Titre: Title:	Hardware aware acceleration in deep neural networks
Auteur: Author:	Mohammad Hossein Askari Hemmat
Date:	2023
Type:	Mémoire ou thèse / Dissertation or Thesis
Référence: Citation:	Askari Hemmat, M. H. (2023). Hardware aware acceleration in deep neural networks [Thèse de doctorat, Polytechnique Montréal]. PolyPublie. https://publications.polymtl.ca/54838/

POLYTECHNIQUE MONTRÉAL
affiliée à l'Université de Montréal

Hardware aware acceleration in deep neural networks

MOHAMMADHOSSEIN ASKARI HEMMAT
Département de génie électrique

Thèse présentée en vue de l'obtention du diplôme de *Philosophiæ Doctor*
Génie électrique

Juin 2023

POLYTECHNIQUE MONTRÉAL
affiliée à l'Université de Montréal

Cette thèse intitulée :

Hardware aware acceleration in deep neural networks

présentée par **MohammadHossein ASKARI HEMMAT**
en vue de l'obtention du diplôme de *Philosophiæ Doctor*
a été dûment acceptée par le jury d'examen constitué de :

Christopher J. PAL, président
Jean Pierre DAVID, membre et directeur de recherche
Yvon SAVARIA, membre et codirecteur de recherche
François LEDUC-PRIMEAU, membre
Vaughn TIMOTHY BETZ, membre externe

ACKNOWLEDGEMENTS

I consider myself fortunate to have been able to meet and receive support from a wide range of individuals including friends, colleagues, mentors, and role models throughout my research journey.

I would like to thank my supervisors Jean Pierre David and Yvon Savaria. I am extremely grateful for the invaluable guidance and support provided by them throughout my research journey. Their unwavering commitment to my success has been a source of inspiration and motivation. Their expertise, insights, and constructive feedback have helped me to navigate the challenges of my research and have played a critical role in the completion of my Ph.D.

I would like to especially thank Sean Wagner and Olexa Bilaniuk for helping me throughout the BARVINN project. I am particularly grateful for the weekly meetings, which provided me with a platform to seek guidance and feedback on my work. Their constructive criticism and insightful comments have helped me to refine my research and produce better results. I would also like to thank Mohammad Pezeshki for inspiring me to embark on my PhD journey through his crash course on Deep Learning.

I would like to express my sincere gratitude to all my co-authors, Alex Hoffman, Christian Perone, Ehsan Sabouri, Francois Leduc-Primeau, Frank K. Gürkaynak, Ivan Lazarevich, Julien Cohen-Adad, Luca Benini, Lucas Rouhier, Matheus Cavalcante, Matteo Perotti, Nizar El Zarif, Olivier Mastropietro, Sina Honari, Sudhakar Sah, Theo Depuis, Yassine Hariri and Yoan Fournier for their invaluable contributions to this project. Without their expertise, dedication, and hard work, this work would not have been possible.

Besides my collaborators, many other individuals have played a significant part in influencing my research and providing their support throughout my academic journey. I would like to thank Alexandre Riviello, Anush Sankaran, Davis Sawyer, Fatemeh Rezvaninejad, Ghouthi Boukli Hacene, Jalal Rezvaninejad, Jonathan Balkind, Kamal Rezvaninejad, Maedeh Rahimnejad, Milad Ebrahimi, Razyie Rezvaninejad, Saad Ashfaq, Shahla Baghkhani , Sivakumar Chidambaram and Yasser Idris.

I am deeply thankful to Rayeheh Rezvaninejad, Yoan Fournier, Theo Depuis for providing me with invaluable feedback and assisting me in writing. This thesis would not have been possible without your support.

Finally, I would like to especially thank my family. I am forever grateful to my parents Zahra and Ataollah, for their unwavering love, support, and encouragement throughout my entire life. Thank you for always being there for me. I would like to express my heartfelt gratitude to my sister and brother, Reyhane and Ali. Their encouragement and belief in me have helped me navigate through

the challenges of my academic journey with confidence and resilience. Finally, I would like to express my deep gratitude to my beloved life partner and companion, Rayeheh Rezvaninejad, who has been my constant source of support, encouragement, and inspiration throughout this journey. I am truly blessed to have her in my life, and I could not have completed this thesis without her love and support.

RÉSUMÉ

Ces dernières années, les réseaux de neurones profonds sont devenus de plus en plus sophistiqués, leur permettant d'accomplir des tâches plus complexes. Cependant, à mesure que leur performance a augmentée, leur taille et leurs exigences en matière de calcul l'ont également. En particulier, pour les dispositifs de pointe où le calcul et la consommation d'énergie sont les plus importants, l'exécution efficace de modèles complexes est un défi. L'une des méthodes efficaces pour réduire les besoins en énergie et la complexité de calcul d'un réseau neuronal profond est appelée quantification. Ce processus implique de faire correspondre des valeurs à virgule flottante sur des valeurs entières de manière à minimiser la perte de précision. En réduisant la précision des paramètres et des calculs intermédiaires, la quantification peut conduire à une inférence plus rapide et à des besoins en mémoire réduits, ce qui est particulièrement bénéfique pour le déploiement de réseaux de neurones sur des appareils à ressources limitées.

Dans cette thèse, notre objectif est de comprendre le fonctionnement de la quantification et son impact sur l'entraînement des réseaux de neurones. De plus, nous explorons les moyens les plus efficaces d'employer la quantification en proposant du matériel dédié innovant. Pour atteindre ces objectifs, cinq articles distincts seront présentés. Le premier article présente un nouvel algorithme de quantification en virgule fixe qui cible spécifiquement l'accélération de l'inférence pour les tâches de segmentation d'images médicales. Les résultats de nos recherches révèlent trois points clés. Premièrement, la quantification peut être exploitée pour améliorer la vitesse de calcul, même dans les applications médicales qui exigent une grande précision. Deuxièmement, nos expériences suggèrent qu'il peut y avoir de légères améliorations de la précision par rapport aux modèles de précision totale lors de l'utilisation de la quantification. Cela nous a conduit à étudier les effets potentiels de régularisation de la quantification. Enfin, nous avons découvert que le matériel informatique standard peut présenter un goulot d'étranglement pour le déploiement efficace de modèles quantifiés, soulignant le besoin de conception de matériel sur mesure.

Dans le deuxième article, en nous appuyant sur les connaissances acquises dans notre premier article, nous avons formulé une hypothèse sur l'effet de régularisation de la quantification. Grâce à notre étude empirique, nous avons constaté que même si tous les niveaux de quantification ne présentent pas cet effet, la quantification 8 bits fournit de manière fiable une forme de régularisation.

Pour répondre aux exigences de calcul des modèles quantifiés, nous présentons des solutions matérielles personnalisées dans le troisième, quatrième et cinquième article. Dans le troisième et quatrième article, nous proposons un accélérateur entièrement personnalisé capable d'exécuter des modèles quantifiés avec une précision arbitraire. Enfin, dans le cinquième article, nous démontrons

les modifications nécessaires requises pour qu'un processeur vectoriel à usage général exécute des modèles quantifiés avec une précision inférieure à l'octet. Ces articles contribuent collectivement à notre objectif d'explorer le potentiel de la quantification et de développer des solutions matérielles efficaces pour son déploiement.

ABSTRACT

Deep neural networks have become increasingly sophisticated in recent years, allowing them to handle more complex tasks. However, as their capabilities have grown, so too has their size and computational demands. Especially, for edge devices where computation and power consumption is of the utmost most importance, running complex models efficiently is a challenge. One of the effective methods to reduce power requirement and computation complexity of deep neural network is called quantization. This process involves mapping floating-point values to integer values in a way that minimizes the loss of accuracy. By reducing the precision of the parameters and intermediate computations, quantization can lead to faster inference and lower memory requirements, which are particularly beneficial for deploying neural networks on resource-constrained devices.

In this dissertation, our goal is to understand how quantization works and its impact on training neural networks. Additionally, we endeavor to explore the most effective means of employing quantization by proposing novel custom hardware. To achieve these objectives, we present five distinct articles. The first article introduces a novel fixed-point quantization algorithm that specifically targets accelerating inference for medical image segmentation tasks. Our research findings reveal three key takeaways. Firstly, quantization can be leveraged to enhance computation speed even in medical applications that demand high precision. Secondly, our experiments suggest that there may be slight improvements in accuracy over full precision models when using quantization. This led us to investigate the potential regularization effects of quantization. Finally, we discovered that commodity hardware may present a bottleneck for the efficient deployment of quantized models, highlighting the need for bespoke hardware designs.

In the second article, building on the insights gained from our first article, we formulated a hypothesis about the regularization effect of quantization. Through our empirical investigation, we found that while not all quantization levels exhibit this effect, 8-bit quantization reliably provides a form of regularization.

To address the computational demands of quantized models, we present custom hardware solutions in the third, fourth, and fifth articles. In the third and fourth articles, we propose a fully customized accelerator capable of running quantized models with arbitrary precision. Finally, in the fifth article, we demonstrate the necessary modifications required for a general-purpose vector processor to run quantized models with sub-byte precision. These articles collectively contribute to our goal of exploring the potential of quantization and developing efficient hardware solutions for its deployment.

TABLE OF CONTENTS

ACKNOWLEDGEMENTS iii

RÉSUMÉ v

ABSTRACT vii

TABLE OF CONTENTS viii

LIST OF TABLES xi

LIST OF FIGURES xii

LIST OF SYMBOLS AND ABBREVIATIONS xiv

LIST OF APPENDICES xvi

CHAPTER 1 INTRODUCTION 1

CHAPTER 2 LITERATURE REVIEW 5

2.1 Deep Neural Networks 5

2.2 Convolutional Neural Networks 7

2.3 Accelerating Computation in Convolutional Neural Networks 10

2.3.1 Convolutional Neural Networks Architectures 10

2.3.2 Pruning 11

2.3.3 Quantization 13

2.3.4 Custom Hardware For Accelerating Convolutional Neural Networks 17

2.4 Summary and Problem Formulation 20

CHAPTER 3 ARTICLE 1 : U-NET FIXED-POINT QUANTIZATION FOR MEDICAL IMAGE SEGMENTATION 22

3.1 Prologue 22

3.1.1 Article Details 22

3.1.2 Context 22

3.1.3 Contributions . . . 23
3.2 Introduction . . . 25
3.3 Related Works . . . 26
3.3.1 Image Segmentation . . . 26
3.3.2 Quantization for Medical Imaging Segmentation . . . 26
3.4 Proposed Quantization . . . 27
3.4.1 Training . . . 28
3.4.2 Observations on U-Net Quantization . . . 29
3.5 Results and Discussion . . . 29
3.6 Power Efficiency . . . 31
3.7 Conclusion . . . 32
CHAPTER 4 ARTICLE 2 : QREG: ON REGULARIZATION EFFECTS OF QUANTIZATION . . . 34
4.1 Prologue . . . 34
4.1.1 Article Details . . . 34
4.1.2 Context . . . 34
4.1.3 Contributions . . . 35
4.2 Introduction . . . 36
4.3 Background . . . 36
4.3.1 Effect of Quantization on Accuracy Improvements . . . 36
4.3.2 Analytical Studies . . . 37
4.3.3 Using Quantization for its Regularization Effect . . . 38
4.4 Quantization Noise as a Regularizer . . . 38
4.4.1 Derivation of Theorem 4.4.1 . . . 40
4.5 Experimental Studies . . . 42
4.5.1 Experiment Setup . . . 42
4.6 Conclusion . . . 46
CHAPTER 5 ARTICLE 3 : RISC-V BARREL PROCESSOR FOR DEEP NEURAL NETWORK ACCELERATION . . . 47
5.1 Prologue . . . 47
5.1.1 Article Details . . . 47
5.1.2 Context . . . 47
5.1.3 Contributions . . . 47
5.2 Introduction . . . 49
5.3 Background . . . 50
5.4 Design . . . 51

5.4.1 Design Decisions and Data Path 51
5.4.2 Pipeline Structure 52
5.4.3 Processing Element Control and Monitoring 52
5.5 Results 53
5.6 Conclusion 56
CHAPTER 6 ARTICLE 4: BARVINN: ARBITRARY PRECISION DNN ACCELERATOR CONTROLLED BY A RISC-V CPU 57
6.1 Prologue 57
6.1.1 Article Details 57
6.1.2 Context 57
6.1.3 Contributions 58
6.2 Introduction 60
6.3 Related works 61
6.4 Architecture 62
6.4.1 Matrix Vector Units 62
6.4.2 PITO 69
6.4.3 Code Generator 71
6.5 Performance Analysis and Results 71
6.5.1 Experimental Setup 71
6.5.2 Discussion 72
6.6 Conclusion 73
CHAPTER 7 ARTICLE 5 : QUARK: AN INTEGER RISC-V VECTOR PROCESSOR FOR SUB-BYTE QUANTIZED DNN INFERENCE 75
7.1 Prologue 75
7.1.1 Article Details 75
7.1.2 Context 75
7.1.3 Contributions 75
7.2 Introduction 77
7.3 Background and Related Works 78
7.3.1 Custom Accelerators 78
7.3.2 Sub-byte Precision on Commodity CPUs 79
7.3.3 Custom RISC-V MPUs 79
7.4 Architecture 80
7.4.1 Bit-Serial Computation 80
7.5 Results and Performance Analysis 83
7.5.1 Performance Analysis 83

7.5.2 Physical Implementation Results 84
7.6 Conclusion 84
CHAPTER 8 GENERAL DISCUSSION 86
CHAPTER 9 CONCLUSIONS AND RECOMMENDATIONS 91

REFERENCES 93

APPENDICES 116

LIST OF TABLES

Table 2.1 Openvino inference timing results for running MNIST classification on Lenet5 10
Table 2.2 Energy consumption of multiplication and accumulation in a 45nm process [1] 14
Table 2.3 Energy consumption of memory access [1] 14
Table 3.1 Dice scores of the quantized U-Net models 32
Table 3.2 Comparing ReLU and Tanh run time using Intel's OpenVino 32
Table 4.1 Regularization effect of quantization measured with relative error improvement 41
Table 5.1 Complexity and performance summary of various RV32I implementations . . 53
Table 5.2 Detailed sythesis results for a PE array and a barrel core on KU115 FPGA . . 55
Table 6.1 Effects of Quantization on Accuracy and Model Size 60
Table 6.2 ResNet9 with different bit precision on CIFAR10 72
Table 6.3 Computation cost of ResNet9 layers on CIFAR10 dataset in BARVINN . . . 72
Table 6.4 Post-synthesis resource utilization of BARVINN 73
Table 6.5 Estimated performance of running CNV model on CIFAR10 on Alveo U250 . 73
Table 6.6 Performance for ResNet-50 model on ImageNet 73
Table 7.1 Quantization of ResNet18 Using LSQ [2] Quantization method 83
Table 7.2 Physical Implementation of Ara and Quark 84

LIST OF FIGURES

Figure 2.1 Max Pool function applied on an image from *Imagenet* dataset 9
Figure 2.2 Structured versus Unstructured pruning. This image was heavily inspired by [3] 13
Figure 2.3 Typical quantization flow. This image was heavily inspired by [2] 15
Figure 2.4 TPU Floor plan 18
Figure 3.1 Sample prediction versus ground truth segmentation results 30
Figure 4.1 Regularization effect of quantization over different quantization level 39
Figure 4.2 Validation objectness loss for YOLOv5n 45
Figure 4.3 Generalization effect of quantization on ResNet18 on CIFAR100 and CIFAR10 45
Figure 5.1 Sample instruction execution in a barrel processor 51
Figure 6.1 BARVINN hardware architecture with MVU array and PITO RISC-V controller 63
Figure 6.2 Channel sizes in models from ONNX Model Zoo. 66
Figure 6.3 Bit-transposed data format for arbitrary precision 66
Figure 6.4 VVP unit with a shifter-accumulator 68
Figure 6.5 Execution flow of a DNN on the MVU array 70
Figure 7.1 Effect of custom instruction on vector registers 78
Figure 7.2 Forward path in a quantized neural network 81
Figure 7.3 Per layer relative speed up of ResNet18 82
Figure 7.4 Roofline plot for Quark 82
Figure 7.5 Ara and Quark system placed-and-routed designs 83
Figure 8.1 Performance of Barvinn running Conv2d 89
Figure A.1 Weight distribution of U-Net when no quantization is used 116
Figure A.2 Dice score of Unet under quantization 119
Figure A.3 Sample prediction versus ground truth segmentation results for EM dataset . 120
Figure A.4 Sample prediction versus ground truth, NIH pancreas dataset 121
Figure A.5 Sample prediction versus ground truth segmentation results for GM dataset . 122
Figure B.1 Generalization gap results under different augmentation 123
Figure B.2 Regularization effect of quantization in classification task CIFAR10 dataset . 124
Figure B.3 Regularization effect of quantization in classification task CIFAR100 dataset 124
Figure B.4 Regularization effect of quantization , CIFAR100 dataset, weight only 125
Figure B.5 Regularization effect of quantization in classification task, weight and activation 125
Figure B.6 Augmentation in NLP task 126
Figure B.7 Regularization effect of quantization in NLP task 126

LIST OF SYMBOLS AND ABBREVIATIONS

AI	Artificial Intelligence
AGU	Address Generation Unit
APB	Advanced Peripheral Bus
ASIC	Application-Specific Integrated Circuit
AXI	Advanced eXtensible Interface
BRAM	Block Ram
CAD	Computer-Aided Design
CMOS	Complementary Metal-Oxide Semiconductor
CPU	Central Processing Unit
CPI	Cycles Per Instruction
CSR	Control and Status Register
CT-Scan	Computed Tomography Scan
DNN	Deep Neural Networks
DSP	Digital Signal Processor
FD-SOI	Fully Depleted Silicon-On-Insulator
FF	Flip Flops
FOSSI	Free and Open Source Silicon
FPGA	Field Programmable Gate Array
FPS	Frame Per Second
GF	Globalfoundries
GHz	Gigahertz
GOPS	Giga Operations Per Second
GPU	Graphics Processing Unit
GEMM	General Matrix Matrix Multiply
GEMV	General Matrix Vector Multiply
HART	Hardware Thread
HPC	High Performance Computing
IOs	Input Outputs
ISA	Instruction Set Architecture
LUTs	Lookup Tables
MAC	Multiply and Accumulate
MHz	Mega Hertz
ML	Machine Learning

MSB	Most Significant Bit
MSE	Mean Square Error
MVU	Matrix Vector Unit
MVP	Matrix Vector Product
NIH	National Institutes of Health
NLP	Natural Language Processing
ONNX	Open Neural Network Exchange
PITO	Petit Multi-Threaded RISC-V Controller
PE	Processing Element
PTQ	Post-Training Quantization
QDNN	Quantized Deep Neural Networks
QAT	Quantization Aware Training
RAM	Random Access Memory
RVV	RISC-V Vector
SGD	Stochastic Gradient Descent
SMT	Simultaneous Multi-Threading
SOTA	State of The Art
SHP	System Hyper Pipelining
STE	Straight Through Estimator
TOPS	Tera Operations Per Second
VVP	Vector Vector Product

LIST OF APPENDICES

Appendix A Supplementary Material for U-Net Fixed-Point Quantization for Medical Image Segmentation 116

Appendix B Supplementary Material For QReg: On Regularization Effects of Quantization 123

CHAPTER 1 INTRODUCTION

Artificial Intelligence (AI) has evolved significantly in recent years, becoming an essential component of many applications. This expansion has also brought with it new hurdles in terms of computing expenses. As AI models have become more sophisticated and capable of handling increasingly complex tasks, they have also become larger and more computationally demanding. This is particularly true for deep learning models, which require a vast amount of data and computation to train. The cost of training these models has become a major barrier to the widespread adoption of AI, as it requires access to large amounts of computational resources, such as powerful GPUs and high-performance computing systems. Additionally, even after a model has been trained, it can still require significant computational resources to run, making it difficult for small or resource-constrained organizations to implement AI solutions. This has led to ongoing research and development efforts aimed at finding ways to make DNN models more computationally efficient and accessible to a wider range of users. As an illustration, GPT-3 [4] is one of the largest language models ever created, with over 175 billion parameters, and training it requires access to significant computational power and storage. With a total computation requirement of 3.14×10^{23} flops for training GPT-3 175B [4], it is extremely difficult for many organizations to implement and use it. On the other hand, even much simpler and smaller tasks such as performing inference on the well-known ImageNet-1k [5] dataset for image classification using ResNet-50 [6] requires 4.12×10^{9} (running on NVIDIA Jetson TX1, : an embedded visual computing system with a 64-bit ARM A57 CPU, a 1 T-Flop/s 256-core NVIDIA Maxwell GPU and 4 GB LPDDR4 of shared RAM) flops of computation power [7]. Although this computational requirement can be satisfied by many recent processors, it is still a challenge to use such processors at the edge efficiently. These have lead to ongoing research and development efforts aimed at DNN models for different tasks to be more computationally efficient and accessible.

To make DNN models more computationally accessible and affordable, researchers and developers have been exploring a range of solutions. Two of the most promising methods are quantization and pruning.

In deep learning, quantization [8] is a technique to reduce memory consumption as well as the computation time of deep neural networks. This method involves reducing the precision of the computations performed by the model, which can significantly lower the amount of memory and computational resources required. Quantization can be applied in different ways, such as reducing the number of bits used to represent weights and activations [2], or using integer arithmetic instead of floating-point arithmetic [9]. By reducing the precision of computations, quantization can speed up the training and inference processes, making it a highly effective solution for reducing the

computational cost.

Pruning [10] is another popular solution for reducing the computational cost of DNN models. This method involves removing neurons or connections that have a minimal impact on the accuracy of the model. Pruning can be applied on weights [11] or filters [12]. By removing unnecessary neurons or connections, pruning can reduce the size of the model and lower the amount of memory and computational resources required for training and inference. Reducing the number of parameters often comes with less computation in a neural network, making the inference processing time faster. Pruning can also make the model more interpretable, as it can provide insight into which parts of the model are most important for its accuracy.

