifies. The way of quantizing the activations, weights, and biases of a layer affects the activation distribution of all layers succeeding it. Additionally, quantizing data-structures independently may not allow for rigorous model compression.

Hence, while quantizing a data-structure of a specific layer, the proposed optimization algorithm considers the quantization error of all data-structures in the same layer as well as the quantization error of data-structures in all the succeeding layers. We refer to these combined errors as the post-training self-distillation error ($\mathcal{E}$). However, we expect the data to be quantized less finely as dispersion increases. This is because the distance between consecutive values in domQ becomes large to better encode outliers. Additionally, as the number of elements in the data-structure increases, the quantization becomes more significant because many occurrences of quantization error propagate to layer output. Thus, the higher the dispersion and size of data-structure get, the more the skew in the statistics of the output activations can be

incurred, which affects the following layers, and therefore degrades model's accuracy.

To take that into account, we assign a weight of importance, π , to each data-structure. Such that, π is a quantitative metric for evaluating the impact of quantizing a data-structure on the output quality of its layer. The quantization error of a data-structure with a large π is more expensive than the error due to quantizing a data-structure with a small π . The value of π is set as follows

$$\pi_\ell^{(d)} = \frac{\sigma_{x_\ell}^{(d)} \cdot \nu_\ell^{(d)}}{\sum_{i \in \{a,w,b\}} \sigma_{x_\ell}^{(i)} \cdot \nu_\ell^{(i)}} \quad (4.15)$$

where π is normalized so as $\sum_{i \in \{a,w,b\}} \pi_\ell^{(i)} = 1$, $\forall \ell \in [L]$, $\sigma_{x_\ell}^{(d)}$ and $\nu_\ell^{(d)}$ are dispersion and size of data-structure type d in layer ℓ , respectively, and $\sigma_{x_\ell}^{(d)}$ is computed as

$$\sigma_{x_\ell}^{(d)} = \sqrt{\frac{1}{\nu_\ell^{(d)}} \cdot \sum_{i=1}^{\nu_\ell^{(d)}} \left(X_{i\ell}^{(d)} - \bar{X}_\ell^{(d)} \right)^2} \quad (4.16)$$

Here, $\bar{X}_\ell^{(d)}$ is the arithmetic mean of $X_\ell^{(d)}$, and $\sigma_{x_\ell}^{(a)}$ is calculated for all of the calibration dataset samples and the mean is taken. Note that the calibration dataset is a subset of the validation dataset used to do what is called QPs calibration, hence the name. Remember that data-structures in different layers have different distributions. Therefore, the errors due to quantizing them must be normalized. Thus, post-training self-distillation error on quantizing the data-structure type d in layer ℓ is calculated as

$$\mathcal{E}(d, \ell) = \sum_{i=\ell}^L \sum_{j \in \{a,w,b\}} \pi_i^{(j)} \cdot \frac{\mathcal{L} \left(X_i^{(j)}; \alpha_i^{(j)}, \beta_i^{(j)}, \mathcal{S}_i^{(j)} \right)}{\mathcal{R}_i^{(j)}} \quad (4.17)$$

where $\mathcal{L}(.;.,.,.) / \mathcal{R}$ is the normalized quantization error, and $\mathcal{R}_i^{(j)}$ is the range of data-structure type j in layer i ,

$$\mathcal{R}_i^{(j)} = X_{i \text{ MAX}+}^{(j)} - X_{i \text{ MAX}-}^{(j)} \quad (4.18)$$

and $X_{i \text{ MAX}+}^{(j)}$ and $X_{i \text{ MAX}-}^{(j)}$ are the largest and smallest FP32 values in the data-structure $X_i^{(j)}$, respectively. It is worth noting that, in the classification task, the last layer of the network produces the probabilities of classifying the input sample for each class. A classifier performs a prediction mapping from the input sample to the model outcome based on these probabilities, the model outcome is the class with the highest probability. Consequently, the classifier fails in predicting the class of the input sample when two, or more, classes have the same highest probability, which might occur when classification is performed on a very-low-precision DNN. Hence, we take into consideration the percentage of unclassified samples (UnClass) as well as the percentage of mispredicted classes (MisPred) in those unclassified samples, and minimize them to maintain the performance, given by

$$\text{UnClass} = \frac{1}{N} \cdot \sum_{i=1}^N \mathcal{I} \left(\left| \mathcal{C}_i^{(1)} \right| > 1 \right) \quad (4.19)$$

$$\text{MisPred} = \frac{1}{N \cdot \mathcal{T}_C} \cdot \sum_{i=1}^N \left(\left| \mathcal{C}_i^{(1)} \right| - 1 \right) \quad (4.20)$$

where $\mathcal{I}(\cdot)$ is the indicator function; it returns 1 if the condition is satisfied and 0 otherwise, $\left| \mathcal{C}_i^{(1)} \right|$ is the number of predicted classes with the same highest probability for the i -th input sample, N is the number of calibration dataset samples, and $\mathcal{T}_C$ is

the number of dataset classes. Based on the discussion above, for optimizing QPs of the data-structure type d in layer ℓ with an initial QPs, denoted as $(\alpha_{\text{init}}, \beta_{\text{init}})$, the optimization framework systematically explores the trade-off between the resolution and range saturation spaces by modulating $(\alpha_{\ell}^{(d)}, \beta_{\ell}^{(d)})$ to minimize the overall post-training self-distillation and network prediction errors, denoted by $\mathcal{F}$, as

$$\begin{aligned} \underset{\alpha_{\ell}^{(d)}, \beta_{\ell}^{(d)}}{\text{minimize}} \quad & \mathcal{F}(d, \ell) = \mathcal{E}(d, \ell) + \text{UnClass} + \text{MisPred} \\ \text{subject to} \quad & \alpha_{\ell}^{(d)} + \beta_{\ell}^{(d)} = \alpha_{\text{init}} + \beta_{\text{init}} \end{aligned} \tag{4.21}$$

Algorithm 5 summarizes the steps involved in finding the optimal QPs. In the initialization step, the optimization framework initializes the quantization *cost*, i.e., the $\mathcal{F}$ error, based on the initial QPs as shown in lines 1 – 5. The search step, highlighted in blue, is used to explore the $2K$ neighboring QPs for the one with the minimum *cost* as defined in Equation (4.21), where K is the search space limit. Given that post-training self-distillation error, $\mathcal{E}$, is a convex function and network prediction error, i.e., $\text{UnClass} + \text{MisPred}$, has a relatively small magnitude, the search space can be safely reduced by setting K to a small value and the global optimum QPs is insured by expanding the search space in the direction of minimum *cost*. In other words, if a neighbor reduces the *cost*, the framework takes a step further and expands the search space to verify the *cost* of additional adjacent QPs. It is worth noting that the forward process is adopted after each quantization operation because the impact of data-structure quantization with various QPs is unpredictable on both *cost* and accuracy. Fortunately, this process is cheap in terms of computation when

Algorithm 5: Quantization Parameters Optimization.

Input: An initial QPs $(\alpha_{\text{init}}, \beta_{\text{init}})$ for the data-structure type d in layer ℓ of a quantized network structure Net_q and model $\{\mathbf{W}, \mathbf{B}\}$ using QPs sets $\{\alpha, \beta, \mathcal{S}\}$, a reference FP32 model $\{\mathcal{W}, \mathcal{B}\}$, and a validation set $\mathcal{T}$.

Output: The optimum QPs and network accuracy (Top-1 and Top-5).

Procedure QParamsOpt($\alpha_{\text{init}}, \beta_{\text{init}}, d, \ell, \text{Net}_q, \{\mathbf{W}, \mathbf{B}\}, \{\alpha, \beta, \mathcal{S}\}, \{\mathcal{W}, \mathcal{B}\}, \mathcal{T}$):

```
1  Initialize  $\alpha_\ell^{(d)}$  and  $\alpha_{\text{best}}$  with  $\alpha_{\text{init}}$ ,  $\beta_\ell^{(d)}$  and  $\beta_{\text{best}}$  with  $\beta_{\text{init}}$ 
2  Forward the network on calibration set
   forward( $\text{Net}_q, \{\mathbf{W}, \mathbf{B}\}, \{\alpha, \beta, \mathcal{S}\}; \text{sample}(\mathcal{T})$ )
3  Calculate the error  $\mathcal{E}(d, \ell)$  using (4.17)
4  Calculate the percent of UnClass samples and MisPred classes using (4.19, 4.20)
5  Initialize the best cost
   Costbest  $\leftarrow \mathcal{E} + \text{UnClass} + \text{MisPred}$ 
6  for  $i \in \{-1, +1\}$  do ▷ Range Saturation vs. Resolution
7       $k \leftarrow 1, j \leftarrow 0$ 
8      while  $k \leq (K + j)$  do ▷ Search Size
9          Update the quantization parameters
            $\alpha_\ell^{(d)} \leftarrow \alpha_{\text{init}} - i \times k, \beta_\ell^{(d)} \leftarrow \beta_{\text{init}} + i \times k$ 
10         Quantize the data-structure  $(d, \ell)$  using (4.10, 4.13)
11         Forward the network on calibration set
           forward( $\text{Net}_q, \{\mathbf{W}, \mathbf{B}\}, \{\alpha, \beta, \mathcal{S}\}; \text{sample}(\mathcal{T})$ )
12         Calculate the error  $\mathcal{E}(d, \ell)$  using (4.17)
13         Calculate the percent of UnClass samples and MisPred classes using
           (4.19, 4.20)
14         Calculate the current cost
           Costcurr  $\leftarrow \mathcal{E} + \text{UnClass} + \text{MisPred}$ 
15         if Costcurr < Costbest then
16             Costbest  $\leftarrow$  Costcurr
17              $\alpha_{\text{best}} \leftarrow \alpha_\ell^{(d)}, \beta_{\text{best}} \leftarrow \beta_\ell^{(d)}, j \leftarrow j + 1$ 
18         end
19          $k \leftarrow k + 1$ 
20     end
21 end
22 Set QPs as  $\alpha_\ell^{(d)} \leftarrow \alpha_{\text{best}}, \beta_\ell^{(d)} \leftarrow \beta_{\text{best}}$ 
23 Quantize the data-structure  $(d, \ell)$  using (4.10, 4.13)
24 Forward the network on validation set
   Accbest  $\leftarrow$  forward( $\text{Net}_q, \{\mathbf{W}, \mathbf{B}\}, \{\alpha, \beta, \mathcal{S}\}; \mathcal{T}$ )
25 return  $\alpha_{\text{best}}, \beta_{\text{best}}, \text{Acc}_{\text{best}}$ 
```

feed-forwarded on calibration and validation sets.

On the other hand, it is important to ensure, before starting to design a mixed low-precision DNN, that the accuracy drop for the initial solution proposed by *pre-processor* is negligible. Therefore, the *forward optimizer* fine-tunes the initial solution by employing Algorithm 5. In doing so, it optimizes the initial QPs for the first layer, after which it moves to the next layer and so on until it reaches and optimizes QPs of the last layer, and hence the name *forward optimizer*. Specifically, for each layer, *forward optimizer* identifies the optimal QPs for weights, then for bias, and finally for activations.

The reason behind adopting this order for layers and data-structures is that the quantization of activations, weights, and biases of a layer does not affect the activation distribution of all the layers that precede it, while on the other hand the activations produced by a layer are affected by how the learnable parameters of the same layer are quantized. Note that while optimizing QPs for a layer, the data-structures in all the succeeding layers are floating-point values, conversely, all the data-structures in preceding layers are fixed-point values.

4.4.4 Bit-Precision Level Reduction

The primary goal of quantization is to reduce the computational cost as well as having a minimal memory footprint to accelerate DNN inference. Recalling that DNNs are designed to achieve superior accuracy on computer vision tasks, such as image classification, it is therefore important to investigate the role each layer plays in forming a good predictor before reducing its bit-precision. In light of the above, we conducted

an experiment to investigate the sensitivity of each data-structure to quantization. The results, which are illustrated in Section 4.5.4, show that quantizing different DNN data-structures causes various levels of network accuracy degradation, and the bit-precision of some data-structures can be significantly reduced without causing noticeable performance degradation.

Even more importantly, though, the results show that data-structures in deep layers are more robust to quantization than the ones in shallow layers. During the forward propagation in DNNs, the stacked layers extract internal features and progressively enhance the dissimilarity between the hierarchical features. Compared to the more separable features in deep layers, the features in shallow layers are distributed on complex manifolds and overlap mutually. Therefore, extracting the meaningful intermediate representations in shallow layers requires the use of more accurate neurons, neurons with high bit-precision level, than the ones in deep layers [286].

Motivated by the aforementioned observations, we propose an iterative accuracy-driven bit-precision reduction framework that designs a quantized DNN flexibly with a mixed-precision rather than a k -bit homogeneous network. The proposed framework aims to find the best quantization level for each data-structure based on the trade-off between accuracy and low-precision requirements. Nonetheless, the search space for choosing the appropriate quantization level of each data-structure in different layers is exponential in the number of layers and data-structures, which makes the exhaustive search impractical. For instance, VGG-16 exposes a search space of size $N_{lv_1}^{2 \times L-2} \times N_{lv_2}^{L+4} = 8^{30} \times 32^{20} > 10^{57}$, where L is the number of layers, N_{lv_1} is the number of possible quantization levels for activations and weights in all layers except

the first and last layers, and N_{lv_2} is the number of possible quantization levels for other data-structures.

To overcome this issue, we present a simple deterministic technique for dividing this huge search space into relatively small feasible sets of solutions. The bit-precision level of the data-structures within each set is determined based on their quantization robustness. With this technique, first and last layers can also be quantized to lower levels without having a drastic impact on performance. As illustrated in Algorithm 6, the proposed framework starts with an initial quantized DNN where the data-structures are in fixed-point representation, each of which with possibly different wordlength from 32 to 8. Thereafter, it progressively quantizes the data-structures until it reaches the maximum compression rate under a given accuracy degradation constraint as elaborated in the three steps given below.

4.4.4.1 Data-Structure Clustering

In typical DNNs, forward propagation analysis demonstrated that the *CONV* operation takes more than 90% of the total computational time [85]. Thus, to improve the computational efficiency, the proposed framework focuses on reducing the bit-precision level of the weights of *CONV* and *FC* layers as well as the activations. This is done through dividing the data-structures into two sets of pairs (d, ℓ) such that d is the data-structure type, $d \in \{a, w, b\}$, and ℓ is the layer ID, $\ell \in [L]$, as

$$\begin{aligned}\mathbf{G}_1 &= \{(a, \ell) : \ell \in [L]\} \cup \{(w, \ell) : \ell \in \mathfrak{L}_p, \text{type}(w, \ell) \in \{\text{CONV}, \text{FC}\}\} \\ \mathbf{G}_2 &= \{(d, \ell) : d \neq a, \ell \in \mathfrak{L}_p, (d, \ell) \notin \mathbf{G}_1\}\end{aligned}\tag{4.22}$$

Here, $\mathfrak{L}_p$ is a set of all layers containing learnable parameters. Thus, the first set, $\mathbf{G}_1$, contains precisely the elements for *CONV* and *FC* weights as well as the activations. On the other hand, the second set, $\mathbf{G}_2$, contains the elements for the learnable parameters not in $\mathbf{G}_1$. Note that there is no need to quantize the activations of the output layer, whereas the activations of other layers, including the input layer, are potential for bit-precision level reduction.

Another important issue that the proposed framework takes into account is the effect of quantizing each data-structure on the memory footprint. As shown in Figure 4.3, the data-structures vary greatly in their need for memory space. One can also note that majority of learnable parameters are concentrated in deep layers. Hence, for instance, reducing the bit-precision of AlexNet *CONV*-1 and *FC*-6 weights to the next level reduces the memory requirements by 4.25 KB and 4.72 MB, respectively.

As it is evident from the given example, the contribution of learnable parameters in deep layers to the compression rate is much more than that of the shallow layers when both are quantized to the next bit-precision level. Accordingly, the proposed framework adopts the two-level clustering approach [291] to represent the data-structures for wordlength reduction based on their type and size, such that

$$\begin{aligned} \mathcal{G}^{(t)} &= \left\{ \mathcal{G}_{\nu_1}^{(t)}, \mathcal{G}_{\nu_2}^{(t)}, \dots, \mathcal{G}_{\nu_{n_t}}^{(t)} : \forall (d_i, \ell_i) \in \mathbf{G}_t, (d_i, \ell_i) \in \mathcal{G}_{\nu_j}^{(t)} \iff \nu_{\ell_i}^{(d_i)} = \nu_j, \right. \\ &\quad \left. t \in \{1, 2\}, 1 \leq i \leq m, m = |\mathbf{G}_t|, 1 \leq j \leq n_t, \right. \\ &\quad \left. n_t = \left| \left\{ \nu_{\ell_1}^{(d_1)}, \nu_{\ell_2}^{(d_2)}, \dots, \nu_{\ell_m}^{(d_m)} \right\} \right| \right\} \end{aligned} \quad (4.23)$$

In other words, the first clustering level maps a data-structure represented as a

2-tuple (d, ℓ) into either $\mathcal{G}^{(1)}$ or $\mathcal{G}^{(2)}$ clusters on the basis of data-structure type as discussed in Equation (4.22). Then, in the second clustering level, the tuples within each Level-1 cluster are grouped into n_t sub-clusters based on their sizes, i.e., n_t -sizes clustering. Consequently, a Level-2 cluster, say $\mathcal{G}_{\nu_j}^{(t)}$, contains all elements in $\mathbf{G}_t$ whose data-structure size is ν_j . Here, n_t is the number of unique sizes in $\mathbf{G}_t$.

4.4.4.2 Clusters Traversal

The order of picking data-structures for quantization plays an important role in achieving rigorous compression for DNNs designed with accuracy degradation constraint. For this purpose, the proposed framework imposes to visit cluster $\mathcal{G}^{(1)}$ before cluster $\mathcal{G}^{(2)}$. Within each Level-1 cluster, the sub-cluster with the larger ν is given a higher priority for reducing the bit-precision level of its data-structures. When visiting a Level-2 cluster, the proposed framework carries out a progressive wordlength reduction to its data-structures, one data-structure at a time, until no further reduction is possible. Thereafter, it advances to the next sibling cluster.

When any of the data-structure belonging to the currently visited Level-1 cluster succeed in reducing the bit-precision level, the proposed framework retraverses all of its Level-2 clusters again. Otherwise, it advances to visit the other Level-1 cluster. In this context, we define an *episode* as a single pass through $\mathcal{G}^{(1)}$ and $\mathcal{G}^{(2)}$ clusters. The cluster traversal algorithm continues to iterate between $\mathcal{G}^{(1)}$ and $\mathcal{G}^{(2)}$ clusters until it converges and no changes in the bit-precision level are observed in an *episode*.

Algorithm 6: Bit-Precision Level Reduction - Part 1.

Input: A quantized network structure $\mathbf{Net}_q$ and model $\{\mathbf{W}, \mathbf{B}\}$ using QPs sets $\{\alpha, \beta, \mathcal{S}\}$, a reference FP32 model $\{\mathcal{W}, \mathcal{B}\}$ and accuracy Acc_{FP32} , an accuracy degradation threshold Δ_{thld} , and a validation set $\mathcal{T}$.

Output: A mixed low-precision deep neural network.

Procedure QReduction($\mathbf{Net}_q, \{\mathbf{W}, \mathbf{B}\}, \{\alpha, \beta, \mathcal{S}\}, \{\mathcal{W}, \mathcal{B}\}, \text{Acc}_{\text{FP32}}, \Delta_{\text{thld}}, \mathcal{T}$):

```

1  Cluster the data-structures using (4.22, 4.23)
    $\mathcal{G}^{(1)} \leftarrow \{\mathcal{G}_{\nu_1}^{(1)}, \dots, \mathcal{G}_{\nu_{n_1}}^{(1)}\}, \mathcal{G}^{(2)} \leftarrow \{\mathcal{G}_{\nu_1}^{(2)}, \dots, \mathcal{G}_{\nu_{n_2}}^{(2)}\}$ 
2  Sort Level-2 clusters in decreasing order such that
    $\nu_1 > \nu_2 > \dots > \nu_{n_t}, \forall t \in \{1, 2\}$ 
3  Initialize the traversal algorithm
    $visit[i] \leftarrow true, \forall i \in \{1, 2\}$  ▷ Visit all clusters
    $t \leftarrow 1$  ▷ Start with  $\mathcal{G}^{(1)}$ 
    $\lambda \leftarrow 0$  ▷ Restrict the degradation threshold
4  repeat ▷ Level-1 traversal
5  |    $visit[t] \leftarrow false$ 
6  |    $j \leftarrow 1$ 
7  |   while  $j \leq n_t$  do ▷ Level-2 traversal
8  |   |   Select a candidate data-structure in  $\mathcal{G}_{\nu_j}^{(t)}$  cluster
   |   |    $(d, \ell, \alpha_{\text{new}}, \beta_{\text{new}}), change \leftarrow \text{Reduce}(\mathcal{G}_{\nu_j}^{(t)}, \Delta_{\text{thld}} \cdot \mathcal{I}(t \neq 2), \lambda, \dots)$ 
9  |   |   if  $change$  then ▷ Perform the reduction
10 |   |   |   Update the quantization parameters
   |   |   |    $\alpha_{\ell}^{(d)} \leftarrow \alpha_{\text{new}}, \beta_{\ell}^{(d)} \leftarrow \beta_{\text{new}}$ 
11 |   |   |   Quantize the data-structure  $(d, \ell)$  using (4.10, 4.13)
12 |   |   |   if  $\alpha_{\ell}^{(d)} + \beta_{\ell}^{(d)} = 1$  then ▷ Saturated data
13 |   |   |   |    $\mathcal{G}_{\nu_j}^{(t)} \leftarrow \mathcal{G}_{\nu_j}^{(t)} - \{(d, \ell)\}$ 
14 |   |   |   end
15 |   |   |    $visit[i] \leftarrow true, \forall i \in \{1, 2\}$ 
16 |   |   else ▷ Advance to next Level-2 cluster
17 |   |   |    $j \leftarrow j + 1$ 
18 |   |   end
19 |   end
20 |   if not  $visit[t]$  then ▷ Advance to other Level-1 cluster
21 |   |    $t \leftarrow t + (-1)^{t+1}$ 
22 |   end
23 |    $\lambda \leftarrow 1$ 
24 until not  $visit[t]$ 
25 return  $\{\mathbf{W}, \mathbf{B}\}, \{\alpha, \beta, \mathcal{S}\}$ 

```

Algorithm 7: Bit-Precision Level Reduction - Part 2.

Input: A quantized network structure Net_q and model $\{\mathbf{W}, \mathbf{B}\}$ using QPs sets $\{\alpha, \beta, \mathcal{S}\}$, a reference FP32 model $\{\mathcal{W}, \mathcal{B}\}$ and accuracy Acc_{FP32} , a quantized data-structure set $\mathcal{G}_{\nu_j}^{(t)}$, a degradation threshold Δ_{thld} and mask λ , and a validation set $\mathcal{T}$.

Output: A selected candidate data-structure for the wordlength reduction, if exists.

Procedure $\text{Reduce}(\mathcal{G}_{\nu_j}^{(t)}, \Delta_{\text{thld}}, \lambda, \text{Net}_q, \{\mathbf{W}, \mathbf{B}\}, \{\alpha, \beta, \mathcal{S}\}, \{\mathcal{W}, \mathcal{B}\}, \text{Acc}_{\text{FP32}}, \mathcal{T})$:

```

1   Initialize the reduction parameters
     $cand \leftarrow \{\}, climb \leftarrow false$ 
2   start:
3   for all  $u \in \{1, \dots, U\}, U = |\mathcal{G}_{\nu_j}^{(t)}|$  in parallel do
4        $sol_u \leftarrow \{\}$ 
5       if  $climb$  then ▷ Hill climb reduction
6            $\theta_{high} \leftarrow \theta_{low} \leftarrow 2$ 
7       else
8           if  $\alpha_{\ell_u}^{(d_u)} + \beta_{\ell_u}^{(d_u)} \leq 8$  then ▷ Gradual
9                $\theta_{high} \leftarrow \theta_{low} \leftarrow 1$ 
10          else ▷ Aggressive
11               $\theta_{high} \leftarrow \alpha_{\ell_u}^{(d_u)} + \beta_{\ell_u}^{(d_u)} - 1, \theta_{low} \leftarrow 1$ 
12          end
13      end
14      while  $\theta_{low} \leq \theta_{high}$  do ▷ Binary search
15           $\theta_u \leftarrow \lceil (\theta_{low} + \theta_{high}) / 2 \rceil$ 
16          Optimize the quantization parameters
           $(\alpha_u^{new}, \beta_u^{new}, \text{Acc}_u^{new}) \leftarrow \text{QParamsOpt}(\alpha_{\ell_u}^{(d_u)}, \beta_{\ell_u}^{(d_u)} - \theta_u, d_u, \ell_u, \dots)$ 
17           $\Delta_u \leftarrow (\text{Acc}_{\text{FP32}} - \text{Acc}_u^{new})$ 
18           $\varphi \leftarrow \lambda + (1 - \lambda) \cdot \theta_u \cdot \nu_{\ell_u}^{(d_u)} / \sum_{\ell \in [L]} \nu_{\ell}^{(d_u)}$ 
19          if  $\Delta_u \leq \varphi \cdot \Delta_{\text{thld}}$  then ▷ Valid reduction
20               $sol_u \leftarrow \{(d_u, \ell_u, \theta_u, \Delta_u, \alpha_u^{new}, \beta_u^{new})\}$ 
21               $\theta_{low} \leftarrow \theta_u + 1$ 
22          else  $\theta_{high} \leftarrow \theta_u - 1$ 
23      end
24       $cand \leftarrow cand + sol_u$ 
25  end
26  if  $cand \neq \phi$  then ▷ Select a candidate for reduction
27      return  $\text{select}(cand), true$ 
28  else ▷ No candidate reductions
29      if not  $climb$  then ▷ Hill climbing
30           $climb \leftarrow true$ 
31          go to start
32      else return  $(\text{None}, \dots, \text{None}), false$ 
33  end

```

4.4.4.3 Intra-Cluster Bit-Precision Level Reduction

At this stage, the framework receives a set data-structures, each in fixed-point representation, for the currently visited Level-2 cluster as $\mathcal{G}_{\nu_j}^{(t)} = \{(d_1, \ell_1), \dots, (d_U, \ell_U)\}$, and then, a data-structure, say (d_u, ℓ_u) , is chosen and quantized to one of the next bit-precision levels from $(\alpha_{\ell_u}^{(d_u)} + \beta_{\ell_u}^{(d_u)} - 1)$ to 1, where $1 \leq u \leq U$, and $U = |\mathcal{G}_{\nu_j}^{(t)}|$. It is noteworthy that during wordlength reduction process, data-structures belonging to $\mathcal{G}_{\nu_j}^{(t)}$ may have different bit-precision levels ranging from 32 to 2.

Here, all the data-structures in $\mathcal{G}_{\nu_j}^{(t)}$ compete with each other to be picked and quantized to a next bit-precision level. More precisely, each data-structure is assigned its next level of bit-precision, and then, quantized as discussed in Section 4.4.2, and finally the quantization parameters are optimized based on the discussion in Section 4.4.3. Note that the reduction in the bit-precision level is done through lessening the fractional length. As thus, to reduce the bit-precision level of data-structure (d, ℓ) by θ bits, the bit-precision reduction framework updates $\beta_{\ell}^{(d)}$ as $\beta_{\ell}^{(d)} = \beta_{\ell}^{(d)} - \theta$. When a data-structure is quantized to the minimum bit-precision level, it will be removed from its Level-2 cluster and therefore will not be considered for further reduction.

To determine the next bit-precision level for data-structure (d, ℓ) , we advocate the use of the gradual reduction, i.e., $\theta = 1$, when $\alpha_{\ell}^{(d)} + \beta_{\ell}^{(d)} \leq 8$. On the other hand, we adopt the use of the binary search algorithm to look for the lowest acceptable bit-precision level, in terms of accuracy, when $\alpha_{\ell}^{(d)} + \beta_{\ell}^{(d)} > 8$. While the aggressive reduction speeds up the reduction process in the initial stages, the gradual reduction prohibits the potential early presence of bottleneck quantization layer [292]. Such a

layer prevents other data-structures in the pipeline from being quantized to the next level without violating the accuracy degradation constraint.

Upon completion of the competition, the information about all the data-structures that successfully reduced their bit-precision level without violating the accuracy degradation threshold (Δ_{thld}) will be in the candidate set, referred to as *cand*. These information include the optimum quantization parameters ($\alpha_u^{\text{new}}, \beta_u^{\text{new}}$) and the amount of accuracy drop (Δ_u) after reducing the bit-precision level of data-structure (d_u, ℓ_u) by an amount of θ_u bits. If more than one candidate was observed during any reduction stage, only one candidate is accepted. Therefore, we defined the operation **select**(*cand*) to be the selection of the candidate sol_u as $(d_u, \ell_u, \alpha_u^{\text{new}}, \beta_u^{\text{new}})$ based on the following ranking order: (i) reduction amount ($\gg \theta$), (ii) amount of accuracy drop ($\ll \Delta$), (iii) layer ID ($\gg \ell$), and (iv) data-structure type (w , then a , then b).

On the other hand, we apply the hill climbing optimization technique [191] when no candidates are found for wordlength reduction. Hill climbing heuristic is adopted to avoid getting stuck in a local optima quantized DNN solution through accepting a bad wordlength reduction solution. Here, we need to emphasize that while the framework tries to reduce bit-precision level, it also tries smartly and innovatively to add a positive error that reduces the overall network error by selecting α and β which results in the lowest $\mathcal{F}$ as mentioned in Equation (4.21). We found hill climbing to be very useful for this.

It is worth noting that reducing the wordlength of a data-structure can impact the level of quantization that other data-structures can reach. Thus, to prevent an early over-quantized data-structure, the bit-precision reduction framework rejects candidate

data-structures that cause degradation more than their size ratio of the degradation threshold times the reduction amount during the first traversal visit of Level-1 data-structures. In other words, a data-structure (d_u, ℓ_u) that is a member of $\mathcal{G}^{(1)}$ can be a candidate during the first traversal visit of $\mathcal{G}^{(1)}$ elements if and only if,

$$\Delta_u \leq \Delta_{\text{thld}} \cdot \theta_u \cdot \frac{\nu_{\ell_u}^{(d_u)}}{\sum_{\ell \in [L]} \nu_{\ell}^{(d_u)}} \quad (4.24)$$

Note that the bit-precision reduction framework allows $\mathcal{G}^{(2)}$ data-structures to reduce their bit-precision level only when they do not cause further accuracy degradation. As such, the proposed framework is accuracy-driven as it decides when and to what extent to quantize each data-structure in order to have the minimum accuracy degradation and the maximum compression rate. Furthermore, *FxP-QNet* does not require domain experts, and more important, it frees the human labor from exploring the vast search space to choose the appropriate quantization level for each data-structure in various layers.

4.5 Experiments and Results

In this section, we first describe our experimental settings and then proceed to analyze the rounding strategies used to determine the initial QPs. Next, we discuss and compare the performance of the proposed post-training self-distillation and network prediction errors function, denoted as $\mathcal{F}$, with the commonly used loss functions to further demonstrate the effectiveness of the proposed quantization parameters opti-

mization framework. We also investigate the sensitivity of data-structures in different layers to quantization. Then, we employ the proposed *FxP-QNet* framework to design mixed low-precision networks for three widely used typical DNNs and discuss the results and findings. Finally, we compare the mixed low-precision DNNs designed using *FxP-QNet* framework with those designed by the state-of-the-art low bit-precision quantization frameworks.

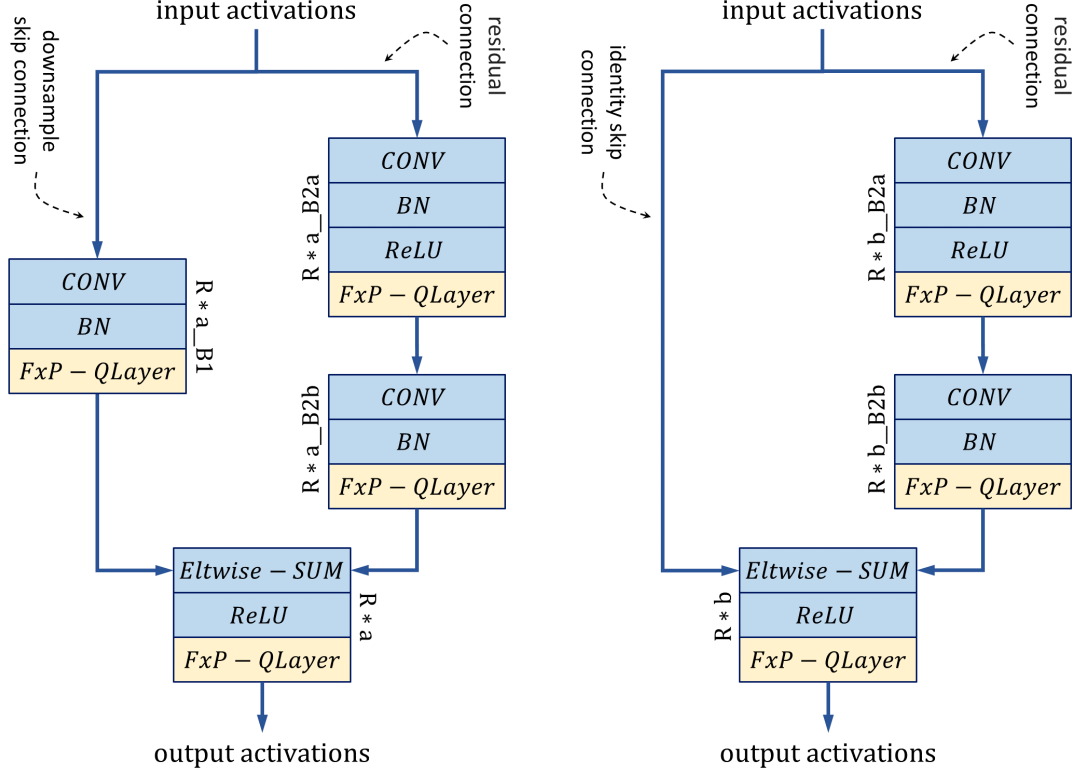
4.5.1 Experimental Settings

To demonstrate the versatility of *FxP-QNet* in designing mixed low-precision DNNs, we conduct extensive experiments on the ImageNet large scale visual recognition challenge (ILSVRC) dataset [28], which is known as one of the most challenging image classification benchmark so far. The ILSVRC-2012 ImageNet dataset has about 1.28 million images for training and 50 thousand images for validation, all are natural high-resolution images. Each image is annotated as one of 1,000 classes. To avoid using the training set, which is the motivation part for post-training quantization, samples from the ILSVRC-2012 validation set are used to do what is called calibration of QPs. Hence, it is referred to as the calibration set. The goal is to keep the calibration set as small as possible. Using the validation images, we report the evaluation results with two standard measures; top-1 accuracy/error-rate and top-5 accuracy/error-rate.

For those experiments, we used our *FxP-QNet* to design mixed low-precision networks for three benchmark architectures, namely, AlexNet [26], VGG-16 [27], and ResNet-18 [48], as they are well-known in the field of image classification and have been extensively studied in the literature. The *FxP-QNet* is implemented on top of

Caffe [58], a popular deep learning framework. The pre-trained full-precision models are taken from the Caffe Model Zoo without any fine-tuning or extra retraining. For the ResNet-18 model, we use the publicly available re-implementation by Facebook. Details of ResNet-18, VGG-16, and AlexNet architectures are presented next.

- **AlexNet** consists of five *CONV* layers and three *FC* layers. Each of these layers, except for the last *FC* layer, is followed by *BatchNorm* and *ReLU* layers. In addition, three *MAX-POOL* layers are employed with the first, second, and last *CONV* layers.
- **VGG-16** is similar to AlexNet architecture in terms of the number of *FC* layers. However, it contains five groups of *CONV* layers, and each group is followed by a *MAX-POOL* layer. The first and second groups consist of two *CONV* layers each, while each of the last three groups consist of three *CONV* layers according to model-D configurations.
- **ResNets-18** is a five-stage network followed by an *AVG-POOL* and finally a *FC* layer. The first stage consists of *CONV*, *BatchNorm*, *ReLU*, and *MAX-POOL* layers. Each of the remaining four stages consists of two blocks, where layers in the residual connection of each block learn the residual, hence the name residual network. The first residual block contains a downsample skip connection, while the second residual block uses an identity skip connection. Figure 4.6 illustrates the structure of these residual blocks after inserting *Fxp-QLayers* for activations quantization.



(a) Downsample skip connection block.

(b) Identity skip connection block.

Figure 4.6: The different types of basic residual blocks used in ResNet-18; (a) with downsample skip connection, and (b) with identity skip connection. The name of each computational block is also given where, for instance, the star symbol in the name $R \star a_B2b$ denotes the stage ID, $R \star a$ means the first residual block in this stage, $R \star a_B2$ refers to the second branch in this block, i.e., the residual skip connection, and finally $R \star a_B2b$ indicates the second computational block in this branch.

4.5.2 Analysis of Rounding Strategies used to Determine the

Initial Quantization Parameters

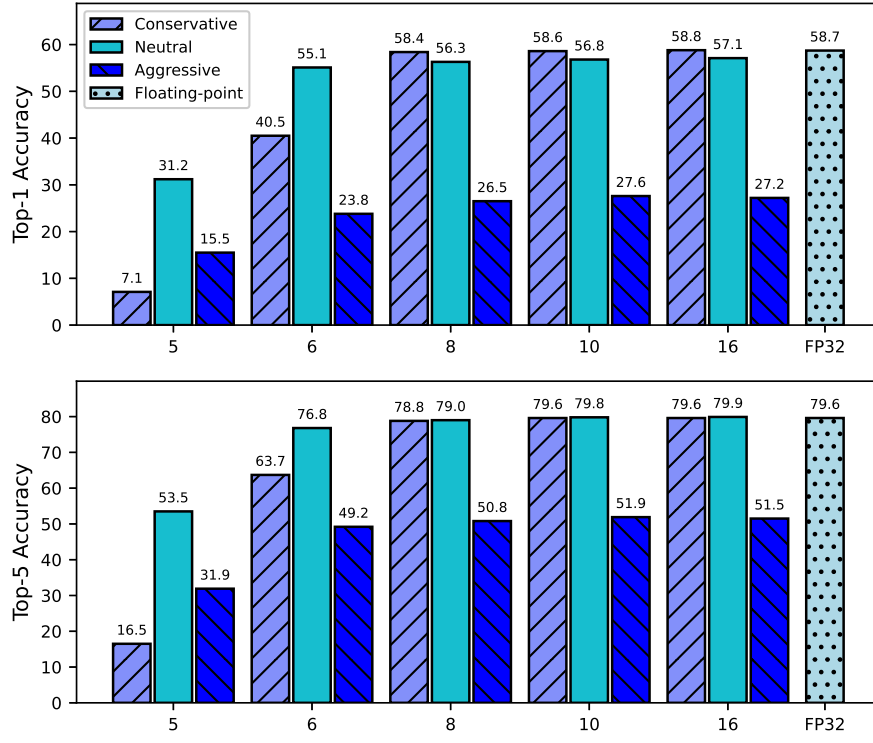
The design of a mixed low-precision DNN starts with the *preprocessor* that generates initial QPs for each data-structure based on a given wordlength and round strategy as discussed in Equation (4.9). Specifically, *preprocessor* adopts three rounding strategies for this purpose; the conservative, neutral, and aggressive strategies. In this section, we evaluate the use of each rounding strategy in the performance of quantized DNNs

considering five bit-precision levels, namely, 16, 10, 8, 6, and 5 bits. Recall from Section 4.4.1.3 that these bit-precision levels are used to quantize $\mathbf{G}_1$ data-structures set defined in Equation (4.22), except for activations and weights in first and last layers, while all other data-structures are represented in the 32-bit fixed-point format.

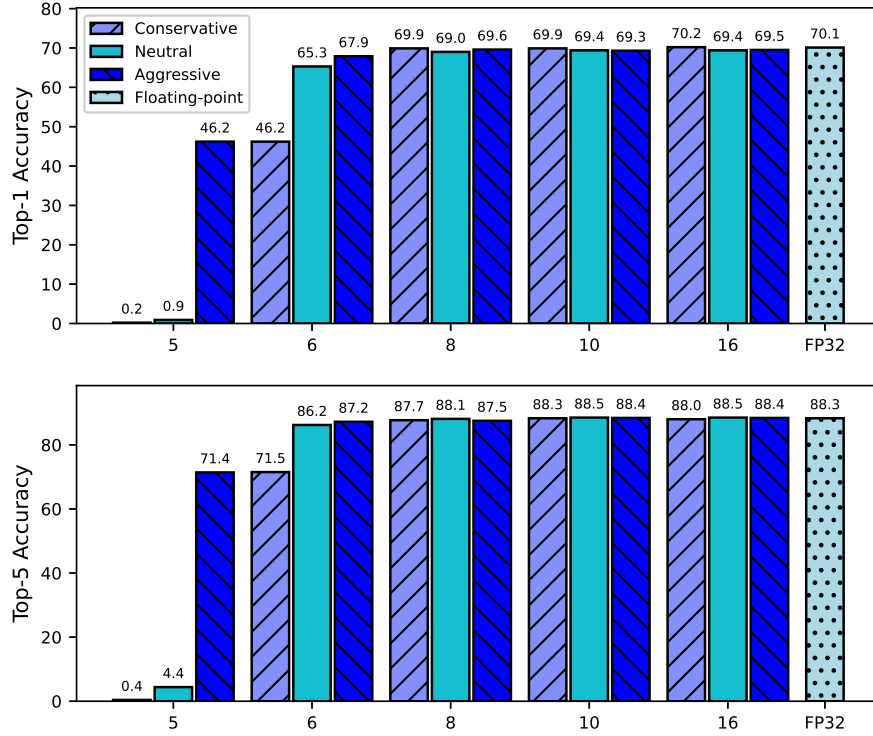
The data ranges, which are used for the quantization of activations, are collected after forwarding the network on the calibration set comprising of 800 images. Then, the top-1 and top-5 accuracies are measured on 200 test images. After 5 runs of each experiment, the average test accuracies for AlexNet, VGG-16, and ResNet-18 architectures are shown in Figure 4.7. Note that we use very small sets in this experiment to demonstrate the efficacy of the quantization solution proposed by *preprocessor* even with limited data.

First of all, we can observe that the rounding strategy plays a crucial role in determining network’s performance. One can also notice that the conservative strategy achieves better performance in all the considered architectures for 8-, 10-, and 16-bit fixed-point quantization. Focusing on 8-bit results, it is clear that the bit-precision reduction framework can start the design of mixed low-precision networks with an initial solution that causes less than 0.4% accuracy drop.

Going down to 6-bit quantized data, the obtained QPs by the neutral strategy increases the top-1 accuracy gap with those of the second-best strategy to 14.6% and 5.4% for AlexNet and ResNet-18, respectively. The top-5 accuracy results also show similar behavior where conservative strategy degrades AlexNet and ResNet-18 top-5 accuracies by 13.1% and 6.1%, respectively. On the other hand, the best results for the 6-bit initial solution of VGG-16 architecture come from the aggressive strategy



(a) AlexNet architecture.



(b) VGG-16 architecture.

Figure 4.7: The performance of (a) AlexNet, (b) VGG-16, and (c) ResNet-18 architectures after quantization using the initial QPs determined by the rounding strategies for each of the studied bit-precision levels.

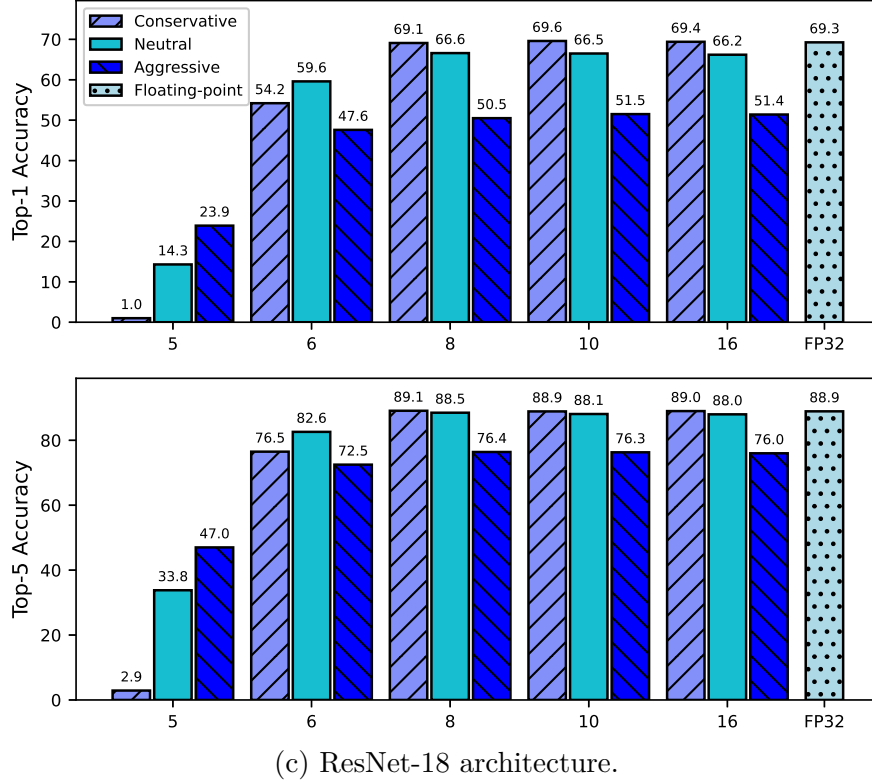


Figure 4.7: The performance of (a) AlexNet, (b) VGG-16, and (c) ResNet-18 architectures after quantization using the initial QPs determined by the rounding strategies for each of the studied bit-precision levels (continued).

as it has a 2.6% and 1% improvement for the top-1 and top-5 accuracies over neutral strategy, respectively.

For the 5-bit quantization, we can see that neutral strategy achieves the best top-1/5 results for AlexNet, whereas for VGG-16 and ResNet-18, the aggressive strategy performs better than other strategies. One can also note that the homogeneous quantization of activations and weights to 6 bits, or even fewer number of bits, without retraining causes a severe information loss. Thus, using 6 bits as a starting wordlength for the bit-precision reduction framework is not appropriate due to the significant accuracy drop. In subsequent experiments, we use the best strategy found in this section for each architecture and wordlength as *preprocessor* strategy to generate initial QPs

and report the numbers based on that. Note that we also analyzed several methods for rounding quantized data in Equation (4.10), such as the ceil, floor, round, and truncate, and found the round method to be the best. For the sake of brevity, we do not show these experiments.

4.5.3 Analysis of Optimization Strategies for Quantization Parameters

In this section, we demonstrate the effectiveness of the proposed post-training self-distillation and network prediction errors function, $\mathcal{F}$, defined in Equation (4.21), in optimizing QPs. In order to do that, we use Algorithm 5 as a standalone framework to optimize the QPs regardless of the optimization function mentioned in lines 5 and 14. We set the search space limit, K in line 8, to 5 so that the 10-nearest QPs neighbors are checked to make sure we get the one with the minimum *cost*. We compare $\mathcal{F}$ with three popular optimization functions; top-1/5 error-rate, $L1$ error, and $L2$ error. The top-1/5 error-rate is equal to one minus the top-1/5 accuracy. The $L2$ error, denoted by $\mathcal{L}$ in this chapter, is calculated as stated in Equation (4.14). On the other hand, the $L1$ error, also named mean absolute error, is computed as

$$L1(X, \tilde{X}) = \frac{1}{N} \cdot \sum_{i=1}^N |X_i - \tilde{X}_i| \quad (4.25)$$

where $\tilde{X}$ is the low-precision representation of X that is computed as mentioned in Equation (4.1), and N is the size of X . We use VGG-16 as a test case to compare the performance of these four optimization functions. Additionally, we use 50 images

Table 4.2: Homogeneous data-structure quantization using different optimization methods for designing 16-, 8-, and 6-bit fixed-point network of VGG-16 architecture.

Configuration			Top-1/5 Error	Range Eqn. (4.9)	$L1$ Eqn. (4.25)	$\mathcal{L}$ Eqn. (4.14)	$\mathcal{F}$ Eqn. (4.21)
16-bit Fixed-Point	Calib	Top-1	74.29%	70.86%	71.50%	70.86%	70.86%
		Top-5	90.86%	91.14%	91.75%	91.14%	91.14%
	Test	Top-1	66.51%	67.83%	67.68%	67.83%	67.83%
		Top-5	88.09%	88.91%	88.80%	88.91%	88.91%
8-bit Fixed-Point	Calib	Top-1	75.43%	70.57%	55.43%	71.14%	69.71%
		Top-5	90.00%	91.14%	80.00%	91.14%	91.14%
	Test	Top-1	65.14%	67.66%	55.74%	67.60%	67.83%
		Top-5	86.77%	88.86%	79.23%	89.14%	88.71%
6-bit Fixed-Point	Calib	Top-1	74.86%	66.29%	35.14%	67.43%	68.00%
		Top-5	87.14%	90.00%	57.71%	87.14%	88.86%
	Test	Top-1	62.17%	66.40%	33.20%	64.69%	66.83%
		Top-5	84.11%	87.63%	58.23%	86.26%	88.06%

to calibrate the QPs with a given optimization function and then test the classification accuracy of the quantized fixed-point network on 5,000 images. The size of the datasets is defined with the aim of measuring the effectiveness of the proposed optimization function even when given a very small sample set for calibration.

In the experiments, the *preprocessor* generates the initial QPs, referred to as the range-based QPs, to homogeneously quantize $\mathbf{G}_1$ data-structures into 16-, 8-, and 6-bit fixed-point numbers, whereas $\mathbf{G}_2$ data-structures are quantized to 32-bit fixed-point numbers. It is noteworthy that $\mathbf{G}_1$ denotes the weights of VGG-16 model as well as the activations of all layers, and on the other hand, $\mathbf{G}_2$ denotes the biases of VGG-16 model. Thereafter, the *forward optimizer* is used to optimize range-based QPs with a given optimization function. After running each experiment on 10 randomly selected datasets, the average calibration and test accuracies are reported in Table 4.2.

Focusing on $\mathcal{L}$ and the proposed $\mathcal{F}$ optimization functions, we see that they both achieve similar performance results for 16-bit quantization level. With 8-bit data quantization, $\mathcal{F}$ performs slightly better than $\mathcal{L}$ top-1 accuracy on test datasets. Going down to 6-bit, it can be seen that the quantized networks achieves top-1 accuracy of 66.83% and top-5 accuracy of 88.06% on test datasets by using our $\mathcal{F}$ optimization function. Comparatively, $\mathcal{L}$ optimization function achieves 64.69% and 86.26% in top-1 and top-5 accuracies on test datasets, respectively. In fact, $\mathcal{L}$ optimization function brings 1.71% and 1.37% decrease in initial solution top-1 and top-5 accuracies, respectively. Apparently, range-based quantized networks are better than low-precision networks optimized with $\mathcal{L}$ optimization function.

As $\mathcal{F}$ achieves better accuracy than optimization functions that directly and independently seek to minimize the quantization error for each layer and data-structure, this suggests that simply fitting a quantizer to its corresponding data distribution may not be optimal for task performance. As seen, $\mathcal{F}$ optimization function can quantize state-of-the-art network to 8-bit fixed-point format with negligible accuracy loss. Furthermore, the 6-bit data quantization using the proposed optimization function brings within 1% accuracy drop on test datasets. We believe this is due to the fact that our optimization function formulation is able to balance range-resolution trade-off effectively.

In Figure 4.8, we analyze the optimized clipping thresholds for a few 6-bit quantized weight data-structures in VGG-16 architecture, highlighting the importance of range-resolution trade-off. Remember that the dynamic range of a 6-bit number in signed 2's complement fixed-point representation is $[-32 \times 2^{-\beta}, 31 \times 2^{-\beta}]$, where β denotes the

fractional length. As seen with *CONV1_1* layer, the calibrated clipping thresholds move out from their initialized values by 1 integer bin in the log-domain, favoring dynamic range over resolution. For other layers, the clipping thresholds move in from their initialized values by up to 2 integer bins in the log-domain, which in turn leads to a higher resolution and, therefore, smaller round-off error upon quantizing the data within the clipping thresholds. From these results, we argue that for better accuracy of quantized models, weights in shallow layers should focus on saturating their range. On the other hand, as we go deeper into the network, layers should focus more on increasing their resolution.

For top-1/5 error-rate when used as an optimization function, it achieves the best accuracy results on calibration sets for all considered quantization levels, but its low-precision networks suffer significant accuracy drop when tested with unseen samples. Hence, the generalization ability of $\mathcal{F}$ -based quantized networks is better than that of the error-rate-based. For $L1$ optimization function, 16-bit data quantization incurs almost no accuracy drop on test sets. Going down to 8- and 6-bit fixed-point weights and activations, the quantized networks with $L1$ optimization function crash. The reason for this is due to the low sensitivity of $L1$ optimization to outliers which have a potentially important impact on the underlying dot-product operation. Hence, $L1$ optimization is not a robust error measure for finding the best design for low-precision networks. To summarize, the experimental results show that the proposed optimization function achieves better performance than other tested optimization functions in all of the three bit-precision cases.

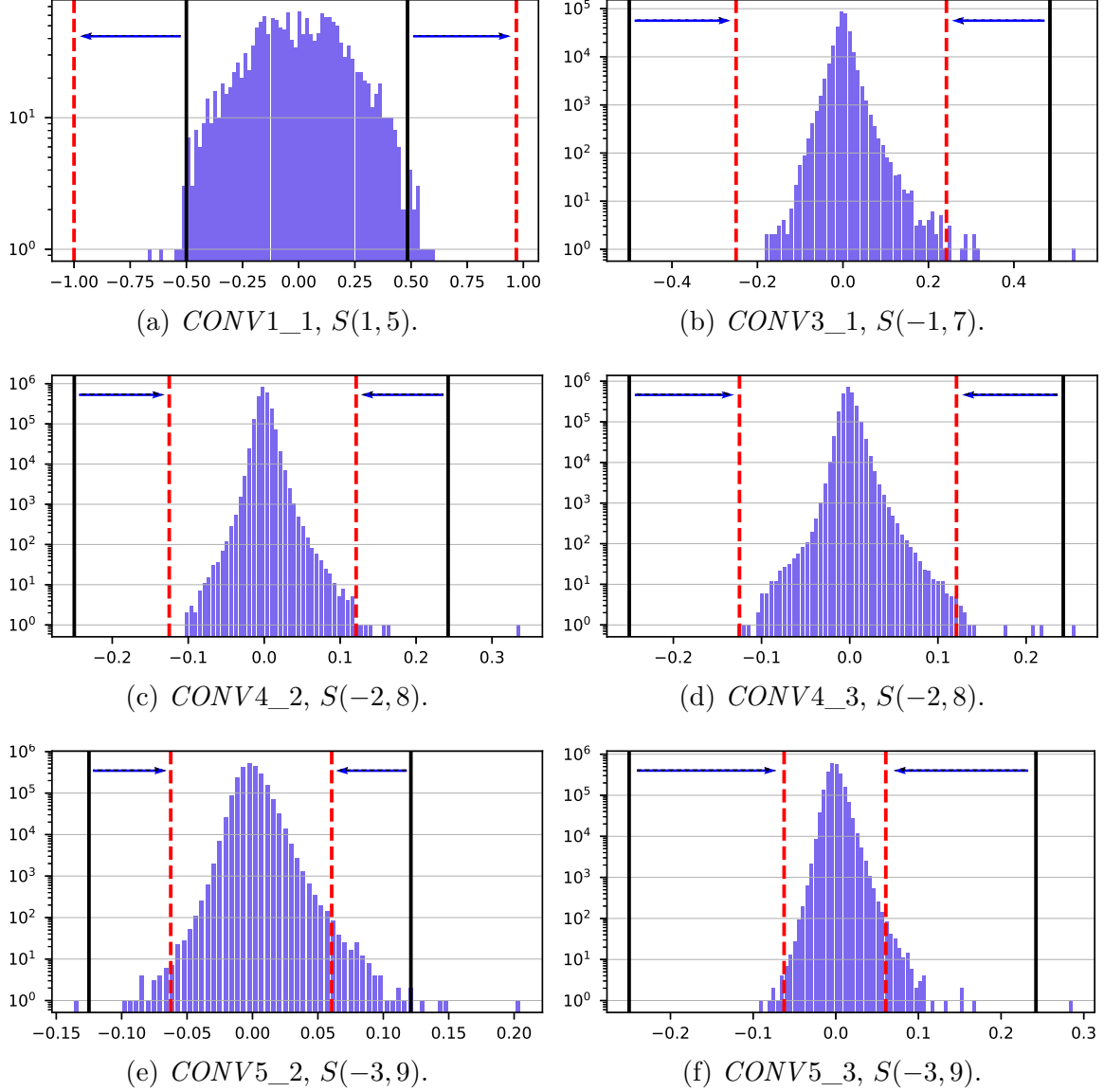


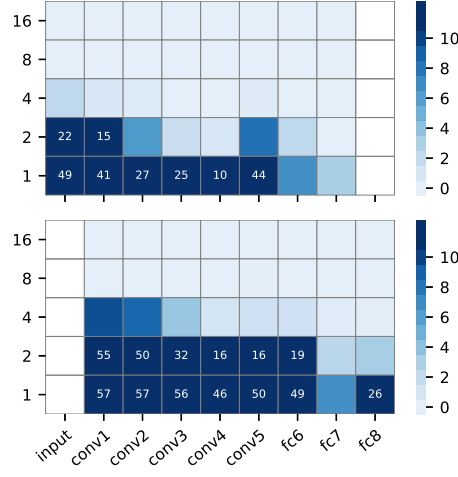
Figure 4.8: Weight distributions of VGG-16 layers that changed their clipping thresholds by non-zero integer amount in log-domain. Initial and calibrated clipping thresholds are indicated as black-solid and red-dashed lines, respectively. The optimized fixed-point representation for each of these layers are also given.

4.5.4 Analysis of Data-Structures Sensitivity to Quantization

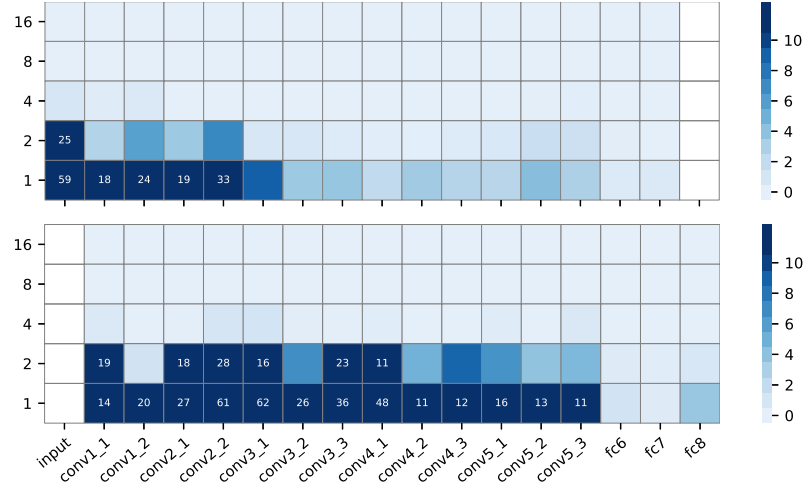
In this experiment, we investigate the role each layer plays in forming a good predictor. To this end, we designed a test that shows the robustness of each layer to quantization. The test differentiates between activations and weights robustness. For each layer, we quantize its weights and output activations separately, while keeping the data-structures for other layers intact. After optimizing the QPs, the performance of the network is evaluated by measuring the top-1 error-rate. To measure the robustness of activations/weights in the ℓ -th layer, we compare the measured top-1 error-rate to that of the original full-precision network. When a network exhibits a significant increase in top-1 error-rate after quantizing ℓ -th layer activations/weights, we say that the ℓ -th layer activation/weight data-structure is sensitive to quantization, and otherwise, it is called robust to quantization.

Each data-structure is quantized into five levels; 16-, 8-, 4-, 2-, and 1-bit fixed-point format. A dataset of size 5,000 images is used for QPs calibration. Thereafter, top-1 error-rate is measured on a test dataset containing another 5,000 images. Observing the robustness results, as shown in Figure 4.9, several conclusions can be drawn; (i) overall, quantizing different DNN data-structures causes various levels of accuracy drop, (ii) data-structures in deep layers are more robust to quantization than the ones in shallow layers, and (iii) the bit-precision level can be reduced up to 8 bits without causing noticeable performance degradation.

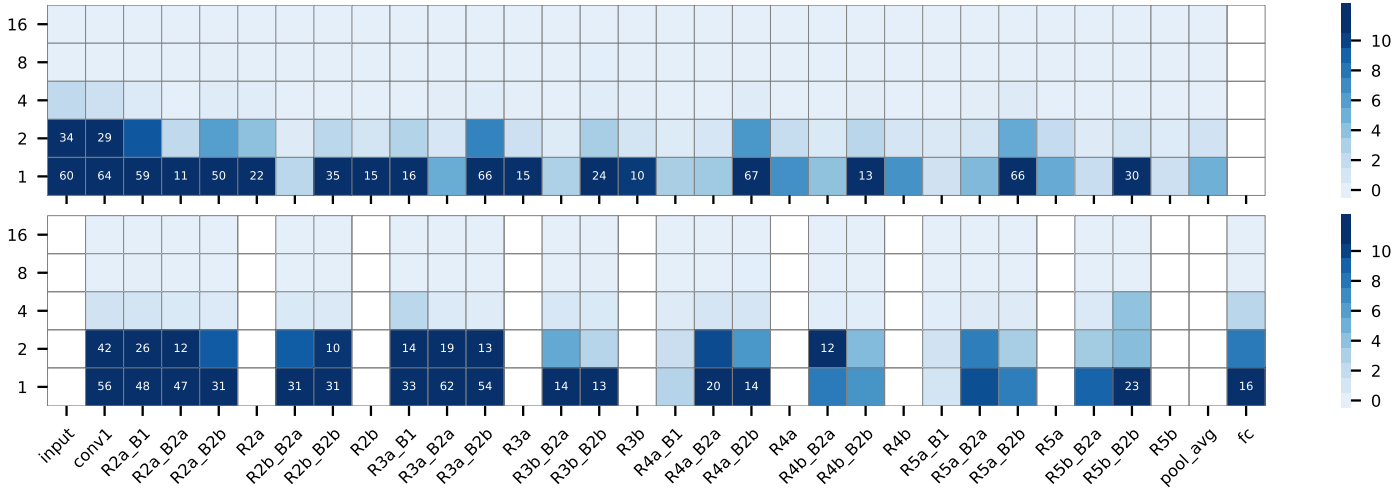
For AlexNet, we can see that almost all layers are sensitive to rigorous quantization with activations being slightly less sensitive than weights. A plausible explanation for



(a) AlexNet architecture.



(b) VGG-16 architecture.



(c) ResNet-18 architecture.

Figure 4.9: Data-structures robustness to quantization for (a) AlexNet, (b) VGG-16, and (c) ResNet-18. For each architecture, the *top* figure represents the activations robustness, while the *bottom* figure is designated for weights robustness. The rows of each figure correspond to the quantization level, while the columns designate robustness at different network layers. The white column means that the data-structure in that layer is not quantized because it is not needed.

this could be attributed to the size of AlexNet architecture. When the network is small, all layers are vigilant participants in forming a good predictor. As the network size increases, which is the case for VGG-16, deep layers become more robust to quantization. The reason for this is due to more abstraction occurring in deep layers. More interestingly, the results show that VGG-16 *FC* layers, which contributes to about 89% of VGG-16 model size, can use aggressively quantized weights.

With regard to ResNet-18 architecture, we found that the sensitivity patterns are not as easily pronounced as those of AlexNet and VGG-16. However, focusing on the weights of ResNet-18, we can interpret the *CONV* layers in the residual connections of each stage as a sub-network with sensitivity characteristics roughly similar to AlexNet and VGG-16. Furthermore, by comparing the results of the *CONV* layer in the downsample skip connection, i.e., weights in *R★a_B1*, at the different residual stages, we can see that the robustness increases for the deep layers in a similar fashion to what was observed in the weights of VGG-16. Concerning activations sensitivity, unlike AlexNet and VGG-16 architectures that place sensitive layers at the bottom of the network, ResNet-18 distributes them across the network. More precisely, we can notice that the activation data-structure of the last *CONV* layer in each residual connection is more sensitive to quantization than that of the other layers. This could be potentially attributed to the higher dynamic range of activations in these layers.

According to the results and findings that we observed in this section, the layers in a DNN should not be treated in a monolithic way. Furthermore, we argue that better preservation of sensitive data-structures information, through higher wordlength allocation, boosts the performance of a quantized DNN evidently. At the same time, it

allows for further compression in deep layers which are likely to constitute the highest ratio of model size as seen with VGG-16. Therefore, in the proposed framework, we check the sensitivity of the data-structure to quantization, line 24 in Algorithm 5, and accordingly either accept or reject the reduction, lines 19 – 24 in Algorithm 7 - Part 2. Also, we can clearly see that the last *FC* layer in each architecture is less robust to quantization than the *FC* layers preceding it, if any. Thus, the *preprocessor* generates an initial solution where the weights and input activations of the last *FC* layer are at 32-bit fixed-point bit-precision level.

4.5.5 Mixed Low-Precision Quantization

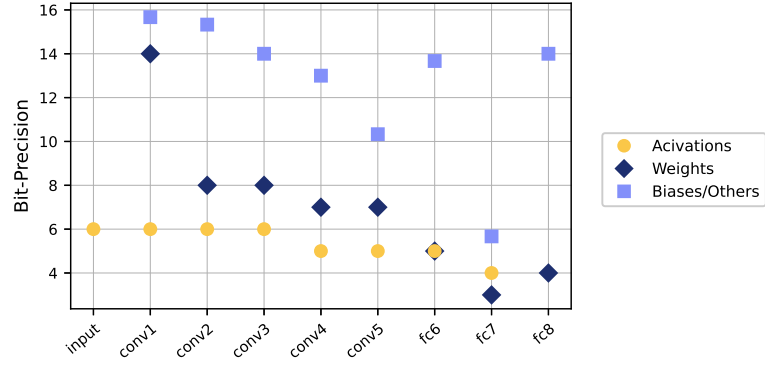
Unlike previous sections where only *preprocessor* or both *preprocessor* and *forward optimizer* units were used for evaluation, this section employs all the components of *FxP-QNet* to design and evaluate a mixed low-precision DNN. We should emphasize here that our work focuses on a hardware-friendly design of quantized DNNs that maps well on generic fixed-point hardware. We use the full ILSVRC-2012 ImageNet validation dataset to evaluate the performance of quantized DNNs, while a sample set of 1,000 images is used to calibrate the quantization parameters. The top-1/5 accuracy degradation threshold is set to 1%, 1%, and 2% for AlexNet, VGG-16, and ResNet-18, respectively. We assign a higher accuracy degradation threshold to ResNet-18 due to its complex structure and lighter model with significantly less parameters compared to AlexNet and VGG-16.

4.5.5.1 Wordlength Allocation and Network Performance

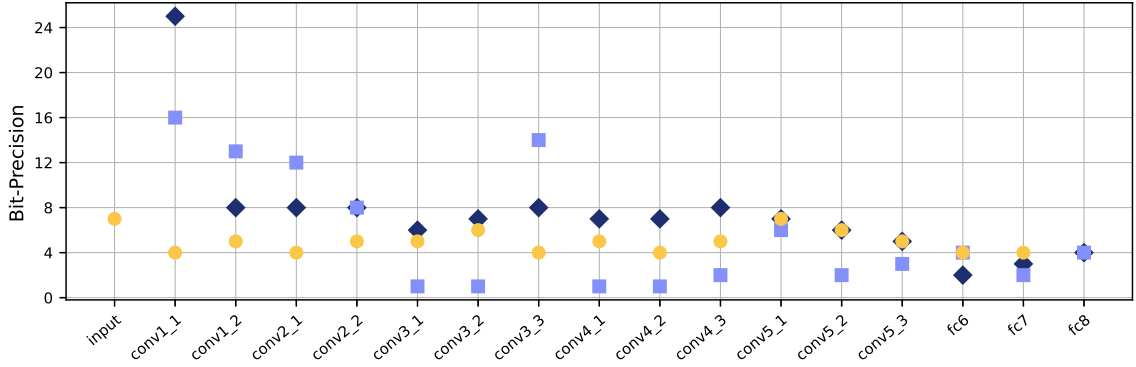
In Figure 4.10, we visualize the allocated wordlength for the data-structures of each layer in AlexNet, VGG-16, and ResNet-18 architectures. As can be seen, *FxP-QNet* shows a spectrum of varying wordlength allocations to the layers. In all cases, there is significant heterogeneity in the wordlength assignment. From the results, we can also notice that the number of bits allocated to each data-structure is consistent with our findings in the sensitivity analysis. Thus, we conclude that *FxP-QNet* succeeds in distinguishing different layers and their varying robustness to quantization while choosing their respective wordlength.

Furthermore, we can see that all MAC operations are performed on a maximum of 8-bit numbers, except for the first *CONV* layer. We believe that assigning higher bits to the weights of the first layer allows for more compression and higher performance for the quantized DNNs. Given that the *FC*-6 layer performs a 102.76 mega floating-point multiply operation, with 2-bit integer weights, it is now possible to replace these expensive operations with simple integer add-subtract operations resulting in a significant improvement in computation and energy efficiency.

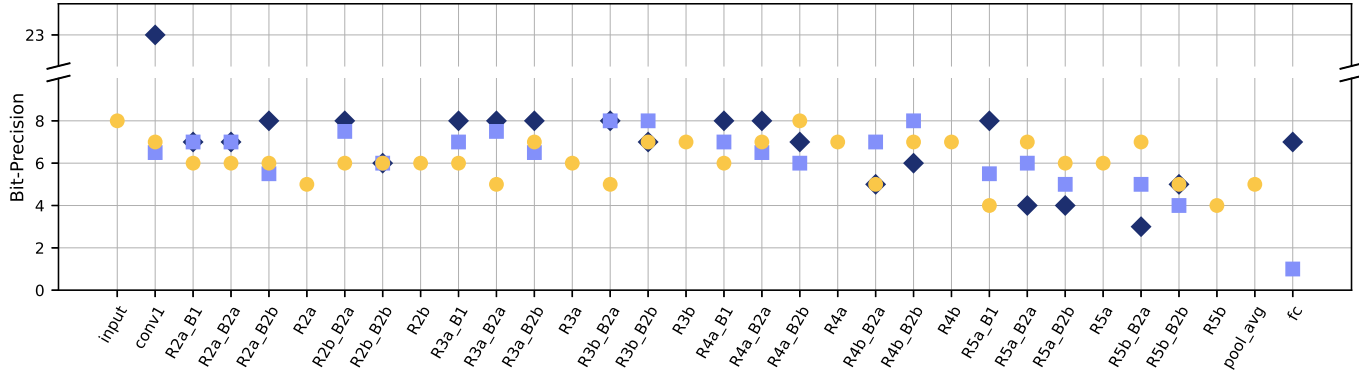
We can also observe that *FxP-QNet* assigns less bits to the weights of *FC* layers than that of *CONV* layers in AlexNet and VGG-16. Intuitively, this is because the size as well as the robustness of the former is much more than the latter. Remember that big data-structures, in terms of memory, are visited early for quantization, and at that time, the *network designer* has a relatively large space to work on quantization before the accuracy degradation threshold is violated. Comparatively, the weights of



(a) AlexNet architecture.



(b) VGG-16 architecture.



(c) ResNet-18 architecture.

Figure 4.10: Quantization wordlengths allocated to the data-structures of each layer for (a) AlexNet, (b) VGG-16, and (c) ResNet-18 architectures. For Biases/Others data-structures, we show the average allocated wordlength in each layer in case of more than one parameter exist in that layer.

the *FC* layer in ResNet-18 are allocated with 3 more bits, on average, than the weights of *CONV* layers in the residual connections of the last stage due to the same reasons.

The results also show that the first and last layers can be quantized to lower bit-precision levels without having a drastic effect on the accuracy. For instance, the proposed framework was able to reduce the bit-precision level of the first layer weights and activations of AlexNet by 18 and 26 bits from their initial 32 bits level while degrading the top-1 accuracy only by 0.06% and 0.08%, respectively. With regards to the last layer of AlexNet, the proposed framework successfully reduced the wordlength to 4 bits for both the activations and the weights. This reduction was accompanied by a 0.12% and 0.02% increase in top-1 and top-5 error-rates, respectively.

A summary of the average bit-precision level assigned to each data-structure as well as the classification accuracy achieved for the evaluated architectures is presented in Table 4.3. Because the post-training quantization works available in the literature for integer-only deployment do not exceed the 8-bit limit, we make a comparison with the accuracy of the full-precision baselines. Accuracy of FP32 baselines is reported as validated on our end. Additionally, we compare the performance of the mixed-precision networks obtained by *FxP-QNet* with the uniform-precision networks designed by the *preprocessor* and *forward optimizer*.

More precisely, the *preprocessor* is used to design a uniform network, in which the bit-precision level of each data-structure is set to the average bit-precision level obtained by *FxP-QNet*. Thereafter, we employ the *forward optimizer* to further improve the uniform networks designed by the *preprocessor*. Note that we set the search space limit in Algorithm 5 to 1 in order to speed up the optimization process. This is

Table 4.3: Quantization results of AlexNet, VGG-16, and ResNet-18 on ImageNet validation dataset. The baseline network performs operations with FP32 type. In comparison, the quantized DNNs perform operations with integer arithmetic. Furthermore, we compare the performance of mixed-precision networks designed by *FxP-QNet* with the networks that assign the same number of bits to all layers.

Architecture	Benchmark	Bit-Precision			Accuracy	
		Activations	Weights	Biases	Top-1	Top-5
AlexNet	Baseline (Floating-Point)	32	32	32	57.20%	79.68%
	Uniform (Preprocessor)	6	5	11	28.95%	50.99%
	Uniform (Forward Optimizer)				50.64%	74.13%
	<i>FxP-QNet</i> (Mixed-Precision)	5.85	4.48	10.33	56.25%	78.80%
VGG-16	Baseline (Floating-Point)	32	32	32	68.35%	88.44%
	Uniform (Preprocessor)	5	3	4	0.11%	0.54%
	Uniform (Forward Optimizer)				18.41%	37.68%
	<i>FxP-QNet</i> (Mixed-Precision)	4.66	2.68	3.34	67.40%	87.50%
ResNet-18	Baseline (Floating-Point)	32	32	32	66.62%	87.49%
	Uniform (Preprocessor)	7	5	6	49.93%	74.85%
	Uniform (Forward Optimizer)				59.76%	83.00%
	<i>FxP-QNet</i> (Mixed-Precision)	6.32	4.81	5.52	64.63%	86.16%

because we found that searching only adjacent QPs with expanding the search space in the direction of minimum *cost* insures global optimum QPs when $\mathcal{F}$ is used as an optimization function.

First, for the uniform quantization, the networks designed by the *preprocessor* recorded relatively very low accuracy compared to the baseline, especially for VGG-16. It can be seen that the classification in VGG-16 is rarely performed correctly. On the other hand, we can observe the effectiveness of the *forward optimizer* in improving the performance of the homogeneous networks designed by the *preprocessor*. The *forward optimizer* increased the top-1 accuracy for AlexNet, VGG-16, and ResNet-18 by 21.69%, 18.30%, and 9.83%, respectively. However, there is still a large gap between the performance of the full-precision networks and uniformly quantized networks even

after optimizing the quantization parameters. Thus, a uniform assignment of the bits is not the desired choice for maintaining accuracy.