In this thesis, we chose to concentrate on quantization as a means of improving the computational efficiency of DNN models. While pruning, both structured and unstructured, can be applied before or after training and can be effectively utilized with specialized hardware, it is not as widely used for accelerating computation during training compared to quantization. Quantization, on the other hand, specifically Int8 quantization, can be applied both before and after training and is a simple technique that is easily integrated into existing deep learning frameworks. The effectiveness of quantization in reducing computational cost while preserving model accuracy has been extensively demonstrated, solidifying its position as the preferred solution for accelerating computation in DNN models within the context of our thesis.

In this thesis, our focus is on the acceleration of sub-byte quantization and its impact on training, inference, and the model's generalizability. While Int8 quantization is widely supported by most state-of-the-art machine learning frameworks and hardware, we aim to explore the potential of sub-byte quantization. To effectively utilize sub-byte quantization in neural networks, specialized hardware that supports low-precision computation is required. At present, there are no commercially available general processors such as CPUs or GPUs that can efficiently handle sub-byte computation in a neural network. To fully take advantage of quantization, one must consider customizing hardware through the development of Application-Specific Integrated Circuits (ASICs) [13–15] or to use re-configurable hardware [16–18]. In the quest for efficiently running quantized neural networks, throughout this dissertation, our research question is:

"*What is the effect of quantization on neural network learning and how to run quantized neural networks efficiently?*"

In the remainder of this dissertation, we present the following articles aimed at answering our research question:

- Chapter 3 presents "*U-Net Fixed-Point Quantization for Medical Image Segmentation*" [19]. As we will discuss in Chapter 3, this project sparked the research question of this dissertation.

The primary goal of Chapter 3 is to introduce a novel fixed-point quantization algorithm for the task of image segmentation. This work is motivated by the need to address the challenge of employing quantization in critical applications like medical image segmentation without sacrificing accuracy. During our investigation into appropriate quantization techniques, we discovered that in certain quantization scenarios, not only was accuracy not diminished, but the quantized models demonstrated increased accuracy compared to models utilizing full precision. Later in Chapter 4, we will investigate this phenomenon in more depth. Additionally, this chapter aims to highlight the potential for acceleration of computation on standard hardware through the use of fixed-point quantization. In Chapter 5, 6 and 7, we will propose custom architectures to accelerate computations in quantized neural networks.

- Chapter 4 presents "*QReg: On Regularization Effects of Quantization*" [20] which seeks to explore how quantization operates as a form of noise in neural networks, both during training and inference. By analyzing this noise propagation, the study demonstrates that quantization has a regularizing effect on the models. To further establish this claim, empirical experiments were conducted on various deep learning tasks, including classification, object detection, and natural language processing, using different datasets and levels of quantization. The results of these experiments revealed that the regularizing effect of quantization is most pronounced in 8-bit quantized models.

- Chapter 5 and 6 presents "*RISC-V barrel processor for deep neural network acceleration*" [21] and "*BARVINN: Arbitrary Precision DNN Accelerator Controlled by a RISC-V CPU*" [22] in which we first present a small, low power RISC-V controller capable of interleaving hardware threads. PITO is specifically made to provide fine-grained multithreading. We then used PITO in BARVINN to control an array of 8 processing elements called MVUs to perform different neural network functions. BARVINN is capable of accelerating most DNN functions with arbitrary precision. BARVINN has a code generator that takes in a quantized model in ONNX format and generates RISC-V code for PITO to run.

- Chapter 7 "*Quark: An Integer RISC-V Vector Processor for Sub-byte Quantized DNN Inference*" [23]. Like BARVINN, Quark is designed to run quantized models in arbitrary precision. However, unlike BARVINN, Quark is a vector processor designed for general-purpose use. In this chapter, we investigate, for a vector processor, what is needed to run quantized models with arbitrary precision efficiently. We then compare the performance of Quark with Ara, a general-purpose vector processor, and show the benefits of using dedicated instructions for low-precision computation.

Before diving into the main articles, we review necessary background in Chapter 2. Chapters 3

through 7 present the aforementioned articles. Chapter 8 contains a general discussion of our results and highlights potential areas of improvement for the methods proposed throughout this thesis. Lastly, Chapter 9 concludes this dissertation.

CHAPTER 2 LITERATURE REVIEW

In this chapter, we will go through the basic building blocks of this research. We will start by describing a simple deep neural network in Section 2.1. Then, in Section 2.2 we will discuss more complicated neural networks, specifically, convolutional neural networks (CNN). We describe basic components within a CNN and will identify the most computationally intensive part of a CNN. In Section 2.3, we will review various methods for accelerating computation in a neural network. We will also provide hardware solutions for accelerating computation in a neural network. Finally in Section 2.4, we will formulate a question that our research proposal tries to answer.

2.1 Deep Neural Networks

In this section, we will discuss the basic components of Deep Neural Networks. A DNN is made of stacking neural layers. Each neural layer is composed of many artificial neurons. Each neuron performs a simple affine transformation. An activation function is used in the output of the neuron which determines whether the output will be triggered or not. We can formally define this transformation as follows: considering a set of inputs $\mathbf{x} = \{x_1, x_2, ..., x_d\}$ coming from d neurons, and a set of weights $\mathbf{w} = \{w_1, w_2, ..., w_d\}$ that corresponds to each input, the affine transformation is defined as:

$$h(\mathbf{x}) = g\Big(\sum_{i=1}^{d} w_i x_i + b\Big), \tag{2.1}$$

where $g(.)$ is an activation function. Equation 2.1 can be written in form of vector transformation as follows:

$$h(\mathbf{x}) = g(\mathbf{w}^T\mathbf{x} + b) \tag{2.2}$$

The bias term b in Equation 2.1 allows the output of the neuron to be shifted before activation. There are various options for the activation function $g(.)$. The most common is the Rectified Linear Unit (ReLU). This activation function simply ignores the negative values and passes the positive values as is:

$$g(z) = \begin{cases} 0 & z \leq 0, \\ z & z > 0 \end{cases} \tag{2.3}$$

ReLU is the most commonly used activation function in any type of neural net. However, in some applications, ReLU is not the best activation function to use. From Equation 2.3, it is clear that the ReLU function is not bounded. However, in some networks [24], a bounded activation function is more desirable. Due to its simplicity, ReLU is very easy to compute. Although it is very simple, we can show how efficiently ReLU can be computed in practice. In Section 3 we will show that using Intel's OpenVino [25] inference engine with high performance `gemm_blas` and `avx2` instructions, computing ReLU is almost 6 times faster than Tanh activation function. Compared to other types of activation functions, ReLU has found much success with computer vision networks. As will be explained later in this chapter, our focus is mainly on Convolutional Neural Networks. Hence, our default activation function of choice will be a ReLU function.

To create a simple neural network, consider an artificial neural layer defined as having Equation 2.1 and Equation 2.3 back to back. Stacking multiple of these artificial neural layers back to back creates a simple neural network which is known as a Multilayer Perceptron (MLP) network. The following equation provides a formal definition of a multi-layer neural network:

$$\mathbf{h}^{(k)}(\mathbf{x}) = g(\mathbf{b}^{(k)} + \mathbf{W}^{(k)}\mathbf{h}^{(k-1)}(\mathbf{x})), \tag{2.4}$$

Here $\mathbf{h}^{(k-1)}$ defines the output of the k^{th} layer in a multi-layer neural network and for the first layer, we simply use the input vector $\mathbf{h}^{(0)}(\mathbf{x}) = \mathbf{x}$.

So far, we have described the relationship between the inputs and the outputs of a neural network. However, the weight vector has not been described. Initially, the weight vector can take any random values, however, to reliably approximate inputs (according to [26], MLP networks can be used as universal approximators), we need to "train" the weight vectors. In neural networks, training is the continuous process of updating parameters (weights) to minimize the cost. The update process is achieved using the Stochastic Gradient Descent (SGD) algorithm and minimizing cost is achieved by minimizing the error between the labeled data (actual output) and the prediction of the network (network output). Consider a neural network with the input vector $\mathbf{x}$, parameters θ and label $\mathbf{y}$, the optimization criterion for predicting $i = 1, ..., N$ labels is defined as follows:

$$\sum_{i=1}^{N} L(f(x, \theta), y_i) + \lambda\Omega(\theta) \tag{2.5}$$

where the $\lambda\Omega(\theta)$ is the regularization term that is used to prevent overfitting. $f(x, \theta)$ is the output of the network given the input vector $\mathbf{x}$ and parameter θ. Finally, $L(f(x, \theta), y_i)$ is a function that takes the label $\mathbf{y}$ and network's output and computes the loss. Depending on the task, the loss function can be different. However, for a multi-class classification task (which is the main focus of this research) a negative log-likelihood with softmax loss is usually used. This loss function is defined as below:

$$L(f_c(x,\theta), y_c) = -\sum_{c=1}^{C} y_c \log f_c(x,\theta) \tag{2.6}$$

Now to update the parameters using SGD, we can use the following algorithm:

Algorithm 1 Simple SGD algorithm to update network parameters θ

Initialize θ

while *Stopping Criterion not met* **do**

  Sample mini-batch of m examples from training set $x^1, ..., x^m$ and targets y^i

  $\Delta \leftarrow +\frac{1}{m}\nabla_\theta \sum_i L(f(x_i,\theta), y_i)$

  $\theta \leftarrow \theta - \lambda\Delta$

end

Using the algorithm defined in Algorithm 1, we can backpropagate the gradient of the loss with respect to parameters and update the parameters.

Although MLPs can be used in many applications, they are inefficient. Specifically, the number of connections can increase rapidly as the network grows in both length and depth. To overcome this problem, we can connect the neurons in a specific way to extract features from the input as much as possible with fewer connections. For instance, in an image, because of the nature of any image, the pixels around any given pixel contain redundant information. We can take advantage of this phenomenon when connecting our neurons. Convolutional Neural Networks [27] take this into account to build more efficient neural networks. In the next section, we focus on the fundamental idea behind CNNs and describe why they are much more efficient compared to MLPs.

2.2 Convolutional Neural Networks

Convolutional Neural Networks (CNNs) are a specialized form of a neural network that are designed to process data with a grid-like structure [28]. As an example, CNNs can be utilized for classifying images [5], speech recognition [29], image segmentation [30, 31], object detection [32] and many other applications. Without loss of generality, in this research proposal, our main focus is accelerating CNNs for image data. Since their inception [33], CNN models have evolved. In general, CNN models consist of the following building layers: Convolutional Layers, Pooling Layers, and Fully Connected Layers. In the following, we will briefly review the computations involved in these neural layers and we will explain why accelerating the computation in the Convolutional Layer has the biggest impact on accelerating the overall computation within a CNN. The term *Convolution* in Convolutional Neural Network resembles the mathematical *Convolution* function. In mathematics, the convolution of two continuous function $f(x)$ and $g(x)$, $x \in \mathbb{R}$, is defined as below:

$$f(x) * g(x) =: \int_{-\infty}^{+\infty} f(\tau)g(x-\tau)d\tau \tag{2.7}$$

and for discrete values where n is defined on a set of integers:

$$f[n] * g[n] =: \sum_{m=-\infty}^{+\infty} f(m)g(n-m) \tag{2.8}$$

However, in CNNs, we perform *cross-correlation* which is a slightly different operation compared to the convolution function. The only difference is that in *cross-correlation* we do not flip the kernels. Hence, a "Convolution" in a CNN with weight matrix $W \in \mathbb{R}^{m \times n}$ is performed on the two-dimensional input image X as follows:

$$Y[i,j] = X * W = \sum_m \sum_n X[i+m, j+n]W[m,n] \tag{2.9}$$

In a general case, where both X and W are tensors and we have multiple channels in our convolution layer, the pseudo-code for computing the convolution of input and weight tensors can be expressed as a set of nested loops as illustrated in Algorithm 2 where oC is the number of output channels, oH and oW is the height and width of output tensor, iC is the number of input channel and k is the kernel size. In this algorithm, we assumed input tensor is provided by $x[iC, H, W]$ and we perform convolution with filter tensor $k[oC, iC, k, k]$. We also assumed padding and stride of 1. Hence, input width and height is the same as output width and height.

Algorithm 2 Pseudo code for computing convolution of input $x[iC, oH, oW]$ and weight tensor $k[oC, iC, k, k]$ with multiple kernel channels and stride and padding of 1.

```
1:  for oc in 1..oC do
2:    for h in 1..oH do
3:      for w in 1..oW do
4:        for ic in 1..iC do
5:          for i in 1..k do
6:            for j in 1..k do
7:              y[oc,h,w] += x[ic, h+i, w+j] * weights[o, ic, i, j]
8:            end for
9:          end for
10:       end for
11:     end for
12:   end for
13: end for
```

As can be seen, computing convolution is a very expensive task. For instance, for convolution operation defined in Algorithm 2, we need to perform $oC * oH * oW * iC * k * k$ multiplication

operations. Algorithm 2 shows that there is an opportunity to process most of the calculation in parallel. In a multi-core processor, one solution [34] proposed to accelerate this computation is to dispatch multiple kernel channels (line 5 and 6) to different cores and depending on the core instruction set architecture (ISA), perform a vector operation to compute the rest (line 1, 2, 3 and 4). In Chapter 5 and 6, we will introduce a specialized multi-core processor and accelerator that uses multiple matrix vector units to accelerate computation in CNNs by running most of these inner loops in parallel.

Pooling Layers in CNNs are used to reduce the spatial size of the representation using simple non-linear downsampling. The input to the Pooling Layer is divided into multiple non-overlapping partitions and in the case of Max Pooling operation, the maximum value of the partition is used instead of the entire partition. This effectively shrinks the spatial size of the representation. In practice, Pooling Layers are inserted after every convolution layer which progressively reduces the number of parameters in a CNN.

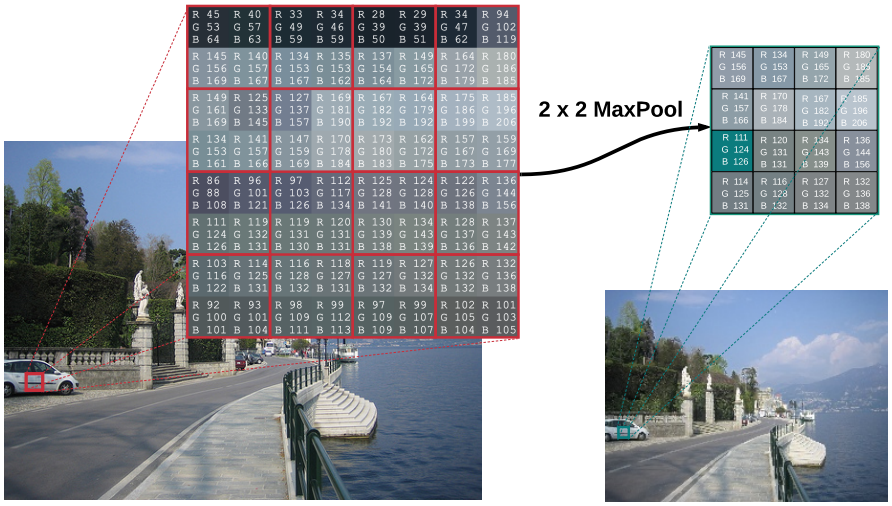


Figure 2.1 Max Pool function applied on an image from *Imagenet* dataset. Input image is an RGB image of size `3x500x375` and after Max Pool, the output image is an RGB image of size `3x250x187`

Figure 2.1 illustrates that Max Pool function is applied on an image from the Imagenet dataset [35]. We used `MaxPool2d` function in Pytorch [36] to generate the output image. We used a stride of 2 with a kernel size of 2 in `MaxPool2d` function. Although Max Pool helps a lot to reduce the network size and complexity, compared to convolution, it is a simpler function to implement [37, 38].

Another type of neural layer that is usually used mostly in the output layers of CNNs is the Fully Connected Layers. With equation 2.4, we have already covered the computation that is involved in these types of layers. Although, in general, Fully Connected layers are much more compute intensive than Convolutional Layers, apart from classification layer, they are not very common to use in vision

tasks. For instance, in VGG16 [39] network, 90% of the weights are in the Fully Connected Layers. On the other hand, these weights account for only 1% of the total floating-point operations. Table 2.1 illustrates the Intel's Openvino inference engine [25] timing results for running MNIST [40] classification on Lenet-like model [27]. The model consists of three Convolutional layers, each followed by a Pooling Layer. After re-ordering, the output is then fed into three Fully Connected layers. The table shows that in total, the Convolution Layers are consuming 119 μs to complete. With 45 μs, Fully Connected Layers are consuming the most time after the Convolutional Layers. Finally, the Pooling Layers are consuming the least amount of time to complete with only 16 μs.

Table 2.1 Openvino inference timing results for running MNIST classification on Lenet5

Layer Type	Instruction Type	Execution time in μs
Convolution	jit_avx2_FP32	26
Pooling	jit_avx_FP32	10
Convolution	jit_sse42_FP32	44
Pooling	jit_avx_FP32	6
Convolution	jit_avx2_FP32	49
Reorder	jit_uni_FP32	10
FullyConnected	gemm_blas_FP32	32
FullyConnected	gemm_blas_FP32	9
FullyConnected	gemm_blas_FP32	4

Although vector instructions (`jit_avx2_FP32`, `jit_avx_FP32` and `jit_sse42_FP32`) can improve the inference time, there are other solutions that can be exercised to reduce the inference time. In the following sections, we will review different methods for accelerating computation in CNNs.

2.3 Accelerating Computation in Convolutional Neural Networks

In the previous sections, we reviewed the building blocks of Deep Neural Networks. As we mentioned, in this work our main focus is CNNs. In this section, we review various methods to accelerate computation in the inference time of a CNN. We will review both software-oriented methods (since the nature of these methods, except Quantization, are somewhat Hardware agnostic and they can be implemented without any special support from hardware) and hardware-oriented methods.

2.3.1 Convolutional Neural Networks Architectures

The neural network architecture has a great impact on the overall performance of the network. Typically, convolutional neural networks consist of a Convolutional Layer, followed by a Pooling Layer and finally a Batch Normalization Layer. We can encapsulate these three layers into a Neural Block. Although it seems very straightforward, the number of these Neural Blocks that are connected one after another and the way they are configured has a huge impact on the final size and performance

of the network. One must tailor a network for its specific need to get an optimized network size with enough capacity to correctly classify the given dataset. As an example, consider the classification of the Imagenet dataset [35]. With 60 million parameters, 7 Convolution Layers, and 720 million MAC (multiply and add) operations, Alexnet [35] can achieve top-1 accuracy of 57.2%. On the other hand, mobilenet-v1 [41] with 4.2 million parameters and 569 million operations can achieve top-1 accuracy of 70.6%. The better performance was achieved using Depthwise Separable Convolution in which the usual channel-wise spatial convolutions are followed by pointwise 1×1 convolution to change the dimension. According to [41], the operation cost of a Depth-wise Separable Convolution for a layer with M input channel, N output channel, and with a kernel size of D_K and feature map size of D_F is defined as below:

$$D_k.D_k.M.D_F.D_F + M.N.D_F.D_F \tag{2.10}$$

However, for a standard convolution it is:

$$D_k.D_k.M.D_F.D_F.N \tag{2.11}$$

Which gives the computation reduction of:

$$\frac{D_k.D_k.M.D_F.D_F + M.N.D_F.D_F}{D_k.D_k.M.D_F.D_F} = \frac{1}{N} + \frac{1}{D_K^2} \tag{2.12}$$

As an example of D_K equal 3, a performance boost of 8 to 9 can be achieved. Finding the best architecture for a given task is very tedious. Neural Architecture Search (NAS) algorithms have outperformed manually designed architectures for image classification task [42, 43]. Although NAS has shown impressive results, the process of finding an optimal architecture requires hours of training [44]. In the next section, we will explore another solution that can reduce the number of computations in a neural network without hours of training.

2.3.2 Pruning

The idea of removing unimportant weights from a neural network dates back to early 1990s with Yan Lecun's "*Optimal Brain Damage*" paper [45]. Even with small networks, most weights have little contribution to the output. In recent years, numerous innovative pruning techniques have been proposed [3, 6, 10, 45–73]. In the following section, we will provide an overview of some of these methods. Pruning methods can be classified into two main categories: structured and unstructured. In structured pruning [50, 51, 53, 54, 57, 58, 60, 61, 63, 73], we remove entire filters or channels, from the network. For instance in [47], a method for pruning filters in a Convolutional Layer was proposed.

Given a filter $F_{i,j} \in \mathbb{R}^{n_i \times k \times k}$ (i is used to iterate layers and j to iterate features) with n_i kernels $\mathcal{K} \in \mathbb{R}^{k \times k}$ of j^{th} Convolutional Layer, we can rank each filter by L_1 norm of the kernel weights:

$$s_j = \sum_{l=1}^{n_i} |\mathcal{K}_l| \tag{2.13}$$

We can then prune m filters that have the smallest norm. Note that the kernels in the next Convolutional Layer corresponding to the pruned feature map are also removed. Using un-pruned weights, a new network is built with the same structure but far fewer parameters. In [47], authors reported an accuracy drop of only 1.04% while pruning 24.2% of the weights for a ResNet-34 [6] model trained on Cifar10 [74] dataset.

In [49], authors proposed a new pruning algorithm that tries to preserve the accuracy of the pruned network. They formulated this as the following combinatorial optimization problem:

$$min_{\mathcal{W}'} |\mathcal{C}(\mathcal{D}|\mathcal{W}') - \mathcal{C}(\mathcal{D}|\mathcal{W})| \quad s.t \quad \|\mathcal{W}\|_0 \leq B \tag{2.14}$$

where $\mathcal{D}$ is the training examples, $\mathcal{C}(\mathcal{D}|\mathcal{W})$ is the cost function of the network given the parameter $\mathcal{W}$ with respect to training examples and L_0 norm ($\|\mathcal{W}\|_0$ bounds the number of non-zero parameters B in $\mathcal{W}'$. By solving this optimization problem, authors claimed a 10x theoretical reduction of 3D-convolutional filters with a small drop in accuracy in a recurrent gesture classifier [49].

On the other hand, in unstructured pruning [3, 45, 65, 70, 72], we remove individual weights or connections from the network. Various methods have been proposed to find out which individual weight or connection needs to be removed. One of the earliest ideas was proposed in [45], in which the authors proposed to use saliency to assess the importance of each weight, which was computed using the diagonal elements of the Hessian matrix. In [70], authors developed an iterative pruning technique using ℓ_1 regularization, followed by fine-tuning step. [56] proposed connection splicing to restore weights that were pruned incorrectly and incorporated it into the training process for dynamic weight learning. [10] employed a 3-stage pipeline to prune weights and achieved a 49× compression ratio on VGG-16, with only minor accuracy loss.

Figure 2.2 provides a visual representation of the differences between structured and unstructured pruning. The image shows that from left to right, the compression rate increases as the network becomes more sparse. From right to left, the computational complexity decreases as more filters are removed. Structured pruning is generally easier to implement and leads to more predictable results, while unstructured pruning can result in a more aggressive reduction of the network size with more potential for performance gains. Ultimately, the choice of which pruning technique to use will depend on the specific needs of the application and the resources available.

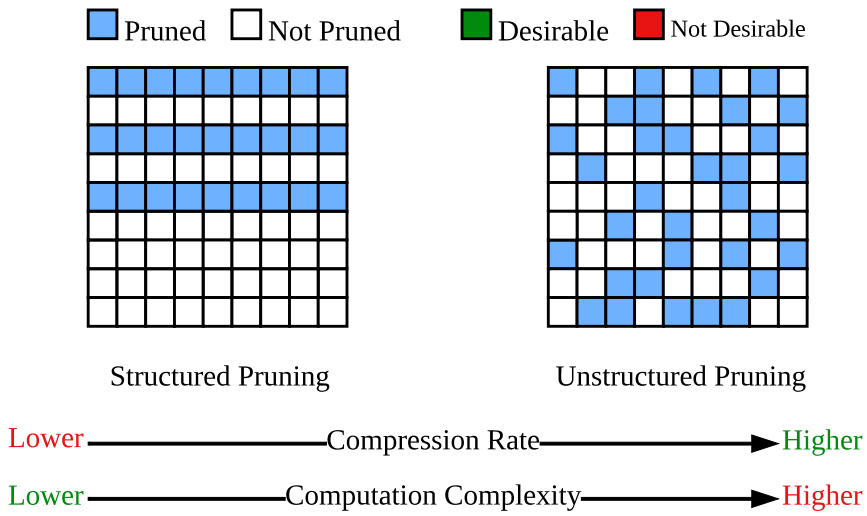


Figure 2.2 Structured versus Unstructured pruning. This image was heavily inspired by [3]

Most of the time, the accuracy degradation that is expected can be tolerated with the new smaller, and faster network. In pruning, we re-train (fine-tune) the new network to gain back some of the lost accuracies. Most pruning algorithms can be tuned to prune a certain percentage of the weights in the network. This is important since in practice, pruning is an iterative process and too much of it can result in non-recoverable accuracy for the new network. Although pruning shows impressive results for reducing the overall size of the network, in reality, the fine-tuning process of the pruned network to gain back the dropped accuracy is challenging. This is mostly because, with pruning, we are reducing the capacity of the network to learn. Also, because of the smaller network capacity, the pruned network has a higher generalization error. In the next section, we review quantization techniques that can be used to reduce the network size and accelerate the overall computation, while preserving the generalization error as much as possible.

2.3.3 Quantization

Quantization is the task of approximating a high precision (floating-point) model with a lower bit resolution model (usually 1 to 8 bits) by quantizing weights and activations of the network. Deep neural network (DNN) quantization has been an active research topic [2, 3, 8–10, 19, 20, 72, 75–110] and the first attempts to quantize a neural network dates back to the 1990s [75, 111, 112].

The primary objective of quantization is to reduce the precision of calculations while preserving the accuracy of the results. Nonetheless, the pertinent question remains as to what extent quantization can lead to improvements in power consumption and computational speed. Tables 2.2 and 2.3 show

the power consumption to perform multiplication, accumulation, and memory access in 45 nm CMOS process [1] (the *pJ* and *nJ* refer to the power consumption that is measured in *Joule* or *J* in short. *pJ* is pico-joule 10^{-12}, *nJ* is nano-joule 10^{-9}.).

Table 2.2 Energy consumption of multiplication and accumulation in a 45nm process [1]

Operation	MUL	ADD
8-bit Integer	0.2pJ	0.03pJ
32-bit Integer	3.1pJ	0.1pJ
16-bit Floating-Point	1.1pJ	0.4pJ
32-bit Floating-Point	3.7pJ	0.9pJ

Table 2.3 Energy consumption of memory access [1]

Memory Size	Cache Access
8KB	10pJ
32KB Integer	20pJ
1MB Floating-Point	100pJ
DRAM	1.3-2.6nJ

Table 2.2 and 2.3 show that as the number of bits used to perform an operation increases, the power consumption also increases. On the other hand, it can be clearly observed that operating on integers is considerably more power efficient than using floating-point values. Chapter 7 will provide further insight into the potential benefits of using integer values. One of the significant advantages is the reduction in die area required on an ASIC, leading to an overall decrease in power consumption.

Quantization not only enhances computation speed but also accelerates specific operations such as multiplication and division. For example, on the Intel Core i7 4770 @ 3.40GHz processor, 32-bit integer multiplication exhibits a threefold increase in throughput compared to floating-point data types [113]. This speedup can partially be attributed to the reduced memory requirements of fixed-point numbers, which results in less data movement and consequently, faster computation.

With the advantages of quantization in mind, let us now explore the various methods used for quantizing deep neural networks. Quantization can be applied to a model after or while training. Quantization after training (post-training quantization) or PTQ [114–117] can be done statically or dynamically. In post-training static quantization, weights are quantized ahead of time and during a calibration process on the validation set, a scale and bias are computed for the activations. In post-training dynamic quantization, much like post-training static quantization, the weights are quantized ahead of time but the activations are dynamically quantized at inference. Dynamic quantization is useful for models where model execution time is dominated by the time it takes to load weights for the model e.g. LSTMs [24].

One difficulty with quantization is that it can cause a deviation in the model parameters, leading it to move away from the point where it was optimized during training with floating-point precision. To overcome this issue, a potential solution is to utilize Quantization Aware Training (QAT) to train the model with quantized parameters. In QAT, the quantization parameters are learned while other

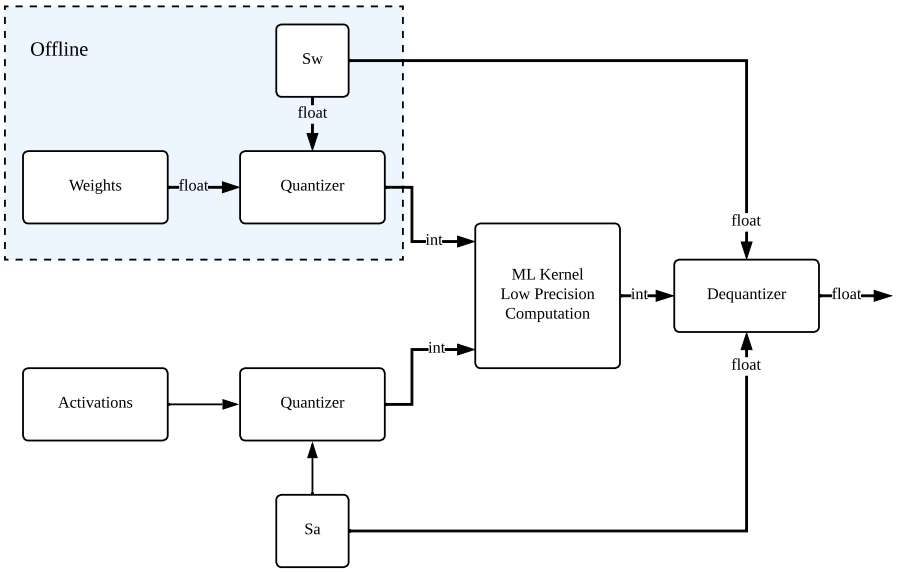


Figure 2.3 Typical quantization flow. This image was heavily inspired by [2]

parameters in the network are learned. Figure 2.3 illustrates a typical feed-forward flow for most quantization methods. As explained before, the idea is to perform the most computation intensive parts in low precision and use scaling to retrieve the accuracy degradation. Given input tensor t, scaling factor s and b bits to quantize, the quantized tensor $\bar{t}$ can be calculated as below:

$$\bar{t} = round(clip\ (\frac{t}{s}, -2^{b-1}, 2^{b-1} - 1)) \tag{2.15}$$

The $clip(r, z_1, z_2)$ function returns r with values below z_1 set to z_1 and values above z_2 set to z_2. For dequantization, we can retrieve the approximated floating-point tensor as below:

$$\hat{t} = \bar{t} \times s \tag{2.16}$$

During QAT (Quantization-Aware Training), when training a model, scaling factors for both weights and activations need to be learned. These scaling factors are treated as regular parameters in the network. However, to achieve optimal results, as noted in [2], it may be necessary to adjust the gradient while updating these scaling parameters.

During QAT, the forward and backward pass are performed in floating-point, but the model parameters are quantized after each gradient update. In general, compared to PQT, QAT methods achieve higher performance (accuracy) for all quantization levels. This performance boost comes at the cost of extra fine-tuning or even full training. Nevertheless, QAT enables the use of quantization with low precision, or even binary quantization.

Handling the non-differentiable round function, which is a commonly used quantization operator, is a crucial aspect of backpropagation in QAT. The gradient of this operator is zero almost everywhere, so the Straight Through Estimator (STE) [118] is used to approximate the gradient. STE is a technique that allows backpropagation to be applied to models with discrete or non-differentiable components. During the forward pass, the non-differentiable component is applied as usual, but during the backward pass, the gradient from the previous layer is passed through without modification. This "straight through" approach enables the flow of gradient information through the non-differentiable operation, enabling the model to be trained using standard gradient-based optimization algorithms. While STE is a commonly used method, other methods have also been explored in the literature, including stochastic neuron [118], combinatorial optimization [119], target propagation [120], and Gumbel-softmax [121]. In practice, most QAT methods use STE to approximate the gradient.

Previous studies have discovered that it is advantageous to not only modify the model parameters but also learn quantization parameters during QAT. To illustrate, PACT [96] trains the model to acquire the optimal clipping ranges of activations when uniform quantization is used. Similarly, QIT [122] trains the model to acquire quantization steps and levels in the context of non-uniform quantization. In addition, LSQ [2] utilizes a novel gradient estimate to learn scaling factors for non-negative activations during QAT, while LSQ+ [123] broadens this concept to encompass general activation functions such as swish and h-swish. LCQ [124] proposes a novel non-uniform quantization method which optimizes quantization levels via a learnable companding function that can flexibly and non-uniformly control the quantization levels of weights and activations. LCQ introduces a new weight normalization technique called limited weight normalization (LWN) to stabilize the training for quantization.

In addition to the aforementioned methods, researchers have pushed QAT methods to its limit by proposing binary quantization. The research on ultra low bit quantization of DNNs was initially pioneered by BinaryConnect and XNOR-Net. BinaryConnect [125] is a quantization method that binarizes variables in the feed-forward path using a rounding function and straight-through estimator for backpropagation. While it works well on small datasets, other types of quantization such as Xnor-Nets and ternary weight networks are recommended for larger datasets. XNOR-Nets [126] involves using Binary Weight Networks (BWN) and XNOR networks to binarize both weights and activations, with an approximation of convolution and methods such as BNorm (binary normalization), BinActive (binary activation), BinConv (binary convolution), and Pooling used to reduce information loss due to binarization.

In addition to Xnor-Nets and BinaryConnect, many other ultra low bit quantization methods have been proposed [127–137], among others. Despite these advancements, the accuracy degradation remains too high for these methods to be widely used in real-world applications.

In this thesis, our main focus is on sub-byte quantization techniques as they can provide a balance between model accuracy and compression. Sub-byte quantization methods enable us to use a limited number of bits to represent each weight or activation, which can greatly reduce the model size and memory requirements. However, we avoid using binary quantization since it typically results in a significant degradation of model accuracy. In Chapters 5, 6 and 7, we will see that our custom hardware follows a similar feed-forward computation graph as shown in Figure 2.3. This allows us to apply various quantization techniques and utilize arbitrary precision to strike a balance between accuracy and performance.

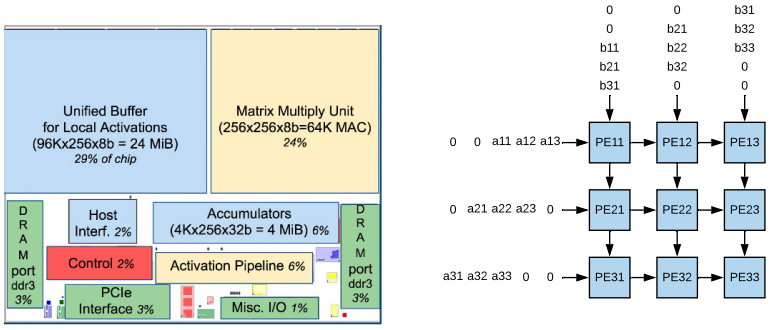
2.3.4 Custom Hardware For Accelerating Convolutional Neural Networks

In previous sections, we primarily discussed software-oriented methods that can enhance neural network performance. Our aim was to identify techniques that are agnostic to hardware. However, it is important to note that the most effective acceleration methods are achieved when software methods are designed with a deep understanding of the underlying hardware. In this section, we will examine custom hardware accelerators that are designed to accelerate computation in Convolutional Neural Networks.

Custom ASIC Neural Accelerators:

The race for designing an ASIC for accelerating Neural networks dates back to the 1990s. The Padamavati/Space project [138] that was started in 1987 was a 170,000 32-bit associative processor designed and built with 1.2 um CMOS process on a 5.8 by 7.9 mm^2 die and was able to run at 8MHz. It was designed to process natural language and images using Lisp and Prolog programming languages. HiPNet-1 [139] is another custom chip that was designed with a 2um CMOS process and was capable of running at 20MHz. These custom processors did not get much attention since neural networks faded in popularity in the 1990s. Also, Moore's law favored general processors for these applications. Fast forward to 2023, all major technology companies have invested in designing a custom neural chip. Google announced its 4th version of Tensor Processing Unit (TPU) [140]. Google TPU is a custom hardware that is specialized in accelerating Deep neural network models. In TPU V1 [13], the Matrix Multiply unit (that is used to compute the product of weights and activations), plus the local unified buffer has consumed more than 24 % of the TPU's die floor plan.

The Matrix Multiply unit in TPU is a Systolic Array [141] architecture. A systolic array is a two-dimensional grid of Processing Elements (PE). Each PE is equipped with a small MAC unit with some glue logic to store and forward data. Compared to a General Processing Unit (GPU), the PEs in a GPU are full-featured processors where that all share a common memory unit. The simplicity of a PE in a systolic array allows it to achieve a higher density of the MAC unit. Figure 2.4(b) illustrates how systolic arrays can multiply two matrices. At each clock cycle, the PEs multiply their



(a) (b)

Figure 2.4 2.4(a) shows the TPU's floor plan depicting the die area used by every block. The data buffers (blue) use 37% , the compute block (yellow) uses 30% , the I/O (green) uses 10% and the control (red) uses 2% of the die area. The image also shows that the matrix multiply unit uses a significant portion of the die. The image source has been taken from [13]. Figure 2.4(b) illustrates how systolic arrays can multiply matrix a and b

input together and add them to their partial sums. The result is then shifted to the PE's neighbors on the next clock cycle. With this architecture, TPU claimed to achieve much better throughput and performance per watt compared to an Nvidia GPU [141]. At peak performance of 92 TOPS/s (92×10^{12} **8-bit** bfloat operations per second) 700MHz, TPU consumes 40 Watts of power [13]. On the other hand, Nvidia K80 GPU (which has 2 dies per card) consumes 98 Watts of power providing 2.8 TOPS/s (2.8×10^{12} **32-bit** floating-point operations per second) running at 560MHz [13].

In academia, many notable accelerators were designed for ASIC and FPGA flows. **Celerity** is an open-source 511-core RISC-V [142] tiered accelerator fabric [143]. It is a 5×5 mm System On a Chip (SoC) taped out in TSMC 16nm process. It includes five RISC-V RV64G [144] cores for the general-purpose tier, 496 RV32IM [144] RISC-V cores in the massively parallel tier which are connected to a complex HLS generated (High-Level Synthesis) BNN accelerator [143]. It runs at 1.4GHz and at this clock frequency, it has 694.4 INT32 GOPS and it consumes 7.47 Watts.

Eyeriss [14] is another Convolutional Neural Network accelerator that was designed with energy efficiency in mind. It consists of a PE array of 14 ×12 constructed in the form of a systolic array. Each PE has a local memory of 1KB and shares a global buffer of 108KB with other PEs. Each PE is capable of performing a MAC operation with 16-bit fixed-point precision. This chip was implemented in a 65 nm CMOS process and runs at 250 MHz. It is capable of running convolutions in AlexNet at 35 frames per second and consumes only 278 milli-Watts of power. This energy efficiency partly comes from skipping MAC operations when the inputs are zero. The authors claimed that his technique alone resulted in 45% energy consumption reduction.

In addition to the aforementioned custom accelerators, in the following we will review custom accelerators with support for low-precision and arbitrary precision computation.

Loom [145] is an accelerator for quantized DNNs that is capable of variable-precision inference, in both DNN weights and activations, at the *sub-layer* level. This enables Loom to exploit variability in precision as needed across layers, or to trade bit-precision (and therefore speed) for model accuracy. Loom supports variable precision by exploiting bit-serial execution, rather than a bit-parallel one. A Loom engine consists of $128 \times 16 = 2048$ Serial Inner-Product units (SIPs), each of which performs 16 1×1-bit products per cycle. An $m \times n$-bit product requires mn clock cycles, but the small size of the SIPs allows a very large number of them to be synthesized, compensating high latency with high throughput.

RaPID [146] is a specialized ASIC accelerator targeting ultra-low-precision neural networks. It supports both training and inference, and both floating-point (IEEE FP16, Hybrid-FP8) and fixed-point (2/4-bit) numbers. RaPiD contains a ring bus interconnecting 4 cores, each containing an 8×8 systolic array of processing elements. Each PE is equipped with a 128-bit-wide SIMD unit with separate FP and INT pipelines. This separation allows *double-pumping* (the execution of two instructions per clock cycle) the INT pipeline to be used for inference, allowing for as much as an 8x increase in throughput and 10.6x higher energy efficiency. When RaPiD is implemented with 4 cores in 7 nm EUV technology, it can be clocked at 1.5GHz, to achieve 12.8 TFLOPS in FP16, 25.6 TFLOPS in HFP8, and 102.4 TOPS in INT4.

FPGA based Neural Accelerators:

So far, we have discussed custom neural accelerators that are specifically designed for ASIC implementation. While these chips offer outstanding performance, their high non-recurring engineering costs make them unaffordable for many. In contrast, FPGA-based accelerators offer configurable designs at a fraction of the development cost of custom chips. In the following section, we will explore some of the most noteworthy FPGA-based accelerators. One of the key advantages of FPGA accelerators is their ability to easily adopt custom data types. Moreover, as new and innovative neural network architectures continue to emerge, FPGA-based designs can be configured to leverage these advances with some development effort.

fpgaConvNet [18] is an automatic design scheme for mapping CNNs on FPGAs. In most CNN models, storing weights of even a single layer can exceed the on-chip RAM available to the FPGA. fpgaConvNet addressed this issue by introducing a folding factor for each layer which allows folding the feature maps within a layer. CNNs for fpgaConvNet is formulated as a streaming problem and the automated tool models the computation as Synchronous Dataflow (SDF). This form of computation modeling allows fpgaConvNet to co-optimize the generated hardware based on the input CNN workload and the target device. For instance for AlexNet, fpgaConvNet produces a hardware that is

capable of running at 125MHz and has a performance of 155.81 GOPs (each op is a mac of two 16-bit fixed-point numbers) per second.

Caffeine [147] is another FPGA accelerator that contains 1058 MAC units. Each MAC is capable of performing multiplication and addition of two 16-bit fixed-point numbers. To run AlexNet, Caffeine runs at 200MHz and has a peak performance of 310 GOPs per second, and consumes only 25 Watts of power. Compared to Nvidia K40 GPU with the same load, Caffeine consumes $\frac{1}{10th}$ of the power.

SnowFlake [148] is a scalable FPGA-hosted accelerator for neural network inference. It is composed of "compute clusters" (one only on the Zynq ZC706), each of which contains a memory interface and a RISC-type processor driving 4 "compute units", interconnected by a data distribution network. Each such compute unit contains 4 vector multiply-accumulate (vMAC) units performing sixteen 16×16-bit scalar MACs per clock cycle. A 16KB per-vMAC buffer stores neural-network weights and a 64KB, shared, per-compute-unit buffer stores data vectors. A total of 256 MACs / clock cycle / cluster can be performed. The RISC processor is 5-stage pipelined and orchestrates data movements and computations using instructions from a custom ISA. The SnowFlake design does not support sub-16-bit precision math, preferring to focus on high occupancy of wide MAC units. At 250 MHz, one compute cluster delivers 64 GMACs (16×16-bit) with an efficiency of 91-95% for several well-known CNN architectures, such as AlexNet, GoogleNet and ResNet.

DNNBuilder [149] is a toolchain that automatically generates efficient FPGA implementations of DNNs from their descriptions in a high-level framework such as TensorFlow. To support a variety of DNNs and FPGAs, DNNBuilder uses a parameterized processing engine (PE) that is tuned under the latency, throughput and resource constraints of the target FPGA. Arbitrary quantization is supported for both weights and activations, and across layers, though only 4-, 8- and 16-bit are demonstrated. DNNBuilder was demonstrated with as an accelerator capable of 4 TOPS at 8-bit precision. Like FINN, the entire DNN is implemented in the generated logic, which limits the size of the DNN that can be fit onto the FPGA.