In contrast, in the case of the proposed *FxP-QNet*, it significantly reduced the wordlength of model parameters as well as intermediate activations while achieving comparable accuracy with full-precision baselines. Furthermore, the results show that the proposed *FxP-QNet* dramatically outperforms uniform quantization methods. For the top-1 accuracy on VGG-16, *FxP-QNet* improves the *forward optimizer* result by 48.99%. The gap to the full-precision model is only 0.95%. The results on AlexNet and ResNet-18 are similar, only 0.95% and 1.99% loss in top-1 classification accuracy is shown compared with the baseline models.

Thus, it is easy to conclude that mixed-precision quantization has advantages over its homogeneous counterparts in terms of performance and model size. It is noteworthy that despite our method not needing any retraining and fine-tuning, it still poses a very competitive quantization scheme. It can be directly deployed without any retraining or fine-tuning, making our quantization scheme highly appealing for hardware implementation.

4.5.5.2 Memory Requirements and Footprint

At any step of the inference, a pair of temporary activation matrices and the whole set of model parameters must be present in the memory. As the quantized input and output activations of a layer are temporarily stored, a read-write (RW) memory is usually used for this purpose. On the other hand, the quantized learnable parameters are not changed during the lifetime of DNN inference, so a read-only (RO) memory

is usually adopted to store the frozen parameters. It is worth noting that the last layer of each residual block in ResNet-18 performs an element-wise sum on the input activations from the skip connection and the residual connection. Thus, if considering a DNN consisting of L layers where M of them with learnable parameters, $M \leq L$, the above requirement is translated as

$$\text{RW}_R(\mathcal{Q}_a) = \max \left(\sum_{\ell_{\text{inp}} \in \text{INP}_\ell^{(a)}} \text{MFP}(a, \ell_{\text{inp}}, \mathcal{Q}_a) + \text{MFP}(a, \ell, \mathcal{Q}_a) \quad , \quad \forall \ell \in [L] \right) \quad (4.26)$$

$$\text{RO}_R(\mathcal{Q}_w, \mathcal{Q}_b) = \sum_{\ell \in [M]} \left(\text{MFP}(w, \ell, \mathcal{Q}_w) + \text{MFP}(b, \ell, \mathcal{Q}_b) \right) \quad (4.27)$$

where $\mathcal{Q}_d$ indicates whether the data-structure type d , $d \in \{a, w, b\}$ is in full-precision (FP32) or quantized to fixed-point (FxP) format, $\text{INP}_\ell^{(a)}$ is a set of all layers providing input activations to layer ℓ , and the function $\text{MFP}(d, \ell, \mathcal{Q}_d)$ returns the memory footprint in MB for the data-structure type d in layer ℓ based on its the size, ν , as well as its quantization status, $\mathcal{Q}_d$, as

$$\text{MFP}(d, \ell, \mathcal{Q}_d) = \frac{\nu_\ell^{(d)}}{2^{23}} \times \begin{cases} 32 & , \quad \mathcal{Q}_d = \text{FP32} \\ k_\ell^{(d)}, & \mathcal{Q}_d = \text{FxP} \end{cases} \quad (4.28)$$

where $k_\ell^{(d)}$ is the bit-precision level of the data-structure d in layer ℓ . Note that we neglected the memory footprint for the shift and *FxP-QLayer* parameters in the aforementioned computations because their size, in terms of bytes, is too small as we are applying a layer-wise quantization. Thus, we calculate the compression rate of

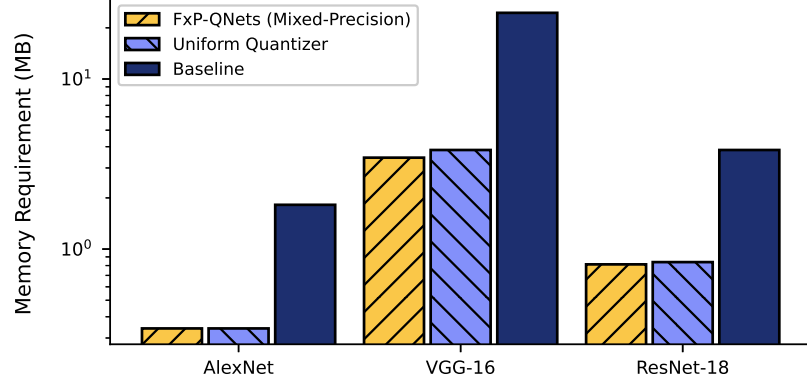
activations and model parameters as follows

$$\text{COMP}_{\text{Act}} = \text{RW}_R(\text{FP32}) / \text{RW}_R(\text{FxP}) \quad (4.29)$$

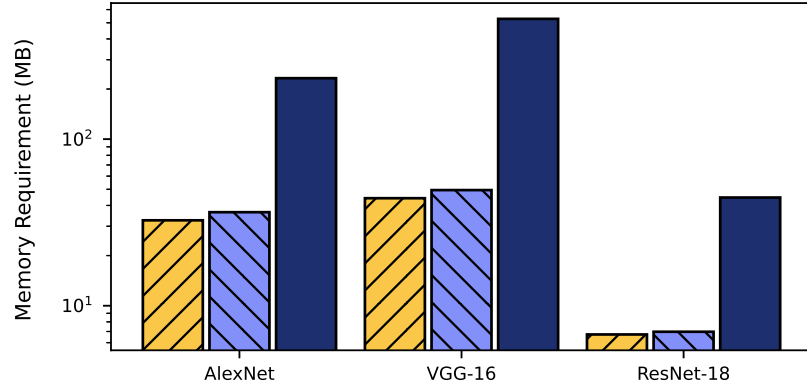
$$\text{COMP}_{\text{Model}} = \text{RO}_R(\text{FP32}, \text{FP32}) / \text{RO}_R(\text{FxP}, \text{FxP}) \quad (4.30)$$

Figure 4.11 reports the memory requirements of the proposed *FxP-QNet*. For the read-write memory requirement, the results show that *FxP-QNet* is 11% and 3% less memory-demanding than using the uniform quantization methodology for VGG-16 and ResNet-18, respectively. Furthermore, the read-only memory demand for *FxP-QNet* is 12%, 12%, and 4% less than uniform quantizer for AlexNet, VGG-16, and ResNet-18, respectively. Thus, it is easy to conclude that mixed-precision quantization has advantages over its uniform counterpart in terms of memory requirements. Nevertheless, the achievements of the uniform quantizer come at the cost of being less accurate than the mixed-precision quantizer.

Compared to baseline requirements, the quantized DNNs reduce the read-write and read-only memory requirements by about 1.5 MB and 200 MB for AlexNet, 21 MB and 483.5 MB for VGG-16, 3 MB and 38 MB for ResNet-18, respectively. To view from the off-chip bandwidth side, a 128-bit off-chip bandwidth can transfer 4 filter weight data per cycle when using full-precision weights. But after *FxP-QNet* quantization, it can transfer 28, 47, and 26 filter weight data per cycle, on average, for AlexNet, VGG-16, and ResNet-18 architectures, respectively, without extra power consumption for data-movement.



(a) Read-write memory.



(b) Read-only memory.

Figure 4.11: The requirement for (a) read-write memory, and (b) read-only memory, in mega byte (MB), for AlexNet, VGG-16, and ResNet-18 architectures.

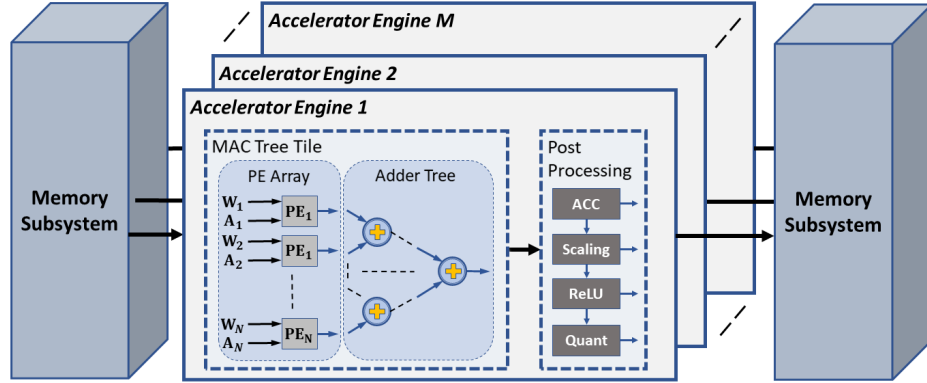
4.5.5.3 Hardware Efficiency

The proposed hardware architecture consists of M accelerator engines that work in parallel to increase the computational throughput. Each accelerator engine entails a MAC tree tile and a post-processing unit as illustrated in Figure 4.12a. The MAC tree tile contains N processing elements (PEs) and an adder tree. Each PE performs multiplication on a k_w -bit input weight filter and a k_a -bit input activation feature. Then, the resultant products are accumulated using the binary adder tree. On the other hand, the post-processing unit includes the hardware circuitry required for partial

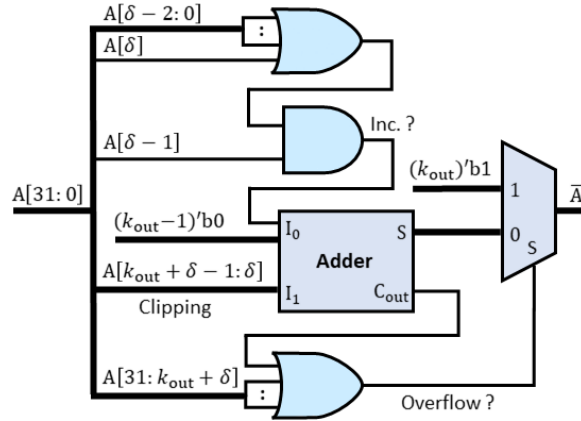
result and bias accumulation, *BatchNorm* and *AVG-POOL* scaling, *ReLU* activation, and quantization. Additionally, two memory subsystems are employed to derive the accelerator engines and hold output data.

We modeled the behavior of the hardware accelerator in Verilog and synthesized it using Xilinx Vivado Design Suite targeting the Artix-7 FPGA (XC7A100TCSG324-1). We kept all synthesis properties to their default. In this experiment, we evaluate the hardware efficiency when implementing ResNet-18 model. Precisely, we consider the test-case of implementing the hardware accelerator for *CONV* layers that require a large number of 3×3 MAC operations to produce final output activations. It is noteworthy that 3×3 is the most widely used kernel size in DNNs. The depth of the PE array is adjusted to achieve the highest possible utilization and data reuse, whereas the value of M is constrained by the hardware resources.

We compare the performance of the proposed *FxP-QNet* mixed-precision quantization with the conventional 8-bit linear quantization scheme, referred to as *Quant8*. The *Quant8* scheme implements each PE using an 8-bit fixed-point multiplier. After computing MAC tree output, *Quant8* converts the result to a floating-point number and performs the computations of the post-processing unit using IEEE-754 single format floating-point arithmetic. The arithmetic operations on the floating-point numbers are implemented using Vivado IP Integrator. Note that *Quant8* is a popular quantization technique in Pytorch and TensorRT [241]. Conversely, *FxP-QNet* scheme performs all the operations with fixed-point arithmetic, including the quantization process which is performed through *FxP-QLayer*. In doing so, PEs and *FxP-QLayers* are configured based on the wordlength allocation and the quantization parameters obtained from



(a) High-level architecture of the proposed hardware accelerator.



(b) *FxP-QLayer* hardware design.

Figure 4.12: The architecture of (a) the proposed hardware accelerator, and (b) *FxP-QLayer* based on the right-shift (δ) and output bit-width (k_{out}) parameters obtained during the design phase by *FxP-QNet* for ResNet-18 model.

Table 4.4: Implementation results for ResNet-18 on Artix-7.

Scheme		Num. PEs	LUTs	FFs	DSPs	Power (mW)	F_{max} (MHz)
Quant8		512	51,408	31,176	48	665	100
<i>Fxp-QNet</i>	LCL	384	25,448	11,772	6	363	135
	MCL	512	12,239	10,284	1	210	135

the bit-precision level reduction framework.

For *Fxp-QNet*, we consider two scenarios as it allocates a different bit-precision level to different layers. These scenarios evaluate the hardware accelerator when it is used to perform the computations of (i) the least compact layer (LCL), i.e., the *CONV* layer with the largest weights and activations wordlengths allocated, and (ii) the most compact layer (MCL), i.e., the *CONV* layer with the smallest weights and activations wordlengths allocated. We utilize the dynamic partial reconfiguration feature offered by modern FPGAs to configure the hardware accelerator at run-time. The overhead introduced by the reconfiguration can be compensated by using the reconfigured hardware accelerator many times, which is the case for the accelerator engines. Thus, the reconfiguration overhead becomes negligible with respect to the duration of the computation interval.

Table 4.4 shows the FPGA hardware resource usage in terms of the look-up tables (LUTs), flip-flops (FFs), and digital signal processors (DSPs). It also provides the power consumption, the maximum operating frequency (F_{max}), and the number of PEs implemented for each scheme. The number of PEs in *Quant8* is limited by the available resources in the device. Consequently, the hardware accelerator for *Quant8* scheme

is implemented with 512 PEs. On the other hand, *FxP-QNet* PEs is constrained by the allocated resources to the reconfigurable region. Therefore, the *FxP-QNet* LCL uses 25% less PEs while the *FxP-QNet* MCL employs the same number of PEs as in *Quant8*. Note that the size of the reconfigurable region, in terms of hardware resources, involves a trade-off between high acceleration and low reconfiguration overhead.

The effectiveness of *FxP-QNet* can be seen from the utilized resources. In the worst-case scenario, *FxP-QNet* uses 50.5%, 62.2%, and 87.5% less LUTs, FFs, and DSPs, respectively, than *Quant8* to design a hardware accelerator integrating 384 PEs. This is despite the fact that the operating frequency of *FxP-QNet* LCL is $1.35\times$ that of *Quant8*, and the former is $1.83\times$ better in saving power than the latter on average. Overall, it is evident that *FxP-QNet* has a comparable hardware efficiency for the worst-case allocated wordlength, but a significant improvement in hardware cost, power efficiency, and operating frequency for best-case wordlength allocated.

Concerning the overhead of quantizing activations, *Quant8* de-quantizes resultant activations from the MAC tile and then re-quantizes them back after *BatchNorm* computation due to the use of FP32 quantization scalars. That is why linear quantization techniques adopt floating-point arithmetic for the computations of post-processing unit, which consumes much energy and hardware resources. Hence, *Quant8* has a non-negligible quantization overhead. In contrast, *FxP-QNet* is tailored for integer-only deployment. Both *BatchNorm* and quantization are performed with integer arithmetic. Specifically, *FxP-QLayer* reduces the bit-precision level of a 32-bit integer input activation (the output of *BatchNorm* and *ReLU* layers) on-the-fly using an adder, a multiplexer, and logic gates, as shown in Figure 4.12b, and its implementation uses

LUTs only. The multiplexer selects outputs between the clipped-then-rounded value and the maximum representable value in case of an overflow. Consequently, the overhead of *FxP-QLayer* is negligible.

4.5.5.4 Comparison with Prior Work

In this section, we present the results of *FxP-QNet* and other quantization frameworks considering seven criteria as summarized in Table 4.5. For each framework, we indicate whether the inference computations are hardware-friendly because of using integer-only arithmetic or they include floating-point operations. We also mention whether the framework facilitates rapid deployment of quantized models or requires a training phase. Furthermore, we show the quantization granularity for each framework as it reflects the additional computational overhead during quantization. Moreover, we highlight whether the framework designs a homogeneous or a mixed-precision quantized DNN.

For the computation cost comparison, we refer to the average wordlength of input activations and weights used to perform the calculation of convolution and fully-connected layers. On the other hand, the compression denotes the amount of reduction in memory requirements for activations and model parameters. Last but not the least, since all the other reported works have different baselines, we use the degradation in top-1 and top-5 accuracy for comparison rather than using absolute accuracy numbers. Note that some of these frameworks report several results such as when quantizing DNN models to different bit-precision levels. In this case, we present the best results that meet our compression criteria for fair comparison.

Table 4.5: Quantization results of AlexNet, VGG-16, and ResNet-18 on ImageNet validation dataset. We calculate the compression rate of activations and model parameters using Equation (4.29) and Equation (4.30), respectively. The bit-precision level represents the average wordlength used in the calculation of *CONV* and *FC* layers.

Architecture	Framework	Integer-Only Inference	Training/ Fine-Tuning	Quantization Granularity	Mixed-Precision	Bit-Precision Level		Compression Rate		Degradation	
						Activations	Weights	Model	Activations	Top-1	Top-5
AlexNet	DoReFa-Net [167]	No	Yes	Per-Layer	No	13.47 ^(1,2)	4.03 ^(1,2)	7.89×	2.58×	1.50%	0.20%
	WRPN [262]	No	Yes	Per-Layer	No	13.47 ^(1,2)	4.03 ^(1,2)	7.89×	2.58×	1.40%	N/A
	PACT [263]	No	Yes	Per-Layer	No	13.47 ^(1,2)	4.03 ^(1,2)	7.89×	2.58×	1.00%	-0.70%
	KDE-KM [278]	No	No	Per-Group	No	32	4	7.99×	1.00×	9.09%	6.97%
	LQ-Nets [273]	No	Yes	Per-Channel	No	13.47 ^(1,2)	4.03 ^(1,2)	7.89×	2.58×	4.40%	3.40%
	TSQ [266]	No	Yes	Per-Channel	No	13.47 ^(1,2)	4.03 ^(1,2)	7.89×	2.58×	0.50%	1.00%
	MMSE [244]	No	No	Per-Group	No	4	4	8.00×	5.13×	2.15%	1.21%
	ACIQ [246]	No	No	Per-Channel	Yes	5.53	8	4.00×	4.00×	4.32%	N/A
	QIL [268]	No	Yes	Per-Layer	No	13.47 ^(1,2)	4.03 ^(1,2)	7.89×	2.58×	3.70%	N/A
	SegLog [280]	No	No	Per-Layer	No	32	5	6.40×	1.00×	N/A	1.73%
	MP-QNN [286]	No	Yes	Per-Layer	Yes	13.47 ^(1,2)	3.19 ^(1,2)	9.95×	2.57×	3.42%	4.30%
	<i>FxP-QNet</i> (ours)	Yes	No	Per-Layer	Yes	5.79	4.48	7.14×	5.33×	0.95%	0.88%
VGG-16	LogQuant [261]	Yes	Yes	Per-Layer	No	4.46 ⁽¹⁾	4.11	7.79×	8.00×	N/A	6.40%
	Going Deeper [88]	Yes	No	Per-Layer	No	8.00	4.43	7.23×	4.00×	1.14%	0.40%
	KDE-KM [278]	No	No	Per-Group	No	32	4	7.99×	1.00×	3.79%	2.19%
	TSQ [266]	No	Yes	Per-Channel	No	2.51 ^(1,2)	2.89 ^(1,2)	11.07×	16.0×	2.00%	0.70%
	MMSE [244]	No	No	Per-Group	No	4	4	8.00×	7.87×	2.82%	1.40%
	ACIQ [246]	No	No	Per-Channel	Yes	4.07	4.12	7.76×	5.33×	1.10%	N/A
	ALT [274]	Yes	Yes	Per-Layer	No	8	4	8.00×	4.00×	0.40%	0.20%
	SegLog [280]	No	No	Per-Layer	No	32	5	6.40×	1.00×	N/A	0.74%
	MXQN [279]	No	No	Per-Layer	No	6.43 ⁽¹⁾	9.68 ⁽²⁾	3.30×	5.33×	1.05%	0.50%
	<i>FxP-QNet</i> (ours)	Yes	No	Per-Layer	Yes	4.55	2.68	11.93×	7.11×	0.95%	0.94%
ResNet-18	DoReFa-Net [167]	No	Yes	Per-Layer	No	9.67 ⁽¹⁻³⁾	4.72 ⁽¹⁻³⁾	6.71×	1.00×	2.90%	2.00%
	PACT [263]	No	Yes	Per-Layer	No	9.67 ⁽¹⁻³⁾	4.72 ⁽¹⁻³⁾	6.71×	1.00×	2.30%	1.40%
	KDE-KM [278]	No	No	Per-Group	No	32	4	7.90×	1.00×	7.91%	5.15%
	LQ-Nets [273]	No	Yes	Per-Channel	No	5.01 ^(1,2)	4.29 ^(1,2)	7.37×	1.00×	2.10%	1.60%
	MMSE [244]	No	No	Per-Group	No	4	4	7.93×	6.75×	2.23%	1.21%
	ACIQ [246]	No	No	Per-Channel	Yes	4.28	4.18	7.57×	4.00×	2.70%	N/A
	QIL [268]	No	Yes	Per-Layer	No	8.91 ⁽¹⁻³⁾	3.78 ⁽¹⁻³⁾	8.36×	1.00×	4.50%	N/A
	CCQ [257]	No	Yes	Per-Layer	Yes	N/A	3.71 ⁽³⁾	8.52×	N/A	2.60%	N/A
	MP-QNN [286]	No	Yes	Per-Layer	Yes	11.66 ⁽¹⁻⁴⁾	3.24 ⁽¹⁻³⁾	9.73×	1.00×	4.27%	3.20%
	MXQN [279]	No	No	Per-Layer	No	9.65 ⁽¹⁾	8	3.98×	2.71×	0.63%	0.40%
	DoubleQExt [287]	No	No	Per-Layer	Yes	8	5.77 ⁽⁵⁾	5.55×	4.00×	1.79%	1.04%
	<i>FxP-QNet</i> (ours)	Yes	No	Per-Layer	Yes	6.29	4.81	6.65×	4.57×	1.99%	1.33%

Note: (1) Do not quantize first layer. (2) Do not quantize last layer. (3) Do not quantize downsample skip connection layer. (4) Do not quantize the input activations to first residual stage. (5) Quantizes the weights to either 5 or 8 bits.

Here, we should emphasize that our work focuses on a hardware-friendly design of quantized DNNs that accelerates the inference, reduces the memory requirements, and at the same time is tailored for integer-only deployment. Thus, it is completely unfair to compare our framework with those interested mainly in speeding up the MAC operation by quantizing activations and weights to a low wordlength before performing MAC operation, and then multiplying each resulting activation by a FP32 scalar to recover the dynamic range. Nevertheless, we make such a comparison to show the effectiveness of *FxP-QNet*.

Overall, from Tables 4.5, it is evident that despite the *FxP-QNet*-quantized architectures perform integer-only inference and are not retrained, they achieve the best trade-off between the computational efficiency, memory requirement, and classification accuracy. For AlexNet architecture, we achieve the highest I/O activation compression ratio ($5.62\times$) and the second-highest compression ratio for model parameters after MP-QNN [286]. This is despite the fact that the latter framework has a top-1 accuracy degradation of 3.42% even though it employs training and floating-point multiplication to maintain network accuracy, whereas *FxP-QNet* has degradation of 1.95% only.

For VGG-16 architecture, *FxP-QNet* achieves the highest model compression ratio ($11.93\times$). Furthermore, we can see that *FxP-QNet* performs better than those frameworks that support integer-only deployment; it attains higher accuracy, in general, with the smallest model size. Note that the improvement in ALT [274] classification accuracy comes at the cost of computational efficiency as it uses 3.45 and 1.32 more bits on average for activations and weights, respectively, during the calculation of *CONV*

and *FC* layers. With regards to frameworks including floating-point arithmetic, and focusing on TSQ [266], the framework with the lowest number of bits involved in the computation of *CONV* and *FC* layers, this wordlength reduction comes at the cost of being less accurate, even though TSQ is a quantization-aware training framework, in addition to having a large memory overhead since it requires a FP32 scalar for each output channel in each layer.

Additionally, we also compare *FxP-QNet* with other frameworks when quantizing a deeper and a more complex network structure such as ResNet-18 and show that the results attained by *FxP-QNet* are competitive even though *FxP-QNet* requires no retraining or fine-tuning and does not require floating-point scaling, as well. For instance, *FxP-QNet* is $1.68\times$ better in increasing the compression rate as compared to MXQN [279] with only a 0.93% higher top-5 accuracy degradation. Further, while the latter framework performs the MAC operation using 9.65-bit activations and 8-bit weights, on average, the former is more computationally efficient because it uses 6.29-bit activations and 4.81-bit weights. Compared to DoubleQExt [287], *FxP-QNet* reduces memory requirements for model parameters by about 1.35 MB with only 0.20% and 0.29% extra drop in top-1 and top-5 accuracies, respectively. This is despite the fact that the former framework requires floating-point arithmetic to transform the output of MAC operations, whereas *FxP-QNet*-quantized ResNet-18 is designed for integer-only deployment.

4.6 Conclusion

In this chapter, we presented *FxP-QNet*, a framework to efficiently quantize DNN data-structures to low-precision numbers in dynamic fixed-point representation. Specifically, the *FxP-QNet* computes the low-precision representation of DNN data-structures as mentioned in Equations (4.10) and (4.13). Furthermore, we experimentally demonstrated that optimizing the quantization parameters for each layer and data-structure independently produces a local optimum solution. Thus, *FxP-QNet* selects the quantization parameters that minimize the post-training self-distillation error as well as network prediction errors as summarized in Algorithm 5. Thereafter, the chapter presented an iterative accuracy-driven bit-precision level reduction framework. *FxP-QNet* employs this framework to flexibly design a quantized DNN with a mixed-precision taking into account the robustness of data-structures to quantization as illustrated in Algorithm 6.

With all the techniques aforementioned, *FxP-QNet* transforms current DNNs to hardware-friendly networks while still preserving accuracy at a high rate. The appealing aspects in our framework are as follows; (i) designing a pure fixed-point DNNs without expensive floating-point operations and thus providing unique computational and hardware efficiencies, (ii) adopting layer-wise quantization and therefore there is almost no additional hardware cost overhead for low-precision scaling as bit-shift operation is used for this, (iii) reducing quantization-induced degradation enough so as to make quantization-aware training unnecessary, (iv) employing a simple and very fast-to-implement quantization process, and (v) facilitating rapid deployment of pre-

trained off-the-shelf models.

Finally, the chapter demonstrated the effectiveness of *FxP-QNet* through extensive experiments on several CNN-based architectures on the benchmark ImageNet dataset. The results shown that *FxP-QNet*-quantized AlexNet, VGG-16, and ResNet-18 can reduce the overall memory requirements by $7.16\times$, $10.36\times$, and $6.44\times$ with only 0.95%, 0.95%, and 1.99% accuracy drop on ImageNet validation dataset, respectively.

Evaluating the presented technique on other deep learning architectures such as recurrent neural networks (RNNs) and Transformer models is an interesting future work. Moreover, the current work was evaluated on image classification task. It would be interesting to see how *FxP-QNet* would perform on other related tasks like object detection, and image segmentation or even in other domains such as natural language processing. Future work shall also explore channel-wise quantization, which could be potentially useful for rigorous quantization of advanced network structures like MobileNets.

CHAPTER 5

HARDWARE-DRIVEN TRAINING-AWARE QUANTIZATION FOR ENABLING EFFICIENT DNN INFERENCE ON EDGE DEVICES WITH MIXED LOW-PRECISION INTEGER-ONLY ARITHMETIC

The deployment of deep learning neural networks (DNNs) on edge devices requires small model sizes to accommodate the limited on-device memory, as well as low latency and high energy efficiency to maintain user engagement. However, modern DNN architectures are primarily designed to improve application accuracy regardless of their complexity and computational efficiency. Therefore, there is a high demand for a hardware-aware automated framework that designs specialized DNNs optimized for a deep learning (DL) hardware accelerator on a particular edge device. In this chapter, we present a novel end-to-end methodology for enabling the deployment of DNNs on resource-limited devices while maintaining model’s accuracy. Specifically, the proposed framework aims at learning the bit-precision level of every weight and activation data-structure that fulfill the memory constraints of a particular target device. This is achieved through the stochastic gradient descent (SGD), which learns the mixed low-precision levels along with model parameters. Additionally, we propose to quantize the weights with either the fixed-point quantizer or the $\log_2$ -based quantizer to represent them accurately and maintains their dynamic range at low wordlength. The quantization method is also learned during model training. By representing activations and weights as integer numbers, the hardware accelerator can achieve efficient DNN inference as integer-only operations are used during the feed-forward computation while the expensive floating-point operations are avoided. The experimental results demonstrate an end-to-end deployment of an integer-only MobileNet network with top-1 accuracy of 72.3% on a device with only 2.5MB of memory space for model parameters (1.53MB) and activations (0.97MB), improving the top-1 accuracy of previously published integer-only deployment by 4.3%.

5.1 Introduction

In computer vision, a better DNN design is characterized by its significant improvement in accuracy. However, such achievement usually comes at the expense of designing highly memory demanding, computationally expensive, and power hungry networks. The large energy consumption and storage overhead prevent DNNs from being incorporated into edge device applications. To bridge the gap between the complexity of DNNs and the low computational power and tight memory of these devices, many researches in recent years have begun to focus on designing new DNNs families. For instance, SqueezeNet [180] and its variants [44], [293], [294] are designed for classification, object detection, and LiDAR point-cloud segmentation tasks with reduced model size. Following SqueezeNet, ShiftNet [295] achieves further model size reduction. Recent works change the focus from reducing model size to minimizing the number of floating-point operations (FLOPs) per image during network inference. MobileNet-V1 [296] and MobileNet-V2 [297] replace the computationally expensive spatial convolution (*CONV*) with depth-wise separable *CONV*. ShuffleNet [298] achieves further reduction in the FLOPs count using group *CONV* and shuffle operations.

Even though these networks achieve significant computational improvement while maintaining, or even improving, model's accuracy, the complexity of these algorithms in terms of their structure and memory access pattern required have grown. The critical need for low-power consumption in today's real-world applications, such as Internet of things (IoT) and unmanned aerial vehicle (UAV) applications [89], [90], to guarantee a long battery lifetime eliminates the deployment of these networks on

graphic processing units (GPUs) at the edge. For energy-efficient solutions, several efforts have been made to implement dedicated DNN hardware accelerators using field programmable gate arrays (FPGAs) [91]. This is because they facilitate the implementation of customized memory hierarchies and hardware architectures with bit-level parallelism, which are critical for efficient data transfers and processing. Nevertheless, DNN computations involve floating-point operations that are slow and resource consuming when implemented on FPGAs. Representing model parameters, activations, and partial results as 32-bit floating-point (FP32) values is also a huge burden on device memory, battery capacity and replacement requirements.

A promising approach to overcome the aforementioned challenges is to replace the expensive floating-point data and arithmetic with low-precision, 8 bits or less, integer data and arithmetic by quantization [105], [230]. With the network topology unchanged, quantization has the potential to significantly reduce inference latency through the use of more efficient integer operations and increasing the arithmetic intensity, that is, the ratio of operations performed to each memory byte accessed [227], [229]. Furthermore, quantization reduces overall memory transaction required between on-chip and off-chip memory storage, providing large power savings.

In general, DNNs can be quantized through incorporating the quantization with the training process, this approach is referred to as quantization-aware training. On the other hand, pre-trained models can be quantized using sophisticated methods, this scheme is denoted by post-training quantization. In this work, we follow the quantization-aware training approach because it is known to be more efficient than the post-training quantization approach in recovering the accuracy drop caused by

quantization. Hence, we assume that the full-size training dataset is available. However, designing efficient and accurate low-precision DNNs for applications in embedded and wearable IoT devices is difficult due to the challenges below.

A) **Heterogeneous quantization robustness:** DNN layers have different structures and therefore different properties related to the quantization process [105], [247]. Furthermore, they have a different arithmetic intensity on hardware, some layers are computation bounded while others are memory bounded. Thus, homogeneously quantizing DNN layers, i.e., allocating the same number of bits to all layers, produces sub-optimal design. In terms of accuracy, quantizing all layers to the same relatively low-precision level causes a noticeable decrease in accuracy because it does not take into account the layers’ sensitivity to quantization.

On the other hand, efficient design of low-precision DNN must consider the issue of scalability. Thereby, it should not be assumed that DNN data-structures can fit into on-chip memory. During inference, model parameters and network activations are transferred between off-chip memory and on-chip memory on need base. Given that the need for memory space varies greatly from layer to layer, homogeneous quantization of DNN layers improves the average-case memory space and off-chip memory access requirements. But in fact, the focus should be on optimizing the worst-case scenario, i.e., reducing the bit-precision level for layers that demand more memory space and accesses.

Considering VGG-16 [27] layers as a motivating example, we showed in [105] that the post-training quantization of the fifth *CONV* layer weights, *CONV3-1*, to 4-bit integer values causes a $3.53\times$ further decrease in accuracy and an $8\times$ less im-

provement in memory footprint compared to its *CONV5-1* counterpart. Spurred by that, this work adopts the *layer-wise mixed low-precision quantization* to flexibly assign different bit-precision levels to different layers and data-structures.

B) Meeting target device requirements: During DNN inference, a memory space is allocated for the input and output activations temporarily required by a single layer as well as for a data block of model parameters. Overall, the total memory allocation is the maximum memory required to occupy a pair of I/O activations plus model parameters required across all layers. With the drive to put DNNs on low-power devices now-a-days, such as battery-powered drones and IoT devices, designers and researchers often find themselves constrained by the limited hardware resources available on the target device. Therefore, a framework is needed to ensure that the designed DNN fits into the available memory space.

Moreover, it is sometimes required to ensure that entire activations can be stored in on-chip memory to avoid data transfer with external memory, which greatly improves inference latency and power efficiency [64]. To the best of our knowledge, there is no systematic method in the literature to design a DNN that meets the memory requirements of a specific target device. All available quantization methods require users to determine the target bit-precision level rather than the target memory space.

Therefore, this work presents a framework for designing DNNs that *fulfill the memory requirements of a particular target device*. Specifically, the proposed framework allows users to separately define target memory space of activations and weights or the overall target memory space. Then, it learns the optimal

quantization levels for activations and weights that meet the specified requirements while maintaining network accuracy and improving inference latency.

- C) **Achieving optimal hardware efficiency:** To accelerate DNN inference and maintain accuracy level, most of the previous quantization methods [275], [276] employ a linear quantizer that maps the FP32 values to low-precision integers and FP32 scalar(s). The integer arithmetic is then used to perform the computation of the multiply-and-accumulate (MAC) operation in *CONV* and fully-connected (*FC*) layers. Thereafter, the output activations of these layers are multiplied with the FP32 scalar(s) to improve accuracy by recovering layer’s dynamic range, i.e., the span of numbers that the quantizer can represent. Other operations, such as batch normalization (*BatchNorm*) and pooling (*POOL*), are performed with floating-point arithmetic. Then, the resultant FP32 activations are quantized before being used by the next *CONV* or *FC* layer, which adds a non-negligible computational overhead. On the other hand, the quantization method in [275] improves the latency of MAC multiplication but uses FP32 accumulators and partial results. Thus, hardware accelerators based on these techniques still need to implement expensive floating-point multipliers and accumulators to handle activation update.

However, compared to 8-bit integer operations, the FP32 multiplier and accumulator consume $18.5\times$ and $30\times$ more power, respectively [228]. Additionally, the 8-bit integer multiplier and accumulator require $27.5\times$ and $116\times$ less area (hardware resources) than their FP32 counterparts, respectively. Considering that IoT devices operate on battery power and have limited hardware resources, and mi-

microcontroller units in edge devices usually have a limited instruction set, and thus, FP32 arithmetic may not be supported, there is a substantial need for a framework that enables the integer-only deployment of DNNs.

With the aim of achieving the highest possible efficiency, this work proposes a *fixed-point quantizer*, a hardware-aware quantization method. All operations performed in the deployed DNNs based on the proposed quantizer are integer arithmetic whereas floating-point arithmetic is not involved at all. This is because the fixed-point quantization replaces the FP32 scaling in linear quantization for activation update with bit-shift and round-off operations.

D) **Bottleneck DNN data-structures:** The majority of weight parameters are concentrated in the deep layers and have a bell-shaped distribution as shown in Figure 5.1. To meet the memory requirements of resource-limited devices, these layers must be compressed at a higher rate compared to the parameters in shallow layers. The problem now is that quantizing these weights to a very-low bit-precision level, such as 4 bits or less, using the fixed-point quantizer may not be optimal for accuracy.

This is because the fixed-point quantizer employs a uniform step size between the consecutive values in the dynamic range, aka the quantization bins. However, the larger proportion of the quantization bins should be logically devoted to small magnitude weights to represent them accurately as they constitute the majority of weights. At the same time, some bins, one or more, must be used to represent relatively large magnitude weights to maintain the dynamic range of the FP32 weights. Hence, these layers become a bottleneck to achieve a high compression

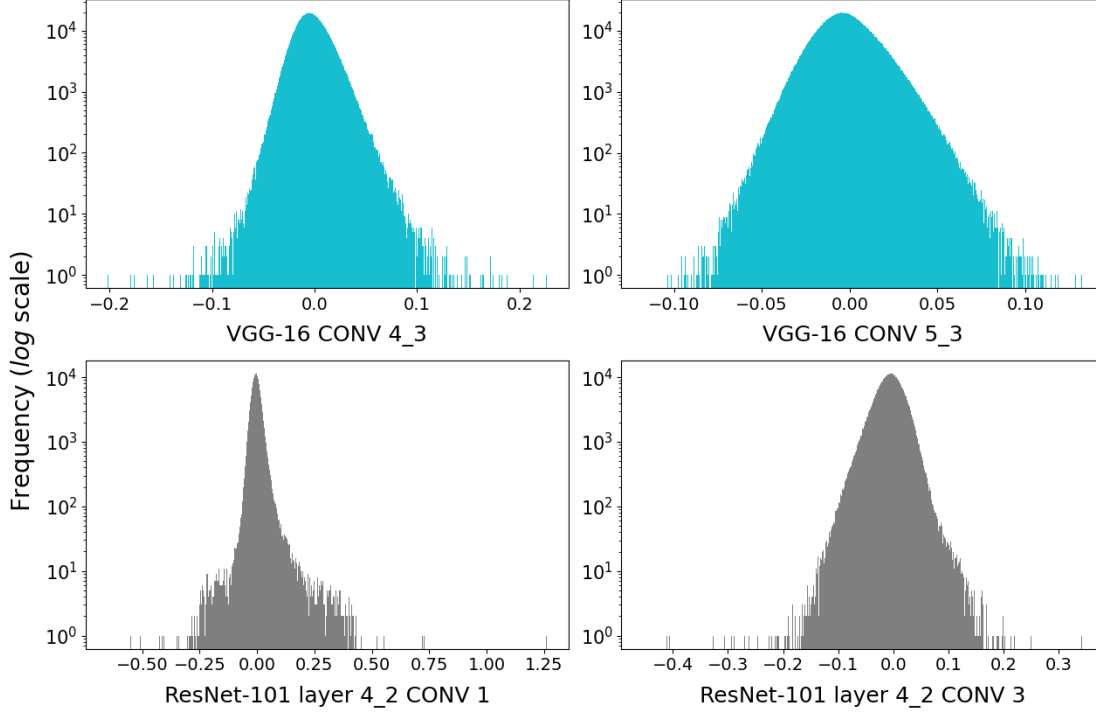


Figure 5.1: The bell-shaped distribution in VGG-16 and ResNet-101 layers.

ratio without causing a noticeable accuracy drop.

To overcome this issues, this work proposes a $\log_2$ -based *quantizer* that converts the FP32 weights of these layers to integer numbers in power-of-2 format. In this way, the non-uniform quantization scalar allows for better representation of the low-precision weights in deep layers. In addition, the power-of-2 format enables the use of high bit-precision activations, such as 8 bits, while maintaining computational efficiency by substituting large digital multipliers with bit-wise left-shift operators.

E) **Intractable design space:** The search space for choosing the appropriate quantization level of each data-structure in different layers on a particular hardware accelerator is exponential in the number of layers and data-structures, which makes the exhaustive search impractical. The need for mixed-precision quan-

tization as well as mixed-quantization methods enlarges the design space further to a range hard to fully explore. For instance, assuming only quantizing the activations and weights of *CONV* and *FC* layers, and each data-structure can choose a different quantization level from a predefined set, the design space contains $(N_{lv_1} \times (N_{lv_2} + N_{lv_3}))^L$ possible architectures.

Here, L is the number of layers, and N_{lv_1} , N_{lv_2} , and N_{lv_3} is the number of possible quantization levels for activations, fixed-point weights, and power-of-2 weights, respectively. Specifically, the design of a mixed-precision quantized MobileNet-V2 with the wordlengths $\{8, 4, 2\}$, $\{8, 6, 4, 2\}$, and $\{5, 4, 3\}$ for activations, fixed-point weights, and power-of-2 weights, respectively, exposes a search space of size $(3 \times (4 + 3))^{54} > 2.5 \times 10^{71}$. Considering that DNN training typically takes days or even weeks, exploring the design space of low-precision MobileNet-V2, a widely used model for classification and detection, by training each possible design to choose the one with the higher accuracy is infeasible even with such simplified design choices.

To simplify the exploration of this huge design space and avoid training each candidate design, the authors in [284] proposed a rule-based heuristic. However, as models become deeper, the exploration space increases exponentially, making it almost infeasible to rely on hand-crafted strategies. Therefore, rule-based quantization policies are usually sub-optimal, and cannot generalize well from one model to another. Alternatively, previous works [282], [283] adopt reinforcement learning (RL) to guide the search. Generally, automated design space search approaches employ a controller to sample and train designs from the search space. To reduce

the training cost, sampled designs are trained on a smaller dataset or trained for fewer epochs. The accuracy of the trained networks is then used to train the controller for improved future sampling. However, training the controller is still time-consuming as it usually relies on thousands of trained designs. To automate and accelerate the design process, this work proposes gradient formulas for *learning quantization levels and quantization methods through SGD*. In this way, quantization methods and levels are learned in conjunction with model parameters using gradient-based optimization such as SGD that minimizes task loss.

Previous quantization schemes in [237], [255], [263], [274]–[276], [299] focused on using backpropagation with SGD to learn the quantization parameters that minimize the task loss for a target bit-precision level. In this work, we extend these techniques by learning the bit-precision levels for each layer along with their quantization parameters through backpropagation. Learning the quantization mapping that satisfies target memory constraints by seeking to minimize task loss is appealing to us because it directly seeks to improve on the metric of interest. At the same time, it avoids the huge computations and extensive exploration phase in [282], [283] that use RL.

A DL hardware accelerator with small memory capacity and fewer logic gates consumption are what industry and academia expect. Hence, in this chapter, we present a novel end-to-end methodology for enabling the deployment of DNNs on resource-limited devices while maintain model’s accuracy. Specifically, we propose a framework, illustrated in Figure 5.2, for quantizing DNN data-structure to integer numbers based on a mixed-precision scheme. The proposed framework learns the bit-precision levels that satisfies the memory limitations and other resource constraints of a given device

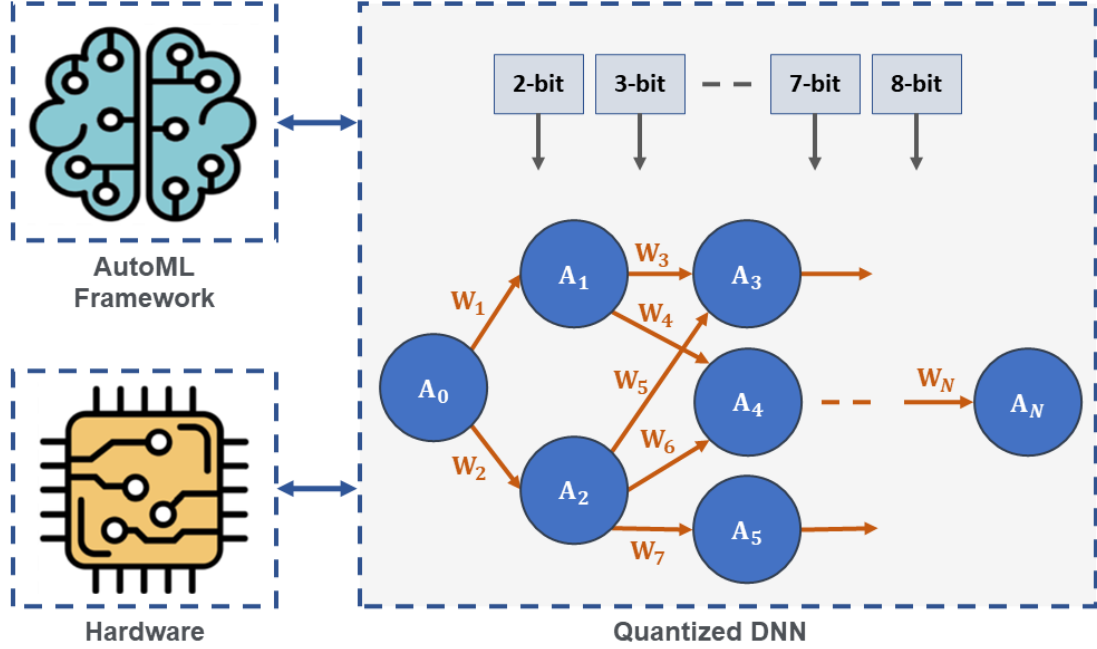


Figure 5.2: Hardware-aware automated ML framework for mixed low-precision quantization.

and minimizes the task loss to preserve model’s performance. This is achieved through the SGD, which learns the mixed low-precision levels along with model parameters. Additionally, we propose to quantize the weights with either the fixed-point quantizer or the $\log_2$ -based quantizer. The quantization method is also learned during model training using the Gumbel Softmax technique [300] aiming to represent the FP32 values accurately at low wordlength while maintaining their dynamic range.

The learning process is extremely fast compared with previous RL based methods [282], [283]. The loss used to train the network consists of both the cross-entropy loss that leads to better accuracy and the hardware-aware loss that penalizes the network memory requirements on the target device. To maintain the accuracy of the quantized network, we employ the knowledge distillation [272] technique to transfer the knowledge from a deeper network to the low-precision quantized network. By

representing activations and weights as integer values, the DL hardware accelerator achieves efficient DNN inference as the expensive floating-point operations are avoided during the feed-forward computations. The experimental evaluation is conducted over the MobileNet-V2, a model family known for run-time efficiency, on ImageNet [28] dataset. We argue that this is a representative problem for extreme-edge-devices, not yet solved and much harder than quantizing over-parameterized networks such as AlexNet [26] and VGG-16 [27]. The contributions of this chapter can be summarized as below:

1. We introduce an automated framework for design mixed low-precision DNNs.

The proposed framework does not require any domain experts to determine the wordlengths for different layers, it allows users to separately define target memory space for activations and weights or the overall target memory space. Then, it learns the optimal quantization levels that meet the specified requirements while maintaining network accuracy at a high level. This feature is not present in existing quantization schemes as they require users to input the wordlength of each data-structure.

2. The proposed framework integrates the target hardware architecture into the optimization loop so that it can directly satisfy the memory, computation, and logic resource constraints. For different hardware architectures, the framework can offer a customized DNN that is perfectly tailored to the target hardware architecture.
3. We present a detailed description of a quantization methodology that converts the FP32 weights and activations to fixed-point numbers. Additionally, we in-

introduce another quantization methodology that quantizes the FP32 weights in their $\log_2$ -based format to power-of-2 numbers. Thus, these quantization methods enable the implementation of a highly resource/energy-efficient DL hardware accelerator that perform only integer operations.

4. We propose gradient formulas to automate and accelerate the learning of quantization levels and methods through SGD. We adopt the Gamble Softmax technique to facilitate the learning of the optimal quantization method for very weight parameter.
5. We propose a simple heuristic to bring the magnitude of the quantization parameter updates into better balance with weight updates, which we found to improve convergence.
6. We present an adaptive layer-wise learning rate technique to overcome the activation instability due to weight quantization to low-precision levels, and thus optimizing the learning.
7. Examining the proposed methodology on the benchmark ImageNet dataset, we show that the integer-only deployment of MobileNet-V2 network on an edge device with 1.53MB weight’s memory and 0.97MB activation’s memory classify images with 72.3% top-1 accuracy, which is 4% higher than previous reported integer-only implementation.

The rest of the chapter is organized as follows. Section 5.2 provides background information about image classification task. It also gives a brief overview of knowledge distillation and its benefits for optimizing the training of low-precision and/or small convolutional neural networks (CNNs). Section 5.3 presents a literature re-

view of the existing DNNs quantization frameworks. In Section 5.6, we discuss the proposed framework that designs mixed low-precision DNNs driven by the memory constraints of a target architecture. Section 5.4 presents the quantization methodology for integer-only deployment. Section 5.8 demonstrates the effectiveness of the proposed framework via extensive experiments on MobileNet family of DNN architectures. Finally, we conclude and describe the future directions of this chapter in Section 5.9.

5.2 Background and Terminology

In this section, we discuss image classification task, a classical and widely used computer vision task in today’s real-world applications. Additionally, this section provides a brief overview of knowledge distillation and its benefits for optimizing the training of low-precision and/or small CNNs.

5.2.1 CNN-based Image Classification

In computer vision, image classification refers to the process of assigning a specific category, such as a person, dog, or cat, for a given sample image. Input images for classification typically contain only one prominent object. To carry out this task, a CNN architecture consisting of a feature extractor and a classifier is usually designed. The feature extractor is basically stacked *CONV* layers with *BatchNorm*, non-linearity rectified linear unit (*ReLU*), and/or *POOL* layers in between. The shallow *CONV* layers extract the low-level features from the input image, such as the edges, orienta-

tions, and colors. On the other hand, deep *CONV* layers make use of the enhanced dissimilar hierarchical features to capture high-level features such as faces and furs.

The *BatchNorm* layer smooths the extracted features and helps the network to learn these features faster while preventing the occurrence of overfitting. The *ReLU* activation function replaces negative activations with zero to help networks to learn and model complex and non-linear functions, i.e., $ReLU(x) = \max(0, x)$. The *ReLU6* and *LeakyReLU* are newer versions of the *ReLU* non-linear activation function. The former restricts the maximum value to 6, $ReLU6(x) = \min(\max(0, x), 6)$, while the latter multiplies negative activations with λ instead of zero, $LeakyReLU(x, \lambda)$ is $\lambda \times x$ when $x < 0$, and x otherwise. The *POOL* layer helps in extracting scale, positional, and rotational invariant features as well as reducing feature map (FM) dimensionality and suppressing noise by applying the minimum, maximum, or average functions. Finally, the classifier employs the inner product function through *FC* layers or equivalent point-wise *CONV* layers to compute the score of each class (also known as logits). Thus, the number of output values from the network classifier equals the number of dataset classes.

The ImageNet large scale visual recognition challenge (ILSVRC) dataset [28] is one of the most challenging image classification benchmarks so far. The ILSVRC-2012 dataset has about 1.28 million images for training and 50 thousand images for validation, all are natural high-resolution images. Each image is annotated as one of the 1,000 classes. The top- k accuracy is the measure used to report the precision of classification models. The prediction is considered correct when the actual label is among the first k labels with the highest predicted probability. The top-1 and

top-5 accuracies are the most popular reported measures for the model’s classification accuracy on benchmark datasets.

To optimize image classification models, the backpropagation algorithms such as SGD and Adam are used. Specifically, these algorithms employ a loss function to compute the error between the predicted scores and expected labels. The backpropagation algorithm performs the forward pass to compute the loss based on the present model parameters. Thereafter, it carries out the backward pass to compute the gradient of the loss function with respect to model parameters. Finally, it updates these parameters by a factor of the gradients (depending on the values of the learning rate, momentum, and other hyperparameters) but in the opposite direction to minimize network loss. A popular loss function for classification problems is the cross-entropy loss $\mathcal{H}(\bullet, \bullet)$, which is computed by

$$\mathcal{H}(y, \hat{y}) = - \sum_{c=1}^C y_c \times \log(\hat{y}_c) \quad (5.1)$$

where y is the expected probability of each class label (also called ground-truth labels) for an input sample in the dataset and $\hat{y}$ is the predicted probability of each class label by the network for that input sample, which is calculated from the logits $\tilde{y}$ of the classifier using the standard softmax function as

$$\hat{y} = \sigma(\tilde{y}) = \frac{\exp(\tilde{y}_c)}{\sum_{i=1}^C \exp(\tilde{y}_i)} \quad , \quad \forall c \in [C] \quad (5.2)$$

MobileNets [296], [297] is a widely used CNN architecture for real-world classifi-

cation applications on mobile devices. It is designed to have a lightweight but accurate model. This is achieved by replacing the computational-intensive and memory-intensive standard *CONV* with a depth-wise *CONV* followed by a point-wise *CONV*. Together, these two layers form the so-called depth-wise separable *CONV* block. Specifically, the depth-wise *CONV* layer uses N single-channel filters each of which is of size $K \times K$. Conversely, the point-wise *CONV* layer employs M filters of size 1×1 , each of which contains N channels. Here, N and M are the number of input FMs and output FMs, respectively. In this way, depth-wise *CONV* filters the features in the input channels, and then these features are combined to build new features through point-wise *CONV*. Unlike standard *CONV* and point-wise *CONV* layers where all filters are applied into all of the input channels, each filter of depth-wise *CONV* is applied into one input FM to produce the corresponding output FM, i.e., depth-wise *CONV* maintains the number of channels as is. Thus, depth-wise separable *CONV* reduces the computational cost of standard *CONV* by $(1/N + 1/K^2)$.

Compared to VGG-16 [27] and GoogleNet [29] architectures, MobileNet-V2 [297] performs $51\times$ and $5.2\times$ less MAC operations per image and its model size is $39.3\times$ and $1.9\times$ smaller with further 0.3% and 2% improvement in top-1 accuracy, respectively. However, this comes at the cost of being a complex structure. For instance, MobileNet-V2 uses the bottleneck residual module as a building block. This block consists of a point-wise *CONV*, followed by a depth-wise separable *CONV* block. Each *CONV* layer in the bottleneck residual block is followed by *BatchNorm* and *ReLU6* except the last point-wise *CONV* layer which comes without any non-linearity.

Additionally, the first point-wise *CONV* layer in each bottleneck residual block

is responsible for expanding the number of channels in the block’s internal outputs by a given expansion factor. Therefore, the bottleneck block removes its first point-wise *CONV* when the expansion factor is 1. Furthermore, each bottleneck block uses a residual skip connection when the number of input channels going into it is the same as the number of output channels coming out of it. Table 5.1 shows the overall architecture of MobileNet-V2. The block’s stride refers to the stride of the depth-wise *CONV* kernel, the stride of point-wise *CONV* kernels is always 1. Note also that the given stride in Table 5.1 is for the first bottleneck residual block, other repeated blocks, if any, use a stride of 1. Overall, MobileNet-V2 has 53 *CONV* layers (1 standard *CONV*, 17 depth-wise *CONV*, and 35 point-wise *CONV*).

5.2.2 Knowledge Distillation

Edge devices have limited memory and computational capacity, which makes the deployment of highly complex DNNs on edge devices very challenging. Therefore, smaller and quantized DNNs are indispensable for edge devices. However, such networks usually have a significant accuracy gap from the original full-precision networks. To tackle this problem, we use the knowledge distillation [272] to learn the simpler quantized network, referred to as the student network, to mimic the behavior of the complex teacher network, which usually has a deeper structure with a larger number of parameters and trained in full-precision. Precisely, the student network distills the teacher’s knowledge by learning the parameters that minimize the cross-entropy loss between its predicted class probabilities and the teacher’s predicted class probabilities, thus obtaining similar predictions.

Table 5.1: MobileNet-V2 overall architecture [297].

Input	Module	Expansion Factor	Output Channels	Stride	Repeating Number
$224 \times 224 \times 3$	Standard <i>CONV</i> (Kernel 3×3)	—	32	2	1
$112 \times 112 \times 32$	<i>BatchNorm</i>	—	16	—	1
	<i>ReLU6</i>	—		—	
	Bottleneck Residual	1		1	
$112 \times 112 \times 16$	Bottleneck Residual	6	24	2	2
$56 \times 56 \times 24$	Bottleneck Residual	6	32	2	3
$28 \times 28 \times 32$	Bottleneck Residual	6	64	2	4
$14 \times 14 \times 64$	Bottleneck Residual	6	96	1	3
$14 \times 14 \times 96$	Bottleneck Residual	6	160	2	3
$7 \times 7 \times 160$	Bottleneck Residual	6	320	1	1
$7 \times 7 \times 320$	Point-wise <i>CONV</i>	—	1280	1	1
$7 \times 7 \times 1280$	<i>BatchNorm</i>	—	—	—	1
	<i>ReLU</i>				
	Average <i>POOL</i> (Kernel 7×7)				
$1 \times 1 \times 1280$	Point-wise <i>CONV</i> (i.e., <i>FC</i>)	—	1000 (for ILSVRC)	—	1

To provide a softer knowledge transfer, the softmax temperature function is used instead of the standard softmax function because it better expresses the degree of association between the different classes [301]. The softmax temperature function employs a temperature parameter $\mathcal{T}$, $\mathcal{T} \in [1, 20]$, to calculate the class probabilities from the logits as,

$$\hat{y} = \sigma(\tilde{y}; \mathcal{T}) = \frac{\exp(\tilde{y}_c / \mathcal{T})}{\sum_{i=1}^C \exp(\tilde{y}_i / \mathcal{T})}, \quad \forall c \in [C] \quad (5.3)$$

At a very low temperature, $\mathcal{T} = 1$, the softmax temperature function is equiv-

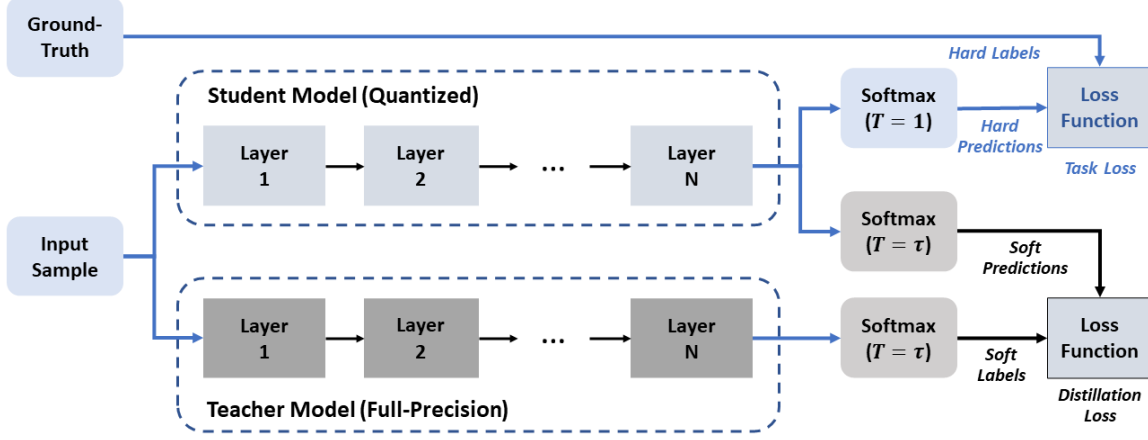


Figure 5.3: Knowledge distillation from teacher to student (quantized) model.

alent to the standard softmax function. As $\mathcal{T}$ grows, the resulting distribution of class probabilities becomes softer, providing more information about the classes that the teacher found more similar to the predicted class. As shown in Figure 5.3, the backpropagation algorithm incorporates both the distillation loss and the standard task loss to optimize the weights of the student model (teacher model parameters are frozen) with the following loss function,

$$\begin{aligned} \mathcal{L}_{task-distill} = & \lambda \times \mathcal{H}\left(\sigma\left(\tilde{y}^{(t)}; \mathcal{T} = \tau\right), \sigma\left(\tilde{y}^{(s)}; \mathcal{T} = \tau\right)\right) + \\ & (1 - \lambda) \times \mathcal{H}\left(y, \sigma\left(\tilde{y}^{(s)}; \mathcal{T} = 1\right)\right) \end{aligned} \quad (5.4)$$

where λ tunes the weight of the loss with the soft labels and the loss with hard labels, $\lambda \in [0, 1]$, $\mathcal{H}(\bullet, \bullet)$ is the cross-entropy loss function discussed in Equation (5.1), $\sigma(\bullet; \bullet)$ refers to the softmax function parameterized by the temperature $\mathcal{T}$ as defined in Equation (5.3), y is the ground-truth labels, $\tilde{y}^{(s)}$ and $\tilde{y}^{(t)}$ are the logits of the student and teacher networks, respectively.

5.3 Related Work

To meet the memory constraints on edge devices, Rusci et al. [284] proposed to gradually reduce the bit-precision of the most memory-demanding data-structures. Specifically, when the memory space for the pair of input and output activations of a layer exceeds the read-write memory constraint, the wordlength of output activations is reduced to the next level if and only if they have a higher wordlength or a higher memory footprint than that of input activations. On the other hand, the wordlength of weights with the highest memory footprint and lowest layer’s index is reduced until meeting the read-only memory constraint. PACT strategy [263] is adopted to asymmetrically quantize weights and activations to unsigned integers. Therefore, before performing the *CONV* operation, these values are shifted using 8-bit layer-wise bias parameters, which introduces a latency overhead and consumes further hardware resources due to the additional subtractions. To achieve the integer-only deployment, the authors combined the FP32 quantization scalars for activations and weights with *BatchNorm* layer parameters. Then, they converted the resulting parameters to fixed-point numbers with 32 bits for the integer field and 8 bits for the fractional field. Thus, large-size digital multipliers and barrel-shifters are needed for *BatchNorm* scaling, fairly complex circuits.

PACT [263] adopts a bounded activation function to clip the activations to the range $[0, \alpha]$. Then, it quantizes the clipped activations into k -bit integer numbers using the scalar $\alpha/(2^k - 1)$. PACT learns the optimal clipping value, α , for each activation layer during network training. Concerning weights, PACT transforms them to the

range $[0, 1]$ through tanh function, and then quantizes the transformed weights in the same way as activations. Finally, an affine transform is used to map PACT quantized weights into $[-1, 1]$.

ReLeQ [282] and HAQ [283] adopt RL to learn the wordlength of each data-structure in a layer-wise manner that imposes the predefined hardware constraints. Specifically, they use two neural networks (NNs), each of which has 3 *FC* layers, to predict the next quantization level for each layer. Thereafter, ReLeQ uses the predicted bit-precision levels to quantize weights using the proposed technique in WRPN [262], whereas HAQ quantizes both weights and activations on each explored wordlength in the same way as TensorRT [241]. Lastly, both frameworks fine-tune the quantized model to recover performance before moving on to the next quantization cycle. This process is repeated until the predefined constraints are met. Although the results of ReLeQ and HAQ are auspicious, the exploration phase is extensive and can be significantly time-consuming. The prediction networks are trained during solution space exploration according to the quantization reward on accuracy and memory.

DNAS [256], proposed by Facebook and Berkeley AI Research, formulates the mixed-precision assignment problem as a neural architecture search problem. For each network block, DNAS search for the bit-precision configuration, denoted as (k_A, k_W) , of the *CONV* layers so that all *CONV* layers within the same block represent their activations and weights with k_A and k_W bits, respectively. Specifically, DNAS allows *CONV*s layers to represent their activations and weight either FP32 values or as integer numbers using one of the manually specified configurations. Out of all possible configurations, DNAS selected five configurations with activations bit-width is always

non-less than weights bit-width. Then, the SGD is used to select the configuration that optimizes the loss function. Note that DNAS follows PACT to quantize weights and activations. In other words, it uses FP32 quantization scalars for activations and weights to recover the floating-point dynamic range. A major drawback of DNAS is that it replicates the entire model for each candidate bit-precision level. Hence, it needs huge memory space while training the network. In this work, we use the same idea but to learn the optimal quantization method. Additionally, we extend DNAS to also learn the bit-precision level that meets system requirements and supports integer-only deployment to provide exceptional efficiency.

ALT [274], LSQ [275], and PROFIT [276] defined optimized gradient formulae to learn low-precision parameters and quantization scalar of each layer when quantizing FP32 models to the desired precision. While PROFIT and IBM’s LSQ use FP32 quantization scalars, ALT restricts them to power-of-2. Furthermore, PROFIT proposed a training method in which sensitive weights, that cause activation instability, are progressively frozen. During network training, LSQ and PROFIT use the low-precision integers scaled with the FP32 quantization scalar as input to matrix multiplication layers, which is algebraically not equivalent to the results on an integer-based hardware implementation of the MAC operation as performing floating-point multiplication and addition on floating-point values will introduce a rounding error. It should be noted that quantization and rescaling are done using floating-point arithmetic in linear quantization (such as in PROFIT and LSQ), whereas they are just simply bit-shift operations in fixed-point quantization.

5.4 Integer-Only Inference

Machine learning frameworks, such as PyTorch and TensorFlow, use the FP32 representation and arithmetic for model parameters and activations. To avert using floating-point arithmetic during model deployment on hardware devices, these frameworks must train DNNs for integer computations. For this purpose, this section presents a quantization methodology that transforms DNN data-structures into low-precision integer values during network training and gradient formulas to improve training convergence.

5.4.1 Quantization Methodology

To allow using fractional numbers on low-cost integer hardware, we defined a quantizer, denoted as Q_{FXP} , that maps floating-point values to fixed-point numbers. Specifically, the Q_{FXP} converts a FP32 data-structure, or a high-precision fixed-point data-structure, $\mathcal{X}$ into a low-precision integer data-structure $\overline{\mathcal{X}}$ and a common scaling factor $2^{-\beta}$, $\beta \in \mathbb{Z}$, in a layer-wise fashion. This is done by transforming the original values to a new domain, and then, clipping and discretizing these transformed values. Given the transformer slop α and the quantization interval $[\text{QL}_{\min}, \text{QL}_{\max}]$, the quantizer computes $\overline{\mathcal{X}}$ as follows

$$\textbf{Transformation:} \quad \hat{\mathcal{X}} = \text{clip}\left(\frac{\mathcal{X}}{\alpha}, \text{QL}_{\min}, \text{QL}_{\max}\right) \quad (5.5)$$

$$\textbf{Q}_{\text{FXP}} \textbf{ Discretization:} \quad \overline{\mathcal{X}} = \left\lfloor \hat{\mathcal{X}} \right\rfloor \quad (5.6)$$

Here, $\text{clip}(x, \min, \max)$ returns x with values less than $\min$ set to $\min$ and values greater than $\max$ set to $\max$, and the $\lfloor \bullet \rfloor$ represents the round operation where the half is rounded to the nearest even to prevent bias [289]. As for activations, we quantize the relatively high-precision integer values similarly except that the transformed values get simply truncated rather than rounded, because of the lighter hardware implementation (i.e. a shift operation). The quantization interval is used to handle various distributions across the layers. For instance, when discretizing to k -bit values, $\text{QL}_{\min}$ and $\text{QL}_{\max}$ are equal to negative and positive $2^{k-1} - 1$, respectively, for symmetric distribution data-structure. On the other hand, $\text{QL}_{\min}$ and $\text{QL}_{\max}$ are set to 0 and $2^k - 1$, respectively, in case of non-negative activation distribution. Thus, each integer number, say $\overline{\mathcal{X}}_i$, in the quantized data-structure that contains N elements is interpreted to a rational number expressible as

$$\text{Scaled } \mathbf{Q}_{\text{Fxp}} \text{ Discretization: } \quad \tilde{\mathcal{X}}_i = \overline{\mathcal{X}}_i \times 2^{-\beta} = \overline{\mathcal{X}}_i \gg \beta, \quad \forall i \in [N] \quad (5.7)$$

That is, the quantized representation of $\mathcal{X}$ at the floating-point scale is $\overline{\mathcal{X}}$ right-shifted β bit positions after the decimal point. It is noteworthy that model parameters are quantized offline before model deployment. Hence, we use a floating-point transformation scalar, $\alpha \in \mathbb{R}$, for model parameters to improve the training convergence. On the other hand, activations require on-the-fly quantization during the forward propagation. Therefore, the quantizer transforms the distribution of the integer input activations with $2^{-\beta}$ instead of α . The transformation scalar (α), the fixed-point scaling factor ($2^{-\beta}$), and the bit-precision level (k , and thus $\text{QL}_{\min}$ and

$QL_{\max}$) are learned during the backpropagation using the SGD as will be illustrated in Sections 5.4.2 and 5.6.

Unlike previous quantization techniques in [275], [276] where the division in Equation (5.5) is performed with floating-point arithmetic for quantizing activations, the Q_{FXP} replaces the division operation with simply bit-shift operation. Thus, it reduces the complexity of hardware implementation in other quantization techniques. Accordingly, to perform the computations of the *CONV* and *FC* layers in a lower bit-precision domain, the high-precision fixed-point activations $\mathcal{A}$ and the FP32 weights $\mathcal{W}$ are quantized to their low-precision integer representations $\overline{\mathcal{A}}$ and $\overline{\mathcal{W}}$, respectively. This is done based on their transformation scalars and quantization intervals as stated in Equation (5.5) and (5.6). In this way, $\mathcal{A}$ and $\mathcal{W}$ is approximated to $\overline{\mathcal{A}} \times 2^{-\beta_{\mathcal{A}}}$ and $\overline{\mathcal{W}} \times 2^{-\beta_{\mathcal{W}}}$, respectively. Thereafter, the MAC operation, the basis of mathematics in the *CONV* and *FC*, is performed on a data block of $\overline{\mathcal{A}}$ ($\overline{\mathcal{A}}_{b_i}$) and a data block of $\overline{\mathcal{W}}$ ($\overline{\mathcal{W}}_{b_i}$) with integer arithmetic, as illustrated in Figure 5.4. Thus, the output result is calculated as $\mathbf{MAC}(\overline{\mathcal{A}}_{b_i}, \overline{\mathcal{W}}_{b_i}) \times 2^{-(\beta_{\mathcal{A}} + \beta_{\mathcal{W}})}$.

However, adopting a uniform scalar to quantize weights to very-low bit-precision level such as 4 bits or less, might not be optimal for some layers. Most layer weights follow a bell-shaped distribution with the majority of FP32 values centered around zero and a small portion of the weights having large magnitude values. With a fixed step size between quantized values, it becomes challenging to efficiently quantize these weights using Q_{FXP} . To overcome this issue and inspired by the effectiveness and efficiency of the logarithmic-based quantization [302], we propose another quantizer, referred to as Q_{Log_2} , that converts the FP32 weights to power-of-2 values. More pre-

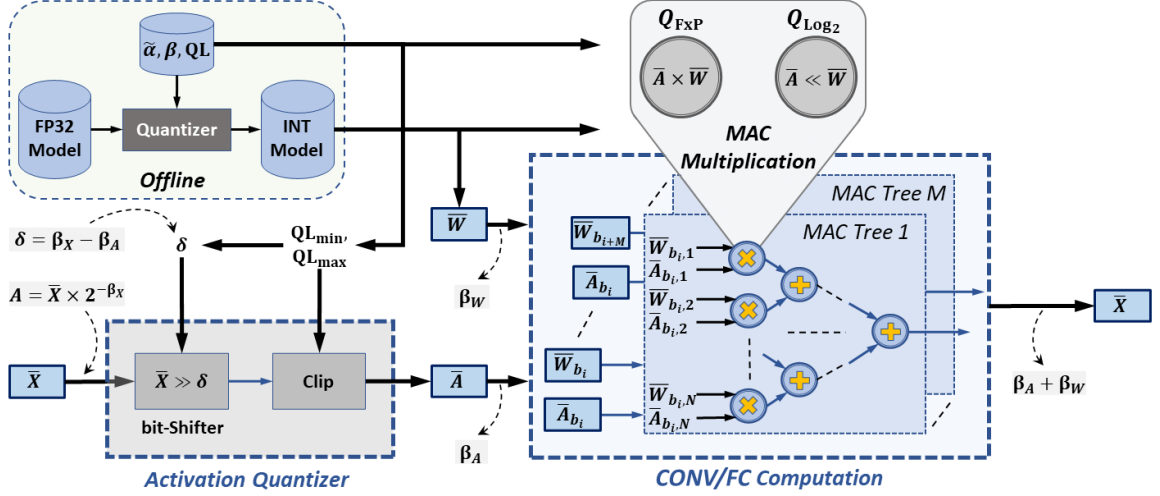


Figure 5.4: An overview of the low-precision computation of the *CONV* and *FC* layers when quantizing activations and weights to fixed-point values using the Q_{FXP} or Q_{Log_2} quantizer.

cisely, the Q_{Log_2} first transforms the FP32 weights as illustrated in Equation (5.5) with $\text{QL}_{\text{max}} = -\text{QL}_{\text{min}} = 2^{(2^{k-2}-1)}$ for a k -bit quantization of symmetric distribution weights. Then, the transformed values are quantized in their logarithmic representation. Thus, Q_{Log_2} results in the data-structure $\overline{\overline{\mathcal{X}}}$ that contains the exponent of the non-zero values when represented as power-of-2 numbers, such that

$$\mathbf{Q_{Log_2 \ Discretization:}} \quad \overline{\overline{\mathcal{X}}} = \begin{cases} \left\lfloor \log_2 (|\hat{\mathcal{X}}_i|) \right\rfloor, & |\hat{\mathcal{X}}_i| > 1 \\ 0, & \text{Otherwise} \end{cases} \quad (5.8)$$

where $|x|$ returns the absolute value of x . Consequently, each value in the quantized data-structure $\overline{\overline{\mathcal{X}}}$ is decoded as

$$\text{Scaled } Q_{\text{Log}_2} \text{ Discretization: } \tilde{\mathcal{X}}_i = 2^{-\beta} \times \begin{cases} \text{sign}(\hat{\mathcal{X}}_i) \times 2^{\overline{\overline{\mathcal{X}}_i}}, & |\hat{\mathcal{X}}_i| > 0.5 \\ 0, & \text{Otherwise} \end{cases} \quad (5.9)$$

Accordingly, Q_{Log_2} quantizer allows the use of relatively high-precision activations, such as 8 bits, and at the same time maintains the computational efficiency through quantizing weights to power-of-2 values. As it is evident from Equation (5.9), Q_{Log_2} quantizer uses two additional bits besides the bits required to store the integer exponent $\overline{\overline{\mathcal{X}}_i}$ to keep track of the sign and true-zeros. In Figure 5.5, we quantize the weights of the last point-wise *CONV* layer of MobileNet-V2 into 4 bits using the proposed quantization methodology. Although Q_{Log_2} quantizer uses fewer quantization bins than Q_{FXP} quantizer, the trained 4-bit power-of-2 weights improve the top-1 and top-5 accuracies of the trained 4-bit uniform weights by 0.47% and 0.28%, respectively.

The reason behind not adopting Q_{Log_2} for quantizing activations is related to the hardware implementation of the log-base-2 operation. Specifically, designing a log-based quantizer to handle activation bit-precision reduction during the inference incurs additional hardware complexity, defeating the original purpose of improving the computational and area efficiency of large size multipliers. On the other hand, weights quantization is performed offline before starting the inference. Hence, no additional overhead is incurred on quantizing model parameters during hardware execution.