2.4 Summary and Problem Formulation

This chapter reviewed the basic building blocks of a neural network. We reviewed how a convolutional neural network works and explored the fundamental components of a convolutional neural network. We reviewed popular methods to accelerate computation in neural networks. We reviewed how pruning and quantization can be used to reduce computation in a neural network. Compared to pruning, quantization can be more versatile and effective in practice. Although pruning can dramatically reduce model size and hence required computation, it is challenging to maintain the accuracy of the model without making the kernels too sparse. Finally, we reviewed some of the

most notable hardware accelerators that are specifically designed to accelerate computation for QDNNs. FPGA accelerators offer several advantages over general-purpose GPUs, including lower power consumption and greater flexibility in supporting different data types. Unlike GPUs, which are limited to a fixed set of supported data types, FPGA-based designs can easily adopt new data types. While custom ASICs are a good option for optimizing the performance per watt metric, their non-recurring engineering costs make them less accessible to many.

We believe that quantization can significantly improve the efficiency of deep neural networks, and our research aims to explore this concept further. In the following chapters, our goal is to not only understand the impact of quantization on model accuracy but also to develop custom hardware solutions that can accelerate quantized models without sacrificing performance. Through our work, we hope to contribute to the development of more energy-efficient and practical deep learning systems.

CHAPTER 3 ARTICLE 1 : U-NET FIXED-POINT QUANTIZATION FOR MEDICAL IMAGE SEGMENTATION

3.1 Prologue

3.1.1 Article Details

U-Net Fixed-Point Quantization for Medical Image Segmentation
MohammadHossein AskariHemmat, Sina Honari, Lucas Rouhier, Christian S. Perone, Julien Cohen-Adad, Yvon Savaria and Jean-Pierre David.

Published in the proceedings of the 2019 Medical Imaging and Computer Assisted Intervention (MICCAI), Hardware Aware Learning Workshop, October 24 2019.

3.1.2 Context

As part of IVADO's "Accelerating Bio-Medical Image Segmentation" project, our group was tasked with proposing and implementing the necessary algorithms and systems to facilitate the deployment of bio-medical models on edge devices. Given the resource constraints inherent in edge device deployment, including limitations on power consumption and computational capacity, our initial approach for the project was to utilize Xilinx FPGAs and employ compressed algorithms to reduce computational and memory requirements. Our literature review indicated that quantization is a suitable technique that can provide both compression and reduce computation complexity. For such custom deployment, we proposed to implement custom hardware for quantized neural network inference.

Later on, each part of this project became a big part of my Ph.D. journey. While proposing a new fixed-point quantization technique, we realized for some quantization levels, we are achieving higher accuracy compared to the base floating models. Soon we found out that this has been partially reported in other quantization papers as well. As we will discuss in Chapter 4, model quantization and its effects on training became the bedrock of part of my Ph.D. While fixed-point quantization was proving to be more effective, our experiments faced the challenge of real-world hardware deployment. In Chapter 5, 6 and 7, we will discuss how we overcame these challenges by proposing new custom hardware for quantized neural networks.

3.1.3 Contributions

The novelty of this work is as follows: We analyzed the distribution of parameters in U-Net and realized that compared to other models, U-Net requires a higher dynamic range. On the other hand, due to the limitation of our target hardware (Xilinx FPGAs) and since floating-point arithmetic requires more power compared to integer arithmetic, we proposed a fixed-point quantization. In most quantization techniques, a floating-point scaler is used to re-scale the quantized values back to floating point. This is mostly used to re-gain the dynamic range to represent the network parameters and reduce error. Compared to other quantization techniques, our fixed-point quantization does not require any floating point re-scaler.

Several subsequent studies have cited our proposed fixed-point quantization method. Independent studies, such as [109, 110, 150, 151] have compared our proposed method, sometimes referring to it as $FixQ$ technique, against their proposed techniques. To ease the reproducibility of our technique, we have published our code at `https://github.com/hossein1387/U-Net-Fixed-Point-Quantization-for-Medical-Image-Segmentation` Github repository. Our project has been well-received by the research community, as evidenced by the over 70 stars on our Github repository at the time of writing this thesis. Referring to our quantization technique as $FixQ$, [109] validates our results and implementation and states: " *For methods with code publicly available (which include DoReFa +* $PACT^2$ $APoT^3$ $FixQ^4$ *and* $TernaryNet^5$*), we integrate their implementation into our training code with only necessary modification ...*" which shows our results were reproducible by independent studies.

Personal Contributions MohammedHossein AskariHemmat was a key contributor to the conceptualization and composition of this work. Christian S. Perone executed the first implementation of U-Net training on the Spinal Cord Grey Matter dataset, while MohammedHossein AskariHemmat was responsible for binary quantization. Given the constraints imposed by the target hardware and application, a primary goal was to minimize the model size through quantization. As such, binary quantization was initially employed as the most aggressive form of compression. However, it was determined that although binary quantization significantly improved the runtime and performance of the U-Net, it was insufficient in meeting the accuracy requirements. With the assistance of Sina Honari, a comprehensive analysis of the network parameters during training was performed and we found out that binary quantization did not encompass the necessary dynamic range of these parameters. To address this issue, a fixed-point quantization algorithm was proposed and implemented to permit the exploration of different dynamic ranges for the floating component of each parameter. This approach enabled us to attain near-floating-point precision with only 6 bits of quantization for the Spinal Cord Grey Matter dataset. Lucas Rouhier further extended this work by adding support for two new datasets (Electron Microscopy and NIH Pancreas). Throughout the

project, Julien Cohen-Adad, Yvon Savaria, and Jean-Pierre David provided supervision and valuable feedback, which greatly contributed to the final form of the paper.

U-Net Fixed-Point Quantization for Medical Image Segmentation

3.2 Introduction

Image segmentation, the task of specifying the class of each pixel in an image, is one of the active research areas in the medical imaging domain. In particular, image segmentation for biomedical imaging allows identifying different tissues, biomedical structures, and organs from images to help medical doctors diagnose diseases. However, manual image segmentation is a laborious task. Deep learning methods have been used to automate the process and alleviate the burden of segmenting images manually.

The rise of Deep Learning has enabled patients to have direct access to personal health analysis [152]. Health monitoring apps on smartphones are now capable of monitoring medical risk factors. Medical health centers and hospitals are equipped with pre-trained models used in medical CADs to analyse MRI images [153]. However, developing a high precision model often comes with various costs, such as a higher computational burden and a large model size. The latter requires many parameters to be stored in floating point precision, which demands high hardware resources to store and process images at test time. In medical domains, images typically have high resolution and can also be volumetric (the data has a depth in addition to width and height). Quantizing the neural networks can reduce the feedforward computation time and most importantly the memory burden at inference. After quantization, a high precision (floating point) model is approximated with a lower bit resolution model. The goal is to leverage the advantages of the quantization techniques while maintaining the accuracy of the full precision floating point models. Quantized models can then be deployed on devices with limited memory such as cell-phones, or facilitate processing higher resolution images or bigger volumes of 3D data with the same memory budget. Developing such methods can reduce the required memory to save model parameters potentially up to 32x in memory footprint. In addition, the amount of hardware resources (the number of logic gates) required to perform low precision computing, is much less than a full precision model [8]. In this paper, we propose a fixed point quantization of U-Net [30], a popular segmentation architecture in the medical imaging domain. We provide comprehensive quantization results on the Spinal Cord Gray Matter Segmentation Challenge [154], the ISBI challenge for segmentation of neuronal structures in electron microscopic stacks [155], and the public National Institute of Health (NIH) dataset for pancreas segmentation in abdominal CT scans [156]. In summary, this work makes the following contributions:

- We report the first fixed point quantization results on the U-Net architecture for the medical

image segmentation task and show that the current quantization methods available for U-Net are not efficient for the hardware commonly available in the industry.

- We quantify the impact of quantizing the weights and activations on the performance of the U-Net model on three different medical imaging datasets.
- We report results comparable to a full precision segmentation model by using only 6 bits for activation and 4 bits for weights, effectively reducing the weights size by a factor of $8\times$ and the activation size by a factor of $5\times$.

3.3 Related Works

3.3.1 Image Segmentation

Image segmentation is one of the central problems in medical imaging [157], commonly used to detect regions of interest such as tumors. Deep learning approaches have obtained the state-of-the-art results in medical image segmentation [158, 159]. One of the favorite architectures used for image segmentation is U-Net [30] or its equivalent architectures proposed around the same time; ReCombinator Networks [160], SegNet [31], and DeconvNet [161], all proposed to maintain pixel level information that is usually lost due to pooling layers. These models use an encoder-decoder architecture with skip connections, where the information in the encoder path is reintroduced by skip connections in the decoder path. This architecture has proved to be quite successful for many applications that require full image reconstruction while changing the modality of the data, as in the image-to-image translation [162], semantic segmentation [30, 31, 161] or landmark localization [160, 163]. While all the aforementioned models propose the same architecture, for simplicity we refer to them as U-Net models. U-Net type models have been very popular in the medical imaging domain and have been also applied to the 3 dimensional (3D) segmentation task [164]. One problem with U-Net is its high usage of memory due to full image reconstruction. All encoded features are required to be kept in memory and then used while reconstructing the final output. This approach can be quite demanding, especially for high resolution or 3D images. Quantization of weights and activations can reduce the required memory for this model, allowing to process images with a higher resolution or with a bigger 3D volume at test time.

3.3.2 Quantization for Medical Imaging Segmentation

There are two approaches to quantize a neural network, namely deterministic quantization and stochastic quantization [8]. Although DNN quantization has been thoroughly studied [8, 79, 165],

little effort has been done on developing quantization methods for medical image segmentation. In the following, we review recent works in this field.

Quantization in Fully Convolutional Networks: Quantization has been applied to Fully Convolutional Networks (FCN) in biomedical image segmentation [87]. First, a quantization module was added to the suggestive annotation in FCN. In suggestive annotation, instead of using the original dataset, a representative training dataset was used, which in turn increased the accuracy. Next, FCN segmentations were quantized using Incremental Quantization (INQ). Authors report that suggestive annotation with INQ using 7 bits results in accuracy close to or better than those obtained with a full precision model. In FCN, features of different resolutions are upsampled back to the image resolution and merged together right before the final output predictions. This approach is sub-optimal compared to the U-Net which upsamples features only to one higher resolution, allowing the model to process them before they are passed to higher resolution layers. This gradual resolution increase in reconstruction acts as a conditional computation, where the features of higher resolution are computed using the lower resolution features. As reported in [160], this process of conditional computation results in faster convergence time and increased accuracy in the U-Net type architectures compared to the FCN type architectures. Considering the aforementioned advantages of U-Net, in this paper we pursue the quantization of this model.

U-Net Quantization: In [166], the authors propose the first quantization for U-Net. They introduce 1) a parameterized ternary hyperbolic tangent to be used as the activation function, 2) a ternary convolutional method that calculates matrix multiplication very efficiently in the hamming space. They report 15-fold decrease in the memory requirement as well as 10x speed-up at inference compared to the full precision model. Although this method shows significant performance boost, in Section 3.5 we demonstrate that this is not an efficient method for the currently available CPUs and GPUs.

3.4 Proposed Quantization

We propose fixed point quantization for U-Net. We start with a full precision (32 bit floating point) model as our baseline. We then use the following fixed point quantization function to quantize the parameters (weights and activation) in the inference path:

$$quantize(x, n) = (round(clamp(x, n) << n)) >> n \tag{3.1}$$

where the $round$ function projects its input to the nearest integer, $<<$ and $>>$ are shift left and right operators, respectively. In our simulation, shift left and right are implemented by multiplication and division in powers of 2. The $clamp$ function is defined as:

$$clamp(x, n) = \begin{cases} 2^n - 1 & \text{when} \quad x \geq 2^n - 1 \\ x & \text{when} \quad 0 < x < 2^n - 1 \\ 0 & \text{when} \quad x \leq 0 \end{cases} \tag{3.2}$$

Equation (3.1) quantizes an input $x \in \mathbb{R}$ to the closest value that can be represented by n bits. To map any given number x to its fixed point value we first split the number into its fractional and integer parts using:

$$x_f = abs(x) - floor(abs(x)), x_i = floor(abs(x)) \tag{3.3}$$

and then use the following equation to convert x to its fixed point representation using the specified number of bits for the integer ($ibits$) and fractional ($fbits$) parts:

$$\begin{aligned} to\_fixed\_point(x, ibits, fbits) = sign(x) * quantize(x_i, ibits) \\ + sign(x) * quantize(x_f, fbits) \end{aligned} \tag{3.4}$$

Equation (3.4) is a fixed point quantization function that maps a floating point number x to the closest fixed point value with $ibits$ integer and $fbits$ fractional bits. Throughout this paper, we use $Q^p i.f$ notation to denote that we are using a fixed point quantization of parameter p by using i bits to represent the integer part and f bits to represent the fractional part. Based on our experiments, we did not benefit from an incremental quantization (INQ) as explained in [79]. Although this method could work for higher precision models, for instance when using fixed point $Q^w 8.8$ (Quantizing weights with 8-bit integer and 8-bit fractional parts), for extreme quantization as in $Q^w 0.4$, learning from scratch gave us the best accuracy with the shortest learning time. As shown in Figure A.1, in the full precision case, the weights of all U-Net layers are in $[-1, 1]$ range, hence the integer part for the weight quantization is not required.

3.4.1 Training

For numerical stability and to verify the gradients can propagate in training, we demonstrate that our quantization is differentiable . Starting from Equation (3.2), the derivative is:

$$\forall x \in \mathbb{R}, \forall n \in \mathbb{Z}^+, \quad \frac{\partial}{\partial x} clamp(x, n) = \begin{cases} 0 & \text{when} \quad x \geq 2^n - 1 \\ 1 & \text{when} \quad 0 < x < 2^n - 1 \\ 0 & \text{when} \quad x \leq 0 \end{cases} \tag{3.5}$$

which is differentiable except on the thresholds. To make it completely differentiable, a straight-through estimator (STE), introduced in [167], is used. The STE passes gradients over the thresholds and also over the $round$ function in Equation (3.1).

3.4.2 Observations on U-Net Quantization

Dropout

Dropout [168] is a regularization technique to prevent over-fitting of DNNs. Although it is used in the original implementation of U-Net, we found that when this technique is applied along with quantization, the accuracy drops a lot. Hence, in our implementation, we removed dropout from all layers. This is due to the fact that quantization acts as a strong regularizer, as reported in [8], hence further regularization with dropout is not required. As shown in Figure A.2, for each quantized precision, dropout reduces the accuracy, with the gap being even higher for lower precision quantizations.

Full Precision Layers

It is well accepted to keep the first and the last layers in full precision, when applying quantization [8, 169]. However, we found that in the segmentation task, keeping the last layer in full precision has much more impact than keeping the first layer in full precision.

Batch Normalization

Batch normalization is a technique that improves the training speed and accuracy of DNN. We used the Pytorch implementation of batchnorm. In training, we use the quantization block after the batchnorm block in each layer such that the batchnorm is first applied using the floating point calculations and then the quantized value is sent to the next layer (hence not quantizing the batchnorm block during training). However, at inference, Pytorch folds the batchnorm parameters into the weights, effectively including batchnorm parameters in the quantized model as part of the quantized weights.

3.5 Results and Discussion

We implemented the U-Net model and our fixed-point quantizer in Pytorch. We trained our model over 200 epochs with a batch size of 4. We applied our fixed point quantization along with TernaryNet [166] and Binary [165] quantization on three different datasets: GM [154], EM [155], and NIH [156]. For GM and EM datasets, we used an initial learning rate of $1e-3$, and for NIH

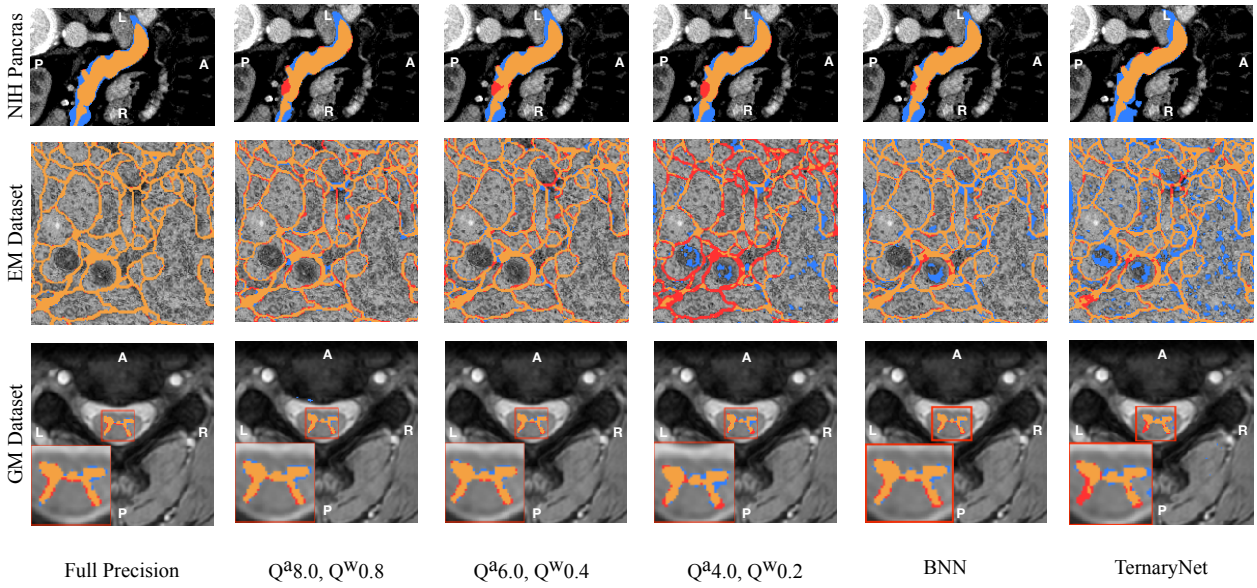


Figure 3.1 Sample prediction versus ground truth segmentation results for NIH Pancreas (top), EM (middle) and GM (bottom) datasets. From left to right, the result of different quantization methods and precisions are reported. Segments in ■ show false positive, segments in ■ show false negative and segments in ■ show true positive

we used initial learning rate of 0.0025. For all datasets we used Glorot for weight initialization and cosine annealing scheduler to reduce learning rate in training. Please check our repository for the model and training details. [1]

The NIH pancreas [156] dataset is composed of 82 3D abdominal CT scan and their corresponding pancreas segmentation images. Unfortunately, we did not have access to the pre-processed dataset described in [166], nevertheless, we extracted 512x512 2-D slices from the original dataset and applied a region of interest cropping to get 7059 images of size 176x112. The final dataset contains 7059 176x112 2-D images which are separated into training and testing dataset (respectively 80% and 20%). For GM and EM datasets, we used the provided dataset as described in [154] and [155] respectively. For both EM and GM datasets, we did not used any region of interest cropping and we used images of size 200x200.

The task of image segmentation for GM and NIH pancreas datasets is imbalanced. As suggested in [154], instead of weighted cross-entropy, we used a surrogate loss for the dice similarity coefficient. This loss is referred to as the dice loss and is formulated as $\mathcal{L}_{dice} = \frac{2\sum_{n=1}^{N} p_n r_n + \epsilon}{\sum_{n=1}^{N} p_n + \sum_{n=1}^{N} r_n + \epsilon}$, where $p_n \in [0, 1]$ and $r_n \in \{0, 1\}$ are prediction and ground truth pixels respectively (with 0 indicating

[1] https://github.com/hossein1387/U-Net-Fixed-Point-Quantization-for-Medical-Image-Segmentation

not-belonging and 1 indicating *belonging* to the class of interest) and ϵ is the noise added for numerical stability. For the EM dataset, using a weighted sum of cross entropy and dice loss produced the best results.

Figure 3.1 along with Table 3.1 show the impact of different quantization methods on the aforementioned datasets. Considering the NIH dataset, Figure 3.1(top) and Table 3.1 show that despite using only 1 and 2 bits to represent network parameters, Binary and TernaryNet quantizations produce results that are close to the full precision model. However, for other datasets, our fixed point Q^a6.0, Q^w0.4 quantization surpasses Binary and TernaryNet quantization. The other important factor here is how efficient these quantization techniques can be implemented using the current CPUs and GPUs hardware. At the time of writing this paper, there is no commercially available CPU or GPU that can efficiently store and load sub-8-bit parameters of a neural network, which leaves us to use custom functions to do bit manipulation to make sub-8-bit quantization more efficient. Moreover, in the case of TernaryNet, to apply floating point scaling factors after ternary convolutions, floating point operations are required. Our fixed point quantization uses only integer operations, which requires less hardware footprint and use less power compared to floating point operations. Finally, TernaryNet uses Tanh instead of ReLU for the activations. Using hyperbolic tangent as an activation function increases training time [35] and execution time at inference. To verify it, we evaluated the performance of ReLU and Tanh in a simple neural network with 3 fully connected layers. We used the Intel's OpenVino [25] inference engine together with high performance `gemm_blas` and `avx2` instructions. Table 3.2 reports the results obtained when ReLU is used instead of Tanh at training and it shows that inference time can decrease by up to 8 times. These results can be extended to U-Net, since activation inference time is a direct function of the input size. To compensate for the computation time, TernaryNet implements an efficient ternary convolution that can decrease processing time by up to 8 times. At inference, an efficient Tanh function that uses only two comparators can be implemented to perform Tanh for ternary values. Considering accuracy, when Tanh is used as an activation function, the full precision accuracy is lower compared to ReLU [166]. We observe similar behavior in the results reported in Table 3.1. Our fixed point quantizer provides a flexible trade-off between accuracy and memory, which makes it a practical solution for the current CPUs and GPUs, as it does not require floating-point operations, and leverages the more efficient ReLU function. As opposed to BNN and TernaryNet quantizations, Table 3.1 shows that our approach for quantization of U-Net provides consistent results over 3 different datasets.

3.6 Power Efficiency

The main goal of quantization is to reduce calculation precision while maintaining accuracy. One might wonder why this is important. An insight can be found in tables 2.2 and 2.3 reported

Table 3.1 Dice scores of the quantized U-Net models on EM (left) GM (middle) and NIH (right) datasets. The last two rows show results for Binary and TernaryNet quantizations. Other rows report results obtained for different weights and activations quantization precisions. For the GM and EM datasets, we also report results when Tanh is used instead of ReLU as the activation function.