Unlike the work in [302], Q_{Log_2} quantize the weights in their fixed-point domain instead of their original floating-point domain. The trained weights typically lie within

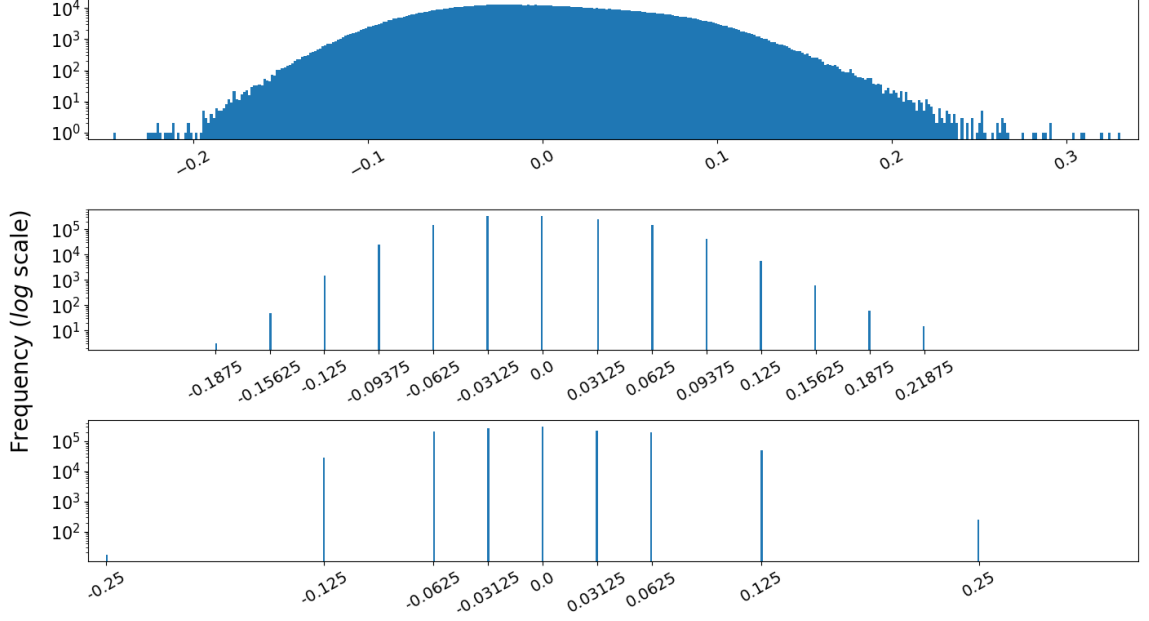
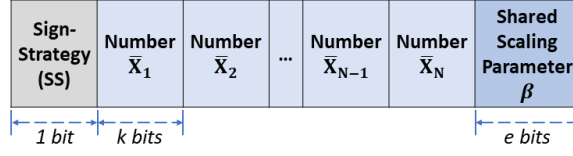


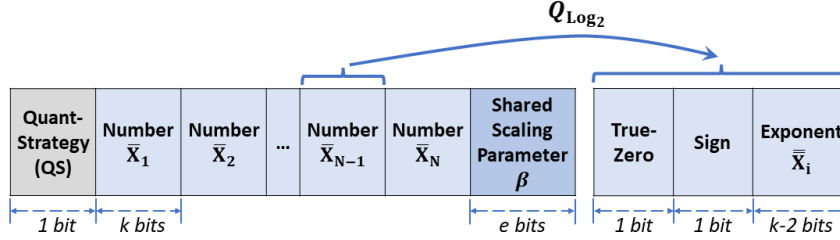
Figure 5.5: The distribution of the original 32-bit floating-point weights (*top*), 4-bit quantized weights using Q_{Fxp} quantizer (*middle*), and 4-bit quantized weights using Q_{Log_2} quantizer (*bottom*) of the last point-wise *CONV* layer in MobileNet-V2.

the interval $[-1, 1]$. Therefore, quantizing these floating-point values in their logarithmic representation would result in negative exponents, $\overline{\overline{\mathcal{X}}} \in \mathbb{Z}^-$. For instance, quantizing the floating-point value 0.0147 directly to power-of-2 gives the exponent -6 . The multiplication between an integer activation and 2^{-6} is actually a right-shift operation. Therefore, there is a need to design a hardware circuitry to handle the error caused by truncating those 6 least significant activation bits. However, this incurs hardware and computational inefficiencies which defeats the original purpose of using power-of-2 weights. On the other hand, by representing the floating-point value as a fixed-point number, $0.0147 \approx 15 \times 2^{-10}$, and then converting the integer number to power-of-2 value, $15 \approx 2^4$, the multiplication operation in the MAC unit can be efficiently computed by shifting the activation 4 positions to the left.

Figure 5.6 shows the general format used to represent quantized data-structures.



(a) Quantized activations format.



(b) Quantized weights format.

Figure 5.6: Generalized format for (a) quantized activations and (b) quantized weights.

For activations, the first bit indicates whether the discrete values are in k -bit unsigned representation or in k -bit signed 2's complement representation. On the other hand, the first bit in the quantized weight data-structure indicates whether Q_{FXP} quantizer or Q_{Log_2} quantizer is used to convert weights to k -bit values. In case of using Q_{Log_2} quantizer, the first two bits denote true-zero weight value and the sign of weight value, respectively, while the remaining bits indicate the exponent. Regarding the memory space required to store a quantized data-structure, it needs $N \times k$ bits for the internal representation of the numbers plus the memory space needed to store the sign/quantization-strategy and the scaling parameter, i.e., $1 + e$ bits, $e \leq 8$, thus the storage overhead of the additional $e + 1$ bits is marginal because they are shared across many numbers.

5.4.2 Gradient Formulas for Learning Quantization Parameters and Model Parameters

To learn the quantizer scalar, each layer of activations and each layer of weights employ a distinct trainable parameter referred to as $\tilde{a}$. This parameter is represented as a 32-bit floating-point value and used to calculate the value of the transformation scalar (α) and the fixed-point scaling factor ($2^{-\beta}$), such that

$$\alpha = \frac{\text{softplus}(\tilde{a})}{\text{QL}_{\max}} \quad (5.10)$$

$$\beta = -\lfloor \log_2(\alpha) \rfloor \quad (5.11)$$

where $\text{QL}_{\max}$ is the largest magnitude quantization bin and the softplus function is used to constrain α to always be positive, which improves the stability of the transformation stage, and it is calculated by

$$\text{softplus}(x) = \log(\exp(x) + 1) \quad (5.12)$$

The initial value of $\tilde{a}$ for both activation and weight data-structures is set based on the largest magnitude value as $\log(\exp(\max(|\mathcal{X}|)) - 1)$. The problem now is that learning $\tilde{a}$ and the quantized model parameters during the backpropagation using the present formula is not possible due to the non-differentiable round function in Equations (5.6) and (5.8). To solve this issue, PACT [263] estimates the output of the round operation to be equal to the input, $[x] = x$. Thus, it removes the round operation from the forward equation and then differentiates such that $\partial \tilde{\mathcal{X}} / \partial \alpha = 0$

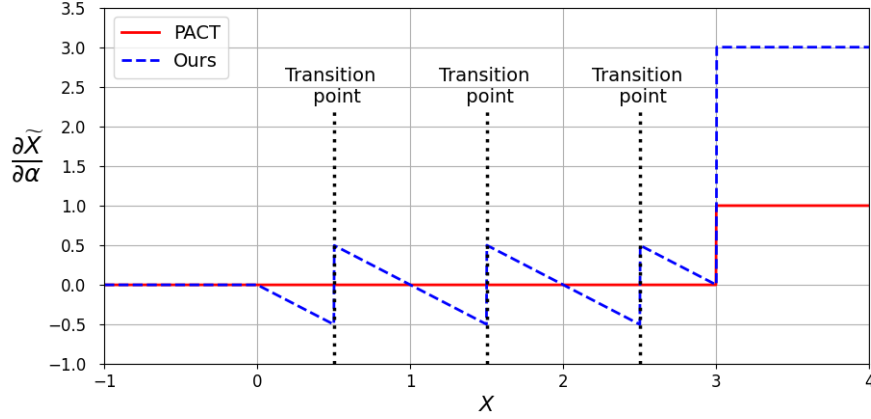


Figure 5.7: The gradient of the quantizer output with respect to the transformation scalar α when quantizing a positive activation data-structure to 2-bit integer values and $\tilde{a}$ is initialized to $\log(\exp(2^2 - 1) - 1)$.

where $QL_{\min} \leq \mathcal{X}/\alpha \leq QL_{\max}$. With such an approximation, PACT only utilizes the gradients from the outside of the clipping interval.

In this work, we provide a simple way to approximate the gradient to the transformation scalar that is sensitive to the quantization transition points. Specifically, we adopt the straight-through estimator [303] to approximate the gradient through the round operation as a pass-through operation, i.e., $\frac{\partial f(\mathcal{X})=\lfloor \mathcal{X} \rfloor}{\partial \mathcal{X}} = 1$. Additionally, we approximate the gradient through the $2^{\lfloor \log_2(x) \rfloor}$ function as a pass-through operation to provide stable update of learnable parameters. Accordingly, the gradient through the quantizer to the transformation scalar is calculated based on the training loss by

$$\frac{\partial \tilde{\mathcal{X}}}{\partial \alpha} = \begin{cases} \lfloor \mathcal{X}/\alpha \rfloor - \mathcal{X}/\alpha, & QL_{\min} \leq \mathcal{X}/\alpha \leq QL_{\max} \\ QL_{\min}, & \mathcal{X}/\alpha < QL_{\min} \\ QL_{\max}, & \mathcal{X}/\alpha > QL_{\max} \end{cases} \quad (5.13)$$

Figure 5.7 compares the gradients of our quantizer output and PACT quantizer output with respect to the transformation scalar. One can notice that the gradient employed by PACT is zero everywhere below the clip point (3 in this example), whereas the gradient employed by our quantizer is sensitive to the distance between inputs and transition points. As the distance from $\mathcal{X}$ to a transition point decreases, $|\partial\tilde{\mathcal{X}}/\partial\alpha|$ increases and thereby providing a finer grained optimization by preventing a large jump in $\tilde{\mathcal{X}}$.

Concerning the gradients of the quantizer output with respect to model parameters, we empirically found that adopting the straight-through estimator for the clipping operation provides a finer grained optimization. In this way, we utilize all the gradients to optimize all the parameters (not only those within the quantization interval) and learn good low-precision weights such that

$$\frac{\partial\tilde{\mathcal{W}}}{\partial\mathcal{W}} = \frac{2^{\lfloor\log_2(\alpha)\rfloor}}{\alpha} \quad (5.14)$$

In [304], the authors show that the best convergence of the training loss can be achieved by making the average update magnitude of each layer parameter equal to its average magnitude. Motivated by that, we consider that each quantizer scalar parameter ($\tilde{a}$) should also have its update magnitude to its parameter magnitude proportion similar to that of the quantizer input data (activations and weights)

$$\frac{\partial\tilde{\mathcal{X}}/\partial\tilde{a}}{\tilde{a}} \approx \frac{\|\partial\tilde{\mathcal{X}}/\partial\mathcal{X}\|_2}{\|\mathcal{X}\|_2} \quad (5.15)$$

However, the update to $\tilde{a}$ grows with the square root of the number of quantized elements. This is because more elements are summed across when computing the gradient of the quantizer output with respect to α as shown in Equation (5.13). To correct for this, we compute the gradient through the transformation scalar to the quantizer scalar parameter as

$$\frac{\partial \alpha}{\partial \tilde{a}} = \frac{1}{\text{QL}_{\max} \times (\exp(-\tilde{a}) + 1)} \times \frac{1}{\sqrt{\mathcal{V}}} \quad (5.16)$$

where $\mathcal{V}$ is the size of the quantized data-structure. The first part of the aforementioned equation represents the normal gradient computed for Equations (5.10) and (5.12), while the second part is used to bring the magnitude of quantizer scalar updates into better balance with quantizer input data updates, and thus improves convergence.

5.4.3 Batch Normalization and Pooling Quantization

The *BatchNorm* layers are widely used in today's DNN architectures specially for accelerating the training phase [288]. This is achieved by normalizing the input activations ($\mathcal{A}$) to have zero mean and unit variance distributions, and then scaling the normalized activations to any other distribution through a linear transform function as

$$\text{BatchNorm}(\mathcal{A}) = \frac{\mathcal{A} - \mu}{\sqrt{\sigma^2 + \epsilon}} \times \gamma + \beta \quad (5.17)$$

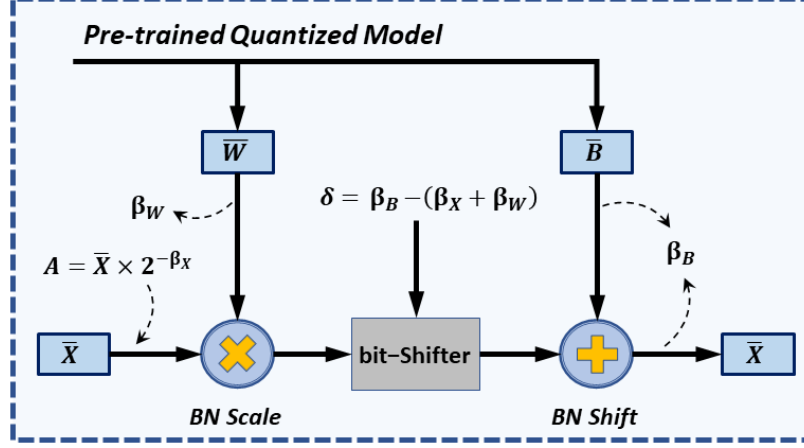


Figure 5.8: An overview of the low-precision computation of the *BatchNorm* layer.

where μ , σ^2 , γ , and β are trainable channel-wise parameters, and ϵ is a small constant used for numerical stability. One can note that this kind of formulation required floating-point arithmetic and thus hinders integer-only deployment. To use integer arithmetic only, the authors in [289] fold *BatchNorm* layer parameters into *CONV* layer parameters before quantization. However, this caused a severe decrease in accuracy because *BatchNorm* parameters are very sensitive to quantization. Instead, we simplify the *BatchNorm* function similar to our previous work in [105] as

$$\text{BatchNorm}(\mathcal{A}) = \mathcal{A} \times \mathcal{W} + \mathcal{B} \quad (5.18)$$

where $\mathcal{W}$ and $\mathcal{B}$ are channel-wise parameters calculated as

$$\begin{aligned} \mathcal{W} &= \frac{\gamma}{\sqrt{\sigma^2 + \epsilon}} \\ \mathcal{B} &= \beta - \frac{\gamma \times \mu}{\sqrt{\sigma^2 + \epsilon}} \end{aligned} \quad (5.19)$$

Then, we carry out the *BatchNorm* computation with integer arithmetic by quan-

tizing $\mathcal{W}$ and $\mathcal{B}$ to 16-bit and 32-bit fixed-point numbers, respectively, as discussed in Section 5.4.1. Thus, the hardware design for *BatchNorm* layer, illustrated in Figure 5.8, requires one multiplier to carry out *BatchNorm* scaling, one adder to carry out *BatchNorm* shift, and a bit-shifter to align the binary point of *BatchNorm* scaling result to that of *BatchNorm* shift parameter before accumulation. In contrast to the *BatchNorm* design proposed in [284] which uses barrel-shifter due to employing a dedicated shift value (δ) for each channel, the bit-shifter parameter δ in this work is an 8-bit signed integer constant for all channels, thereby reducing the hardware complexity of *BatchNorm* implementation.

On the other hand, for *AVG-POOL* layer, the reciprocal of the patch size (named as $\aleph$) is quantized into a 16-bit fixed-point number. Then, the elements of each *AVG-POOL* window are accumulated using the adder tree as shown in Figure 5.4, and the result is multiplied by the integer representation of $\aleph$ in a similar way to *BatchNorm* scaling illustrated in Figure 5.4.

5.5 Learning the Optimal Layer-Wise Quantization

Method

In this section, we discuss the technique employed to determine the optimal quantization method for the weights of *CONV* and *FC* layers. It is noteworthy mentioning that DNN weights can either be quantized with Q_{FXP} or Q_{Log_2} quantizer. On the other hand, the activations are always quantized to fixed-point values through the Q_{FXP} quantizer. To avoid evaluating each candidate quantized solution one-by-one, which

is a time and computational consuming process, the network learns the appropriate quantization method through backpropagation.

Formally, we represent the network as a computational directed acyclic graph whose vertices indicate layer activations and edges denote *CONV/FC* operations. Each two consecutive vertices are connected with two edges, each of which has its own trainable weights $\mathcal{W}_\ell^{(e)}$, bit-precision level $k_{\ell,\mathcal{W}}^{(e)}$ and quantizer scalar $\tilde{a}_{\ell,\mathcal{W}}^{(e)}$ for the low-precision weights, and bit-precision level $k_{\ell,\mathcal{A}}^{(e)}$ and quantizer scalar $\tilde{a}_{\ell,\mathcal{A}}^{(e)}$ for the low-precision activations, as shown in Figure 5.9. Here, e is the edge ID, $e \in \{1, 2\}$, and ℓ is the layer ID, $\ell \in [L]$. To calculate the output of a layer, both edges quantize their input vertex activations in the same way with Q_{FXP} but using their respective bit-precision level and quantizer scalar. Additionally, the first edge quantizes its FP32 weights through Q_{FXP} while the second edge uses Q_{Log_2} to convert its full-precision weights to power-of-2 numbers. Thereafter, they perform their function $\mathcal{F}_\ell(\bullet)$, either a *CONV* or *FC* operation, on the low-precision fixed-point activations $\tilde{\mathcal{A}}_\ell^{(e)}$ and weights $\tilde{\mathcal{W}}_\ell^{(e)}$ as discussed in Section 5.4.1.

To specify which edge output to use during inference, each edge is accompanied with a trainable parameter referred to as $\theta_\ell^{(e)}$. The vector θ is used to stochastically compute the edges mask $\mathcal{M}$. The $\mathcal{M}_\ell^{(e)}$ is a random variable represents the probability of selecting edge e to compute the activations of layer $\ell + 1$, $\mathcal{M}_\ell^{(e)} \in [0, 1]$ and $\sum_{e \in \{1,2\}} \mathcal{M}_\ell^{(e)} = 1$. Hence, the activations $\mathcal{A}_{\ell+1}$ is calculated as follows

$$\mathcal{A}_{\ell+1} = \sum_{e \in \{1,2\}} \mathcal{M}_\ell^{(e)} \times \mathcal{F}_\ell\left(\tilde{\mathcal{A}}_\ell^{(e)}, \tilde{\mathcal{W}}_\ell^{(e)}\right) \quad (5.20)$$

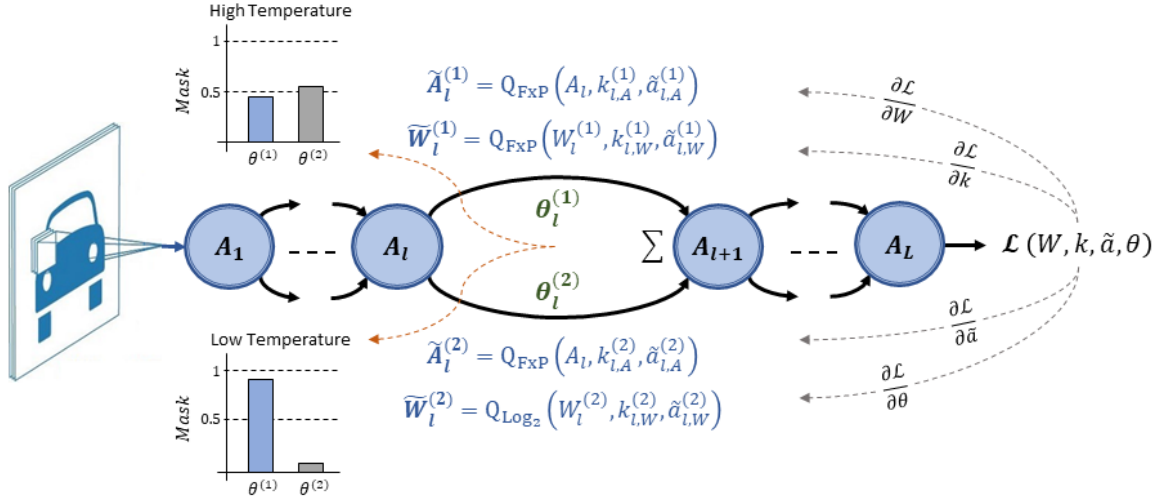


Figure 5.9: An illustration of the stochastic network used to learn the optimal quantization method. The vertices of the stochastic network represent activations, while edges represent $CONV/FC$ operators with different quantization parameters for Q_{FXP} and Q_{Log_2} quantizers.

In this way, solving the optimal quantization method problem is alleviated to solve for the optimal distribution of θ that minimizes the loss function as

$$\min_{\theta} \min_{\mathcal{W}, k, \tilde{a}} \mathcal{L}(\mathcal{W}, k, \tilde{a}, \theta) \quad (5.21)$$

where $\mathcal{L}(\bullet)$ represents the loss function on a target dataset given the architecture characterized by its parameters $\mathcal{W}$, k , $\tilde{a}$, and θ . When considering the differentiability in Equations (5.20) and (5.21), one can notice that the loss function is differentiable with respect to $\mathcal{M}$ and $\tilde{\mathcal{W}}$, hence it is also differentiable with respect to $\mathcal{W}$, k , and $\tilde{a}$ as illustrated in Section 5.4.2. On the other hand, back-propagating the gradients to θ requires defining $\mathcal{M}_{\ell}^{(e)}$ as a continuous random variable or otherwise the learning algorithm will be difficult to converge. For this reason, we adopt the Gumbel Softmax [300] to control the edge selection. The Gumbel Softmax is a well-known tech-

nique for modeling categorical predictors and works with backpropagation. Instead of just executing a single edge through applying a hard sampling, Gumbel Softmax applies a soft sampling and computes the edge mask as

$$\mathcal{M}_\ell^{(e)} = \frac{\exp\left((\theta_\ell^{(e)} + \mathcal{G}_\ell^{(e)})/\mathcal{T}\right)}{\sum_{j \in \{1,2\}} \exp\left((\theta_\ell^{(j)} + \mathcal{G}_\ell^{(j)})/\mathcal{T}\right)} \quad (5.22)$$

where $\mathcal{G}_\ell^{(e)}$ is a random variable sampled from the Gumbel distribution, and $\mathcal{T}$ is a temperature coefficient used to control the behavior of the Gumbel Softmax. When the temperature is very high, $\mathcal{T} \rightarrow \infty$, $\mathcal{M}_\ell^{(e)}$ becomes a continuous random variable following the uniform distribution. Thus, both edges are executed and their outputs are accumulated as illustrated in Equation (5.20). On the other hand, $\mathcal{M}_\ell^{(e)}$ gradually becomes a discrete random variable with a distribution identical to the categorical distribution as the temperature decreases. In other words, as $\mathcal{T} \rightarrow 0$, $\mathcal{M}_\ell$ becomes a one-hot-encoded vector and thus only one edge is executed during evaluation. Thereby, $\mathcal{M}$ has a well-defined gradient with respect to θ and we can update the parameter θ of each layer based on the gradients of the loss function.

To schedule the temperature, we start the training at a high initial temperature $\mathcal{T}_0$. Then, we anneal the temperature in each epoch with the exponential decay schedule using the parameter λ as

$$\mathcal{T} = \mathcal{T}_0 \times \exp(-\lambda \times \text{epoch}) \quad (5.23)$$

where $\mathcal{T}_0$ and λ are hyperparameters for which we select the best-performing quan-

tization method on the validation set and report test performance, and epoch is the training epoch ID.

5.6 Designing Memory-Driven Mixed Low-Precision DNNs

To maximize the throughput of CNN applications on edge devices, a hardware accelerator referred to as a convolutional layer processor (CLP) is typically designed to perform the computationally intensive operations of *CONV* and *FC* layers [55], [73], [92]–[94], [104], [254]. In general, DL accelerators implement a specialized run-time reconfigurable matrix multiplication overlay to efficiently compute the output of *CONV* and *FC* layers. In doing so, they also design customized memory subsystems for temporary input activations, temporary output activations, and weight parameters. As the output activations from one layer become input to the next layer, hardware accelerators are usually designed to use activation memory subsystems in a ping-pong fashion to avoid the data traffic with off-chip memory. On the other hand, to achieve the requirements of critical real-time applications, the whole set of model parameters needs to be presented in on-chip memory for energy efficient and low latency inference. This is because the entire set is required for classifying or detecting objects in a sample input image.

Considering that available commercial FPGA platforms have a limited memory space, and the memory footprint of read-only memory and read-write memory in common microcontroller architectures is a stringent constraint [305], there is a critical

need for a method that decides the optimal layer-wise bit-precision assignment driven by the memory budget of the target platform. Without such a tool, implementing the hardware accelerator on the target edge device may not be feasible. Here, it should be emphasized that the available quantization methods use hand-crafted wordlength allocations to design low-precision DNNs with a minimal possible accuracy drop. Hence, users have to manually select the bit-precision assignments. However, there is no systematic way to determine the optimal bit-precision level for each data-structure that maintains the application’s accuracy. Thus, these methods will likely lead to sub-optimal quantized DNN which may not fit the hardware resources in the target edge device.

In this section, we present an automated method for learning the bit-precision level of DNN data-structures that enables the implementation of the hardware accelerator on a particular target edge device. The proposed method allows users to specify the target memory requirements for activation and weights separately, or the overall memory space for general requirements along with the specifications of the memory subsystems. Then, it learns the optimal quantizers with their associated parameters (bit-precision level k and quantizer scalar $\tilde{a}$) for each data-structure type in each layer that satisfies the memory constraints without noticeable degradation in accuracy.

To learn the optimal bit-precision level for each data-structure of each layer through backpropagation, we defined two gradient formulae; the first one is responsible for reducing the bit-precision level based on the data-structure’s footprint, while the second one is responsible for improving the bit-precision level for those data-structures sensitive to quantization. Specifically, we use SGD to minimize a loss function that consists

of the cross-entropy loss and the hardware-aware loss. While the cross-entropy loss leads to better accuracy, the hardware-aware loss penalizes the memory footprint on the target device to enable DNN deployment using a particular hardware accelerator. The SGD loss function is defined as follows

$$\mathcal{L} = \mathcal{L}_{task-distill} + (\pi_{\mathcal{A}} + \pi_{\mathcal{W}}) \times (\mathcal{L}_{\mathcal{A}} + \mathcal{L}_{\mathcal{W}}) \quad (5.24)$$

where $\mathcal{L}_{task-distill}$ is the cross-entropy loss defined in Equation (5.4) which incorporates both the distillation loss and the standard task loss, $\mathcal{L}_{\mathcal{A}}$ and $\mathcal{L}_{\mathcal{W}}$ are the memory footprint in mega byte (MB) for activations and weights, respectively, for a DNN consisting of L layers, such that

$$\mathcal{L}_{\mathcal{A}} = \frac{1}{2^{23}} \times \left(\mathcal{V}_{\ell,\mathcal{A}} \times \sum_{e=1}^2 \mathcal{M}_{\ell}^{(e)} \times k_{\ell,\mathcal{A}}^{(e)} + \mathcal{V}_{\ell+1,\mathcal{A}} \times \sum_{e=1}^2 \mathcal{M}_{\ell+1}^{(e)} \times k_{\ell+1,\mathcal{A}}^{(e)} \quad , \quad \ell = 0, \dots, L-1 \right) \quad (5.25)$$

$$\mathcal{L}_{\mathcal{W}} = \frac{1}{2^{23}} \times \sum_{\ell=1}^L \mathcal{V}_{\ell,\mathcal{W}} \times \sum_{e=1}^2 \mathcal{M}_{\ell}^{(e)} \times k_{\ell,\mathcal{W}}^{(e)} \quad (5.26)$$

where $k_{\ell,\mathcal{A}}^{(e)}$ and $k_{\ell,\mathcal{W}}^{(e)}$ are the bit-precision levels for activations and weights at the edge e of layer ℓ , respectively, $\mathcal{V}_{\ell,\mathcal{A}}$ and $\mathcal{V}_{\ell,\mathcal{W}}$ are the size of activations data-structure and weights data-structure in layer ℓ , respectively, and $\mathcal{M}_{\ell}^{(e)}$ is the mask for the edge e of layer ℓ .

On the other hand, $\pi_{\mathcal{A}}$ and $\pi_{\mathcal{W}}$ are the memory reduction rates for the activation memory subsystem and the weight memory subsystem, respectively. Our methodology

aims to determine optimal bit-precision levels that satisfy the memory requirements of the target edge device by training the network using the loss function defined in Equation (5.24) for a specified number of epochs. Thus, we want memory reduction rates to be directly proportional to the distance to the target memory space. Also, we want to have higher reduction rates as the training algorithm approaches the end of training to ensure that we reach the target memory requirements by the end of the training phase. Based on these requirements, we defined a reward function $\mathcal{R}_{\mathcal{W}}$ to encourage weight parameters to use low-precision levels as

$$\mathcal{R}_{\mathcal{W}} = \lambda \times \mathcal{R}_{\mathcal{W}} + (1 - \lambda) \times \begin{cases} \left(1 + \frac{\mathcal{L}_{\mathcal{W}} - \Phi_{\mathcal{W}}}{\mathcal{L}_{\mathcal{W}}}\right)^{\exp\left(\sigma \frac{\text{epoch}}{\text{total epochs}}\right)}, & \mathcal{L}_{\mathcal{W}\text{-INT}} > \Phi_{\mathcal{W}} \\ 0, & \text{Otherwise} \end{cases} \quad (5.27)$$

Here, λ and σ are hyperparameters used to smooth the reward function and accelerate the bit-precision reduction process, respectively, $\mathcal{L}_{\mathcal{W}}$ is the current memory space for weight parameters considering k as floating-point values, $\mathcal{L}_{\mathcal{W}\text{-INT}}$ is the current memory space for weight parameters calculated after round k into discrete wordlengths, $\Phi_{\mathcal{W}}$ is the target memory space for weight parameters, epoch is the training epoch ID, and total epochs is the number of training epochs used for designing the low-precision DNN. Then, we defined the memory reduction rate for weight memory subsystem as follows

$$\pi_{\mathcal{W}} = \frac{\mathcal{R}_{\mathcal{W}}}{\mathcal{L}_{\mathcal{W}}} \quad (5.28)$$

The reward function for activations ($\mathcal{R}_{\mathcal{A}}$) and the memory reduction rate for activation memory subsystem ($\pi_{\mathcal{A}}$) are calculated in the same way as that for weights.

Here, it should be emphasized that $\pi_{\mathcal{A}}$ and $\pi_{\mathcal{W}}$ are floating-point numbers that do not require gradients. Thus, during the forward process, reward value is accumulated with the task loss function in Equation (5.24).

The SGD loss function given in Equation (5.24) is responsible for reducing the bit-precision levels based on the memory footprint of data-structures, the larger the memory footprint of a data-structure, the lower the bit-precision level assigned. However, data-structures in different layers have different sensitivity levels to quantization. Therefore, we define gradient formulae to improve the bit-precision level of quantization-sensitive data-structures based on quantization interval $[\text{QL}_{\min}, \text{QL}_{\max}]$ in Equation (5.5), where

$$\text{QL}_{\max} = \begin{cases} 2^{(2^{k-2}-1)}, & \text{symmetric } Q_{\text{Log}_2} \text{ quantization} \\ 2^{k-1} - 1, & \text{symmetric } Q_{\text{FXP}} \text{ quantization} \\ 2^k - 1, & \text{non-negative } Q_{\text{FXP}} \text{ quantization} \end{cases} \quad (5.29)$$

$$\text{QL}_{\min} = \begin{cases} -\text{QL}_{\max}, & \text{symmetric } Q_{\text{Log}_2} \text{ quantization} \\ -\text{QL}_{\max}, & \text{symmetric } Q_{\text{FXP}} \text{ quantization} \\ 0, & \text{non-negative } Q_{\text{FXP}} \text{ quantization} \end{cases} \quad (5.30)$$

The gradient formula for increasing the bit-precision level $\left(\frac{\partial \mathcal{L}}{\partial \text{QL}_{\max}}\right)$ takes into

account the rounding error ($\mathcal{E}_{round}$) and the truncation error ($\mathcal{E}_{trunc}$) defined as

$$\frac{\partial \mathcal{L}}{\partial \text{QL}_{\max}} = \min(0, \mathcal{E}_{trunc}) - \mathcal{E}_{round} \quad (5.31)$$

$$\mathcal{E}_{trunc} = 2^{\lfloor \log_2(\alpha) \rfloor} \times \begin{cases} -\sum_{i=1}^v \partial \mathcal{L} / \partial \tilde{\mathcal{X}}_i, & \mathcal{X} / \alpha < \text{QL}_{\min} \text{ \& symmetric quantization} \\ \sum_{i=1}^v \partial \mathcal{L} / \partial \tilde{\mathcal{X}}_i, & \mathcal{X} / \alpha > \text{QL}_{\max} \\ 0, & \text{Otherwise} \end{cases} \quad (5.32)$$

$$\mathcal{E}_{round} = 2^{\lfloor \log_2(\alpha) \rfloor} \times \begin{cases} \sqrt{\frac{1}{v} \sum_{i=1}^v \left(\partial \mathcal{L} / \partial \tilde{\mathcal{X}}_i \right)^2}, & \text{QL}_{\min} \leq \mathcal{X} / \alpha \leq \text{QL}_{\max} \\ 0, & \text{Otherwise} \end{cases} \quad (5.33)$$

It is noteworthy that the rounding error measure is calculated by the root mean square error because it is a standard way to measure the error of a model in predicting quantitative data.

5.7 Improving the Convergence of Low-Precision

Quantized DNNs with Adaptive Learning Rate

Based on the analysis above, we propose a training method that controls the training of each layer to improve the convergence of the designed low-precision DNNs. The

training algorithm goes into three phases; (i) the first phase aims to learn the quantization parameters and quantization methods that satisfy the memory requirements of the target edge device while achieving optimal performance, (ii) the second phase insures the integer-only deployment of the designed DNN by quantizing *BatchNorm* and *POOL* layers, and (iii) the last phase fine-tune the designed low-precision network with adaptive freezing for *CONV* layer parameters to improve the testing accuracy.

Specifically, in the first phase, we construct a stochastic network with the edge selection parameters θ , weights $\mathcal{W}$, bit-precision level $k_{\mathcal{W}}$ and quantizer scalar $\tilde{a}_{\mathcal{W}}$ for the low-precision weights, and bit-precision level $k_{\mathcal{A}}$ and quantizer scalar $\tilde{a}_{\mathcal{A}}$ for the low-precision activations, as shown in Figure 5.9. We train the network with respect to $(k, \tilde{a})$ and θ separately. Also, we train the network with respect to activations bit-precision reduction and weights bit-precision reduction alternately. Even number epochs are used to learn activation quantization parameters, while odd epochs are used to learn the quantization parameters for weights. Different edges can have a different impact on the overall performance. Therefore, each bit-precision reduction epoch is followed by an epoch to learn optimal edges (θ , selection of optimal quantizer) to increase the probability to sample those edges with better performance and to suppress those with worse performance. At the beginning of each θ epoch, we anneal the temperature $\mathcal{T}$ for Gumbel Softmax with the schedule in Equation (5.23). By the end of this phase, the network will have learned the optimal quantizers with their respective parameters that satisfy the memory constraints of edge devices.

On the other hand, the second phase is used to fine-tune the quantized DNN for a few epochs before quantizing *BatchNorm* layers. Then, to use integer arithmetic

only during deployment, the network is trained to learn the quantization parameters for *BatchNorm* and *POOL* layers as discussed in Section 5.4.3. Finally, the last phase fine-tunes the quantized DNN to improve its accuracy. However, we found that the output activations from *CONV* layers are unstable during this phase, which results in sub-optimal accuracy. The reason for this instability is coming from the low-precision data-structures that keep fluctuating around the transition points due to backpropagation update as they adopt few quantization bins. The lower the bit-precision level gets, the more activation instability can be incurred.

To overcome this issue, we propose to adaptively freeze *CONV* layers which have the highest skew in the statistics of their output activations during training. To measure the activation instability of each *CONV* layer, we calculate the Kullback-Leibler (KL) divergence between the distribution of output activations in the previous training iteration and the distribution of output activations in the current training iteration. We do this calculation for a few number of epochs referred to as the sampling epochs.

By the end of the sampling period, we calculate the average activation instability for each *CONV* layer and assign to each layer a specific number of epochs, between zero and a predefined total number of adaptive learning epochs, to freeze the update of its weight parameters. The number of freeze epochs assigned to each layer is proportional to the amount of layer instability over the highest activation instability found during the last sampling period. This pattern of sampling and adaptive freezing is repeated until the end of the fine-tuning stage. Freezing *CONV* layers that cause the most activation instability removes the fluctuation source, thus allowing the rest

of the layers to converge to a more optimal point.

5.8 Results and Discussions

In this section, we conduct several experiments to demonstrate the consistent effectiveness of our framework by measuring the accuracy of the quantized networks. We implemented the proposed framework in PyTorch and compare it with state-of-the-art methods on the challenging ILSVRC [28] dataset. As our focus is on more efficient models, we extensively study the quantization of MobileNet-V2 [297].

5.8.1 Experimental Settings

The pre-trained MobileNet-V2 model we use in experiments is publicly available in PyTorch Model Zoo. All quantized networks are initialized using weights from a pre-trained full-precision model with equivalent architecture before fine-tuning in the quantized space, which is known to improve performance [306]. Our experiments are performed on the ImageNet dataset. The ILSVRC-2012 ImageNet dataset has about 1.28 million images for training and 50 thousand images for validation, all are natural high-resolution images. Each image is annotated as one of the 1,000 classes. Images were resized to 232×232 , then a 224×224 crop was selected for training, with a horizontal flip applied half the time. At test time, a 224×224 centered crop was chosen.

We use SGD with 0.9 momentum to train the quantized low-precision networks for 90 epochs with a batch size of 256 (128 images per chip). We set the initial learning

rate to $5e-3$ and weight decay to $4e-5$. The reason behind adopting such a low weight decay value is that reducing model precision will reduce the model’s tendency to overfit, and thus also reduce the regularization in the form of weight decay necessary to achieve good performance. We use the cosine learning rate decay with warmup and without restarts [307], [308] to reduce the learning rate. We chose to use cosine learning rate decay in our experiments as it removes the need to select learning rate schedule hyperparameters. Additionally, we use dropout of 0.2.

We use knowledge distillation using the ResNet-101 as a teacher model. We set the softmax temperature $\mathcal{T}$ for knowledge distillation to 1 as we empirically found lower temperatures work better. This is might be because the student model (MobileNet-V2) is very small compared to the teacher model (ResNet-101). If we use a high temperature, the resulting soft-label distribution becomes richer in information, making the student model unable to capture all of this information. We assigned equal weights ($\lambda = 0.5$) to the standard loss and the distillation loss in Equation (5.4). To control Gumbel Softmax functions for learning the optimal quantization method, we use an initial temperature $\mathcal{T}_0$ of 5.0, and we set the decaying factor λ in Equation (5.23) to be 0.096. We set the hyperparameters used to smooth the reward function and accelerate the bit-precision reduction process in Equation (5.27) to 0.99 and 0.3, respectively.

For the final fine-tuning phase of the designed low-precision network, we use 3 epochs for sampling and 7 epochs for the adaptive freezing, with 30 epochs for the entire fine-tuning. To obtain the final inference model, we adopt the model re-parametrization technique to merge multiple computational modules into one at the inference stage. Specifically, we perform an exponentially weighted moving average of

the weight and quantizer scalar parameters of models at different training iteration numbers with a momentum of 0.9997 and use the weighted moving average model purely as the final inference model.

5.8.2 Ablation Study

In this section, we run experiments on the MobileNet-V2 family network using ImageNet dataset to study the effectiveness of each component of the proposed framework. During exploration for the optimal bit-precision level, we randomly select 50 categories from ImageNet dataset and only use their training samples to accelerate the model quantization process. We do not use a teacher model in these experiments. Additionally, we quantize all data-structures using the Q_{FXP} quantizer.

5.8.2.1 Impact of Weight Gradient Pass-Through

In this scenario, we compare weight gradient clipping and weight gradient pass-through when quantizing the MobileNet-V2 model to 2MB (achieve model compression by $6.7\times$). The activation representation adopted in this scenario is a 32-bit floating-point. The purpose of this test scenario is to study the behavior of the clip function defined in Equation (5.5). The default gradient formula through the clip operation is defined as follows

$$\frac{\partial \widehat{\mathcal{W}}}{\partial \mathcal{W}} = \begin{cases} 1/\alpha, & \text{QL}_{\min} \leq \mathcal{W}/\alpha \leq \text{QL}_{\max} \\ 0, & \text{Otherwise} \end{cases} \quad (5.34)$$

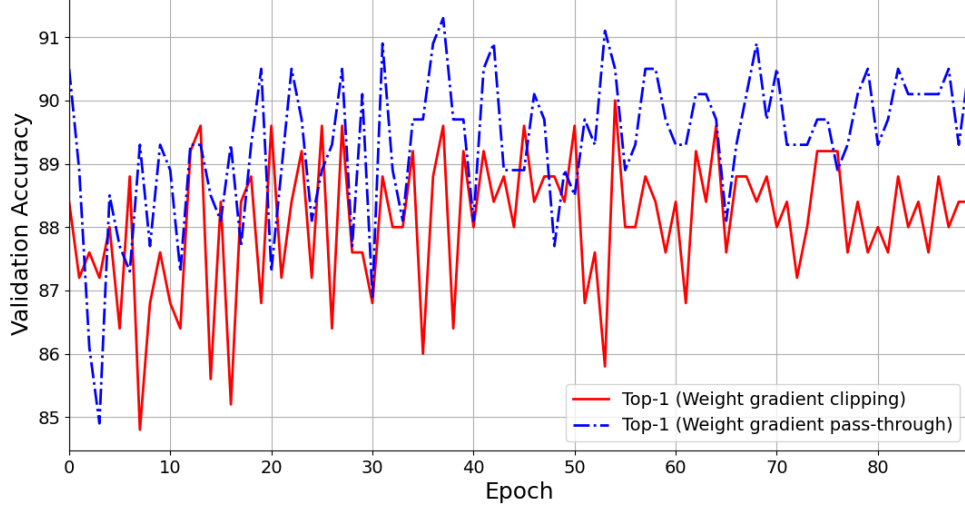


Figure 5.10: The validation top-1 accuracy for a 2MB MobileNet-V2 model when formulating the clip operation to perform weight gradient clipping and weight gradient pass-through.

Using this formulation, weights whose transformed values ($\mathcal{W}/\alpha$) are outside the quantization interval will not be updated. To allow all weight parameters to update their values during backpropagation, we adopt the straight-through estimator to approximate the gradient through the clip operation as a pass-through operation, such that $\partial \widehat{\mathcal{W}} / \partial \mathcal{W} = 1/\alpha$ for all weight parameters. Figure 5.10 shows the top-1 accuracy for both techniques during the bit-precision reduction process. As evident from the figure, allowing all weight parameters to update their values through the backpropagation leads to better convergence (about 1.8% improvement in the validation accuracy). Therefore, we adopt the weight gradient pass-through technique in all subsequent experiments.

5.8.2.2 Impact of Alternately Quantizing Weights and Activations

In this scenario, we impose a specific target memory space for activations and weights. Memory constraints of 512KB and 2MB were set for activation and weights, respectively. Here, we consider two test cases; (i) the first test case quantizes activations and weights at the same time to satisfy the target memory space, and (ii) the second test case quantizes activations and weights alternately one epoch for each. In other words, both activation data-structures and weight data-structures can reduce their bit-precision levels over the same training iteration in the first case, while on the other hand, either activations or weights can reduce their bit-precision level over a single training epoch in the second case. From the results shown in Figure 5.11, we can conclude that joint quantization is definitely not the right quantization methodology. The alternate quantization of activations and weights resulted in a more accurate model that had a 4.3% higher validation top-1 accuracy.

5.8.3 Accuracy on Specific Memory Constraints for Weights and Activations

In this experiment, we quantize MobileNet-V2 model for an edge device with a memory system of size 2.5MB. We use our proposed method to satisfy the memory requirement by utilizing all the components with the settings discussed in Section 5.8.1. Here, we assume that the framework has the flexibility to divide the available memory space between activations and weights to show its effectiveness in assigning the bit-precision levels. For each epoch of training, the network learns the wordlength assignment for

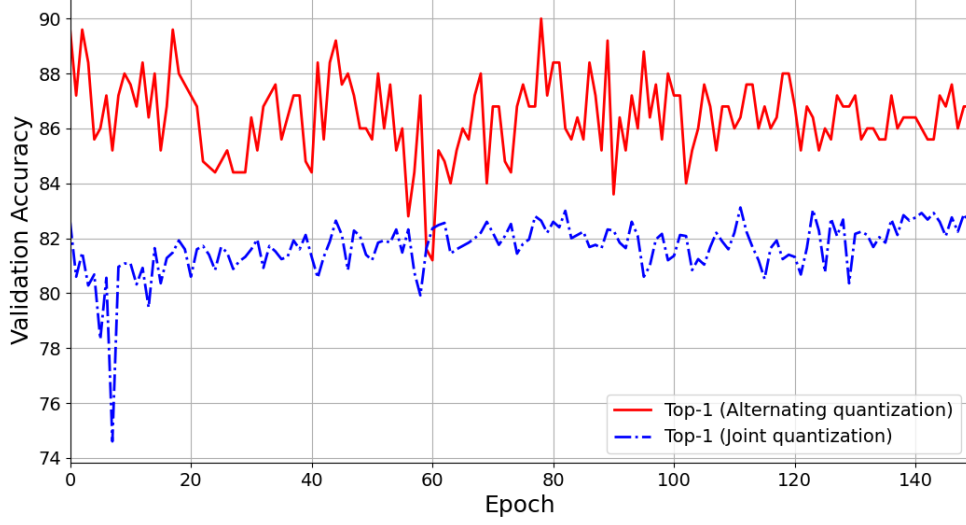


Figure 5.11: The validation top-1 accuracy for the low-precision MobileNet-V2 model that jointly quantizes activations and weights to satisfy the memory requirements of 512KB for activations and 2MB for weights compared to that model that achieves the same memory requirements but by quantizing data-structures alternately.

either the activations or the weights and thereafter learns the quantizer method that minimize the SGD loss function which includes the task loss function, the distillation loss function, and the loss function used to satisfy the given memory requirements.

In Figure 5.12, we present the task-loss and the hardware-aware loss defined in Equation (5.24). During the early 10 epochs, we can see that the framework is focused on reaching design solutions whose memory footprint meets the specified target due to the high hardware-aware loss. Therefore, the memory space requirements are strictly reduced during these epochs as depicted in Figure 5.13. After that, the framework successfully reached solutions that satisfy the memory requirements and continued to optimize the assignment of bit-precision levels and quantizers of each edge in the stochastic network.

We can also notice that in each epoch, the loss is high at the beginning, and

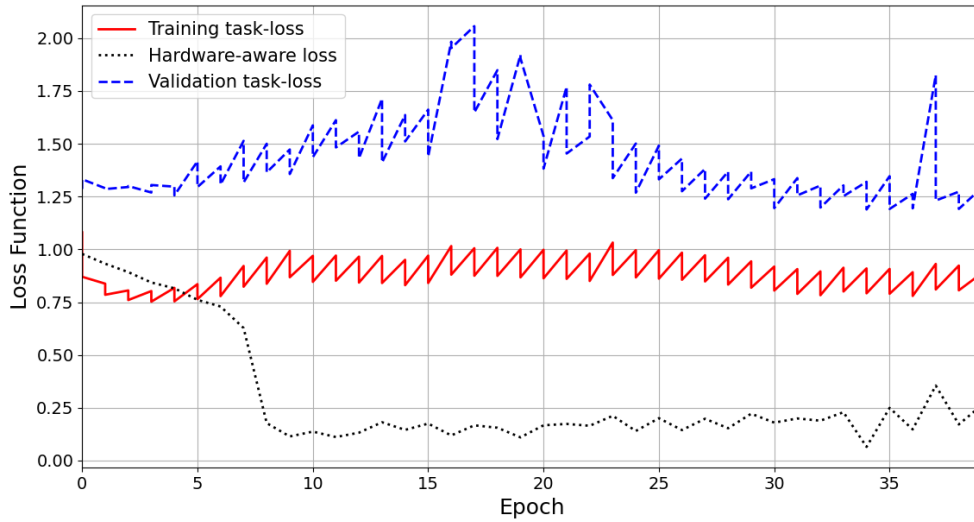


Figure 5.12: The loss function on ImageNet dataset during the exploration for the optimal bit-precision level and the optimal quantizer assignment that satisfy the target memory requirement of 2.5MB.

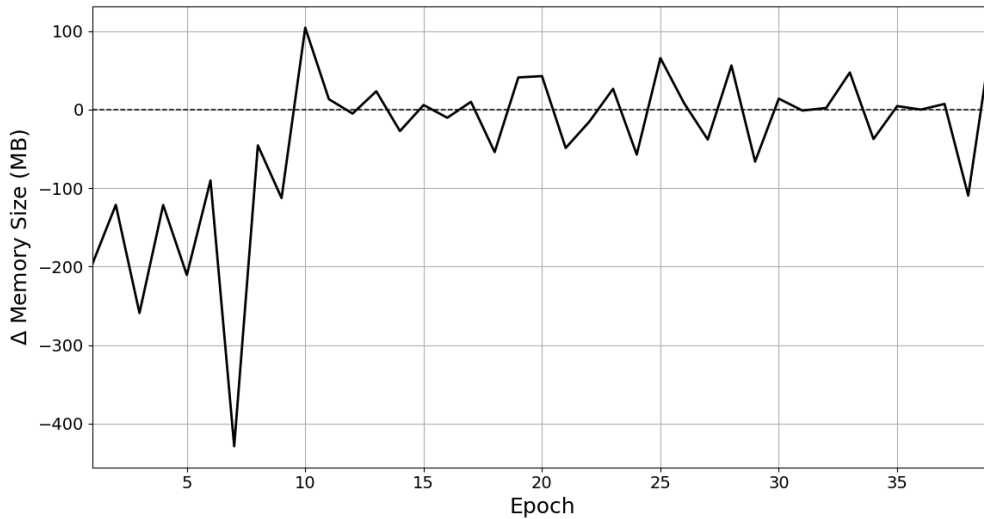


Figure 5.13: The mount of memory reduction, in MB, during the exploration for the optimal bit-precision level and the optimal quantizer assignment that satisfy the target memory requirement of 2.5MB.

then decreases. The reason is that the first operation in each epoch is to optimize the bit-precision assignment, which causes the activations to become unstable due

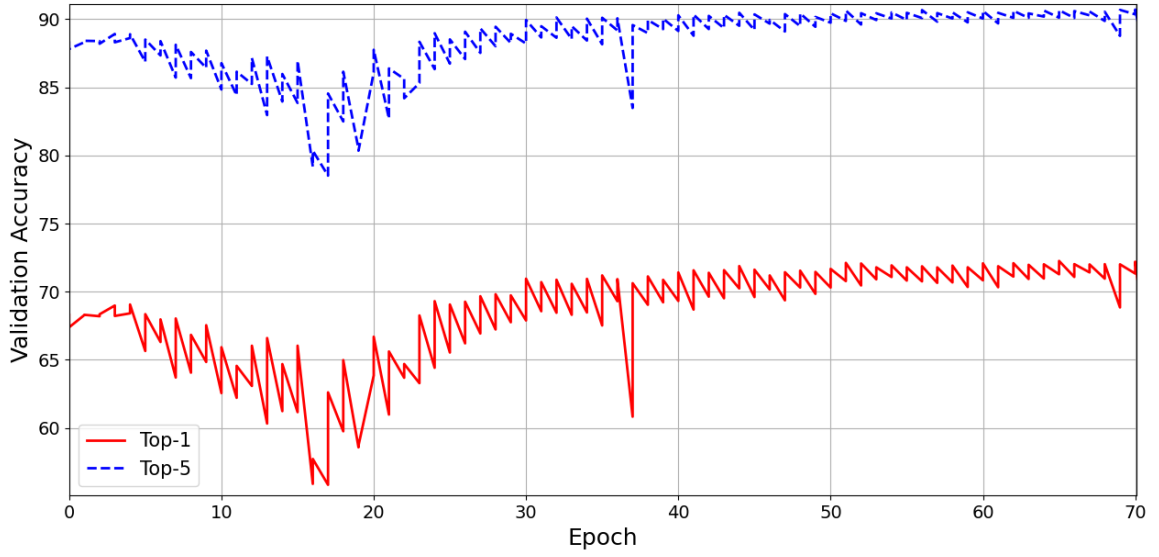


Figure 5.14: The validation accuracy on ImageNet dataset during the exploration for the optimal bit-precision level and the optimal quantizer assignment that satisfy the target memory requirement of 2.5MB.

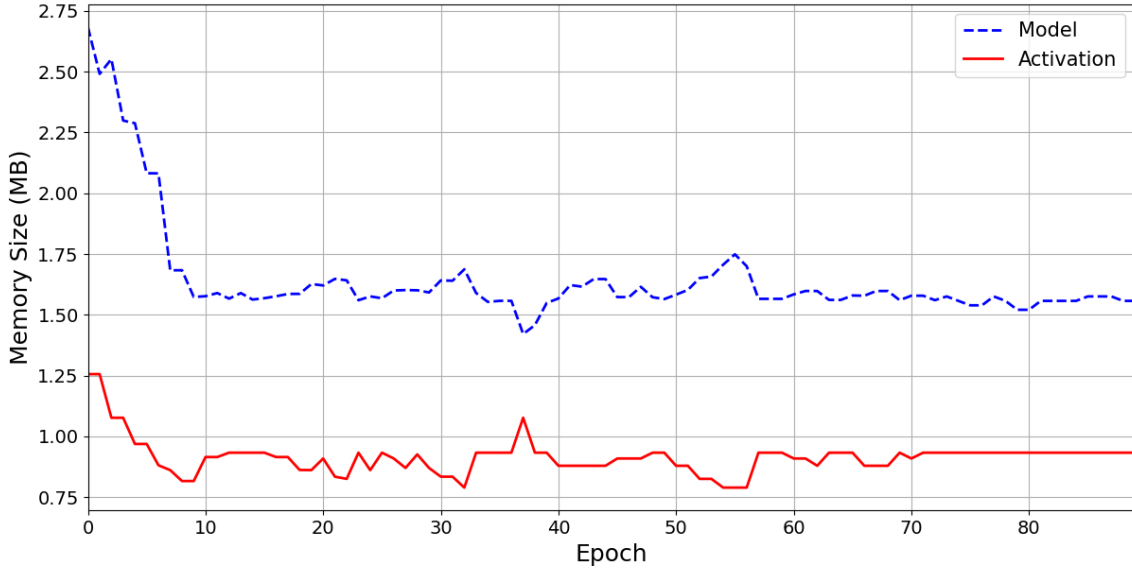


Figure 5.15: The memory size for activations and model parameters during the exploration for the optimal bit-precision level and the optimal quantizer assignment that satisfy the target memory requirement of 2.5MB.

to the large change in the quantization bins which leads to sub-optimal running-mean and running-variance of the *BatchNorm* layers. On the other hand, the second

Table 5.2: The validation accuracy after each design/optimization step.

Design/Optimization Step	Validation Accuracy	
	Top-1	Top-5
Baseline floating-point model	71.56%	90.21%
Baseline floating-point model (+ teacher-student knowledge transfer)	72.54%	91.02%
8-bit fixed-point weights and activations network	71.83%	90.28%
1.57MB MobileNet-V2 model with a 0.93MB activations memory	72.51%	90.92%
1.57MB MobileNet-V2 model with a 0.93MB activations memory (+ <i>BatchNorm</i> quantization)	70.17%	89.61%
1.57MB MobileNet-V2 model with a 0.93MB activations memory (+ truncated activations)	70.11%	89.49%
1.57MB MobileNet-V2 model with a 0.93MB activations memory (+ adaptive learning rate)	72.31%	90.65%

operation in each epoch is to optimize θ to choose the optimal quantizers, since the activations are almost stable during this operation, *BatchNorm* layers can learn finer running-mean and running-variance. As the training algorithm progresses further, the network converges to a more accurate design as shown in Figure 5.14. This proves the effectiveness of the proposed framework in balancing the memory space allocated for activations and weights illustrated in Figure 5.15.

In Table 5.2, we present the validation accuracy after each design step of the training algorithm discussed in Section 5.7. Overall, our framework was able to reduce memory requirements for activations and weights by $7.8\times$ with only 0.23% and 0.37% decrease in validation top-1 and top-5 accuracies, respectively. This improves the top-1 accuracy for a previously published integer-only deployment [284] with the same memory constraints by 4.3%.

In Figures 5.16 and 5.17, we visualize the wordlength allocated and the size of weights and input activations for each *CONV* layer of MobileNet-V2 architecture.

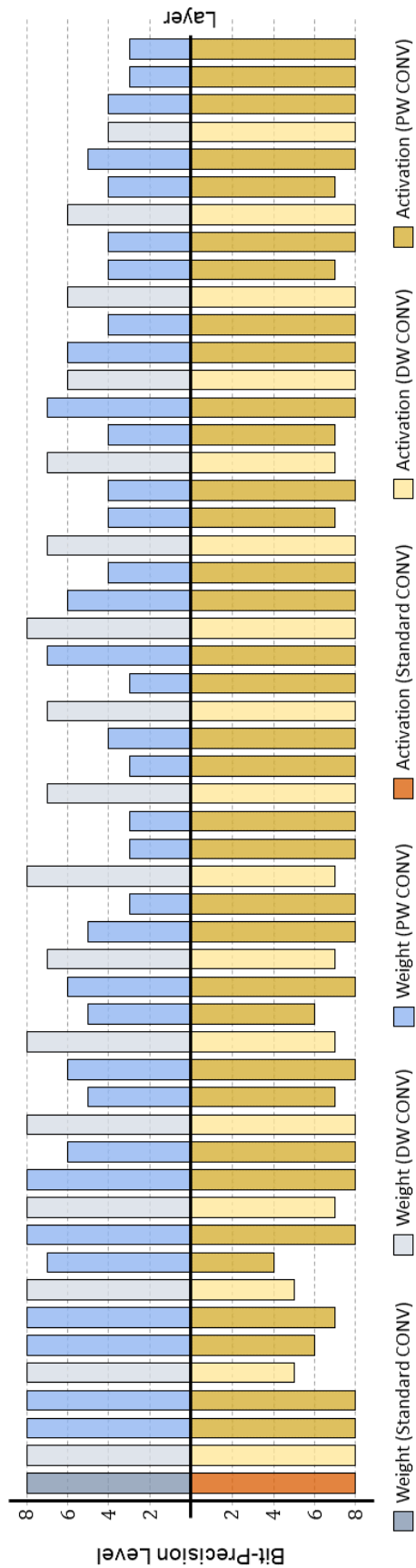


Figure 5.16: Wordlength allocated to weights and input activations for each *CONV* layer of MobileNet-V2 architecture.

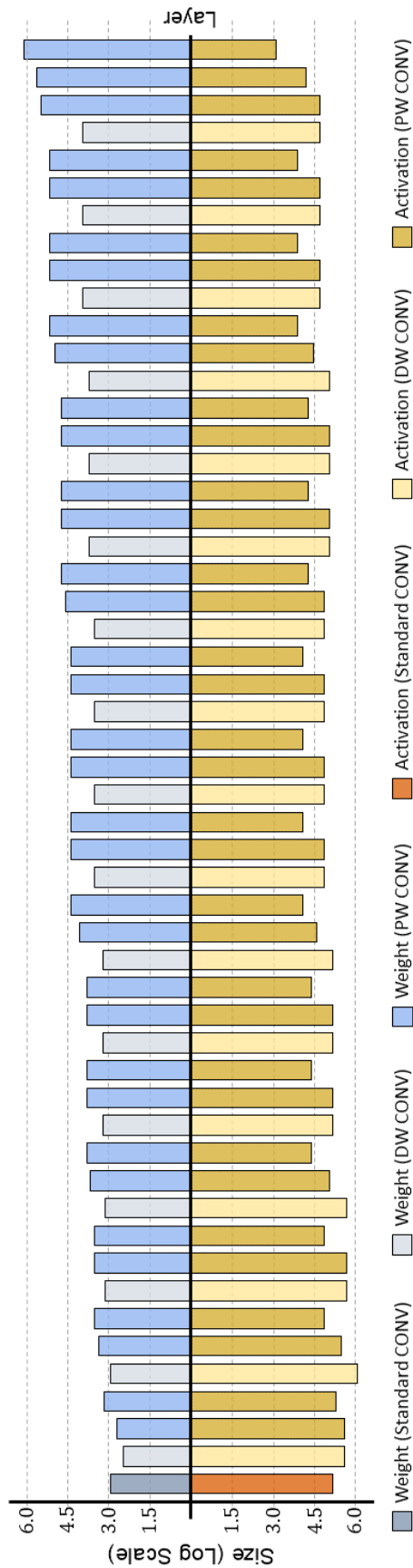


Figure 5.17: Statistics on size of weights and input activations for each *CONV* layer of MobileNet-V2 architecture in log-domain.

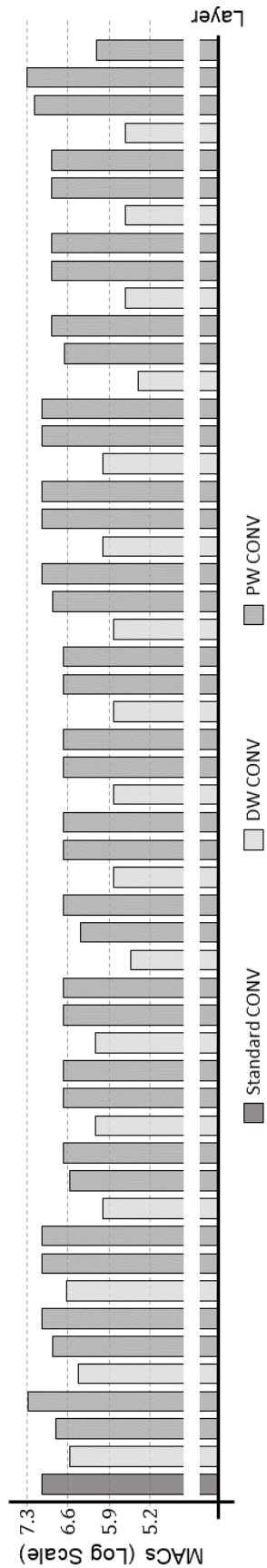


Figure 5.18: Statistics on MAC operations performed by each *CONV* layer of MobileNet-V2 architecture in log-domain.

From these figures, we can observe that our framework assigns more bits to the weights in depth-wise *CONV* layers than point-wise *CONV* layers to preserve the network accuracy at low memory overhead. Intuitively, this is because the number of parameters in the former is much smaller than in the latter. On the other hand, since the activation size in the deeper layers gets smaller, they get assigned more bits. Additionally, our framework allocates fewer bits to point-wise *CONV* layers to prevent them from being computation bounded. This is because they have a large amount of MAC operations as shown in Figure 5.18.

Another important metric to focus on is the classification accuracy of each category to ensure that the quantization framework does not sacrifice one or a subset of the categories to maintain overall accuracy. Figures 5.19 and 5.20 present statistics on validation top-1 and top-5 accuracies for each category. One can notice that the low-precision quantized MobileNet-V2 maintains the number of categories accurately classified for high top-1 and top-5 levels. More importantly, it improves the top-1 accuracy level for imprecise classified categories from $[10 - 14)$ to $[14 - 18)$.

Focusing on the top 10 inaccurately classified categories depicted in Table 5.3, we can see that 70% of inaccurately classified categories by low-precision quantized integer-only network were already in the top 10 inaccurately classified categories by the the baseline floating-point network. However, their top-1 and top-5 accuracies improved by 3.7% and 3.4% on average, respectively. Regarding the remaining 30% of categories that were not previously in the top 10 list of inaccurately classified categories, although their top-1 accuracy decreased by 7.3% on average, they still classify with better accuracy than the most inaccurate classified category in the base-

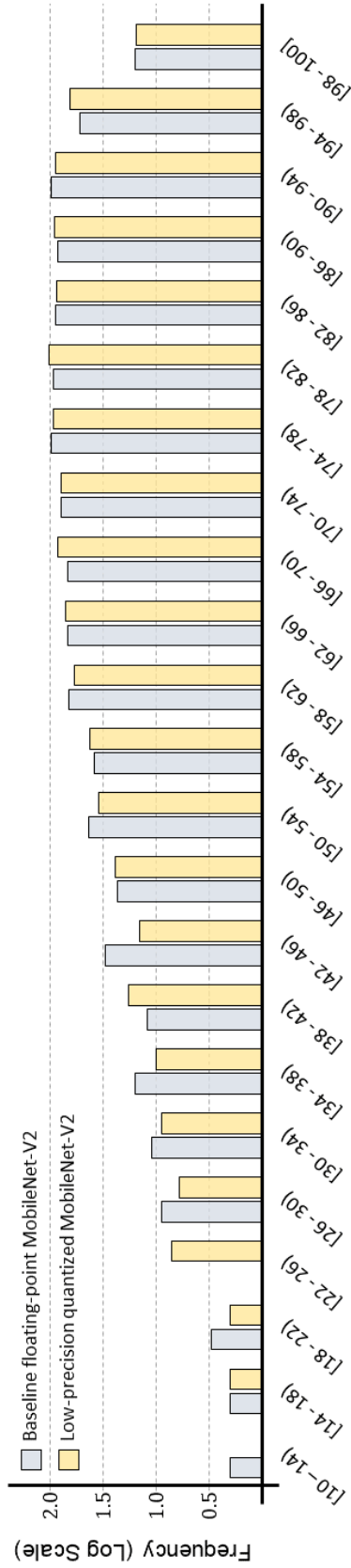


Figure 5.19: Statistics on validation top-1 accuracy for each class in ImageNet dataset on MobileNet-V2 architecture in log-domain.

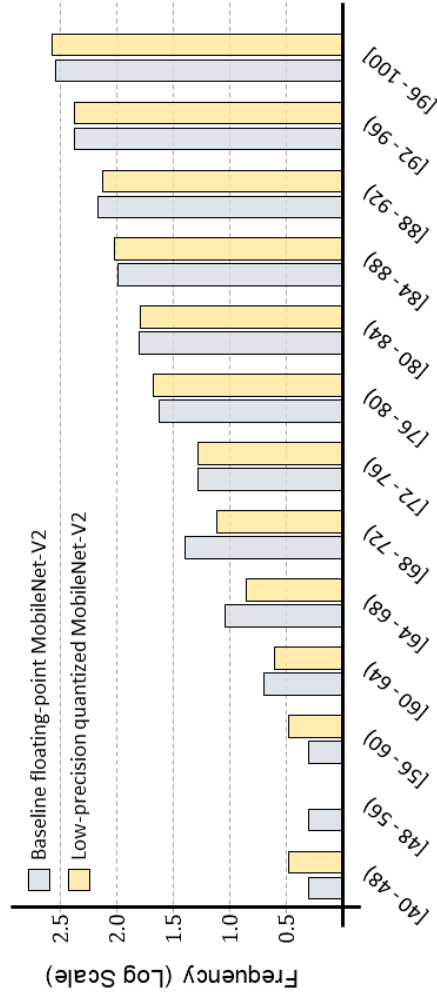


Figure 5.20: Statistics on validation top-5 accuracy for each class in ImageNet dataset on MobileNet-V2 architecture in log-domain.

Table 5.3: The validation accuracy for the top 10 inaccurately classified categories.

Baseline Floating-Point MobileNet-V2			Low-Precision Quantized MobileNet-V2		
Category	Validation Accuracy		Category	Validation Accuracy	
	Top-1	Top-5		Top-1	Top-5
Cleaver, chopper	26%	68%	Hook, claw	24%	70%
Letter opener, paper knife	26%	46%	Projectile, missile	22%	92%
Cassette player	22%	92%	Overskirt	22%	76%
Overskirt	20%	76%	Screen, CRT screen	22%	76%
Hook, claw	20%	60%	Sunglass	22%	58%
Tiger cat	18%	88%	Tiger cat	20%	92%
Laptop, laptop computer	16%	84%	Water jug	18%	84%
Sunglass	16%	62%	Velvet	16%	44%
Screen, CRT screen	12%	70%	Maillot	14%	94%
Velvet	10%	40%	Laptop, laptop computer	12%	88%

line floating-point network, and their top-5 accuracy improved by 4.3% on average. This proves that the proposed framework does not sacrifice some categories in order to satisfy the memory requirements while maintaining overall accuracy.

5.8.4 Comparison with Prior Work

In this section, we conduct three experiments to reduce the memory footprint of MobileNet-V2 to a specific target size and then compare the accuracy and latency of the networks designed using our framework with those designed by PACT [263] and HAQ [283] quantization techniques. While PACT uses the same number of bits for all layers, HAQ adopts flexible wordlengths for activations and weights like our framework. However, both PACT and HAQ use 8-bit weights and activations for the first and last layers.

Specifically, we adopt PACT to quantize MobileNet-V2 weights and activations

Table 5.4: The validation accuracy and latency of low-precision MobileNet-V2 models designed by our framework and PACT and HAQ methods.

Benchmark	Bit-Precision	Validation Accuracy		Latency (ms)
		Top-1	Top-5	
PACT [263]	4	61.39%	83.72%	52.15
HAQ [283]	<i>mixed-precision</i>	66.99%	87.33%	52.12
Ours	<i>mixed-precision</i>	66.74%	87.12%	51.69
PACT [263]	5	68.84%	88.58%	66.94
HAQ [283]	<i>mixed-precision</i>	70.90%	89.91%	66.92
Ours	<i>mixed-precision</i>	70.99%	89.85%	66.68
PACT [263]	6	71.25%	90.00%	82.49
HAQ [283]	<i>mixed-precision</i>	71.89%	90.36%	82.34
Ours	<i>mixed-precision</i>	71.96%	90.38%	82.27

to 4, 5, and 6 bits. Then, we use our framework to design low-precision networks with the memory footprint of PACT’s low-precision networks as the target memory space. Then, we deployed these networks on Xilinx Zynq-7020 FPGA platform using BISMO [254] hardware accelerator that is running at 200 MHz and utilizing 86% and 92% of the look-up table (LUT) and block random-access memory (BRAM) resources, respectively. BISMO accelerates the matrix multiplication operation of *CONV* and *FC* layers by expressing the matrix multiplication as a weighted sum of binary matrix multiplications. Then, it uses bit-serial multipliers which are fed with one-bit digits from $8 \times 8 \times 4$ weights and corresponding activations in parallel to carry out the dot product operation. The results shown in Table 5.4 demonstrate the effectiveness of our framework in achieving a good trade-off between accuracy and latency.

5.9 Conclusion

In this chapter, we presented a novel end-to-end methodology for enabling the deployment of DNNs on resource-limited devices while maintaining model’s accuracy. Specifically, we proposed a framework for quantizing DNN data-structure to integer numbers based on a mixed-precision scheme. The proposed framework learns the bit-precision levels that satisfies the memory limitations of a given device and minimizes the task loss to preserve model’s performance. This is achieved through the SGD, which learns the mixed low-precision levels along with model parameters. Additionally, we proposed to quantize the weights with either the fixed-point quantizer or the $\log_2$ -based quantizer. The quantization method is also learned during model training using the Gumbel Softmax technique aiming to represent the FP32 values accurately at low wordlength while maintaining their dynamic range.

The learning process is extremely fast compared with previous RL based methods. The loss used to train the network consists of both the cross-entropy loss that leads to better accuracy and the memory loss that penalizes the network’s memory requirements on a target device. To maintain the accuracy of the quantized network, we employed the knowledge distillation technique to transfer the knowledge from a deeper network to the low-precision quantized network. By representing activations and weights as integer values, the DL hardware accelerator achieves efficient DNN inference as the expensive floating-point operations are avoided during the feed-forward computations. The experimental evaluation is conducted over the MobileNet-V2 family networks on ImageNet dataset. The experimental results demonstrate an end-to-

end deployment of an integer-only MobileNet network with top-1 accuracy of 72.3% on a device with only 1.57MB of model parameter memory and 0.93MB of activations memory, improving by 4.3% the top-1 accuracy of previously published integer-only deployment with the same memory constraints.

CHAPTER 6

CONCLUSION AND FUTURE RESEARCH DIRECTIONS

6.1 Conclusion

Convolutional neural networks (CNNs) have shown great performance in a wide range of machine learning applications. However, they require significant computational power that cannot be met by general-purpose processors. Nowadays, field programmable gate arrays (FPGAs) have been found to deliver excellent performance in accelerating CNNs due to their ability to create custom designs with different levels of parallelism. Furthermore, FPGAs provide better performance per watt compared to other computing technologies such as graphic processing units (GPUs). In this dissertation, we provided a couple of works for addressing the requirements for efficient implementations of CNNs. These works optimize CNN implementation on FPGA platforms in terms of throughput, latency, energy efficiency, and power consumption. Particularly, we proposed and investigated three works, (i) a CNN accelerator and an accompanying

automated design methodology that employs metaheuristics for optimal partitioning of available FPGA resources to design a high-throughput multiple convolutional layer processors (CLPs), (ii) a framework, referred to as the fixed-point quantizer of deep neural networks (*FxP-QNet*), to efficiently quantize the data-structures of pre-trained CNN-based models to low-precision numbers in fixed-point representation, and (iii) a novel end-to-end memory-driven methodology to enable deployment of CNNs on resource-constrained edge-devices while maintaining model’s accuracy.

In Chapter 2, we reviewed recent developments in the area of acceleration of deep learning neural networks (DNNs) and, in particular, CNNs on FPGAs. The chapter provided a brief overview of deep learning (DL) techniques highlighting their importance, key operations, and applications. Special emphasis is given on CNNs as they have wide applications in the area of detection and recognition and require both central processing unit (CPU) and memory-intensive operations that can be effectively accelerated utilizing FPGA inherent ability to maximize parallelism of operations.

While the chapter briefly touches upon the acceleration techniques for DL algorithms and CNNs from both software and hardware perspectives, the core of that chapter has been the review of recent techniques employed in the acceleration of CNNs on FPGAs. A thorough up-to-date review is provided that illustrates the employment of various possibilities and techniques such as exploitation of parallelism utilizing loop tiling and loop unrolling, effective use of internal memory to maximize data reuse, operation pipelining, and effective use of data sizes to minimize memory footprint, and, to optimize FPGA resource utilization.

The chapter also presented the use of tools for generating register transfer level

(RTL) scripts that not only help in automating the design process but also help in exploring the design space and suggesting efficient hardware. The chapter discusses the use of analytics such as (i) workload analysis in determining the computations that can be parallelized, (ii) optimal loop unrolling factors, (iii) determining access patterns to improve data locality, etc. In addition, a brief review of the use of non-deterministic heuristics in solving NP-hard combinatorial optimization problems in the design and implementation of CNNs has been presented. Finally, the chapter summarized the key features employed by the various FPGA-based CNN acceleration techniques and provided recommendations for enhancing the effectiveness of utilizing FPGAs in CNNs acceleration.

Most of the existing FPGA-based accelerators use a single CLP for processing all convolution (*CONV*) layers. However, the current trend in CNN acceleration is using multiple CLPs to make efficient use of hardware resources. The multiple CLPs design involves optimizing the number of CLPs used, assignment of *CONV* layers to CLPs, the parallelization factors $\langle T_n, T_m \rangle$ for each CLP, and the tiling parameters $\langle T_r, T_c \rangle$ for each *CONV* layer. This imposes an intractable design space where it is almost impossible to find near-optimal configurations through exhaustive search algorithms. Therefore, in Chapter 3, we employed simulated annealing (SA) and tabu search (TS) algorithms to design CNN accelerators using multiple CLPs on FPGAs targeting the optimization of the number of execution cycles.

Experimental results on accelerating AlexNet, GoogLeNet, SqueezeNet 1.1, and VGG-16 architectures demonstrated the effectiveness of the proposed optimization framework in providing efficient accelerators. The results also shown the importance

of using metaheuristics in finding different configurations for a given CNN using a systematic methodology. Compared to a single CLP design, SA-based Multi-CLP and TS-based Multi-CLP accelerate CNN computations by about $1.31\times - 2.37\times$ when targeting Xilinx Virtex-7 FPGAs. Our implementation achieves a performance of 113.92 GFLOPs under 100 MHz working frequency.