Quantization			EM Dataset		GM Dataset		NIH Panceas
Activation	Weight	Parameter Size	Dice Score ReLU	Dice Score Tanh	Dice Score ReLU	Dice Score Tanh	Dice Score
Full Precision		18.48 MBytes	94.05	93.02	56.32	56.26	75.69
Q8.8	Q8.8	9.23 MBytes	92.02	91.08	56.11	56.01	74.61
Q8.0	Q0.8	4.61 MBytes	92.21	88.42	56.10	53.78	73.05
Q6.0	Q0.4	2.31 MBytes	91.03	90.93	55.85	52.34	73.48
Q4.0	Q0.2	1.15 MBytes	79.80	54.23	51.80	48.23	71.77
BNN [165]		0.56 MBytes	78.53	-	31.44	-	72.56
TernaryNet [166]		1.15 MBytes	-	82.66	-	43.02	73.9

in [170]. They show the power consumption in a typical hardware implementation when performing multiplication, accumulation, and memory access. The power consumption increases with the number of bits used to perform most operations. It can be observed that processing with integers is considerably more power-efficient than using floating points. A hardware that is designed to perform integer operations can perform fixed point operations without any modification. In our implementation of the U-Net, for example for the first convolutional layer, the input and output are tensors of shape `[16, 1, 200, 200]` and `[16,64,200,200]`respectively, and we used a kernel size of 3. This means that for this layer alone, we performed about 360 million multiplications. Our quantization method can potentially save 1.3 milli (10^{-3}) joule just by quantizing the first layer to a 8 bit fixed point model.

3.7 Conclusion

In this work, we proposed a fixed point quantization method for the U-Net architecture and evaluated it on the medical image segmentation task. We reported quantization results on three different segmentation datasets and showed that our fixed point quantization produces more accurate and also more consistent results over all these datasets compared to other quantization techniques. We also

Table 3.2 Comparing ReLU and Tanh run time using Intel's OpenVino [25]. Each row illustrates the execution time for a layer of a neural network in micro seconds. It demonstrates that using Tanh as activation can increase execution time by up to 8 times compared to ReLU.

Layer Type	Instruction Type	Execution time in μs **Tanh**	Execution time in μs **ReLU**	Performance Gain of using ReLU over Tanh	Tensor Dimension
`Activation`	`jit_avx2_FP32`	`30`	`5`	`6`	`[100, 100]`
`FullyConnected`	`gemm_blas_FP32`	`20`	`19`	-	-
`FullyConnected`	`gemm_blas_FP32`	`860`	`527`	-	-
`Activation`	`jit_avx2_FP32`	`77`	`9`	`8.6`	`[100, 300]`

demonstrated that Tanh, as the activation function, reduces the base-line accuracy and also adds a computational complexity in both training and inference. Our proposed fixed-point quantization technique provides a trade-off between accuracy and the required memory. It does not require floating-point computation and it is more suitable for the currently available CPUs and GPUs hardware.

CHAPTER 4 ARTICLE 2 : QREG: ON REGULARIZATION EFFECTS OF QUANTIZATION

4.1 Prologue

4.1.1 Article Details

QReg: On Regularization Effects of Quantization
MohammadHossein AskariHemmat, Reyhane AskariHemmat, Alex Hoffman, Ivan Lazarevich, Ehsan Saboori, Olivier Mastropietro, Sudhakar Sah, Yvon Savaria, Jean-Pierre David.
The first version of this work was peer reviewed and published at Hardware Aware Efficient Training (HAET) workshop at the thirty-ninth International Conference on Machine Learning (ICML), July 23 2022.

4.1.2 Context

Previously, the impact of quantization on model generalization has been noted in various experiments where quantized models have achieved higher accuracy compared to their baseline counterparts. Our study, "U-Net Fixed-Point Quantization for Medical Image Segmentation" [19], also found that quantization can act as a regularizer, resulting in improved generalization for 6 and 8-bit quantized models.

Despite this observation, there was limited research available that fully examines the effect of quantization on model generalization. Our work aims to fill this gap by investigating how different levels of quantization impact generalization in various machine learning tasks, across different models and datasets. Our research involves a examination of the impact of quantization on generalization across a diverse set of tasks and models. Through empirical studies, we show that quantization can be a valuable technique for improving generalization performance. This technique can be applied to various machine learning tasks, such as image classification, object detection, and speech recognition, among others. Our findings indicate that quantization can help reduce overfitting and improve the model's ability to generalize to unseen data, thereby leading to better performance on real-world tasks.

4.1.3 Contributions

Previously in the literature, the regularization effect of quantization has been explored, but the specific level of quantization that provides regularization was unclear. Previous studies have also only explored a limited number of machine learning tasks and models. In this work, we present an extensive empirical study to determine the impact of different levels of quantization on training and generalization. Our findings demonstrate that 8-bit quantization offers a reliable form of regularization across various vision and language processing tasks and models.

Personal Contributions MohamamdHossein AskariHemmat contributed to the original idea of this work and writing of the paper. MohamamdHossein AskariHemmat first presented the idea of using quantization as a regularizer in image segmentation and classification tasks. However, we found that for ultra low precision (under 4 bit) quantized models, this phenomenon is not as effective as 8 bit quantized models. We then re-formulated our assumption by trying to study the effect of quantization on model training and generalization. With the help from Reyhaneh Askarihemmat, we tried to model quantization as noise. We found out that during training, quantization noise appears in the loss function. To find out in what capacity this noise is affecting training and generalization, with the help from Alex Hoffman, Ivan Lazarevich, Ehsan Saboori, Olivier Mastropietro and Sudhakar Sah, we started a training setup on more than 100 models, quantization levels and dataset combination distributed over different machine learning tasks (segmentation, classification, object detection and natural language processing). Yvon Savaria and Jean-Pierre David supervised the project and provided valuable feedback forming the paper.

QReg: On Regularization Effects of Quantization

4.2 Introduction

Deep neural network (DNN) quantization has been an active research topic in the past few years. Quantization is the task of approximating high precision (floating point) weights and activations of a model with lower bit resolution counterparts (usually 1 to 8 bit integers). Quantization can be done after (post training quantization) or during training (quantization aware training). Although both methods can compress the model and reduce overall execution time, at the cost of model accuracy, post training quantization usually takes less time to quantize a model. On the other hand, quantization aware training usually requires a full training session and a moderate quantization level (int8) can produce quantized models with little to no accuracy loss compared to their full precision counterparts. In fact, as we will discuss in Section 4.3, there are many existing works claiming that quantization aware training produced accuracy improvements compared to some of the best full precision models. In these studies, the authors usually attribute such accuracy gains to the regularization effect of quantization. This has motivated us to explore how quantization affects training. In this work, we first hypothesize the regularization effect of quantization. Moreover, we hypothesize that this regularization effect is correlated with the quantization level. To confirm our hypothesis, we analytically explore how quantization noise propagates in a network at training time, and how the quantization level is correlated with the amount of regularization. We then empirically explore the regularization effect of quantization by applying different quantization aware training methods on different models and datasets with different quantization levels. In both analytical and empirical studies, we confirm our initial hypothesis.

4.3 Background

The regularization effect of quantization has previously been reported in the literature. We have categorized earlier works into three main lines of work, which we will review briefly in the following subsections.

4.3.1 Effect of Quantization on Accuracy Improvements

In [171], one of the first quantization methods for DNNs, the authors empirically show that modern deep learning optimization techniques such as stochastic gradient descent are compatible with noisy

weights, as they appear in quantization. They also empirically show that noisy weights provide a form of regularization. They confirm their hypothesis by achieving state-of-the-art results on MNIST, CIFAR-10 and SVHN datasets. Similar to this work, authors in [89], argue that weight quantization is a form of noise injection to weights and as shown in DropConnect [95], this noise injection can improve the generalization performance. However, both studies are only limited to small-scale classification tasks. Their studies are also limited to one quantization technique. Furthermore, their hypotheses lack analytical study.

In [88] the authors study the effect of quantization on accuracy and provide an analysis of the regularization effect of quantization. Their study is limited to showing a generalization gap between a quantized model with 8 and 4 bit precision and full precision models on the Imagenet dataset on ResNet50. On the other hand, in addition to their work, they reported the generalization gap of different quantization methods. However, in their experiments, the regularization effect of quantization is not investigated. Furthermore, they did not explain whether this generalization gap is coming from quantization or whether they have used other regularization techniques that might have affected the overall results.

Apart from the papers above, there are many other works reporting accuracy improvement after quantization, [87, 90–93, 172–174]. However, to the best of our knowledge, none of them provided analytical or empirical studies and the authors justified the boost in performance (accuracy) with a conjectured regularization effect of quantization.

4.3.2 Analytical Studies

As presented in 4.3.1, the regularization effect of quantization has been observed in previous works. In the following, we present three papers that provide an analytical study on how quantization affects training.

In [175], the authors claim that applying a constraint on the weights can act as a regularizer. They investigate how DNNs are robust to different types of weight distortion. As an example, for quantization, they studied the effect of binarization proposed in [171] on training. Using proximal methods, the authors showed that applying weight clipping during training can be modeled as a type of regularization that penalizes weights outside the unit ball of the $\ell_\propto$ norm. Their study is limited to applying binarization on weights and they do not explain how different levels of weight quantization affect training. Moreover, like [171] , they only investigate the regularization effect of weight distortion on small models.

A more recent study in [94] provides an analytical overview on how quantization can affect robustness. Although not directly relevant to our work, the authors in this paper provide an analytical study on how

quantization affects training and the loss function. In this paper, the authors propose a regularization based quantization method that, unlike most quantization-aware methods, provides a selectable quantization bit-width after training. Like previous methods, the authors model quantization as a form of noise. Using first order Taylor approximation, they then provide an analytical study on how this noise affects the robustness of DNNs. With their method, they show that if the ℓ_1-norm of the gradient is small, the perturbation (at least to the first-order approximation) can be effectively controlled for various quantization bit-widths. To guarantee the ℓ_1-norm of the gradients, they propose a regularization term in the loss function. As we will show in Section 4.4, we used similar quantization noise modeling and approximation techniques to investigate the effect of quantization on the loss function.

In [97], the authors provide an analytical study on how models with stochastic binary quantization can have a smaller generalization gap compared to their full precision counterparts. In this work, the authors propose a so-called "quasi neural network" to approximate the effect of binarization on neural networks. The authors then derive the neural tangent kernel for the proposed quasi neural network approximation. With this formalization, the authors show that binary neural networks have lower capacity, hence lower training accuracy, but also a smaller generalization gap than their full precision counterparts. Finally, the authors provide some empirical results to support their derivations. However, the authors fail to show the effect of different quantization levels on the generalization gap. As mentioned earlier, the authors only focused their analysis on statistical binary quantization and they did not explore how different quantization methods and precision levels affect generalization. Also, their empirical results are only limited to a two layer model.

4.3.3 Using Quantization for its Regularization Effect

Although very limited, the idea of using quantization solely for its regularization effect has been explored before. In [176], the authors studied the effect of binarization on the loss function. They then proposed an algorithm that uses diagonal Hessian approximation that directly minimizes the loss function with regard to the binarized weights. For training their binarized model, the authors claimed that they only used regularization that is induced by weight binarization. Furthermore they further developed their method in [95] where they show that the same techniques can be applied to weight ternerization.

4.4 Quantization Noise as a Regularizer

The authors in the previous works usually assume that weight quantization is similar to adding noise to a model's weights. However, it is not clear how this noise propagates through the network

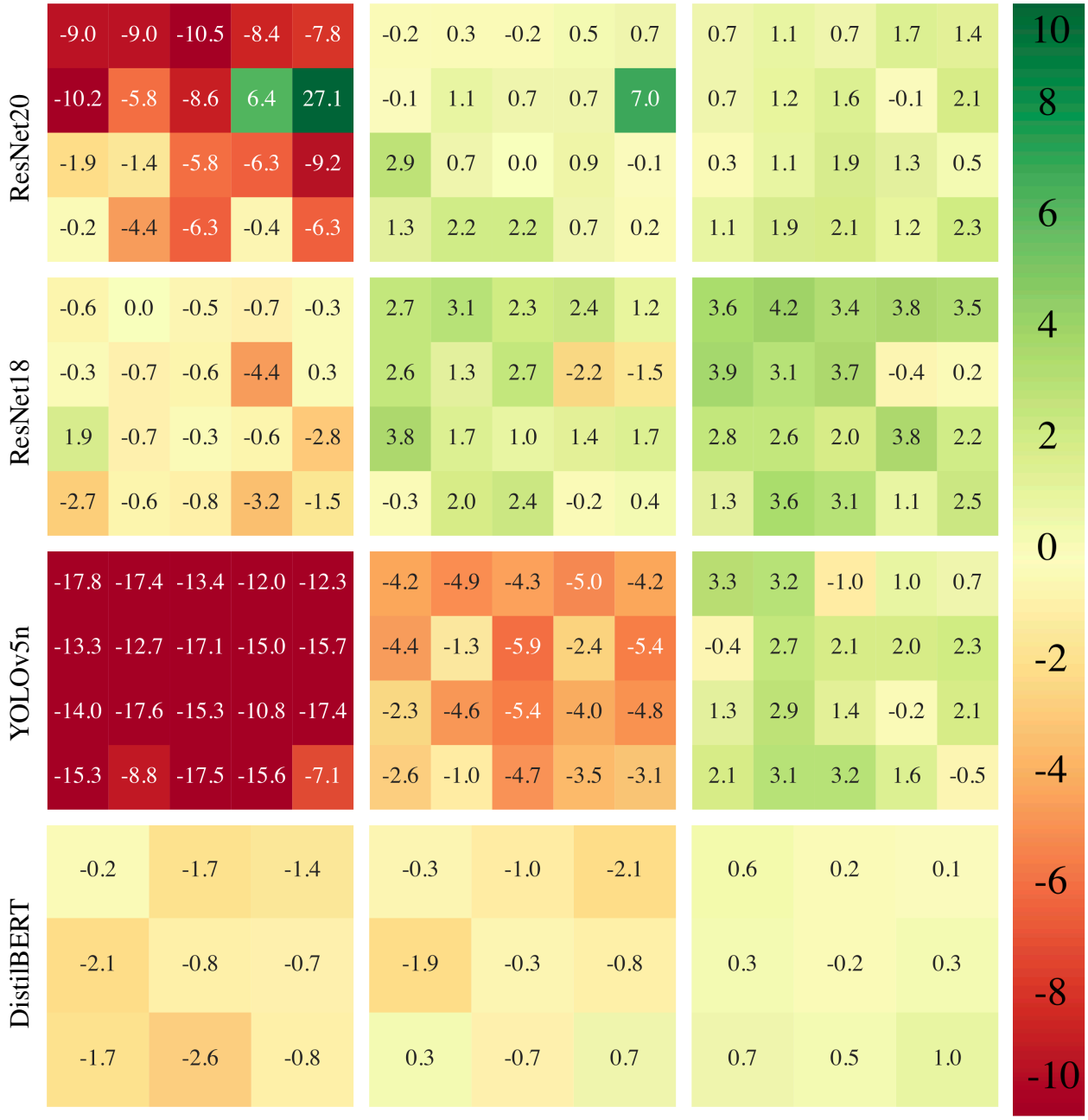


Figure 4.1 Regularization effect of quantization over different quantization level (2 bits, 4 bits and 8 bits), and models. The value in each cell shows the test accuracy difference between the quantized and full precision models. The position of each cell shows the corresponding augmentation in Figure B.1(a) that was applied on test data. Cells are colored by generalization difference, where green cells ■ show quantized model generalized better compared to floating point model and the red cells ■ show the opposite. The color intensity shows the amplitude of the difference according to the color bar shown on the right

at training time. Also, it is not clear how different quantization levels and algorithms affect the magnitude of this noise. In this section, we will show how quantization noise can be modeled as a regularization term. We will then extend our study and show that this noise is directly related to the quantization level.

For simplicity, let us consider a regression problem where the mean square error loss is defined as,

$$\mathcal{L} = \frac{1}{m}\sum_{i=1}^{m} \|\hat{y}_i - y_i\|_2^2, \tag{4.1}$$

where y_i is the target, and $\hat{y}_i = f(x, w)$ is the predicted target of the network f parameterized by w. Weight quantization can be characterized as noise perturbing the weights and activations. Such noise can be described as:

$$f(x, w_q) = f(x, w + \delta) \tag{4.2}$$

Thus, a quantized neural network, effectively has the following loss,

$$\tilde{\mathcal{L}} = \frac{1}{m}\sum_{i=1}^{m} \|\hat{y_i}^q - y_i\|_2^2, \tag{4.3}$$

where $\hat{y_i}^q = f(x, w + \delta)$.

Theorem 4.4.1. *Given the mean square error loss defined in Eq. 4.3, assuming that the quantization noise follows a normal distribution, $\delta \sim \mathcal{N}(0, \sigma I)$, applying a first-order Taylor approximation around the weights of the full precision model, results in the following approximation on the $\tilde{\mathcal{L}}$:*

$$\tilde{\mathcal{L}} \approx \mathcal{L} + \frac{\sigma\delta^2}{m}\sum_{i=1}^{m} \|\nabla_w \hat{y}_i\|_2^2, \tag{4.4}$$

This means that minimizing $\tilde{\mathcal{L}}$ is equivalent to minimizing the loss of a non-quantized neural network with gradient norm regularization.

4.4.1 Derivation of Theorem 4.4.1

From Eq. 4.3, we have,

$$\begin{aligned}\tilde{\mathcal{L}} &= \frac{1}{m}\sum_{i=1}^{m}\left[\|y_i\|_2^2 + \|\hat{y_i}^q\|_2^2 - 2y_i\hat{y_i}^q\right] \\ &= \mathbb{E}_{p(x,y,\delta)}\left[\|y_i\|_2^2 + \|\hat{y_i}^q\|_2^2 - 2y_i\hat{y_i}^q\right]\end{aligned}$$

Table 4.1 Regularization effect of quantization measured with relative error improvement. For each model, we measured the test accuracy of quantized (2-bit, 4-bit and 8-bit) and full precision (FP32) models when 19 different data augmentation methods (presented in B.1(a)) are applied on the original test dataset. We then averaged the results (third column). For each case (row), we used Equation 4.5 to measure the relative test error improvement. Green cells ■ in the last column show that quantized models have less error compared to the full precision model. The red cells ■ show the opposite.

	Quantization Level	Avg Accuracy Augmented Data	Relative Error Improvement Eq.4.5
ResNet20 CIFAR10	2 bits	79.17	-8.32
	4 bits	83.94	2.98
	8 bits	84.07	3.33
	FP32	82.80	0.00
ResNet18 CIFAR100	2 bits	49.43	-0.84
	4 bits	51.76	1.21
	8 bits	53.05	2.39
	FP32	50.40	0.00
YOLOv5n VOC	2 bits	14.66	-7.86
	4 bits	24.90	-2.31
	8 bits	30.34	0.96
	FP32	28.78	0.00

Also, since $\hat{y_i}^q = f(x, w + \delta)$, we can apply a first order Taylor approximation,

$$f(x, w + \delta) = f(x, w) + \delta \nabla_w f(x, w) + \mathcal{O}(\delta^2)$$

Thus, $\hat{y_i}^q \approx \hat{y_i} + \delta \nabla_w \hat{y_i}$.

Re-writing, the expectation on $\tilde{\mathcal{L}}$,

$$\begin{aligned}
\tilde{\mathcal{L}} &= \mathbb{E}_{p(x,y,\delta)} \left[\|y_i\|_2^2 + \|\hat{y_i}^q\|_2^2 - 2 y_i \hat{y_i}^q \right] \\
&\approx \mathbb{E}_{p(x,y,\delta)} [\|y_i\|_2^2 + \|\hat{y_i} + \delta \nabla_w \hat{y_i}\|_2^2 \\
&\quad - 2 y_i (\hat{y_i} + \delta \nabla_w \hat{y_i})] \\
&= \mathbb{E}_{p(x,y,\delta)} [\|y_i\|_2^2 + \|\hat{y_i}\|_2^2 - 2 y_i \hat{y_i} + \delta^2 \|\nabla_w \hat{y_i}\|_2^2 \\
&\quad + 2\delta (\nabla_w \hat{y_i})(\hat{y} - y)] \\
&= \mathcal{L} + \sigma \delta^2 \mathbb{E}_{p(x,y,\delta)} [\|\nabla_w \hat{y_i}\|_2^2]
\end{aligned}$$

Note that since we assume that $\delta \sim \mathcal{N}(0, \sigma I)$, we have $\mathbb{E}_{p(x,y,\delta)}[2\delta(\nabla_w \hat{y_i})(\hat{y} - y)] = 0$.

Direct application of noise on the weights as a form of regularization has been studied in the

literature. In [177] authors argue that noisy weights result in forcing the model to find minima that are insensitive to small perturbations of the weights. In other words, noisy weights encourage finding minima in flat regions and thus lead to better generalization [178].

4.5 Experimental Studies

In this section, we will describe empirically how quantization has a regularization effect. Unlike previous works that was described in Section 4.3, we tried to explore the regularization effect of quantization over different quantization techniques, levels, models, vision tasks and datasets. In the following subsections, we briefly describe our test setup and results.

4.5.1 Experiment Setup

Training Setup

As we described in Section 4.4 we hypothesize that quantization has a regularization effect that is correlated with quantization level (precision). In order to test this hypothesis, we needed a quantization technique that could be used to quantize a model to any quantization precision. For our experiments, we picked Learned Step size Quantization (LSQ) [2] and Parameterized Clipping Activation (PACT) [96]. Since PACT applies quantization only to activations, we used DoReFa [179] quantization for weights. For PACT, DoReFa and LSQ, we used learning rate of 0.1 with momentum of 0.9 for the classification experiment. In our experiments, we initially used quantization as the only regularizer in the network. However, both full precision and quantized models were unable to achieve a good accuracy. Hence, we used weight decay of 0.0001 for both full precision and quantized models in our experiments. We trained both full precision and quantized models for 200 epochs. We used a multi-step learning scheduler with milestones of 50, 90, 130 and 180 with gamma of 0.2. For the YOLO experiments, we have used the standard training setup for training from scratch provided in the YOLOv5 repository [180]. We did not use weights pretrained on the COCO dataset, but rather trained on the VOC dataset from scratch with the standard set of hyperparameters with the exception of data augmentations that have been disabled during training. For the quantized model, we did not quantize the first convolutional layer of the model as well as the last 3 convolutional layers in the detection head. For the NLP experiments, we used a standard BERT fine-tuning setup [181] where we fine-tuned the models for 3 epochs with a learning rate of 2e-5, a batch size of 32 and the maximal sequence length of 128. For data augmentation in the NLP tasks, we used text augmentations provided by the TextAttack framework [182]. For more details on the set of text augmentations we used in our experiments, please refer to B. For DistilBERT quantization, we applied LSQ to weights of all the matrix multiplication operations in the network (including the

multi-head self-attention layers), except for the last two fully-connected layers used for classification.

Models and Datasets

In our experiments, we explored the regularization effect of quantization in different vision tasks, models, quantization levels and datasets. We explored two popular vision tasks, object detection and classification and a natural language processing (NLP) task. For small-scale classification tasks, we tested our hypothesis on CIFAR10 dataset and used ResNet20, MobileNetV1, ResNet18 and ResNet50. For bigger classification tasks we used CIFAR100 dataset and used MobileNetV1, ResNet18, ResNet50. For object detection, we used VOC07+12 dataset and tested our hypothesis on YOLOv5n model taken from the YOLOv5 PyTorch repository [180]. The full list of experiments is available in Appendix B. For each experiment, we trained the model as described in Section 4.5.1. To measure the generalization improvement, in addition to the original test data, we applied 19 different augmentations on the original dataset as provided in [183]. For the NLP tasks, we used a DistilBERT-base model [184] implemented in the HuggingFace-transformers repository [185] and fine-tuned it on the 3 tasks from the GLUE benchmark: SST-2, MNLI and CoLA. The evaluation metrics for SST-2 and MNLI are accuracy and Matthews correlation coefficient for CoLA. We have applied 9 different TextAttack augmentations to the validation sets of the tasks we as provided in Appendix B. We then tested each full precision and quantized model with the aforementioned datasets. To measure generalization improvement, we calculated the test accuracy difference between the full precision model and the quantized model when an augmentation is applied on the corresponding test dataset.

Results

Figure 4.1 illustrates part of the experiments that we ran to test our hypothesis and shows results of twelve different test configuration in twelve squares. Each square consists of 20 cells for classification and 9 for NLP tasks which, as we discussed in Section 4.5.1, correspond to the test accuracy difference between full precision model and quantized models when different augmentation is used. From left to right, each column shows the quantization levels (2-bit, 4-bit and 8-bit) that was used for our tests. From top to bottom, we have tested our hypothesis on ResNet20 on CIFAR10, ResNet18 on CIFAR100, YOLOv5n on VOC and DistilBERT-base on SST-2 dataset respectively. As it can be seen (i.e. the overall cell colors in the right column squares are green), regardless of model, dataset, augmentation or quantization technique, 8-bit quantized models generally have better generalization compared to full precision models. As we discussed in Section 4.3.3, we believe that this performance improvement corresponds to the regularization effect of quantizing the models. As we predicted, this regularization effect is correlated with the quantization level. Moving from left

to right columns, the overall color codes for each cell changes from red to green. This indicates that quantized models are generalizing better compared to their full precision counterparts as we use higher precision quantized models (from 2-bit to 8-bit). Regardless of the model, dataset, data augmentation or quantization technique, the generalization difference shrinks and even get worse as we use less precision for quantization. Another way to evaluate how quantization is helping with better generalization, is to compare the error of quantized and full precision models. Table 4.1 shows the performance improvement relative to model error. We averaged the accuracy of each square in Figure 4.1. We then calculated the performance improvement relative to error using the following formula:

$$ErrorImprovement = log\Big(\frac{100 - fval}{100 - qval}\Big) \times 100\% \tag{4.5}$$

Where $fval$ and $qval$ are the average accuracy of full precision and quantized model over 19 different augmentation setups. Since models on small datasets like CIFAR10 generally have small errors compared to models on more complex datasets (like CIFAR100 or VOC), we used a logarithm in Equation 4.5. Once again, according to Table 4.1, in all cases 8-bit quantized models reduce the error more (i.e, generalizing better) compared to full precision models. On all three models tested over 19 different augmentation setups, all 8-bit quantized models reduce the error when compared to full precision models (i.e last column for 8-bit quantization levels is green for all models). This effect (reducing error) is less clear as we use lower precision models .

To find if int8 quantization does in fact provide the best generalization effect, we performed another experiment. Figure 4.3 illustrates the generalization effect of quantization on ResNet18 tested on CIFAR10 and CIFAR100. We performed the same experiment as discussed before, however, in this experiment, we gradually moved from higher to lower precision. For each quantization step, we measured the generalization improvement, when test and augmented data was used. As it can be seen, from int32 to int8, the generalization improvement gradually increments. However, after int8 quantization, we no longer benefit from the induced regularization. This experiment is again inline with our previous empirical studies.

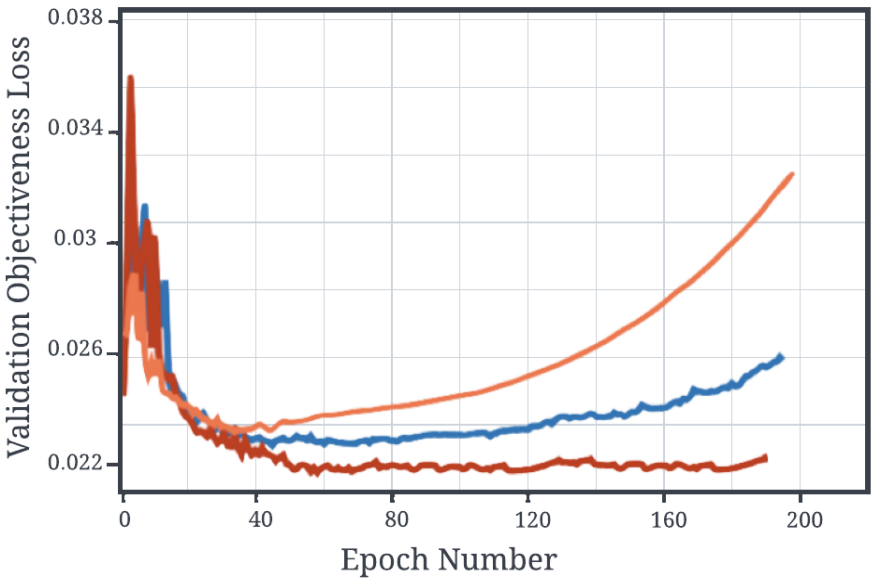


Figure 4.2 Validation objectness loss of 4-bit quantized ■, 8-bit quantized ■ and full precision model ■ for the YOLOv5n model on the VOC dataset. This figure illustrates that unlike the quantized model, the full precision model exhibits heavy overfitting. We believe that the quantized model has better generalization performance due to its regularization effect.

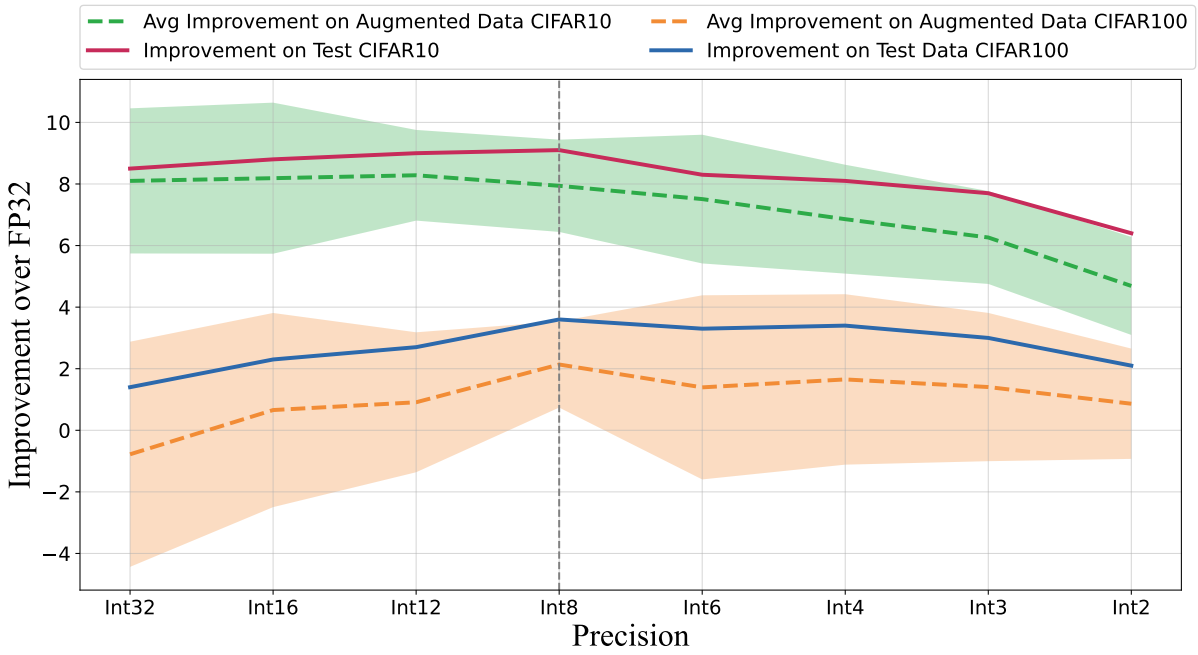


Figure 4.3 Generalization effect of quantization on ResNet18 on CIFAR100 and CIFAR10. Here we are testing the generalization effect of quantization by measuring the improvement of accuracy of quantized models compared to full precision. This figure shows the average improvement of augmented (19 different augmentation) and test data of ResNet18 on CIFAR100 and CIFAR10 over full precision model when different quantization levels are used. It shows that moderate weight quantization (int8) has the best regularization effect over other levels of quantization for both datasets

Our results presented in Figure 4.1, Table 4.1 and Appendix B empirically confirm our hypothesis. Unlike [171], we believe that the regularization effect of quantization is in fact correlated with the quantization level. Our empirical study shows that moderate quantization (8-bit) generally helps models generalize better. In addition to these results, specially for more complex models, we observe that quantized models overfit less at training time. Figure 4.2 illustrates the validation loss of 8-bit, 4-bit quantized and full precision for the YOLOv5n model applied to the VOC dataset. Due to regularization effect of quantization, the quantized models overfit much less compared to full precision model. This, confirms our hypothesis. For the full list of experiments, please refer to Appendix B.

4.6 Conclusion

In this paper, we explored the regularization effect of quantization. We hypothesized that this regularization effect is correlated with the quantization level. To confirm our hypothesis, we provided an analytical study as well as an extensive list of experiments. Our experiments show that the regularization effect of quantization exists regardless of model, dataset, vision task and quantization technique. Finally, we analytically and empirically confirm that this regularization effect is correlated with quantization level. Our empirical study shows that moderate quantization (8-bit) helps models generalize better.

CHAPTER 5 ARTICLE 3 : RISC-V BARREL PROCESSOR FOR DEEP NEURAL NETWORK ACCELERATION

5.1 Prologue

5.1.1 Article Details

RISC-V Barrel Processor for Deep Neural Network Acceleration
MohammadHossein AskariHemmat, Olexa Bilaniuk, Sean Wagner, Yvon Savaria, Jean-Pierre David
The first version of this work was published at Field-Programmable Custom Computing Machines (FCCM) conference 2020. The final version of this work was published at 56th International Symposium on Circuits and Systems (ISCAS) conference, May 28 2021.

5.1.2 Context

To run neural networks on hardware accelerators, some form of software control is required. This control mechanism can be as simple as a state machine. However, due to the fast growing nature of DNN models, one cannot forecast all the upcoming new architectures. On the other hand, sequencing the computation based on the input DNN model can be done at compile time, reducing the need for a high performance modern processor. At runtime, the processing elements are more heavily occupied than the controller. Hence, a small controller with fine grain controllability over each processing element is a desirable solution. PITO is a small processor that allows fine grain control over each MVU. PITO achieves this by providing software programmability over each MVU within the corresponding HART. On the software side, the computation of each layer can be divided among all HARTs, or it can be partitioned into different stages and run on a corresponding HART in a pipelined fashion. Thread communication can be performed using shared memory. In Chapter 6, we will further describe the over all architecture of an accelerator based on array of MVUs and PITO.

5.1.3 Contributions

The idea of barrel processing has been explored before. Most notable in Tera MTA [186] and its Cray MTA-based successors, and the Sun UltraSPARC T1/Niagara processor [187]. However, by the time of developing our idea around barrel processing, to the best of our knowledge, there was no barrel processing based on RISC-V ISA. We presented PITO in FCCM 2020 [188] and later in

ISCAS 2021 [21] with more detailed design and implementation. Later in 2021, Klessydra-T [189] introduced a vector co-processor for multi-threaded edge computing. Compared to PITO, Klessydra-T is a much bigger design. Unlike PITO, Klessydra-T is proposed to accelerate computation on its multi-threaded cores. As described earlier, the intention behind designing PITO was to make a small processor that is capable of controlling an array of processing elements. Furthermore, since PITO leverages thread-level parallelism by switching between different threads every clock cycle, running a computationally intensive task on each HART is very inefficient. Finally, PITO is designed as an open source project and is available at `https://github.com/hossein1387/pito_riscv`.

Personal Contributions: In their paper [98], Olexa Bilaniuk and Sean Wagner proposed Matrix Vector Units (MVUs) as a processing element that employs bit-slicing to run quantized neural networks. However, it lacked the necessary infrastructure for deploying modern deep neural networks. As a crucial part of the architecture, Olexa Bilaniuk proposed the usage of barrel processing to control the array of MVUs. MohammadHossein AskariHemmat implemented the first (at the time of submitting our first paper to FCCM 2020 [188]) barrel processor in RISC-V ISA called PITO. PITO is a RISC-V processor that has 8 HARdware Threads (HARTs) and supports the base RV32I ISA plus part of the privileged mode (CSRs and IRQs). MohammadHossein AskariHemmat designed and verified PITO using SystemVerilog and implemented it on FPGA. During the entire process of designing, implementation of PITO, MohammadHossein AskariHemmat had weekly meetings with Sean Wagner and Olexa Bilaniuk. Sean Wagner and Olexa Bilaniuk equally contributed to the writing of [188] and [21]. Yvon Savaria and Jean-Pierre David supervised the project and provided valuable feedback forming and revising the paper.

RISC-V Barrel Processor for Deep Neural Network Acceleration

5.2 Introduction

Deep neural networks (DNNs) have been a focus of research and development in recent years as use of artificial intelligence in a number of application areas has grown rapidly. With this powerful technology comes significant computational costs. To address these computational costs, the development of a number of different hardware accelerators specific to DNNs has been pursued. While much focus has been on using GPUs, on enhancing CPU vector processing capabilities, and on building custom ASICs for DNNs, field programmable gate arrays (FPGAs) offer the advantage of being flexible devices that can be continually adapted to rapidly evolving DNN methodologies and architectures. Indeed, there have been a number of DNN hardware accelerators implemented in FPGAs recently, such as those reported in [149, 190–192] that take advantage of the programmable nature of FPGAs, and such work to accelerate DNNs on FPGAs will certainly continue.

One approach to DNN accelerator design is to create an accelerator primitive or processing element (PE) that contains the necessary functions for a typical neural network model. In many cases, it is possible to increase throughput by replicating the basic primitives of the accelerators multiple times to create an array of processing elements. There have been numerous such designs for both ASICs and FPGAs reported in the literature [84, 107, 108, 193–196] with varying capabilities, e.g. support for low-precision computations that are suitable for DNNs. Controlling and monitoring the execution of computations on such accelerators can be challenging, and usually requires complex control circuitry to support a range of computation configurations and accelerator features. Further complicating the control mechanisms is coordinating the activities of an array of processing elements. When set to accomplish the same overall task, e.g. computing multiple layers of a DNN simultaneously, such an array requires close coordination of otherwise separate accelerator units to handle data inter-dependencies. While each accelerator primitive can have an independent replica of the control circuitry, there must still be a master controller to coordinate between them. To address these challenges, a simple, programmable approach to accelerator control is needed. One scheme is to use a software programmable controller such as an embedded CPU, one that is connected to and is capable of controlling and coordinating the accelerators. Furthermore, having such a CPU capable of multiple thread execution can ease programming when controlling multiple accelerators simultaneously. In this paper, we present a barrel processor design based on the RISC-V instruction set architecture (ISA) [144] for use as a controller for a DNN accelerator. By making the barrel processor N-way threaded, we may assign one thread to control each of the PEs, while amortizing

the fixed circuit resource costs of each microprocessor over the N threads. The purpose of this processor is to allow concurrent control and coordination of an array of DNN accelerator processing elements with software.

5.3 Background

A barrel processor is form of a fine-grained multithreading processor that exploits thread-level parallelism by switching between different threads on each clock cycle [85]. The aim is to maximize overall utilization of the processor's resources, and instruction throughput. This is similar to the technique of simultaneous multi-threading (SMT) that is used in modern superscalar processors. However, unlike SMT processors, barrel processors do not issue more than one instruction per clock cycle and the pipeline is shared by all threads.

Barrel processors have a number of advantages. First, a multi-stage execution pipeline can be simplified since it does not need to include data hazard detection and handling logic. Data dependencies between instructions are avoided when each thread has only one instruction executing in the pipeline, which will complete before the next instruction in a thread is executed. This applies to branch instruction outcomes as well, so branch prediction logic is also not necessary to avoid thread stalls and pipeline bubbles. The exception to this rule are loads from high-latency memory, which can be dealt with by implementing nearby low-latency memory instead if only a small memory is needed, or by stalling the thread and switching to another. Second, circuit resources can be kept small since the main execution pipeline is shared between threads. Each thread only needs separate circuitry for storage element that contain its state, i.e. data/address registers, program counter, and status registers.

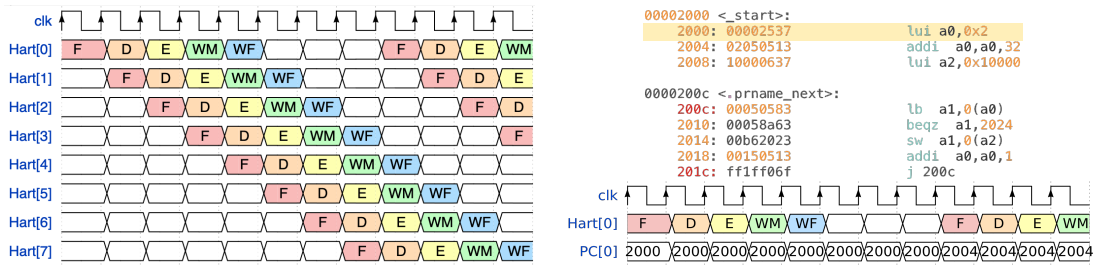
There are several examples of systems with barrel architectures. Notable instances of high-performance computers (HPCs) and processors include the Tera MTA [186] and its Cray MTA-based successors, and the Sun UltraSPARC T1/Niagara processor [187]. In the case of HPC, systems tended to implement several tens of hardware threads to tolerate memory latency by switching to threads that are not stalled. For instance, the Cray Threadstorm processor supported 128 threads [197]. By contrast, barrel processors used in embedded systems typically have nearby low-latency memories, so large hardware thread counts are not needed. Instead, the key motivation for barrel processors in embedded systems is architectural simplicity. Examples of embedded barrel processors include the XMOS XS1 [198] and the Ubicom IP3023 [199]. Given that our need is for embedded control, architectural simplicity is our motivating factor for implementing a barrel processor.

5.4 Design

5.4.1 Design Decisions and Data Path

Based on the needs explained earlier in this paper, we decided to implement the RV32I, which is the base RISC-V instruction set supporting integer arithmetic. We adopted a Harvard architecture with separate instruction and data memories. The processor is an in-order CPU and instructions are executed following compilation order and without any further scheduling. However, a hart scheduler is needed to give access to the area required resources for the hart at each stage. In the fetch stage, each hart needs to fetch instructions from a shared instruction memory. Each hart has its own register file and in the Decode stage, the hart scheduler gives access to the scheduled hart's register file.

The hart scheduler itself uses a strict round robin algorithm. No preemption nor priority is implemented, and every hart is given a fixed timeslot for execution. Figure 5.1 shows how harts are scheduled for execution in our design. Considering the execution for `Hart[0]`, it takes 5 clock cycles for an instruction to be completed. After the 5^{th} clock tick, no more processing associated to `Hart[0]` is performed. The next three slots are given to `Hart[5], Hart[6]` and `Hart[7]`. Thus each hart executes an instruction every 8^{th} cycle of the main clock, hence the CPI of 8. From the perspective of the main CPU, the throughput is one instruction per clock cycle. From the perspective of each hart, each hart runs at one 8^{th} of the main clock speed with a CPI of 1.



(a) (b)

Figure 5.1 Left figure shows 8 harts running in the barrel processor. Right figure shows every 8 clock cycles, the program counter of the associated hart increments, which allows this pipeline to be implemented without any data or control hazard circuitry

In the proposed solution, the instruction and data memory is treated as a scratchpad that is independent of any other memory in the host system. In the default configuration, each memory is 32KB in size and is implemented in the FPGA using on-chip Block RAMs. Memory reads from these Block RAMs require two clock cycles to complete. Load instructions perform all read commands to Block RAMs at the Execute stage. Subsequently, results of those reads are ready two clock cycles later in the Write Back stage and then stored in the specified register. From the perspective of the hart, the read instruction takes one clock cycle, hence there are no stalls on load instructions.

Register files and Control and Status Registers (CSRs) are encapsulated for each hart and there is no communication between threads through these registers. To simplify our design, we decided to implement inter-process communication between harts only through the shared memory. Since there is no dependency between harts, the runtime of the harts is deterministic.