In Chapter 4, we presented *FxP-QNet*, a framework to efficiently quantize DNN data-structures to low-precision numbers in dynamic fixed-point representation. Specifically, the *FxP-QNet* computes the low-precision representation of DNN data-structures as mentioned in Equations (4.10) and (4.13). Furthermore, we experimentally demonstrated that optimizing the quantization parameters for each layer and data-structure independently produces a local optimum solution. Thus, *FxP-QNet* selects the quantization parameters that minimize the post-training self-distillation error as well as network prediction errors as summarized in Algorithm 5. Thereafter, the chapter presented an iterative accuracy-driven bit-precision level reduction framework. *FxP-QNet* employs this framework to flexibly design a quantized DNN with a mixed-precision taking into account the robustness of data-structures to quantization as illustrated in Algorithm 6.

With all the techniques aforementioned, *FxP-QNet* transforms current DNNs to hardware-friendly networks while still preserving accuracy at a high rate. The appealing aspects in our framework are as follows; (i) designing a pure fixed-point DNNs without expensive floating-point operations and thus providing unique computational and hardware efficiencies, (ii) adopting layer-wise quantization and therefore there is almost no additional hardware cost overhead for low-precision scaling as the bit-shift

operation is used for this, (iii) reducing quantization-induced degradation enough so as to make quantization-aware training unnecessary, (iv) employing a simple and very fast-to-implement quantization process, and (v) facilitating rapid deployment of pre-trained off-the-shelf models. Finally, the chapter demonstrated the effectiveness of *FxP-QNet* through extensive experiments on several CNN-based architectures on the benchmark ImageNet dataset. The results shown that *FxP-QNet*-quantized AlexNet, VGG-16, and ResNet-18 can reduce the overall memory requirements by $7.16\times$, $10.36\times$, and $6.44\times$ with only 0.95%, 0.95%, and 1.99% accuracy drop on ImageNet validation dataset, respectively.

In Chapter 5, we presented a novel end-to-end memory-driven methodology for enabling the deployment of DNNs on resource-limited devices while maintaining the model’s accuracy. Specifically, we proposed a framework for quantizing DNN data-structure to integer numbers based on a mixed-precision scheme. The proposed framework learns the bit-precision levels that satisfy the memory limitations of a given device and minimizes the task loss to preserve the model’s performance. This is achieved through the stochastic gradient descent (SGD), which learns the mixed low-precision levels along with model parameters. Additionally, we proposed to quantize the weights with either the fixed-point quantizer or the $\log_2$ -based quantizer. The quantization method is also learned during model training using the Gumbel Softmax technique aiming to represent the 32-bit floating-point (FP32) values accurately at low wordlength while maintaining their dynamic range.

The learning process is extremely fast compared with previous reinforcement learning (RL) based methods. The loss used to train the network consists of both the

cross-entropy loss that leads to better accuracy and the memory loss that penalizes the network’s memory requirements on a target device. To maintain the accuracy of the quantized network, we employed the knowledge distillation technique to transfer the knowledge from a deeper network to a quantized network. By representing activations and weights as integer values, the DL hardware accelerator achieves efficient DNN inference as the expensive floating-point operations are avoided during the feed-forward computations. The experimental evaluation is conducted over the MobileNet-V2 family networks on ImageNet dataset. The experimental results demonstrate an end-to-end deployment of an integer-only MobileNet network with top-1 accuracy of 72.3% on a device with only 1.53MB of model parameter memory and 0.97MB of activations memory, improving by 4.3% the top-1 accuracy with respect to previously published implementation.

6.2 Future Research Directions

In future work, we consider improving the quality of the resource usage model, discussed in Chapter 3, in predicting digital signal processor (DSP) slices required by taking into account those used in control logic, loop indexing, and address calculation.

Evaluating the presented technique in Chapter 4 on other deep learning architectures such as recurrent neural networks (RNNs) and Transformer models is an interesting future work. Moreover, the work in Chapter 4 was evaluated on an image classification task. It would be interesting to see how *FxP-QNet* would perform on other related tasks like object detection, and image segmentation or even in other

domains such as natural language processing.

Future work shall also explore channel-wise quantization, which could be potentially useful for the rigorous quantization of advanced network structures like MobileNets. In future work, we consider improving the CLP design by implementing a bit-serial matrix multiplication overlay, which allows for increasing the operating frequency, and thus minimizing the latency of the deployed low-precision DNNs.

REFERENCES

- [1] Y. Bengio *et al.*, “Learning deep architectures for ai,” *Foundations and trends® in Machine Learning*, vol. 2, no. 1, pp. 1--127, 2009.
- [2] J. Schmidhuber, “Deep learning in neural networks: An overview,” *Neural networks*, vol. 61, pp. 85--117, 2015.
- [3] I. Goodfellow, Y. Bengio, A. Courville, and Y. Bengio, *Deep learning*. MIT Press Cambridge, 2016, vol. 1.
- [4] L. Zhang, S. Wang, and B. Liu, “Deep learning for sentiment analysis: A survey,” *Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery*, p. e1253, 2018.
- [5] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, “Learning representations by back-propagating errors,” *Nature*, vol. 323, no. 6088, p. 533, 1986.
- [6] Rumelhart, David E and Hinton, Geoffrey E and Williams, Ronald J, “Neurocomputing: Foundations of research,” *ch. Learning Representations by Back-propagating Errors*, pp. 696--699, 1988.

- [7] M. A. Nielsen, *Neural networks and deep learning*. Determination Press, USA, 2015, vol. 25.
- [8] T. Weyand, I. Kostrikov, and J. Philbin, “Planet-photo geolocation with convolutional neural networks,” in *European Conference on Computer Vision*. Springer, 2016, pp. 37--55.
- [9] Mathworks. What Is Deep Learning? [Online]. Available: <https://www.mathworks.com/discovery/deep-learning.html/>, 2018.
- [10] Y. LeCun, Y. Bengio, and G. Hinton, “Deep learning,” *Nature*, vol. 521, no. 7553, p. 436, 2015.
- [11] A. Deshpande. A Beginner’s Guide To Understanding Convolutional Neural Networks [Online]. Available: <https://adeshpande3.github.io/A-Beginner%27s-Guide-To-Understanding-Convolutional-Neural-Networks/>, 2018.
- [12] J. Hauswald, Y. Kang, M. A. Laurenzano, Q. Chen, C. Li, T. Mudge, R. G. Dreslinski, J. Mars, and L. Tang, “Djinn and tonic: DNN as a service and its implications for future warehouse scale computers,” in *ACM SIGARCH Computer Architecture News*, vol. 43, no. 3. ACM, 2015, pp. 27--40.
- [13] J. Yue-Hei Ng, M. Hausknecht, S. Vijayanarasimhan, O. Vinyals, R. Monga, and G. Toderici, “Beyond short snippets: Deep networks for video classification,” in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2015, pp. 4694--4702.

- [14] Y. LeCun, B. E. Boser, J. S. Denker, D. Henderson, R. E. Howard, W. E. Hubbard, and L. D. Jackel, “Handwritten digit recognition with a back-propagation network,” in *Advances in neural information processing systems*, 1990, pp. 396--404.
- [15] P. Barros, S. Magg, C. Weber, and S. Wermter, “A multichannel convolutional neural network for hand posture recognition,” in *International Conference on Artificial Neural Networks*. Springer, 2014, pp. 403--410.
- [16] A. Graves, A.-r. Mohamed, and G. Hinton, “Speech recognition with deep recurrent neural networks,” in *Acoustics, speech and signal processing (icassp), 2013 IEEE international conference on*. IEEE, 2013, pp. 6645--6649.
- [17] P.-S. Huang, X. He, J. Gao, L. Deng, A. Acero, and L. Heck, “Learning deep structured semantic models for web search using clickthrough data,” in *Proceedings of the 22nd ACM international conference on Conference on information & knowledge management*. ACM, 2013, pp. 2333--2338.
- [18] O. Abdel-Hamid, A.-r. Mohamed, H. Jiang, L. Deng, G. Penn, and D. Yu, “Convolutional neural networks for speech recognition,” *IEEE/ACM Transactions on audio, speech, and language processing*, vol. 22, no. 10, pp. 1533--1545, 2014.
- [19] P. Y. Simard, D. Steinkraus, and J. C. Platt, “Best practices for convolutional neural networks applied to visual document analysis,” in *null*. IEEE, 2003, p. 958.

- [20] S. Lai, L. Xu, K. Liu, and J. Zhao, “Recurrent convolutional neural networks for text classification.” in *AAAI*, vol. 333, 2015, pp. 2267--2273.
- [21] Y. Kim, “Convolutional neural networks for sentence classification,” *arXiv preprint arXiv:1408.5882*, 2014.
- [22] R. Collobert and J. Weston, “A unified architecture for natural language processing: Deep neural networks with multitask learning,” in *Proceedings of the 25th international conference on Machine learning*. ACM, 2008, pp. 160--167.
- [23] R. Sarikaya, G. E. Hinton, and A. Deoras, “Application of deep belief networks for natural language understanding,” *IEEE/ACM Transactions on Audio, Speech and Language Processing (TASLP)*, vol. 22, no. 4, pp. 778--784, 2014.
- [24] A. Karpathy, G. Toderici, S. Shetty, T. Leung, R. Sukthankar, and L. Fei-Fei, “Large-scale video classification with convolutional neural networks,” in *Proceedings of the IEEE conference on Computer Vision and Pattern Recognition*, 2014, pp. 1725--1732.
- [25] J. Mutch and D. G. Lowe, “Multiclass object recognition with sparse, localized features,” in *Computer Vision and Pattern Recognition, 2006 IEEE Computer Society Conference on*, vol. 1. IEEE, 2006, pp. 11--18.
- [26] A. Krizhevsky, I. Sutskever, and G. E. Hinton, “Imagenet classification with deep convolutional neural networks,” *Advances in neural information processing systems*, vol. 25, pp. 1097--1105, 2012.

- [27] K. Simonyan and A. Zisserman, “Very deep convolutional networks for large-scale image recognition,” *arXiv preprint arXiv:1409.1556*, 2014.
- [28] O. Russakovsky, J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma, Z. Huang, A. Karpathy, A. Khosla, M. Bernstein *et al.*, “Imagenet large scale visual recognition challenge,” *International journal of computer vision*, vol. 115, no. 3, pp. 211--252, 2015.
- [29] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich, “Going deeper with convolutions,” in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2015, pp. 1--9.
- [30] S. Ren, K. He, R. Girshick, and J. Sun, “Faster r-cnn: Towards real-time object detection with region proposal networks,” in *Advances in neural information processing systems*, 2015, pp. 91--99.
- [31] K. Korekado, T. Morie, O. Nomura, T. Nakano, M. Matsugu, and A. Iwata, “An image filtering processor for face/object recognition using merged/mixed analog-digital architecture,” in *VLSI Circuits, 2005. Digest of Technical Papers. 2005 Symposium on*. IEEE, 2005, pp. 220--223.
- [32] H. Li, Z. Lin, X. Shen, J. Brandt, and G. Hua, “A convolutional neural network cascade for face detection,” in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2015, pp. 5325--5334.

- [33] U. Muller, J. Ben, E. Cosatto, B. Flepp, and Y. L. Cun, “Off-road obstacle avoidance through end-to-end learning,” in *Advances in neural information processing systems*, 2006, pp. 739–746.
- [34] R. Hadsell, A. Erkan, P. Sermanet, J. Ben, K. Kavukcuoglu, U. Muller, and Y. LeCun, “A multi-range vision strategy for autonomous offroad navigation,” *Proc. Robotics and Applications (RA07)*, vol. 1, no. 7, 2007.
- [35] P. Sermanet, R. Hadsell, M. Scoffier, M. Grimes, J. Ben, A. Erkan, C. Crudele, U. Miller, and Y. LeCun, “A multirange architecture for collision-free off-road robot navigation,” *Journal of Field Robotics*, vol. 26, no. 1, pp. 52–87, 2009.
- [36] B. Blanco-Filgueira, D. García-Lesta, M. Fernández-Sanjurjo, V. M. Brea, and P. López, “Deep learning-based multiple object visual tracking on embedded system for iot and mobile edge computing applications,” *arXiv preprint arXiv:1808.01356*, 2018.
- [37] P. D. McNelis, *Neural networks in finance: gaining predictive edge in the market*. Academic Press, 2005.
- [38] P. J. Lisboa and E. C. Ifeachor, *Artificial neural networks in biomedicine*. Springer Science & Business Media, 2000.
- [39] P. W. Mirowski, Y. LeCun, D. Madhavan, and R. Kuzniecky, “Comparing SVM and convolutional networks for epileptic seizure prediction from intracranial EEG,” in *Machine Learning for Signal Processing, 2008. MLSP 2008. IEEE Workshop on*. IEEE, 2008, pp. 244–249.

- [40] G. E. Dahl, T. N. Sainath, and G. E. Hinton, “Improving deep neural networks for LVCSR using rectified linear units and dropout,” in *Acoustics, Speech and Signal Processing (ICASSP), 2013 IEEE International Conference on*. IEEE, 2013, pp. 8609–8613.
- [41] R. Hadsell, P. Sermanet, J. Ben, A. Erkan, M. Scoffier, K. Kavukcuoglu, U. Muller, and Y. LeCun, “Learning long-range vision for autonomous off-road driving,” *Journal of Field Robotics*, vol. 26, no. 2, pp. 120–144, 2009.
- [42] L. Deng, D. Yu *et al.*, “Deep learning: methods and applications,” *Foundations and Trends® in Signal Processing*, vol. 7, no. 3–4, pp. 197–387, 2014.
- [43] R. Girshick, J. Donahue, T. Darrell, and J. Malik, “Rich feature hierarchies for accurate object detection and semantic segmentation,” in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2014, pp. 580–587.
- [44] B. Wu, F. N. Iandola, P. H. Jin, and K. Keutzer, “Squeezedet: Unified, small, low power fully convolutional neural networks for real-time object detection for autonomous driving,” in *CVPR Workshops*, 2017, pp. 446–454.
- [45] J. E. Dayhoff, *Neural network architectures: an introduction*. Van Nostrand Reinhold New York, 1990.
- [46] Y. LeCun and Y. Bengio, “Convolutional networks for images, speech, and time series,” *The handbook of brain theory and neural networks*, vol. 3361, no. 10, p. 1995, 1995.

- [47] M. D. Zeiler and R. Fergus, “Visualizing and understanding convolutional networks,” in *European conference on computer vision*. Springer, 2014, pp. 818--833.
- [48] K. He, X. Zhang, S. Ren, and J. Sun, “Deep residual learning for image recognition,” in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016, pp. 770--778.
- [49] —, “Delving deep into rectifiers: Surpassing human-level performance on imagenet classification,” in *Proceedings of the IEEE international conference on computer vision*, 2015, pp. 1026--1034.
- [50] A.-r. Mohamed, G. E. Dahl, G. Hinton *et al.*, “Acoustic modeling using deep belief networks,” *IEEE Trans. Audio, Speech & Language Processing*, vol. 20, no. 1, pp. 14--22, 2012.
- [51] Image-Net. The ImageNet Large Scale Visual Recognition Challenge (ILSVRC) [Online]. Available: <http://image-net.org/challenges/LSVRC/>, 2018.
- [52] O. Nomura and T. Morie, “Projection-field-type VLSI convolutional neural networks using merged/mixed analog-digital approach,” in *International Conference on Neural Information Processing*. Springer, 2007, pp. 1081--1090.
- [53] T. M. Chilimbi, Y. Suzue, J. Apacible, and K. Kalyanaraman, “Project Adam: Building an efficient and scalable deep learning training system.” in *OSDI*, vol. 14, 2014, pp. 571--582.

- [54] Y. LeCun, B. Boser, J. S. Denker, D. Henderson, R. E. Howard, W. Hubbard, and L. D. Jackel, “Backpropagation applied to handwritten zip code recognition,” *Neural computation*, vol. 1, no. 4, pp. 541--551, 1989.
- [55] 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*, 2015, pp. 161--170.
- [56] A. Yazdanbakhsh, J. Park, H. Sharma, P. Lotfi-Kamran, and H. Esmailzadeh, “Neural acceleration for gpu throughput processors,” in *Proceedings of the 48th International Symposium on Microarchitecture*. ACM, 2015, pp. 482--493.
- [57] G. Hinton, L. Deng, D. Yu, G. E. Dahl, A.-r. Mohamed, N. Jaitly, A. Senior, V. Vanhoucke, P. Nguyen, T. N. Sainath *et al.*, “Deep neural networks for acoustic modeling in speech recognition: The shared views of four research groups,” *IEEE Signal processing magazine*, vol. 29, no. 6, pp. 82--97, 2012.
- [58] Y. Jia, E. Shelhamer, J. Donahue, S. Karayev, J. Long, R. Girshick, S. Guadarrama, and T. Darrell, “Caffe: Convolutional architecture for fast feature embedding,” in *Proceedings of the 22nd ACM international conference on Multimedia*. ACM, 2014, pp. 675--678.
- [59] A. Vasudevan, A. Anderson, and D. Gregg, “Parallel multi channel convolution using general matrix multiplication,” in *Application-specific Systems, Architect-*

- tures and Processors (ASAP)*, *2017 IEEE 28th International Conference on*.
IEEE, 2017, pp. 19--24.
- [60] K. Guo, L. Sui, J. Qiu, J. Yu, J. Wang, S. Yao, S. Han, Y. Wang, and H. Yang, "Angel-eye: A complete design flow for mapping cnn onto embedded FPGA," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 37, no. 1, pp. 35--47, 2018.
- [61] E. Nurvitadhi, G. Venkatesh, J. Sim, D. Marr, R. Huang, J. Ong Gee Hock, Y. T. Liew, K. Srivatsan, D. Moss, S. Subhaschandra *et al.*, "Can FPGAs beat GPUs in accelerating next-generation deep neural networks?" in *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. ACM, 2017, pp. 5--14.
- [62] J. Misra and I. Saha, "Artificial neural networks in hardware: A survey of two decades of progress," *Neurocomputing*, vol. 74, no. 1-3, pp. 239--255, 2010.
- [63] H. Esmailzadeh, A. Sampson, L. Ceze, and D. Burger, "Neural acceleration for general-purpose approximate programs," in *Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE Computer Society, 2012, pp. 449--460.
- [64] S. Han, X. Liu, H. Mao, J. Pu, A. Pedram, M. A. Horowitz, and W. J. Dally, "Eie: efficient inference engine on compressed deep neural network," in *Computer Architecture (ISCA)*, *2016 ACM/IEEE 43rd Annual International Symposium on*. IEEE, 2016, pp. 243--254.

- [65] L. Du, Y. Du, Y. Li, J. Su, Y.-C. Kuan, C.-C. Liu, and M.-C. F. Chang, “A reconfigurable streaming deep convolutional neural network accelerator for internet of things,” *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 65, no. 1, pp. 198–208, 2018.
- [66] W. Vanderbauwhede and K. Benkrid, *High-performance computing using FPGAs*. Springer, 2013.
- [67] A. Putnam, A. M. Caulfield, E. S. Chung, D. Chiou, K. Constantinides, J. Demme, H. Esmailzadeh, J. Fowers, G. P. Gopal, J. Gray *et al.*, “A reconfigurable fabric for accelerating large-scale datacenter services,” *ACM SIGARCH Computer Architecture News*, vol. 42, no. 3, pp. 13–24, 2014.
- [68] Y. Liang, K. Rupnow, Y. Li, D. Min, M. N. Do, and D. Chen, “High-level synthesis: productivity, performance, and software constraints,” *Journal of Electrical and Computer Engineering*, vol. 2012, p. 1, 2012.
- [69] J. Cong, B. Liu, S. Neuendorffer, J. Noguera, K. Vissers, and Z. Zhang, “High-level synthesis for fpgas: From prototyping to deployment,” *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 30, no. 4, pp. 473–491, 2011.
- [70] A. Canis, J. Choi, M. Aldham, V. Zhang, A. Kammoona, J. H. Anderson, S. Brown, and T. Czajkowski, “Legup: high-level synthesis for fpga-based processor/accelerator systems,” in *Proceedings of the 19th ACM/SIGDA international symposium on Field programmable gate arrays*. ACM, 2011, pp. 33–36.

- [71] T. Serre, L. Wolf, S. Bileschi, M. Riesenhuber, and T. Poggio, “Robust object recognition with cortex-like mechanisms,” *IEEE Transactions on Pattern Analysis & Machine Intelligence*, no. 3, pp. 411–426, 2007.
- [72] P. Joshi. What Is Local Response Normalization In Convolutional Neural Networks [Online]. Available: <https://prateekvjoshi.com/2016/04/05/what-is-local-response-normalization-in-convolutional-neural-networks/>, 2018.
- [73] N. Suda, V. Chandra, G. Dasika, A. Mohanty, Y. Ma, S. Vrudhula, J.-s. Seo, and Y. Cao, “Throughput-optimized openc1-based fpga accelerator for large-scale convolutional neural networks,” in *Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, 2016, pp. 16–25.
- [74] M. Denil, B. Shakibi, L. Dinh, N. De Freitas *et al.*, “Predicting parameters in deep learning,” in *Advances in neural information processing systems*, 2013, pp. 2148–2156.
- [75] V. Nair and G. E. Hinton, “Rectified linear units improve restricted boltzmann machines,” in *Proceedings of the 27th international conference on machine learning (ICML-10)*, 2010, pp. 807–814.
- [76] A. Karpathy. Convolutional Neural Networks for Visual Recognition [Online]. Available: <http://cs231n.github.io/convolutional-networks/>, 2018.
- [77] K. He, X. Zhang, S. Ren, and J. Sun, “Identity mappings in deep residual networks,” in *European conference on computer vision*. Springer, 2016, pp. 630–645.

- [78] C. Szegedy, S. Ioffe, V. Vanhoucke, and A. A. Alemi, “Inception-v4, inception-resnet and the impact of residual connections on learning,” in *AAAI*, vol. 4, 2017, p. 12.
- [79] J. Villasenor and W. H. Mangione-Smith, “Configurable computing,” *Scientific American*, vol. 276, no. 6, pp. 66--71, 1997.
- [80] S. D. Brown, R. J. Francis, J. Rose, and Z. G. Vranesic, *Field-programmable gate arrays*. Springer Science & Business Media, 2012, vol. 180.
- [81] M. C. Herbordt, Y. Gu, T. VanCourt, J. Model, B. Sukhwani, and M. Chiu, “Computing models for FPGA-based accelerators,” *Computing in science & engineering*, vol. 10, no. 6, pp. 35--45, 2008.
- [82] B. S. C. Varma, K. Paul, and M. Balakrishnan, *Architecture exploration of FPGA based accelerators for BioInformatics applications*. Springer, 2016.
- [83] G. Lacey, G. W. Taylor, and S. Areibi, “Deep learning on FPGAs: Past, present, and future,” *arXiv preprint arXiv:1602.04283*, 2016.
- [84] C. Farabet, Y. LeCun, K. Kavukcuoglu, E. Culurciello, B. Martini, P. Akselrod, and S. Talay, “Large-scale FPGA-based convolutional networks,” *Scaling up Machine Learning: Parallel and Distributed Approaches*, pp. 399--419, 2011.
- [85] J. Cong and B. Xiao, “Minimizing computation in convolutional neural networks,” in *International conference on artificial neural networks*, vol. 8681. Springer, 2014, pp. 281--290.

- [86] Y.-H. Chen, J. Emer, and V. Sze, “Eyeriss: A spatial architecture for energy-efficient dataflow for convolutional neural networks,” *ACM SIGARCH Computer Architecture News*, vol. 44, no. 3, pp. 367--379, 2016.
- [87] Y. Ma, N. Suda, Y. Cao, J.-s. Seo, and S. Vrudhula, “Scalable and modularized rtl compilation of convolutional neural networks onto FPGA,” in *Field Programmable Logic and Applications (FPL), 2016 26th International Conference on*. IEEE, 2016, pp. 1--8.
- [88] J. Qiu, J. Wang, S. Yao, K. Guo, B. Li, E. Zhou, J. Yu, T. Tang, N. Xu, and S. Song, “Going deeper with embedded FPGA platform for convolutional neural network,” in *Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. ACM, 2016, pp. 26--35.
- [89] G.-J. Horng, M.-X. Liu, and C.-C. Chen, “The smart image recognition mechanism for crop harvesting system in intelligent agriculture,” *IEEE Sensors Journal*, vol. 20, no. 5, pp. 2766--2781, 2020.
- [90] H. Jiang, X. Li, and F. Safara, “Iot-based agriculture: Deep learning in detecting apple fruit diseases,” *Microprocessors and Microsystems*, p. 104321, 2021.
- [91] A. Shawahna, S. M. Sait, and A. El-Maleh, “Fpga-based accelerators of deep learning networks for learning and classification: A review,” *IEEE Access*, vol. 7, pp. 7823--7859, 2019.
- [92] H. Li, X. Fan, L. Jiao, W. Cao, X. Zhou, and L. Wang, “A high performance fpga-based accelerator for large-scale convolutional neural networks,” in *2016*

26th International Conference on Field Programmable Logic and Applications (FPL), 2016, pp. 1--9.

- [93] Y. Shen, M. Ferdman, and P. Milder, "Maximizing cnn accelerator efficiency through resource partitioning," in *2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA)*, 2017, pp. 535--547.
- [94] W. Lu, G. Yan, J. Li, S. Gong, Y. Han, and X. Li, "Flexflow: A flexible dataflow accelerator architecture for convolutional neural networks," in *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2017, pp. 553--564.
- [95] D. F. Bacon, S. L. Graham, and O. J. Sharp, "Compiler transformations for high-performance computing," *ACM Computing Surveys (CSUR)*, vol. 26, no. 4, pp. 345--420, 1994.
- [96] S. Chakradhar, M. Sankaradas, V. Jakkula, and S. Cadambi, "A dynamically configurable coprocessor for convolutional neural networks," in *ACM SIGARCH Computer Architecture News*, vol. 38, no. 3. ACM, 2010, pp. 247--257.
- [97] Y. Ma, Y. Cao, S. Vrudhula, and J.-s. Seo, "Optimizing loop operation and dataflow in FPGA acceleration of deep convolutional neural networks," in *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. ACM, 2017, pp. 45--54.

- [98] V. Sze, Y.-H. Chen, J. Emer, A. Suleiman, and Z. Zhang, “Hardware for machine learning: Challenges and opportunities,” in *Custom Integrated Circuits Conference (CICC), 2018 IEEE*. IEEE, 2018, pp. 1--8.
- [99] R. Hameed, W. Qadeer, M. Wachs, O. Azizi, A. Solomatnikov, B. C. Lee, S. Richardson, C. Kozyrakis, and M. Horowitz, “Understanding sources of inefficiency in general-purpose chips,” *Communications of the ACM*, vol. 54, no. 10, pp. 85--93, 2011.
- [100] S. W. Keckler, W. J. Dally, B. Khailany, M. Garland, and D. Glasco, “Gpus and the future of parallel computing,” *IEEE micro*, vol. 31, no. 5, pp. 7--17, 2011.
- [101] S. Han, H. Mao, and W. J. Dally, “Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding,” *arXiv preprint arXiv:1510.00149*, 2015.
- [102] W. Sung, S. Shin, and K. Hwang, “Resiliency of deep neural networks under quantization,” *arXiv preprint arXiv:1511.06488*, 2015.
- [103] S. Han, J. Kang, H. Mao, Y. Hu, X. Li, Y. Li, D. Xie, H. Luo, S. Yao, Y. Wang *et al.*, “Ese: Efficient speech recognition engine with sparse lstm on FPGA,” in *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. ACM, 2017, pp. 75--84.
- [104] S. M. Sait, A. El-Maleh, M. Altakrouri, and A. Shawahna, “Optimization of fpga-based cnn accelerators using metaheuristics,” *The Journal of Supercomputing*, pp. 1--41, 2022.

- [105] A. Shawahna, S. M. Sait, A. El-Maleh, and I. Ahmad, “Fxp-qnet: A post-training quantizer for the design of mixed low-precision dnns with dynamic fixed-point representation,” *IEEE Access*, vol. 10, pp. 30 202--30 231, 2022.
- [106] J. Cloutier, E. Cosatto, S. Pigeon, F. R. Boyer, and P. Y. Simard, “Vip: An FPGA-based processor for image processing and neural networks,” in *Microelectronics for Neural Networks, 1996., Proceedings of Fifth International Conference on.* IEEE, 1996, pp. 330--336.
- [107] D. F. Wolf, R. A. Romero, and E. Marques, “Using embedded processors in hardware models of artificial neural networks,” in *V Simposio Brasileiro de automação inteligente, Brasil*, 2001.
- [108] K. R. Nichols, M. A. Moussa, and S. M. Areibi, “Feasibility of floating-point arithmetic in FPGA based artificial neural networks,” in *In CAINE.* Citeseer, 2002.
- [109] K. Benkrid and S. Belkacemi, “Design and implementation of a 2d convolution core for video applications on FPGAs,” in *Digital and Computational Video, 2002. DCV 2002. Proceedings. Third International Workshop on.* IEEE, 2002, pp. 85--92.
- [110] F. Cardells-Tormo and P.-L. Molinet, “Area-efficient 2-d shift-variant convolvers for FPGA-based digital image processing,” in *Signal Processing Systems Design and Implementation, 2005. IEEE Workshop on.* IEEE, 2005, pp. 209--213.

- [111] R. G. Gironés, R. C. Palero, J. C. Boluda, and A. S. Cortés, “FPGA implementation of a pipelined on-line backpropagation,” *Journal of VLSI signal processing systems for signal, image and video technology*, vol. 40, no. 2, pp. 189--213, 2005.
- [112] H. Zhang, M. Xia, and G. Hu, “A multiwindow partial buffering scheme for FPGA-based 2-d convolvers,” *IEEE Transactions on Circuits and Systems II: Express Briefs*, vol. 54, no. 2, pp. 200--204, 2007.
- [113] A. W. Savich, M. Moussa, and S. Areibi, “The impact of arithmetic representation on implementing mlp-bp on FPGAs: A study,” *IEEE transactions on neural networks*, vol. 18, no. 1, pp. 240--252, 2007.
- [114] C. Farabet, C. Poulet, J. Y. Han, and Y. LeCun, “Cnp: An FPGA-based processor for convolutional networks,” in *Field Programmable Logic and Applications, 2009. FPL 2009. International Conference on*. IEEE, 2009, pp. 32--37.
- [115] Y. LeCun *et al.*, “Lenet-5, convolutional neural networks,” *URL: <http://yann.lecun.com/exdb/lenet>*, p. 20, 2015.
- [116] M. Sankaradas, V. Jakkula, S. Cadambi, S. Chakradhar, I. Durdanovic, E. Cosatto, and H. P. Graf, “A massively parallel coprocessor for convolutional neural networks,” in *Application-specific Systems, Architectures and Processors, 2009. ASAP 2009. 20th IEEE International Conference on*. IEEE, 2009, pp. 53--60.

- [117] H. P. Graf, S. Cadambi, V. Jakkula, M. Sankaradass, E. Cosatto, S. Chakradhar, and I. Dourdanovic, “A massively parallel digital learning processor,” in *Advances in Neural Information Processing Systems*, 2009, pp. 529–536.
- [118] S. Cadambi, I. Durdanovic, V. Jakkula, M. Sankaradass, E. Cosatto, S. Chakradhar, and H. P. Graf, “A massively parallel FPGA-based coprocessor for support vector machines,” in *2009 17th IEEE Symposium on Field Programmable Custom Computing Machines*. IEEE, 2009, pp. 115–122.
- [119] S. Cadambi, A. Majumdar, M. Becchi, S. Chakradhar, and H. P. Graf, “A programmable parallel accelerator for learning and classification,” in *Proceedings of the 19th international conference on Parallel architectures and compilation techniques*. ACM, 2010, pp. 273–284.
- [120] J. C. Platt, “12 fast training of support vector machines using sequential minimal optimization,” *Advances in kernel methods*, pp. 185–208, 1999.
- [121] B. Bai, J. Weston, D. Grangier, R. Collobert, K. Sadamasa, Y. Qi, O. Chapelle, and K. Weinberger, “Learning to rank with (a lot of) word features,” *Information retrieval*, vol. 13, no. 3, pp. 291–314, 2010.
- [122] J. MacQueen *et al.*, “Some methods for classification and analysis of multivariate observations,” in *Proceedings of the fifth Berkeley symposium on mathematical statistics and probability*, vol. 1, no. 14. Oakland, CA, USA, 1967, pp. 281–297.
- [123] A. Sato and K. Yamada, “Generalized learning vector quantization,” in *Advances in neural information processing systems*, 1996, pp. 423–429.

- [124] 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.
- [125] S. Lawrence, C. L. Giles, A. C. Tsoi, and A. D. Back, "Face recognition: A convolutional neural-network approach," *IEEE transactions on neural networks*, vol. 8, no. 1, pp. 98--113, 1997.
- [126] K. Chellapilla, S. Puri, and P. Simard, "High performance convolutional neural networks for document processing," in *Tenth International Workshop on Frontiers in Handwriting Recognition*. Suvisoft, 2006.
- [127] F. Nasse, C. Thureau, and G. A. Fink, "Face detection using gpu-based convolutional neural networks," in *International Conference on Computer Analysis of Images and Patterns*. Springer, 2009, pp. 83--90.
- [128] J. D. Dixon, "Asymptotically fast factorization of integers," *Mathematics of computation*, vol. 36, no. 153, pp. 255--260, 1981.
- [129] P. L. Montgomery, "A survey of modern integer factorization algorithms," *CWI quarterly*, vol. 7, no. 4, pp. 337--365, 1994.
- [130] C. Farabet, B. Martini, P. Akselrod, S. Talay, Y. LeCun, and E. Culurciello, "Hardware accelerated convolutional neural networks for synthetic vision systems," in *Circuits and Systems (ISCAS), Proceedings of 2010 IEEE International Symposium on*. IEEE, 2010, pp. 257--260.

- [131] C. Farabet, B. Martini, B. Corda, P. Akselrod, E. Culurciello, and Y. LeCun, “Neuflow: A runtime reconfigurable dataflow processor for vision,” in *Computer Vision and Pattern Recognition Workshops (CVPRW), 2011 IEEE Computer Society Conference on*. IEEE, 2011, pp. 109–116.
- [132] R. Collobert, C. Farabet, K. Kavukcuoglu *et al.*, “Torch,” in *Workshop on Machine Learning Open Source Software, NIPS*, vol. 76, 2008.
- [133] D. Grangier, L. Bottou, and R. Collobert, “Deep convolutional networks for scene parsing,” in *ICML 2009 Deep Learning Workshop*, vol. 3, no. 6. Citeseer, 2009, p. 109.
- [134] M. Peemen, A. A. Setio, B. Mesman, and H. Corporaal, “Memory-centric accelerator design for convolutional neural networks,” in *Computer Design (ICCD), 2013 IEEE 31st International Conference on*. IEEE, 2013, pp. 13–19.
- [135] A. Beric, J. van Meerbergen, G. de Haan, and R. Sethuraman, “Memory-centric video processing,” *IEEE Transactions on Circuits and Systems for Video Technology*, vol. 18, no. 4, pp. 439–452, 2008.
- [136] V. Gokhale, J. Jin, A. Dundar, B. Martini, and E. Culurciello, “A 240 g-ops/s mobile coprocessor for deep neural networks,” in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition Workshops*, 2014, pp. 682–687.
- [137] C. Farabet, C. Poulet, and Y. LeCun, “An FPGA-based stream processor for embedded real-time vision with convolutional networks,” in *Computer Vision*

- Workshops (ICCV Workshops), 2009 IEEE 12th International Conference on.*
IEEE, 2009, pp. 878--885.
- [138] S. Williams, A. Waterman, and D. Patterson, "Roofline: an insightful visual performance model for multicore architectures," *Communications of the ACM*, vol. 52, no. 4, pp. 65--76, 2009.
- [139] L.-N. Pouchet, P. Zhang, P. Sadayappan, and J. Cong, "Polyhedral-based data reuse optimization for configurable computing," in *Proceedings of the ACM/SIGDA international symposium on Field programmable gate arrays*, 2013, pp. 29--38.
- [140] K. Ovtcharov, O. Ruwase, J.-Y. Kim, J. Fowers, K. Strauss, and E. S. Chung, "Accelerating deep convolutional neural networks using specialized hardware," *Microsoft Research Whitepaper*, vol. 2, no. 11, 2015.
- [141] C. F. Van Loan, "Matrix computations (johns hopkins studies in mathematical sciences)," 1996.
- [142] C. Zhang, G. Sun, Z. Fang, P. Zhou, P. Pan, and J. Cong, "Caffeine: Towards uniformed representation and acceleration for deep convolutional neural networks," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2018.
- [143] B. Bosi, G. Bois, and Y. Savaria, "Reconfigurable pipelined 2-d convolvers for fast digital signal processing," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 7, no. 3, pp. 299--308, 1999.