5.4.2 Pipeline Structure

In ordinary pipelined CPUs, the controller requires logic to resolve data and control hazards. Consider the example assembly code snippet of Figure 5.1. In an ordinary in-order 5-stage pipeline, for instructions `0x2000` and `0x2004` to be executed correctly, the read-after-write data hazard must be resolved. This could be done by implementing additional circuitry such as forwarding networks, or by simply stalling `0x2004` for 3 clock cycles while waiting for `0x2000` to complete the write-back stage. The processor must also resolve the control hazard at branch instruction `0x2010`. This is ordinarily done by either stalling following instructions until that branch is decided, or by attempting to predict the branch condition result and potentially roll back if the prediction was incorrect. In our RISC-V barrel CPU, we do not need to handle these hazard cases. Under our round-robin scheduling scheme, each hardware thread returns to execution long enough after the register/memory writeback stage that no forwarding paths, prediction units or stalls are required. The corresponding control logic can therefore be trimmed away.

5.4.3 Processing Element Control and Monitoring

The proposed barrel processor is designed specifically to control a neural network accelerator. This accelerator is an array of eight independent PEs, each capable of performing a general matrix-vector (GEMV) product operation. Arbitrary, mixed fixed-point precision operands, either signed or unsigned, are supported with bit depths ranging from 16 down to 1. A PE computes 64 elements of the output vector in parallel and carries out 4096 1-bit multiply-and-accumulate (MAC) operations on every clock cycle. Each PE has local memory for storing elements of the weight tensors, and a local memory for storing data tensors. Executing a GEMV task on a PE, a number of configuration registers that control the memory access patterns on both the weight and data memories need to be set by the CPU.

To control each PE, we considered several mechanisms including custom instructions and memory-mapped IOs. In the case of custom instructions, the parameters to be sent to the neural network accelerator to perform an operation like convolution can be numerous and cannot be encapsulated within a single instruction. In the case of memory-mapped IOs, the processor needs to define a *physical* memory space for control registers and this brings additional overhead to the memory sub-system, because of the need for a virtual-to-physical address translation. Instead, we chose to

control the PEs by implementing a set of custom CSRs as an extension to the base set of the CSRs in the RISC-V specification [200]. Every hart has its own instance of this set of CSRs, which are directly linked to its associated PE. At the software level, these CSRs are accessible through standard read/write CSR instructions. This allows the user to programmably control each PE. Thus, each thread is responsible for controlling and monitoring its PE by writing and reading from the CSRs of its associated hart. Like the register file, the hart scheduler provides access to the correct CSR file when it is being accessed. At the hardware level, the CSRs are accessible by both the associated hart and the accelerator.

A hart will configure and submit a job (a GEMV product operation) to its PE through CSRs. When a hart submits a job to its associated PE, other PEs can be in idle mode to save power or they can be fully utilized by their harts. The configuration of a job can be different from PE to PE i.e different precision for weights, inputs and outputs. This allows for distributed mixed precision GEMV operation among the processing array. To submit a job, the hart needs to send the configurations and a kick start signal. When the PE is busy with the submitted job, the hart can start preparing the next job by sending configuration to the PE. After a job is submitted to the PE, the PE notifies the processor when it completes it by raising an interrupt request. This interrupt will be handled by the interrupt service routine. Once the interrupt is serviced by the hart, the processor can start preparing the next job.

Table 5.1 Complexity and performance summary of various RV32I implementations. Note that, except the column for SHP [201], all other columns show resource usage in a Kintex Ultrascale (-2) 040 (KU040) FPGA. For the SHP design, the numbers show resource usage in a Virtex 6 LX75T (-3ff784) FPGA

	Baseline	Barrel Processor w/o CSRs, 8 Harts	Barrel Processor w/ CSRs, 8 Harts	GRVI Phalanx [202]	PicoRV32 Regular [203]	System Hyper Pipelining (SHP) [201]
LUTs	1111(0.623%)	1988(0.82%)	5712(2.36%)	320(0.13%)	909(0.38%)	2812 (24.15%)
LUTRAMs	48(0.04%)	423 (0.38%)	410 (0.36%)	N/A	48(0.04%)	N/A
FFs	497 (0.1%)	730(0.15%)	13303(2.74%)	N/A	574 (0.12%)	4218 (4.5%)
BRAMs	15 (2.5%)	15 (2.5%)	15 (2.5%)	N/A	15 (2.5%)	N/A
Dynamic Power	0.201W	0.292W	0.372W	0.043W	0.172W	N/A
Frequency	250MHz	250MHz	250MHz	375MHz	400MHz	549MHz
CPI	8	1	1	1.3	4.1	N/A

5.5 Results

We started by designing a single threaded RV32I core in SystemVerilog. Since the code for the base model was intended to be re-used for implementation of the barrel processor, we did not implement any hazard control logic. After testing and verification of this base processor using a SystemVerilog simulator, we implemented logic for barreling. We synthesized our design with Xilinx Vivado targeting the Kintex Ultrascale (-2) 040 (KU040) FPGA. Table 5.1 illustrates the resource utilization

for implementing our barrel processor on the target hardware. The additional resource utilization for the 8-hart barrel processor is about twice the baseline's usage in number of LUTs. However, this is significantly smaller than simply replicating the single-threaded baseline processor 8 times. The PE array runs at 250MHz and can consume a long word (4096 bits) each clock cycle. We designed both our baseline and barrel core to run at 250MHz. We then added the support for CSRs, including 25 custom CSRs per hart to control the PE array. This resulted in a higher LUT count usage compared to our baseline barrel core which does not include any CSRs. However, we will show that this LUT usage is significantly lower than the LUT usage of the PE array.

To test our processor, we used the RISC-V tests available at [204]. Written in assembly, these tests verify the essential functionality of the core. We compiled these tests with the RISC-V gcc compiler and loaded the binary output to our processor. Using the generated binaries, we verified the functionality of the baseline core, the barrel processor and the barrel processor with custom CSRs. For the base core, which lacks hazard control logic, we had to change the RISC-V test source code to inject five `nops` (for the length of the 5-stage pipeline) to avoid any hazards. For the barrel processor, we used the RISC-V source code without any modifications. Each RISC-V test program was replicated for all harts within the core. In addition to these tests, we wrote a behavioral model for RV32I with multiple harts in SystemVerilog. In our verification environment, for every instruction, our predictor model used this behavioral model to predict the output (our expectation) and checked it against the actual values inside the hardware. Using these two test suites, we are confident that our processor complies with the RV32I specification.

We compared our design to GRVI Phalanx [202] on a per core basis. Note that each Phalanx core shares some logic (shifter, multiplier, and byte/halfword load/store) with other cores, and the amount of FPGA resources used by that shared logic is not mentioned. Also, Phalanx core does not support CSR related instructions. Phalanx implements 2 (or 3 configurable) stages, which likely explains in part the extra amount of resource in our 5-stage pipeline. However, each hart within our barrel processor shares most of the data path with other harts. To achieve this, we make a compromise on our per-thread speed. Although the per-core resource usage of Phalanx is quite good, we estimate that supporting 8 hardware threads (core) by replicating the Phalanx core as-is would result in the same resource usage as our 8-thread RV32I. Any memory intensive operation in Phalanx slows downs by half because of 2 to 1 Block RAM to core ratio. The branch instructions in Phalanx is guaranteed to take 2 to 3 clock cycles (for 2 to 3 pipeline stages) which results in CPI of more than 1 (1.3). Our barrel design guarantees a CPI of 1 since branches and other instructions take exactly one clock cycle for each hart.

In [201] each thread could have a variable execution time per time slot. However, in our barrel processor, the hart scheduler assigns fixed-length time slots to each hart. In the System Hyper

Pipelining (SHP) design, to allow variable length execution time for each thread, it is expected that the processor needs to implement hazard control logic. In our barrel implementation, a fixed-length time slot for each hart not only results in a smaller logic footprint for the hart scheduler, but also all the logic required for hazard control is removed. The last column in Table 5.1 illustrates the resource usage of the SHP. To compare the resource usage reported in [201] to a more recent Xilinx FPGA device that we used, we estimated the LUT usage based on the LUTs available in each slice. We used Xilinx's Configurable Logic Block datasheet [205] to map the 703 slices used to implement the SHP processor to LUT counts. Although the SHP design has the flexibility of assigning variable execution time slot to different threads, the estimated LUT and FlipFlop usage for an SHP design is higher than for our barrel core. On the other hand, although the SHP design can achieve higher clock frequency, the effective CPI was not reported in the paper.

Unlike GRVI Phalanx and the SHP design, the source code for PicoRV32 is freely available at [203]. There are three core variants for PicoRV32 from which the regular implementation is the closest implementation to our baseline design. We synthesized the regular PicoRV32 on the same FPGA platform for comparison. Compared to our baseline implementation (no barreling, no CSRs), the PicoRV32 requires almost the same amount of resources. Compared to the PicoRV32, our barrel processor achieves better CPI. The results we obtained come straightforwardly from the synthesizer. None of the aforementioned designs are ideal for controlling the PE array as explained earlier. GRVI cores are too simple and they do not handle many instructions used in our barrel design (especially CSR related instructions). SHP is too complicated and since the design is not open source, it is harder to re-purpose it for our accelerator. Finally, PicoRV32 would require a major design change in the pipeline to be used in our design.

Table 5.2 Detailed sythesis results for a PE array and a barrel core (with CSRs) on a Xilinx KU115 FPGA

	Barrel Processor 8 Harts + 8 PEs	PE array with 8 PEs	Barrel Processor 8 Harts
LUT	169550 (25.56%)	162833 (24.55%)	6781 (1.02%)
FF	114173 (8.61%)	100861 (7.60%)	13312 (1.00%)
BRAM	983 (45.51%)	968 (44.9%)	15 (0.6%)
Dynamic Power	9.147W	8.841W	0.302W
Frequency	250MHz	250MHz	250MHz

To allow performing GEMV operations, we attached the PE array to our barrel processor. Table 5.2 illustrates the synthesis results for 8 harts and 8 PEs. Compared to the PE array, the barrel core is relatively small. In terms of LUT counts, the barrel core is only 4% of the PE array. To test our design, we wrote an assembly code to perform a matrix-matrix product. We used an input matrix size

of 8 by 128 and a weight matrix size of 128 by 128 with a two-bit precision (64 elements per row). The input was divided into 8 vectors, which allowed us to divide this matrix-matrix product into 8 matrix-vector products. Each hart configured its associated PEs with the appropriate configuration and submitted a job to the PE. In total, each matrix-vector product took 16 clock cycles to finish. Since all these matrix-vector products were performed simultaneously on 8 different harts, the result of the entire matrix-matrix product took 16 clock cycles to finish.

5.6 Conclusion

In this paper, we presented a 5-stage RV32I barrel processor supporting 8 harts. We demonstrated how we attached our barrel processor to an array of PEs to perform a matrix-matrix product, a core operation in machine learning applications. In our barrel processor, each hardware thread is responsible for sending control signals to a PE. We showed that compared to the ordinary RV32I processor, a barrel processor needs little extra logic to handle multiple threads, yet it outperforms the base RV32I core when handling multiple harts running simultaneously. Finally, we added a set of custom CSRs to our barrel RV32I processor and successfully controlled an array of PEs to perform matrix-matrix product operation.

CHAPTER 6 ARTICLE 4: BARVINN: ARBITRARY PRECISION DNN ACCELERATOR CONTROLLED BY A RISC-V CPU

6.1 Prologue

6.1.1 Article Details

BARVINN: Arbitrary Precision DNN Accelerator Controlled by a RISC-V CPU
MohammadHossein AskariHemmat, Sean Wagner, Olexa Bilaniuk, Yassine Hariri, Yvon Savaria, Jean-Pierre David
Published at 28th Asia and South Pacific Design Automation Conference (ASP-DAC) Jan 31 2023.

6.1.2 Context

The overall architecture of BARVINN was continuously discussed and developed in weekly meetings from 2019 to 2022 between Sean Wagner, Olexa Bilaniuk and MohammadHossein AskariHemmat. After designing PITO, we realized the need for a complete system to encapsulate PITO and MVU array. Since beginning, we focused on software programability of BARVINN. PITO was a step towards achieveing this goal, however, there was a huge gap between abstraction layers that define models in ONNX or Pytorch to fine grain control signals for MVU. To close this gap, we decided to design a code generator tool (available at `https://github.com/hossein1387/MVU_Codegen` that takes in a quantized model in ONNX format and generates RISC-V code for PITO. Our code generator traverses the ONNX computational graph and produces RISC-V code for each computation node. We later improved our code gen tool to provide performance report for the input model. Currently, our code generator tool is not performing any optimization on the model. However, it provides a better user experience. Our future plan is to integrate our code generator into TVM [206] to benefit from both hardware agnostic optimization and define BARVINN specific optimization passes. In addition to code generator, we added a memory sub-system and host interface to BARVINN to be able to communicate and program PITO and MVU RAMs. Finally, we implemented BARVINN for ALVEO U250 FPGA card.

Additional information and Results: All the experiments and results presented in this chapter were conducted using AMD Vivado 2021.1. In Section 6.5.2, we have included various results comparing BARVINN to other low precision accelerators. Specifically, for the results displayed in Table 6.5, we utilized a 250MHz clock for BARVINN and a 100MHz clock for FINN. Additionally, both FINN and BARVINN utilized on-chip BRAMs for weight and activation storage. The power estimation

data reported in Table 6.4 was obtained using Vivado's power estimation tool.

We conducted a performance comparison between BARVINN and Nvidia Jetson Nano board. For model execution, we utilized the *onnxruntime-gpu* python package and used the ResNet9 model from Section 6.5.1. We tested batch sizes of 128 and 1, resulting in average FPS measurements of 996.12 and 120.83, respectively, after 1000 runs, with an average power consumption of 6.6 W.

With BARVINN, we achieved significantly higher FPS rates of 31001.9 and 1937.6 for 2 and 8-bit precision, respectively. Comparing the power consumption data reported in Table 6.4, we can conclude that our accelerator demonstrates superior throughput per watt compared to programmable devices like the Jetson Nano board.

6.1.3 Contributions

BARVINN is an open-source project that can be accessed on GitHub at `https://github.com/hossein1387/BARVINN`. Its purpose is to serve as a platform for researchers to execute quantized neural networks with any desired level of precision on actual hardware. BARVINN offers a unique capability to run quantized models with arbitrary accuracy in contrast to regular CPUs and GPUs. The project was presented at ASP-DAC 2023 and has been featured on the Free and Open Source Silicon (FOSSI) Foundation website [207].

Personal Contributions: After developing PITO, BARVINN was the next natural step towards a complete accelerator system. It comprises a memory interface, code generator, PITO, MVU array, and the necessary software for running a quantized neural network based on the ONNX model. Under Sean Wagner's supervision, MohammadHossein AskariHemmat designed a code generator that maps the computation nodes of the ONNX model to corresponding computation jobs for deployment on BARVINN. To enable communication with an external host, MohammadHossein AskariHemmat also created a memory interface that enables an external host to program PITO and MVU array. Alongside the memory subsystem, host interface, and code generator, MohammadHossein AskariHemmat also developed an APB bus that enables PITO to write to MVU registers through RISC-V CSR instructions, an MVU wrapper to convert APB requests to low-level signals for MVU registers, and a BARVINN top-level that properly encapsulates and connects all the components. With the assistance of Olexa Bilaniuk, MohammadHossein AskariHemmat developed a C runtime that controls MVUs with multiple threads through software. Furthermore, MohammadHossein AskariHemmat prepared BARVINN for FPGA implementation on the ALVEO U200 card and on Globalfoundries 22FDX FD-SOI technology with the help of Yoan Fournier. MohammadHossein AskariHemmat also made the documentation for BARVINN publicly available at `https://barvinn.readthedocs.io`. During the entire process of designing, implementation of BARVINN, MohammadHossein AskariHemmat had weekly meetings with Sean Wagner and Olexa Bilaniuk. Sean Wagner and Olexa Bilaniuk

contributed to the writing of [22]. Yvon Savaria and Jean-Pierre David supervised the project and provided valuable feedback forming and revising the paper. This project was partially funded by our industrial partners, CMC Microsystem and Mitcas. CMC Microsystem also provided EDA tools and access to ALVEO U200 FPGA card.

BARVINN: Arbitrary Precision DNN Accelerator Controlled by a RISC-V CPU

6.2 Introduction

Deep neural networks (DNNs) traditionally rely on floating point computations. These operations are slow and costly in terms of power consumption and required silicon area compared to fixed-point/integer operations. One way to accelerate computation in a DNN is to use less precision for computation via quantization [8]. This also reduces memory consumption as well as energy consumption. For instance, in a 45 nm process, 8-bit integer multiplication and addition take 0.2 pJ and 0.03 pJ, respectively, while the same operations with 32-bit floating-point values requires 3.7 pJ for multiplication and 0.9 pJ for addition [170]. On an Intel Core i7 4770 running at 3.4GHz, multiplication is more than 3 times faster for fixed-point compared to floating-point [208]. With recent quantization techniques, these benefits are available with little to no loss in model performance and accuracy. In [2, 105], their quantization schemes showed accuracy losses of 1-3% at 2-bit precision on most classification and object detection models. Table 7.1 illustrates the result of applying Learned Scale Quantization (LSQ) [2] with different bit precisions on different models and tasks. Quantized models offer accuracy similar to full precision models, while having smaller size.

Mixed-precision quantization [106, 209–212] further provides finer control to reach an optimal solution by learning different precisions for each layer of a network. In [106], the authors illustrate that using their mixed-precision framework, they reduced model latency and energy consumption by a factor of almost $2\times$ with little drop in accuracy compared with an 8-bit quantized model.

Fully benefiting from low-precision in a DNN requires hardware that natively supports low-precision

Table 6.1 Effects of Quantization on Accuracy and Model Size

Task	Dataset	Model	Precision A/W	Acc/ MAP	Size (MB)
Classification	CIFAR 100	ResNet18	LSQ(2/2)	76.81	2.889
			LSQ(4/4)	76.92	5.559
			LSQ(8/8)	78.45	10.87
			FP32	76.82	42.8
Object Detection	VOC-2007	SSD300-ResNet18	LSQ(2/2)	0.61	10.34
			LSQ(4/4)	0.60	11.81
			LSQ(8/8)	0.68	14.77
			FP32	0.59	32.49

computations. Commodity hardware can perform arbitrary precision arithmetic by transforming data-layout and computing with bit-wise instructions [99]. However, this approach is extremely costly for general processors, because of the overhead for shifting, masking and packing bits to the correct format. At the time of writing this paper, there are no commercially available general processors (CPU or GPU) that can efficiently process data in arbitrary precision.

In this paper, we propose an arbitrary low-precision DNN hardware accelerator called BARVINN. Our accelerator is software programmable and can be integrated in the RISC-V standard development flow. It is designed as a highly optimized computational pipeline for DNNs that introduces low hardware overhead and offers low-power operation. The contributions of our paper are as follows:

- Implementation of a DNN hardware accelerator with arbitrary fixed-point low-precision for matrix-vector multiply operations at high-throughput and low power.
- Implementation of a custom embedded RISC-V CPU to control an array of DNN accelerators by software.
- Data structures for efficiently storing and processing weights and activations for high-throughput serial computation.
- Development of a software code generator for transforming DNNs into RISC-V code that executes on our accelerator.

In section 6.3, we review relevant DNN accelerators from the literature. Section 6.4 presents the architecture of BARVINN. In section 6.5, a detailed performance analysis of BARVINN is provided and compared with other DNN accelerators.

6.3 Related works

Several DNN hardware accelerators supporting quantization and low-precision have been presented in recent years for both FPGA and ASIC targets. Here, we discuss the accelerator architectures most relevant to our work.

Recent FPGA-based accelerators include FINN [103,213], DNNBuilder [149], and FILM-QNN [100]. In FINN and DNNBuilder, a software toolchain is used to map a trained DNN to generated logic modules that are integrated together. An overall processing pipeline is generated and then synthesized for the target device. The advantage of this approach is that the logic efficiently implements a specific DNN with minimal overhead on a device that can be reconfigured to different DNNs at different times. However, this approach requires that all DNN layers be implemented in the logic all at once,

which limits the size of the DNN to the amount of logic resources available on a given FPGA. While FINN supports low-precision down to binary and DNNBuilder down to 4-bit, neither supports arbitrary and mixed-precision at different DNN layers. In contrast, FILM-QNN does support DNN models of arbitrary sizes and quantized DNNs with mixed precisions. However, it is limited to only 4- or 8-bit weights and 5-bit activations due to a bit-packing scheme used with the DSP blocks in the FPGA.

Several ASIC accelerator designs support arbitrary precision. Bit Fusion [214] uses a large array of 2-bit processing elements that can be fused together to perform up to 8-bit operations. Loom [145], and BitBlade [215] employ bit serial computation schemes for added flexibility. While bit-serial computation of any single math computation (e.g. multiplication) inherently requires additional clock cycles and latency over bit-parallel circuits, these designs exploit the large number of computations in a DNN that can be done in parallel. This is done by implementing a large number of bit-serial computational units operating simultaneously, which compensates high latency with high throughput. For example, the Loom engine consists of $128 \times 16 = 2048$ Serial Inner-Product units (SIPs), each of which performs 16 1×1-bit products per cycle.

6.4 Architecture

BARVINN is designed to provide high-throughput and software programmability, while supporting DNNs of arbitrary size and type. The high-level architecture of BARVINN is illustrated in Figure 6.1. It consists of the following main components: 1) an array of Matrix Vector Units (MVU) [104], and 2) a RISC-V CPU called PITO [21] as a controller for the MVU array. The MVUs accelerate common DNN computations such as GEMV, GEMM, and convolutions along with other operations such as batch normalization, ReLU activation, and quantization. PITO coordinates the computations in the MVU array while also handling data transfers to and from the host system.

As it is not possible to foresee all possible neural networks that may crop up in the literature in the future, high-level sequencing of tensor operations for BARVINN is done in software. To control the array of processing elements, unlike the aforementioned accelerators, BARVINN uses the standard RISC-V RV32I ISA. This allows us to leverage the pre-existing software ecosystem. Furthermore, by using a CPU that supports a well known ISA, BARVINN is more flexible and it can be adapted to support new DNN architectures.