- [144] Y. Wang, J. Xu, Y. Han, H. Li, and X. Li, “Deepburning: automatic generation of FPGA-based learning accelerators for the neural network family,” in *Proceedings of the 53rd Annual Design Automation Conference*. ACM, 2016, p. 110.
- [145] K. O. W. Group *et al.*, “The opencl specification version 1.1,” <http://www.khronos.org/registry/cl/specs/opencl-1.1.pdf>, 2011.
- [146] M. S. Abdelfattah, A. Hagiescu, and D. Singh, “Gzip on a chip: High performance lossless data compression on FPGAs using opencl,” in *Proceedings of the International Workshop on OpenCL 2013 & 2014*. ACM, 2014, p. 4.
- [147] Altera. OpenCL Design Examples [Online]. Available: <https://www.altera.com/support/support-resources/designexamples/design-software/opencl.html/>, 2018.
- [148] Nallatech. P395-D8 OpenCL FPGA Accelerator Cards [Online]. Available: http://www.nallatech.com/wp-content/uploads/opencardspb_v1_51.pdf/, 2018.
- [149] Altera. DE5-Net FPGA Kit User Manual [Online]. Available: ftp://ftp.altera.com/up/pub/Altera_Material/Boards/DE5/DE5_User_, 2018.
- [150] R. C. Whaley and J. J. Dongarra, “Automatically tuned linear algebra software,” in *Supercomputing, 1998. SC98. IEEE/ACM Conference on*. IEEE, 1998, pp. 38--38.
- [151] C. Zhang, Z. Fang, P. Zhou, P. Pan, and J. Cong, “Caffeine: Towards uniformed representation and acceleration for deep convolutional neural networks,”

- in *Computer-Aided Design (ICCAD), 2016 IEEE/ACM International Conference on*. IEEE, 2016, pp. 1--8.
- [152] W. Zuo, Y. Liang, P. Li, K. Rupnow, D. Chen, and J. Cong, "Improving high level synthesis optimization opportunity through polyhedral transformations," in *Proceedings of the ACM/SIGDA international symposium on Field programmable gate arrays*. ACM, 2013, pp. 9--18.
- [153] E. A. Lee and D. G. Messerschmitt, "Synchronous data flow," *Proceedings of the IEEE*, vol. 75, no. 9, pp. 1235--1245, 1987.
- [154] S. I. Venieris and C.-S. Bouganis, "FPGAconvnet: A framework for mapping convolutional neural networks on fpgas," in *Field-Programmable Custom Computing Machines (FCCM), 2016 IEEE 24th Annual International Symposium on*. IEEE, 2016, pp. 40--47.
- [155] C. R. Reeves, *Modern heuristic techniques for combinatorial problems. Advanced topics in computer science*. Mc Graw-Hill, 1995.
- [156] L. Cavigelli, M. Magno, and L. Benini, "Accelerating real-time embedded scene labeling with convolutional networks," in *Proceedings of the 52nd Annual Design Automation Conference*. ACM, 2015, p. 108.
- [157] Y. Ma, N. Suda, Y. Cao, S. Vrudhula, and J.-s. Seo, "Alamo: FPGA acceleration of deep learning algorithms with a modularized rtl compiler," *Integration*, 2018.
- [158] M. Lin, Q. Chen, and S. Yan, "Network in network," *arXiv preprint arXiv:1312.4400*, 2013.

- [159] Z. Liu, Y. Dou, J. Jiang, J. Xu, S. Li, Y. Zhou, and Y. Xu, “Throughput-optimized fpga accelerator for deep convolutional neural networks,” *ACM Transactions on Reconfigurable Technology and Systems (TRETs)*, vol. 10, no. 3, pp. 1--23, 2017.
- [160] Y. Guan, H. Liang, N. Xu, W. Wang, S. Shi, X. Chen, G. Sun, W. Zhang, and J. Cong, “FP-DNN: An automated framework for mapping deep neural networks onto FPGAs with RTL-HLS hybrid templates,” in *Field-Programmable Custom Computing Machines (FCCM), 2017 IEEE 25th Annual International Symposium on*. IEEE, 2017, pp. 152--159.
- [161] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard *et al.*, “Tensorflow: a system for large-scale machine learning.” in *OSDI*, vol. 16, 2016, pp. 265--283.
- [162] M. H. Alsuwaiyel, *Algorithms: Design Techniques And Analysis (Revised Edition)*. World Scientific, 2016, vol. 14.
- [163] S. Chetlur, C. Woolley, P. Vandermersch, J. Cohen, J. Tran, B. Catanzaro, and E. Shelhamer, “cudnn: Efficient primitives for deep learning,” *arXiv preprint arXiv:1410.0759*, 2014.
- [164] W. Zaremba, I. Sutskever, and O. Vinyals, “Recurrent neural network regularization,” *arXiv preprint arXiv:1409.2329*, 2014.

- [165] M. Rastegari, V. Ordonez, J. Redmon, and A. Farhadi, “Xnor-net: Imagenet classification using binary convolutional neural networks,” in *European Conference on Computer Vision*. Springer, 2016, pp. 525--542.
- [166] M. Kim and P. Smaragdis, “Bitwise neural networks,” *arXiv preprint arXiv:1601.06071*, 2016.
- [167] S. Zhou, Y. Wu, Z. Ni, X. Zhou, H. Wen, and Y. Zou, “Dorefa-net: Training low bitwidth convolutional neural networks with low bitwidth gradients,” *arXiv preprint arXiv:1606.06160*, 2016.
- [168] M. Courbariaux, I. Hubara, D. Soudry, R. El-Yaniv, and Y. Bengio, “Binarized neural networks: Training deep neural networks with weights and activations constrained to +1 or -1,” *arXiv preprint arXiv:1602.02830*, 2016.
- [169] Y. Umuroglu, N. J. Fraser, G. Gambardella, M. Blott, P. Leong, M. Jahre, and K. Vissers, “Finn: A framework for fast, scalable binarized neural network inference,” in *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. ACM, 2017, pp. 65--74.
- [170] A. Krizhevsky and G. Hinton, “Learning multiple layers of features from tiny images,” Citeseer, Tech. Rep., 2009.
- [171] S. I. Venieris and C.-S. Bouganis, “Latency-driven design for fpga-based convolutional neural networks,” in *Field Programmable Logic and Applications (FPL), 2017 27th International Conference on*. IEEE, 2017, pp. 1--8.

- [172] A. Lavin and S. Gray, “Fast algorithms for convolutional neural networks,” in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2016, pp. 4013--4021.
- [173] S. Winograd, *Arithmetic complexity of computations*. Siam, 1980, vol. 33.
- [174] C. Van Loan, *Computational frameworks for the fast Fourier transform*. Siam, 1992, vol. 10.
- [175] C. Zhang and V. Prasanna, “Frequency domain acceleration of convolutional neural networks on cpu-fpga shared memory system,” in *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. ACM, 2017, pp. 35--44.
- [176] U. Aydonat, S. O’Connell, D. Capalija, A. C. Ling, and G. R. Chiu, “An opencl deep learning accelerator on arria 10,” in *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. ACM, 2017, pp. 55--64.
- [177] L. Lu, Y. Liang, Q. Xiao, and S. Yan, “Evaluating fast algorithms for convolutional neural networks on fpgas,” in *Field-Programmable Custom Computing Machines (FCCM), 2017 IEEE 25th Annual International Symposium on*. IEEE, 2017, pp. 101--108.
- [178] J. Zhang and J. Li, “Improving the performance of opencl-based fpga accelerator for convolutional neural network,” in *Proceedings of the 2017 ACM/SIGDA*

- International Symposium on Field-Programmable Gate Arrays.* ACM, 2017, pp. 25--34.
- [179] T. S. Czajkowski, D. Neto, M. Kinsner, U. Aydonat, J. Wong, D. Denisenko, P. Yiannacouras, J. Freeman, D. P. Singh, and S. D. Brown, "Opencl for fp-gas: Prototyping a compiler," in *Proceedings of the International Conference on Engineering of Reconfigurable Systems and Algorithms (ERSA)*. The Steering Committee of The World Congress in Computer Science, Computer Engineering and Applied Computing (WorldComp), 2012, p. 1.
- [180] F. N. Iandola, S. Han, M. W. Moskewicz, K. Ashraf, W. J. Dally, and K. Keutzer, "Squeezenet: Alexnet-level accuracy with 50x fewer parameters and < 0.5 mb model size," *arXiv preprint arXiv:1602.07360*, 2016.
- [181] W. Xuechao, Y. Cody Hao, Z. Peng, C. Youxiang, W. Yuxin, H. Han, L. Yun, and C. Jason, "Automated systolic array architecture synthesis for high throughput cnn inference on FPGAs," in *Proceedings of the 2017 Design Automation Conference*. ACM, 2017, pp. 1--6.
- [182] Y. Ma, M. Kim, Y. Cao, S. Vrudhula, and J.-s. Seo, "End-to-end scalable FPGA accelerator for deep residual networks," in *Circuits and Systems (ISCAS), 2017 IEEE International Symposium on*. IEEE, 2017, pp. 1--4.
- [183] C. Wang, L. Gong, Q. Yu, X. Li, Y. Xie, and X. Zhou, "Dlau: A scalable deep learning accelerator unit on FPGA," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 36, no. 3, pp. 513--517, 2017.

- [184] Altera. JTAG UART Core [Online]. Available: https://www.altera.com/en_US/pdfs/literature/hb/nios2/n2cpu_nii51009.pdf, 2018.
- [185] Y. Ma, Y. Cao, S. Vrudhula, and J.-s. Seo, “An automatic rtl compiler for high-throughput FPGA implementation of diverse deep convolutional neural networks,” in *Field Programmable Logic and Applications (FPL), 2017 27th International Conference on*. IEEE, 2017, pp. 1--8.
- [186] M. S. Abdelfattah, D. Han, A. Bitar, R. DiCecco, S. OConnell, N. Shanker, J. Chu, I. Prins, J. Fender, A. C. Ling *et al.*, “Dla: Compiler and FPGA overlay for neural network inference acceleration,” *arXiv preprint arXiv:1807.06434*, 2018.
- [187] A. K. Jain, S. A. Fahmy, and D. L. Maskell, “Efficient overlay architecture based on dsp blocks,” in *Field-Programmable Custom Computing Machines (FCCM), 2015 IEEE 23rd Annual International Symposium on*. IEEE, 2015, pp. 25--28.
- [188] W. Liu, D. Anguelov, D. Erhan, C. Szegedy, S. Reed, C.-Y. Fu, and A. C. Berg, “Ssd: Single shot multibox detector,” in *European conference on computer vision*. Springer, 2016, pp. 21--37.
- [189] E. Chung, J. Fowers, K. Ovtcharov, M. Papamichael, A. Caulfield, T. Massengil, M. Liu, D. Lo, S. Alkalay, M. Haselman *et al.*, “Accelerating persistent neural networks at datacenter scale,” in *Hot Chips*, vol. 27, 2017.

- [190] Y. Ma, Y. Cao, S. Vrudhula, and J.-s. Seo, “Optimizing the convolution operation to accelerate deep neural networks on FPGA,” *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, no. 99, pp. 1--14, 2018.
- [191] M. S. Sadiq and Y. Habib, *Iterative Computer Algorithms with Applications in Engineering: Solving Combinatorial Optimization Problems*. Los Alamitos, CA, USA: IEEE Computer Society Press, 1999.
- [192] L. Xie and A. L. Yuille, “Genetic CNN,” in *ICCV*, 2017, pp. 1388--1397.
- [193] L. Rere, M. I. Fanany, and A. M. Arymurthy, “Metaheuristic algorithms for convolution neural network,” *Computational intelligence and neuroscience*, vol. 2016, 2016.
- [194] X. Hu, X. Lu, and C. Hori, “Mandarin speech recognition using convolution neural network with augmented tone features,” in *The 9th International Symposium on Chinese Spoken Language Processing*, 2014, pp. 15--18.
- [195] M. Khalil-Hani and L. S. Sung, “A convolutional neural network approach for face verification,” in *2014 International Conference on High Performance Computing & Simulation (HPCS)*, 2014, pp. 707--714.
- [196] S. S. Farfade, M. J. Saberian, and L.-J. Li, “Multi-view face detection using deep convolutional neural networks,” in *Proceedings of the 5th ACM on International Conference on Multimedia Retrieval*, 2015, pp. 643--650.

- [197] J. Zheng, Y. Wang, and W. Zeng, "Cnn based vehicle counting with virtual coil in traffic surveillance video," in *2015 IEEE International Conference on Multimedia Big Data*, 2015, pp. 280--281.
- [198] R. Wang and Z. Xu, "A pedestrian and vehicle rapid identification model based on convolutional neural network," in *Proceedings of the 7th International Conference on Internet Multimedia Computing and Service*, 2015, pp. 1--4.
- [199] M. M. Lau, K. H. Lim, and A. A. Gopalai, "Malaysia traffic sign recognition with convolutional neural network," in *2015 IEEE International Conference on Digital Signal Processing (DSP)*, 2015, pp. 1006--1010.
- [200] X. Feng, Y. Jiang, X. Yang, M. Du, and X. Li, "Computer vision algorithms and hardware implementations: A survey," *Integration*, vol. 69, pp. 309--320, 2019.
- [201] D. Ghimire, D. Kil, and S.-h. Kim, "A survey on efficient convolutional neural networks and hardware acceleration," *Electronics*, vol. 11, no. 6, p. 945, 2022.
- [202] I. H. Osman and J. P. Kelly, "Meta-heuristics: an overview," *Meta-heuristics*, pp. 1--21, 1996.
- [203] L. R. Rere, M. I. Fanany, and A. M. Arymurthy, "Simulated annealing algorithm for deep learning," *Procedia Computer Science*, vol. 72, pp. 137--144, 2015.
- [204] M. Cho and Y. Kim, "Fpga-based convolutional neural network accelerator with resource-optimized approximate multiply-accumulate unit," *Electronics*, vol. 10, no. 22, p. 2859, 2021.

- [205] Xilinx, “Vivado Design Suite Product Guide: Floating-Point Operator v7.1 [Online],” Available: <https://docs.xilinx.com/v/u/en-US/pg060-floating-point> (2020).
- [206] —, “User Guide: 7 Series FPGAs Memory Resources [Online],” Available: https://docs.xilinx.com/v/u/en-US/ug473_7Series_Memory_Resources (2019).
- [207] S. M. Sait and H. Youssef, *VLSI physical design automation: theory and practice*. World Scientific, 1999, vol. 6.
- [208] S. Kirkpatrick, C. D. Gelatt Jr, and M. P. Vecchi, “Optimization by simulated annealing,” *science*, vol. 220, no. 4598, pp. 671–680, 1983.
- [209] V. Cerny, “Thermodynamical approach to the traveling salesman problem: An efficient simulation algorithm,” *Journal of optimization theory and applications*, vol. 45, no. 1, pp. 41–51, 1985.
- [210] N. Metropolis, A. W. Rosenbluth, M. N. Rosenbluth, A. H. Teller, and E. Teller, “Equation of state calculations by fast computing machines,” *The journal of chemical physics*, vol. 21, no. 6, pp. 1087–1092, 1953.
- [211] H. Youssef, S. M. Sait, and H. Adiche, “Evolutionary algorithms, simulated annealing and tabu search: a comparative study,” *Engineering Applications of Artificial Intelligence*, vol. 14, no. 2, pp. 167–181, 2001.
- [212] F. Glover, “Tabu search - part i,” *ORSA Journal on computing*, vol. 1, no. 3, pp. 190–206, 1989.

- [213] ———, “Tabu search - part ii,” *ORSA Journal on computing*, vol. 2, no. 1, pp. 4--32, 1990.
- [214] F. Glover and E. Taillard, “A user’s guide to tabu search,” *Annals of operations research*, vol. 41, no. 1, pp. 1--28, 1993.
- [215] F. Glover and M. Laguna, “Tabu search,” in *Handbook of combinatorial optimization*. Springer, 1998, pp. 2093--2229.
- [216] Xilinx, “User Guide: VC707 Evaluation Board for the Virtex-7 FPGA [Online],” Available: https://docs.xilinx.com/v/u/en-US/ug885_VC707_Eval_Bd (2019).
- [217] ———, “User Guide: VC709 Evaluation Board for the Virtex-7 FPGA [Online],” Available: <https://docs.xilinx.com/v/u/en-US/ug887-vc709-eval-board-v7-fpga> (2019).
- [218] P. Garcia, D. Bhowmik, R. Stewart, G. Michaelson, and A. Wallace, “Optimized memory allocation and power minimization for fpga-based image processing,” *Journal of Imaging*, vol. 5, no. 1, p. 7, 2019.
- [219] S. Matiz and K. E. Barner, “Inductive conformal predictor for convolutional neural networks: Applications to active learning for image classification,” *Pattern Recognition*, vol. 90, pp. 172--182, 2019.
- [220] G. Huang, Z. Liu, L. Van Der Maaten, and K. Q. Weinberger, “Densely connected convolutional networks,” in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2017, pp. 4700--4708.

- [221] S. Ren, K. He, R. Girshick, and J. Sun, “Faster r-cnn: towards real-time object detection with region proposal networks,” *IEEE transactions on pattern analysis and machine intelligence*, vol. 39, no. 6, pp. 1137--1149, 2016.
- [222] J. Wei, Y. Xia, and Y. Zhang, “M3net: A multi-model, multi-size, and multi-view deep neural network for brain magnetic resonance image segmentation,” *Pattern Recognition*, vol. 91, pp. 366--378, 2019.
- [223] K. He, G. Gkioxari, P. Dollár, and R. Girshick, “Mask r-cnn,” in *Proceedings of the IEEE international conference on computer vision*, 2017, pp. 2961--2969.
- [224] J. Long, E. Shelhamer, and T. Darrell, “Fully convolutional networks for semantic segmentation,” in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2015, pp. 3431--3440.
- [225] Z. Li, Y. Zhang, J. Wang, and J. Lai, “A survey of fpga design for ai era,” *Journal of Semiconductors*, vol. 41, no. 2, p. 021402, 2020.
- [226] F. Liu, W. Zhao, Z. He, Y. Wang, Z. Wang, C. Dai, X. Liang, and L. Jiang, “Improving neural network efficiency via post-training quantization with adaptive floating-point,” in *Proceedings of the IEEE/CVF International Conference on Computer Vision*, 2021, pp. 5281--5290.
- [227] A. Kouris, S. I. Venieris, and C.-S. Bouganis, “Cascade^ cnn: Pushing the performance limits of quantisation in convolutional neural networks,” in *2018 28th International Conference on Field Programmable Logic and Applications (FPL)*. IEEE, 2018, pp. 155--1557.

- [228] M. Horowitz, “1.1 computing’s energy problem (and what we can do about it),” in *2014 IEEE International Solid-State Circuits Conference Digest of Technical Papers (ISSCC)*. IEEE, 2014, pp. 10–14.
- [229] 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. 105, no. 12, pp. 2295–2329, 2017.
- [230] R. Krishnamoorthi, “Quantizing deep convolutional networks for efficient inference: A whitepaper,” *arXiv preprint arXiv:1806.08342*, 2018.
- [231] H. Qin, R. Gong, X. Liu, X. Bai, J. Song, and N. Sebe, “Binary neural networks: A survey,” *Pattern Recognition*, vol. 105, p. 107281, 2020.
- [232] J. Bethge, M. Bornstein, A. Loy, H. Yang, and C. Meinel, “Training competitive binary neural networks from scratch,” *arXiv preprint arXiv:1812.01965*, 2018.
- [233] I. Hubara, M. Courbariaux, D. Soudry, R. El-Yaniv, and Y. Bengio, “Quantized neural networks: Training neural networks with low precision weights and activations,” *The Journal of Machine Learning Research*, vol. 18, no. 1, pp. 6869–6898, 2017.
- [234] ———, “Binarized neural networks,” *Advances in neural information processing systems*, vol. 29, 2016.
- [235] L. Hou, Q. Yao, and J. T. Kwok, “Loss-aware binarization of deep networks,” *arXiv preprint arXiv:1611.01600*, 2016.

- [236] F. Li, B. Zhang, and B. Liu, “Ternary weight networks,” *arXiv preprint arXiv:1605.04711*, 2016.
- [237] C. Zhu, S. Han, H. Mao, and W. J. Dally, “Trained ternary quantization,” *arXiv preprint arXiv:1612.01064*, 2016.
- [238] J. Achterhold, J. M. Koehler, A. Schmeink, and T. Genewein, “Variational network quantization,” in *International Conference on Learning Representations*, 2018.
- [239] L. Hou and J. T. Kwok, “Loss-aware weight quantization of deep networks,” *arXiv preprint arXiv:1802.08635*, 2018.
- [240] S. Gupta, A. Agrawal, K. Gopalakrishnan, and P. Narayanan, “Deep learning with limited numerical precision,” in *International conference on machine learning*. PMLR, 2015, pp. 1737–1746.
- [241] S. Migacz, “8-bit inference with tensorrt,” in *GPU technology conference*, vol. 2, 2017, p. 5.
- [242] J. H. Lee, S. Ha, S. Choi, W.-J. Lee, and S. Lee, “Quantization for rapid deployment of deep neural networks,” *arXiv preprint arXiv:1810.05488*, 2018.
- [243] S. Jain, S. Venkataramani, V. Srinivasan, J. Choi, P. Chuang, and L. Chang, “Compensated-dnn: energy efficient low-precision deep neural networks by compensating quantization errors,” in *2018 55th ACM/ESDA/IEEE Design Automation Conference (DAC)*. IEEE, 2018, pp. 1–6.

- [244] E. Kravchik, F. Yang, P. Kisilev, and Y. Choukroun, “Low-bit quantization of neural networks for efficient inference,” in *Proceedings of the IEEE/CVF International Conference on Computer Vision Workshops*, 2019, pp. 0–0.
- [245] E. Meller, A. Finkelstein, U. Almog, and M. Grobman, “Same, same but different: Recovering neural network quantization error through weight factorization,” in *International Conference on Machine Learning*. PMLR, 2019, pp. 4486–4495.
- [246] R. Banner, Y. Nahshan, and D. Soudry, “Post training 4-bit quantization of convolutional networks for rapid-deployment,” *Advances in Neural Information Processing Systems*, vol. 32, pp. 7950–7958, 2019.
- [247] C. Zhang, S. Bengio, and Y. Singer, “Are all layers created equal?” *arXiv preprint arXiv:1902.01996*, 2019.
- [248] P. Micikevicius, S. Narang, J. Alben, G. Diamos, E. Elsen, D. Garcia, B. Ginsburg, M. Houston, O. Kuchaiev, G. Venkatesh *et al.*, “Mixed precision training,” in *International Conference on Learning Representations*, 2018.
- [249] K. Wang, Z. Liu, Y. Lin, J. Lin, and S. Han, “Haq: Hardware-aware automated quantization with mixed precision,” in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2019, pp. 8612–8620.
- [250] A. Capotondi, M. Rusci, M. Fariselli, and L. Benini, “Cmix-nn: Mixed low-precision cnn library for memory-constrained edge devices,” *IEEE Transactions on Circuits and Systems II: Express Briefs*, vol. 67, no. 5, pp. 871–875, 2020.

- [251] A. Rajagopal, D. Vink, S. Venieris, and C.-S. Bouganis, “Multi-precision policy enforced training (muppet): A precision-switching strategy for quantised fixed-point training of cnns,” in *International Conference on Machine Learning*. PMLR, 2020, pp. 7943--7952.
- [252] S. I. Venieris, I. Panopoulos, I. Leontiadis, and I. S. Venieris, “How to reach real-time ai on consumer devices? solutions for programmable and custom architectures,” *arXiv preprint arXiv:2106.15021*, 2021.
- [253] H. Sharma, J. Park, N. Suda, L. Lai, B. Chau, V. Chandra, and H. Esmaeilzadeh, “Bit fusion: Bit-level dynamically composable architecture for accelerating deep neural network,” in *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2018, pp. 764--775.
- [254] Y. Umuroglu, L. Rasnayake, and M. Sjölander, “Bismo: A scalable bit-serial matrix multiplication overlay for reconfigurable computing,” in *2018 28th International Conference on Field Programmable Logic and Applications (FPL)*. IEEE, 2018, pp. 307--3077.
- [255] A. Mishra and D. Marr, “Apprentice: Using knowledge distillation techniques to improve low-precision network accuracy,” in *International Conference on Learning Representations*, 2018.
- [256] B. Wu, Y. Wang, P. Zhang, Y. Tian, P. Vajda, and K. Keutzer, “Mixed precision quantization of convnets via differentiable neural architecture search,” *arXiv preprint arXiv:1812.00090*, 2018.

- [257] M. F. F. Khan, M. M. Kamani, M. Mahdavi, and V. Narayanan, “Learning to quantize deep neural networks: A competitive-collaborative approach,” in *2020 57th ACM/IEEE Design Automation Conference (DAC)*. IEEE, 2020, pp. 1--6.
- [258] Y. Kang and J. Chung, “A dynamic fixed-point representation for neuromorphic computing systems,” in *2017 International SoC Design Conference (ISOCC)*. IEEE, 2017, pp. 44--45.
- [259] E. Park, J. Ahn, and S. Yoo, “Weighted-entropy-based quantization for deep neural networks,” in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2017, pp. 5456--5464.
- [260] J. Faraone, N. Fraser, M. Blott, and P. H. Leong, “Syq: Learning symmetric quantization for efficient deep neural networks,” in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2018, pp. 4300--4309.
- [261] D. Miyashita, E. H. Lee, and B. Murmann, “Convolutional neural networks using logarithmic data representation,” *arXiv preprint arXiv:1603.01025*, 2016.
- [262] A. Mishra, E. Nurvitadhi, J. J. Cook, and D. Marr, “Wrpn: Wide reduced-precision networks,” *arXiv preprint arXiv:1709.01134*, 2017.
- [263] J. Choi, Z. Wang, S. Venkataramani, P. I.-J. Chuang, V. Srinivasan, and K. Gopalakrishnan, “Pact: Parameterized clipping activation for quantized neural networks,” *arXiv preprint arXiv:1805.06085*, 2018.

- [264] J. Choi, P. I.-J. Chuang, Z. Wang, S. Venkataramani, V. Srinivasan, and K. Gopalakrishnan, “Bridging the accuracy gap for 2-bit quantized neural networks (qnn),” *arXiv preprint arXiv:1807.06964*, 2018.
- [265] Z. Cai, X. He, J. Sun, and N. Vasconcelos, “Deep learning with low precision by half-wave gaussian quantization,” in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2017, pp. 5918--5926.
- [266] P. Wang, Q. Hu, Y. Zhang, C. Zhang, Y. Liu, and J. Cheng, “Two-step quantization for low-bit neural networks,” in *Proceedings of the IEEE Conference on computer vision and pattern recognition*, 2018, pp. 4376--4384.
- [267] S. Lloyd, “Least squares quantization in pcm,” *IEEE transactions on information theory*, vol. 28, no. 2, pp. 129--137, 1982.
- [268] S. Jung, C. Son, S. Lee, J. Son, J.-J. Han, Y. Kwak, S. J. Hwang, and C. Choi, “Learning to quantize deep networks by optimizing quantization intervals with task loss,” in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2019, pp. 4350--4359.
- [269] B. Graham, “Low-precision batch-normalized activations,” *arXiv preprint arXiv:1702.08231*, 2017.
- [270] S.-C. Zhou, Y.-Z. Wang, H. Wen, Q.-Y. He, and Y.-H. Zou, “Balanced quantization: An effective and efficient approach to quantized neural networks,” *Journal of Computer Science and Technology*, vol. 32, no. 4, pp. 667--682, 2017.

- [271] A. Zhou, A. Yao, Y. Guo, L. Xu, and Y. Chen, “Incremental network quantization: Towards lossless cnns with low-precision weights,” *arXiv preprint arXiv:1702.03044*, 2017.
- [272] G. Hinton, O. Vinyals, and J. Dean, “Distilling the knowledge in a neural network,” *arXiv preprint arXiv:1503.02531*, 2015.
- [273] D. Zhang, J. Yang, D. Ye, and G. Hua, “Lq-nets: Learned quantization for highly accurate and compact deep neural networks,” in *Proceedings of the European conference on computer vision (ECCV)*, 2018, pp. 365--382.
- [274] S. R. Jain, A. Gural, M. Wu, and C. Dick, “Trained uniform quantization for accurate and efficient neural network inference on fixed-point hardware,” *arXiv preprint arXiv:1903.08066*, vol. 6, p. 6, 2019.
- [275] S. K. Esser, J. L. McKinstry, D. Bablani, R. Appuswamy, and D. S. Modha, “Learned step size quantization,” in *International Conference on Learning Representations*, 2020.
- [276] E. Park and S. Yoo, “Profit: A novel training method for sub-4-bit mobilenet models,” in *European Conference on Computer Vision*. Springer, 2020, pp. 430--446.
- [277] N. Mellempudi, A. Kundu, D. Mudigere, D. Das, B. Kaul, and P. Dubey, “Ternary neural networks with fine-grained quantization,” *CoRR*, vol. abs/1705.01462, 2017.

- [278] S. Seo and J. Kim, “Efficient weights quantization of convolutional neural networks using kernel density estimation based non-uniform quantizer,” *Applied Sciences*, vol. 9, no. 12, p. 2559, 2019.
- [279] C. Huang, P. Liu, and L. Fang, “Mxqn: Mixed quantization for reducing bit-width of weights and activations in deep convolutional neural networks,” *Applied Intelligence*, pp. 1--14, 2021.
- [280] J. Xu, Y. Huan, B. Huang, H. Chu, Y. Jin, L.-R. Zheng, and Z. Zou, “A memory-efficient cnn accelerator using segmented logarithmic quantization and multi-cluster architecture,” *IEEE Transactions on Circuits and Systems II: Express Briefs*, vol. 68, no. 6, pp. 2142--2146, 2020.
- [281] S. Ye, T. Zhang, K. Zhang, J. Li, J. Xie, Y. Liang, S. Liu, X. Lin, and Y. Wang, “A unified framework of dnn weight pruning and weight clustering/quantization using admm,” *arXiv preprint arXiv:1811.01907*, 2018.
- [282] A. Elthakeb, P. Pilligundla, F. Miresghallah, A. Yazdanbakhsh, S. Gao, and H. Esmailzadeh, “Releq: An automatic reinforcement learning approach for deep quantization of neural networks,” in *NeurIPS ML for Systems workshop, 2018*, 2019.
- [283] K. Wang, Z. Liu, Y. Lin, J. Lin, and S. Han, “Hardware-centric automl for mixed-precision quantization,” *International Journal of Computer Vision*, vol. 128, no. 8, pp. 2035--2048, 2020.

- [284] M. Rusci, A. Capotondi, and L. Benini, “Memory-driven mixed low precision quantization for enabling deep network inference on microcontrollers,” *arXiv preprint arXiv:1905.13082*, 2019.
- [285] Z. Dong, Z. Yao, A. Gholami, M. W. Mahoney, and K. Keutzer, “Hawq: Hessian aware quantization of neural networks with mixed-precision,” in *Proceedings of the IEEE/CVF International Conference on Computer Vision*, 2019, pp. 293--302.
- [286] T. Chu, Q. Luo, J. Yang, and X. Huang, “Mixed-precision quantized neural networks with progressively decreasing bitwidth,” *Pattern Recognition*, vol. 111, p. 107647, 2021.
- [287] J.-C. See, H.-F. Ng, H.-K. Tan, J.-J. Chang, W.-K. Lee, and S. O. Hwang, “Doubleqext: Hardware and memory efficient cnn through two levels of quantization,” *IEEE Access*, 2021.
- [288] S. Ioffe and C. Szegedy, “Batch normalization: Accelerating deep network training by reducing internal covariate shift,” in *International conference on machine learning*. PMLR, 2015, pp. 448--456.
- [289] B. Jacob, S. Kligys, B. Chen, M. Zhu, M. Tang, A. Howard, H. Adam, and D. Kalenichenko, “Quantization and training of neural networks for efficient integer-arithmetic-only inference,” in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2018, pp. 2704--2713.

- [290] R. Banner, Y. Nahshan, E. Hoffer, and D. Soudry, “Acic: Analytical clipping for integer quantization of neural networks,” *arXiv preprint arXiv:1810.05723*, 2018.
- [291] D. T. Larose and C. D. Larose, *Discovering knowledge in data: an introduction to data mining*. John Wiley & Sons, 2014, vol. 4.
- [292] B. Zhuang, C. Shen, M. Tan, L. Liu, and I. Reid, “Towards effective low-bitwidth convolutional neural networks,” in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2018, pp. 7920--7928.
- [293] B. Wu, A. Wan, X. Yue, and K. Keutzer, “Squeezeseg: Convolutional neural nets with recurrent crf for real-time road-object segmentation from 3d lidar point cloud,” in *2018 IEEE International Conference on Robotics and Automation (ICRA)*. IEEE, 2018, pp. 1887--1893.
- [294] B. Wu, X. Zhou, S. Zhao, X. Yue, and K. Keutzer, “Squeezesegv2: Improved model structure and unsupervised domain adaptation for road-object segmentation from a lidar point cloud,” in *2019 International Conference on Robotics and Automation (ICRA)*. IEEE, 2019, pp. 4376--4382.
- [295] B. Wu, A. Wan, X. Yue, P. Jin, S. Zhao, N. Golmant, A. Gholaminejad, J. Gonzalez, and K. Keutzer, “Shift: A zero flop, zero parameter alternative to spatial convolutions,” in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2018, pp. 9127--9135.

- [296] A. G. Howard, M. Zhu, B. Chen, D. Kalenichenko, W. Wang, T. Weyand, M. Andreetto, and H. Adam, “Mobilenets: Efficient convolutional neural networks for mobile vision applications,” *arXiv preprint arXiv:1704.04861*, 2017.
- [297] M. Sandler, A. Howard, M. Zhu, A. Zhmoginov, and L.-C. Chen, “Mobilenetv2: Inverted residuals and linear bottlenecks,” in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2018, pp. 4510--4520.
- [298] X. Zhang, X. Zhou, M. Lin, and J. Sun, “Shufflenet: An extremely efficient convolutional neural network for mobile devices,” in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2018, pp. 6848--6856.
- [299] A. Polino, R. Pascanu, and D. Alistarh, “Model compression via distillation and quantization,” *arXiv preprint arXiv:1802.05668*, 2018.
- [300] E. Jang, S. Gu, and B. Poole, “Categorical reparameterization with gumbel-softmax,” in *International Conference on Learning Representations*, 2017.
- [301] S. Shin, Y. Boo, and W. Sung, “Knowledge distillation for optimization of quantized deep neural networks,” in *2020 IEEE Workshop on Signal Processing Systems (SiPS)*. IEEE, 2020, pp. 1--6.
- [302] Y. Li, X. Dong, and W. Wang, “Additive powers-of-two quantization: An efficient non-uniform discretization for neural networks,” in *International Conference on Learning Representations*, 2020.

- [303] Y. Bengio, N. Léonard, and A. Courville, “Estimating or propagating gradients through stochastic neurons for conditional computation,” *arXiv preprint arXiv:1308.3432*, 2013.
- [304] Y. You, I. Gitman, and B. Ginsburg, “Large batch training of convolutional networks,” *arXiv preprint arXiv:1708.03888*, 2017.
- [305] Y. Zhang, N. Suda, L. Lai, and V. Chandra, “Hello edge: Keyword spotting on microcontrollers,” *arXiv preprint arXiv:1711.07128*, 2017.
- [306] J. L. McKinstry, S. K. Esser, R. Appuswamy, D. Bablani, J. V. Arthur, I. B. Yildiz, and D. S. Modha, “Discovering low-precision networks close to full-precision networks for efficient inference,” in *2019 Fifth Workshop on Energy Efficient Machine Learning and Cognitive Computing-NeurIPS Edition (EMCC2-NIPS)*. IEEE, 2019, pp. 6--9.
- [307] I. Loshchilov and F. Hutter, “Sgdr: Stochastic gradient descent with warm restarts,” *arXiv preprint arXiv:1608.03983*, 2016.
- [308] P. Goyal, P. Dollár, R. Girshick, P. Noordhuis, L. Wesolowski, A. Kyrola, A. Tulloch, Y. Jia, and K. He, “Accurate, large minibatch sgd: Training imagenet in 1 hour,” *arXiv preprint arXiv:1706.02677*, 2017.

VITAE

- Name: Ahmed Ibrahim Ahmed Shawahna.
- Nationality: Palestinian.
- Date of Birth: June 5, 1989.
- Email: *ahmad.shawahna@gmail.com*.
- Permanent Address: Jenin, West Bank, Palestine, Post Code: P2400896.
- Ahmad Shawahna received M.S. in computer engineering from King Fahd University of Petroleum and Minerals (KFUPM), Saudi Arabia, in 2016. He also received the B.Sc degree in computer engineering from An-Najah National University, Palestine, in 2012. In addition, Ahmad Shawahna worked at the Center for Communications and IT Research (CCITR), KFUPM. His research interests include hardware accelerator, deep learning, CNNs, FPGA, wireless security, network security, Internet of Things (IoT), and cloud computing.
- Publications:

1. **Ahmed Shawahna**, Sadiq M. Sait, Aiman El-Maleh and Irfan Ahmad, “*FxP-QNet: A Post-Training Quantizer for the Design of Mixed*

- Low-Precision DNNs With Dynamic Fixed-Point Representation*,” in IEEE Access, vol. 10, pp. 30202-30231, (2022), doi: 10.1109/ACCESS.2022.3157893.
2. Sadiq M. Sait, Aiman El-Maleh, Mohammad Altakrouri and **Ahmed Shawahna**, “*Optimization of FPGA-based CNN accelerators using meta-heuristics*,” in The Journal of Supercomputing, pp. 1-41, (2022), doi: 10.1007/s11227-022-04787-8.
 3. **Ahmed Shawahna**, Sadiq M. Sait and Aiman El-Maleh, “*FPGA-Based Accelerators of Deep Learning Networks for Learning and Classification: A Review*,” in IEEE Access, vol. 7, pp. 7823-7859, (2019), doi: 10.1109/ACCESS.2018.2890150.
 4. **Ahmed Shawahna**, Marwan Abu-Amara, Ashraf S. H. Mahmoud and Yahya Osais, “*EDoS-ADS: An Enhanced Mitigation Technique Against Economic Denial of Sustainability (EDoS) Attacks*,” in IEEE Transactions on Cloud Computing, vol. 8, no. 3, pp. 790-804, 1 July-Sept. (2020), doi: 10.1109/TCC.2018.2805907.
 5. **Ahmed Shawahna**, M. E. Haque and M. E. Tozal, “*Energy harvesting in wireless sensor network with efficient landmark selection using mobile actuator*,” 2017 Annual IEEE International Systems Conference (SysCon), Montreal, QC, Canada, (2017), pp. 1-8, doi: 10.1109/SYSCON.2017.7934761.
 6. H. ur Rahman, F. Azzedin, **Ahmed Shawahna**, F. Sajjad and A. S.

Abdulrahman, "*Performance evaluation of VDI environment*," 2016 Sixth International Conference on Innovative Computing Technology (INTECH), Dublin, Ireland, (2016), pp. 104-109, doi: 10.1109/INTECH.2016.7845102.