6.4.1 Matrix Vector Units

The base configuration of BARVINN is implemented with an array of 8 MVUs. Figure 6.1 shows each MVU is a 64-element vector pipeline with several modules: a) a Matrix Vector Product unit

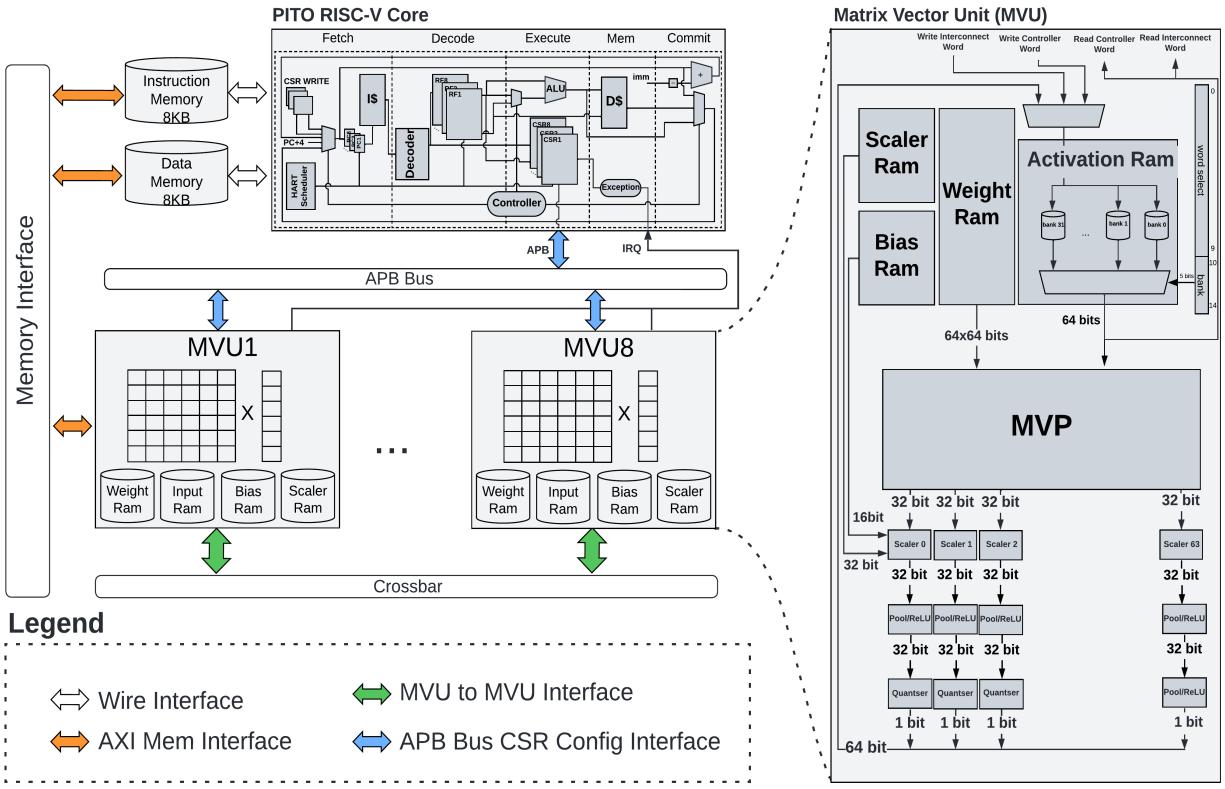


Figure 6.1 BARVINN hardware architecture with MVU array and PITO RISC-V controller. Right side is MVU detail

(MVP), b) RAMs for activations/weights/scalers/biases, c) a scaler unit, d) a pooling/activation unit, and e) a quantizer. MVUs compute 64 output vector elements per clock cycle using a 64 element input data vector from the activation RAM and a 64×64 element matrix from the weight RAM. Activation and weight RAMs store data in low-precision. MVP units operate in low-precision, while subsequent units in the pipeline operate in high-precision fixed-point.

To justify our design choice of operating on 64 element vectors, we analysed over 50 models available at the ONNX Model Zoo [216] to check the input channel size of convolution layers. Figure 6.2 illustrates a distribution of input channel size of all layers among those models. We found that 79% of these models use convolution with input channel sizes that are multiples of 64.

Matrix-Vector Product Units

Matrix operations are carried out by the MVP units. They compute on fixed-point arbitrary precision operands from 1- to 16-bit. Each MVP has 64 vector-vector product (VVP) pipelines. Each VVP has 64 input lanes with 1-bit multipliers, followed by an addition tree with 8-bit output, as shown

in Figure 6.4. On every cycle, 64 bits from the activation RAM are broadcasted to each of the 64 VVPs, while a 64×64 matrix tile from the weight RAM is read with each row of the tile sent to separate VVPs. The VVPs compute a 64-element dot product on 1-bit operands in each pipeline. With 64 VVPs per MVP, the overall output is a 64-element vector.

MVPs compute arbitrary bit precision dot-products using the bit-serial scheme of [104]. Weights and activations can be unsigned or 2's-complement signed fixed-point. Bit-depth is set independently for both, thus allowing for mixed precision. The bit-serial dot-product, shown in Algorithm 3, is a multi-cycle sequence starting with the most significant bits (MSB) from 64 elements of the activation and weight tensors. Bits are multiplied in each lane and results are added together across lanes in an addition tree producing an 8-bit dot product. This is added to an accumulator/shifter (see Figure 6.4). The MSB×MSB result represents the highest order-of-magnitude partial sum of the overall dot product. The MVP then computes the next lower order-of-magnitude partial sum by drawing the needed bit combinations of the operands. When a change in the order-of-magnitude is made, the accumulator is shifted left by 1-bit to align to the order-of-magnitude prior to adding the addition tree output. MVPs are fully pipelined, allowing them to work on different bit combinations at different stages without stalling. The operation completes when the dot products of the least significant bits (LSB) of the operands are computed and accumulated. For b_w-bit weights and b_a-bit activations, the overall operation takes $b_w b_a$ cycles to compute one tile of the output vector. The precision of the operands is configured separately for each MVU, thus each MVU can process different layers with different bit precisions.

Algorithm 3 Bit-serial dot-product

b_a, b_w: activation and weight bit precisions
x, w: activation and weight vectors of size n
j, k: bit position for activations and weights
$accumulator \leftarrow 0$
for $i \leftarrow b_w + b_a$ to 1 **do**
  for all (j, k) where $j + k == i$ **do**
    for $l \leftarrow 0$ to $n - 1$ **do**
      $onebitprod = x_j[l] \times w_k[l]$
      $accumulator \leftarrow accumulator + onebitprod$
    end for
  end for
  shift $accumulator$ left 1-bit
end for
$output \leftarrow accumulator$

Our bit-serial dot product scheme differs from other architectures. The computation scheme in BitFusion is based on computing the individual products of the overall dot-product, that are then

summed. This requires a large number of shift-registers to align and sum partial products. BARVINN and BitBlade instead interchange the ordering of the computation such that partial products of the same magnitude from all individual products are computed first and then summed. This reduces the number shifters needed. BARVINN additionally serializes the computation of partial products of different magnitude, requiring only a single fixed shifter and a single adder tree, whereas BitBlade requires 16 variable shifters and 17 adder trees. BARVINN maintains throughput despite this serialized scheme by parallelizing across a wider number of input operands and producing a larger number of output products per clock cycle. BitFusion and BitBlade are further limited to operand sizes 2, 4, and 8-bit, whereas MVU s in BARVINN and SIPs in Loom support operands of any bit-depth down to 1-bit. However, Loom's data loading scheme restricts the efficiency for general matrix multiply operations when the weight bit depth is below 16, whereas BARVINNs is able to maintain full throughput down to 1-bit.

Memories and Data Layout

Activation and weight RAMs store data in a bit-transposed format shown in Figure 6.3 to exploit bit-serial computation. When precision is greater than 1 bit, tensor elements are organized in blocks where bits of the same order-of-magnitude are stored in the same memory word starting with the MSBs in the lowest address. A block of n elements with precision b requires b memory words of width n. Activation vector elements are in blocks of 64 while weights matrix elements are in blocks of 4096 bits in order to load a 64$\times$64 matrix tile. A transposer module transforms input data from the host into the needed bit-transposed format. Transposition is only needed on the first layer of a DNN since MVUs write back to activation RAM in the bit-transposed format. Weights are pre-processed by a toolchain on the host and loaded into weights RAMs in the expected bit-transposed format.

The layout of the tensors in the RAMs depends on the operation to be performed. For GEMV, activations are organized as vectors with blocks of 64 elements, while weight matrices are organized as a set of 64$\times$64 tiles. For 2D convolutions, layout of activations is $NHWC$, where the channel dimension C is the innermost dimension, followed by width W, height H, and batch size N. The C dimension is the innermost since several common DNNs such as ResNet typically have hidden layer channel depths that are powers of 2, and hence align to the 64 input lanes of the VVPs. When there are more than 64 channels, the first 64 channels are stored in the first block, the second 64 channels are stored into the second block and so on. As an example, an input tensor of [N=1, H=8, W=8, C=256] with 2-bit precision, will have 4 channel blocks, each block will have 64 rows of 2 by 64-bit elements.

Our weight tensor memory layout for 2D convolutions is designed to support efficient execution by interleaving the input channel dimension C_i and output channel dimension C_o. Each weight memory

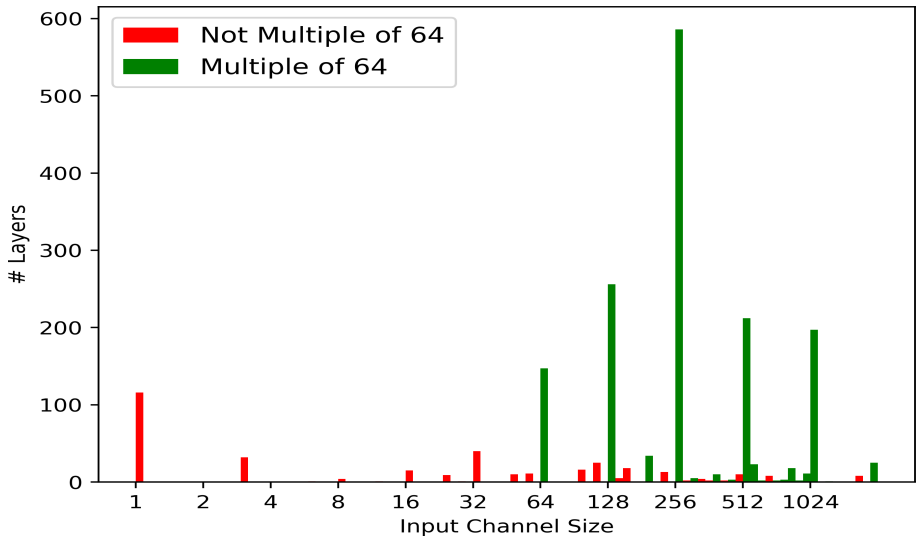


Figure 6.2 Channel sizes in models from ONNX Model Zoo.

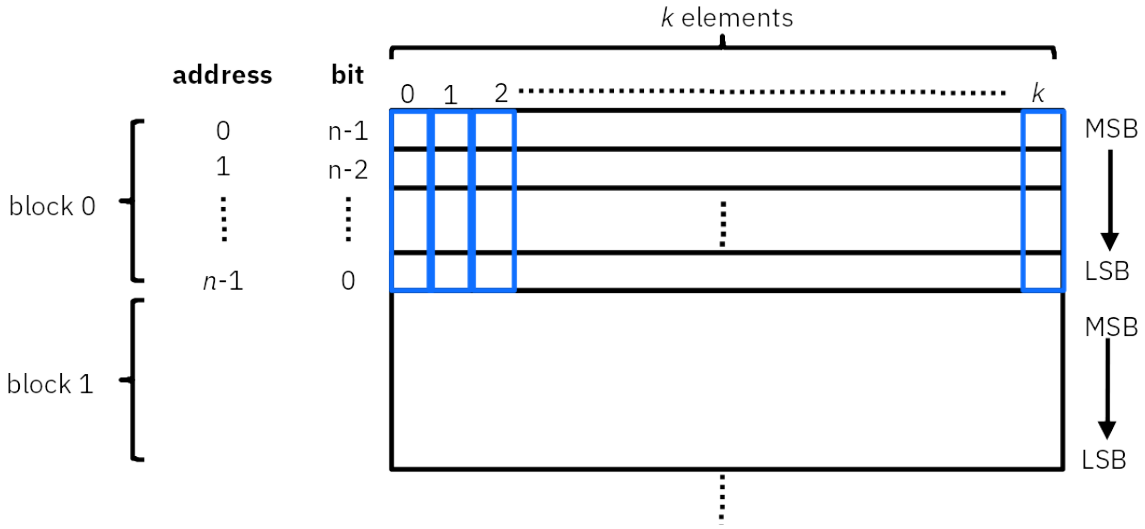


Figure 6.3 Bit-transposed data format for arbitrary precision

word contains 64 subsets from the C_o dimension, with each subset containing 64 elements from the C_i dimension. A contiguous block of b_w words that stores a complete set of bits for the needed weight precision is referred to as a channel block C_b. The layout for 2D convolution weights is $C_{o,s}F_HF_WC_b$, where $C_{o,s} = C_o/64$ are output channel sets, and the kernel size is (F_W, F_H).

Job Configuration and Execution

MVUs are programmed to perform jobs such as GEMV or Conv2D operation. A controller sets configuration registers that orchestrate the sequence of calculations and memory reads to complete

an operation in the MVUs. Once the job is finished, the MVU will generate an interrupt to the controller, indicating that the job is finished and results are ready to be sent back to the host or to trigger subsequent operations on the same MVU or other MVUs. While a MVU is busy, it can be programmed to prepare the next job to minimize idle time.

Each MVU contains address generation units (AGU) that drive the memory access pattern across the activation and weight RAMs. The access pattern is managed by a set of up to five nested loops with parameters setting the number of iterations and the forward or backward address jumps to make on each iteration. The address jump scheme reduces the logic to a set of small accumulators to control the loops and small adders to compute addresses. Innermost loops are usually set to stride over the bit depth of the activations and weights. Outer loops are used to iterate over the bit combinations for the serial dot-product procedure and over the dimensions of the tensors. For GEMV, two nested loops are required for both activations and weights. Conv2D operations are programmed to compute one row of the output activation map per job, requiring four nested loops.

Pipeline Modules

Each MVU has modules downstream from the MVP to implement other DNN operations including a multiplier/adder unit, a pooling/ReLU unit, and a quantizer/serializer unit. These modules operate at high-precision. Fixed-point multiplier/adder units (*Scaler* in Figure 6.1), compute DNN operations such as batch normalization and quantization scaling as in LSQ [2]. Scalers multiply the MVP output by a 16-bit operand sourced from the scaler RAM. In an FPGA, the multiplier is 27×16, which aligns with the port widths of on-chip fixed DSP units. An adder that follows adds 32-bit fixed-point bias terms from bias RAM. Scaler and bias RAMs have independent AGUs. The module that follows combines max pooling and ReLU (*Pool/ReLU* in Figure 6.1), implemented as a comparator with an internal register. For ReLU, the incoming value is checked against the register initially set to 0. The combined MaxPool/ReLU is implemented by programming MVU s to produce data in the sequence needed for a MaxPool window.

The pipeline ends at the quantization/serialization unit (*QuantSer* in Figure 6.1). It takes 32-bit fixed-point data from each of the 64 data paths and serializes them into 64 1-bit outputs. It is programmed to set the output bit-depth and the MSB position from the input word. Combined with scaler units, this is used to implement quantization schemes such as LSQ [2]. Serialized outputs of each datapath are grouped into a single 64-bit word that is sent either to the activation RAM of the same MVU, or to a different MVU via an interconnect.

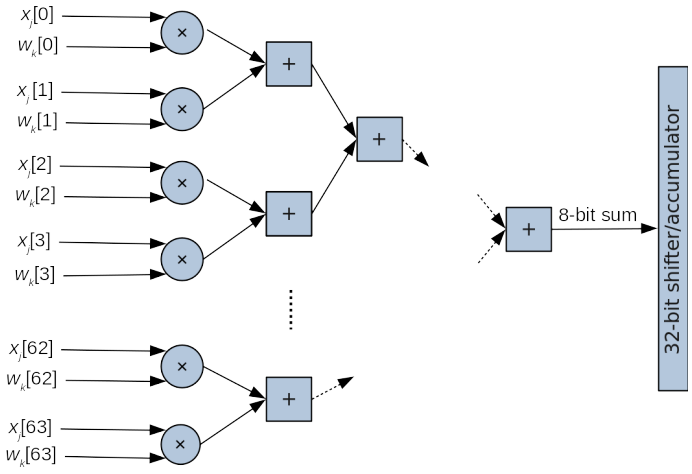


Figure 6.4 VVP unit with a shifter-accumulator. Bit j from 64 elements of the activation tensor x and bit k from 64 elements of the weight tensor w are input in a bit-serial fashion. Note that some input bits and layers of the 5-deep adder tree are not shown

Interconnect

MVUs can send data to each other via an interconnect implemented as an 8-way crossbar switch with broadcast capability. A source MVU is programmed to send its output results in a serialized fashion to a given address in the activation memory of a destination MVU(s). At a destination MVU, a fixed-priority arbitration scheme to the write port of the target MVU activation RAM is used. The interconnect is given highest priority, followed by the controller, then lastly the MVU itself. When multiple MVUs attempt to write to the same destination MVU, a fixed priority scheme determines which MVU can write to its memory.

DNN Mapping

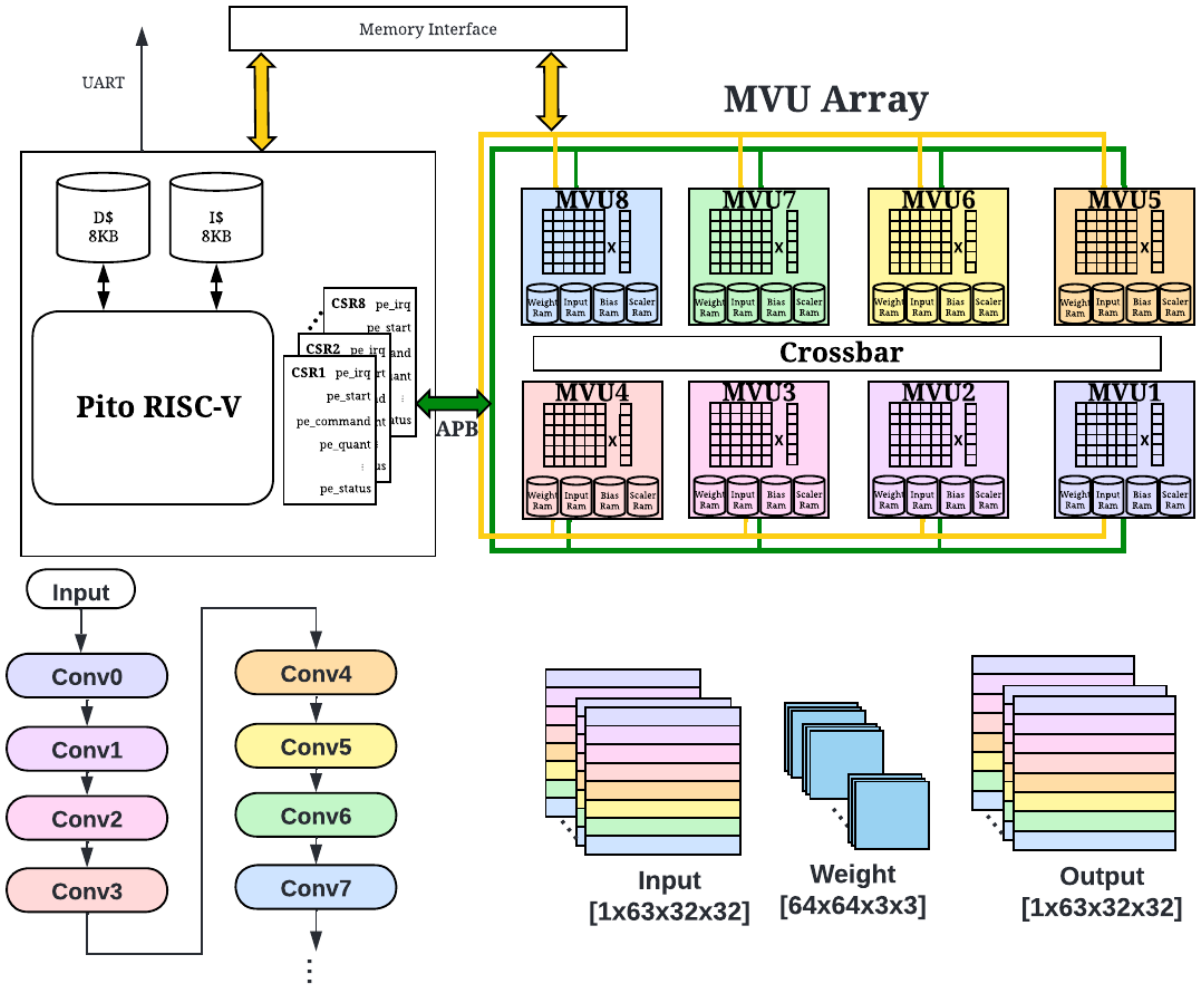
Each MVU can be assigned to different layers of a DNN, such as convolutions and fully-connected layers. Alternatively, a single layer can be split between multiple MVUs with each MVUs processing a subset of the input activations and/or weights. Partial results are forwarded from one MVU to another via the interconnect to process subsequent layers of the network, thus creating an overall processing pipeline through the array. By sending partial results from one MVU to another, subsequent MVUs can begin processing as soon as sufficient data has been received from previous layers. For instance, a MVU processing a 3×3 convolution requires only 3 rows of activations from the previous layer to produce one output row of the layer it is processing. This avoids the need to wait until all outputs from a layer are generated, which reduces latency and idle time. Furthermore, the ability to immediately process partial layer outputs by subsequent MVUs keeps on-chip storage requirements low, since only the partial set of activations required to produce the next layer partial output needs to be stored.

Depending on the performance goal, BARVINN can execute a DNN in either Pipelined mode or Distributed mode. In Pipelined mode (Figure 6.5.a), the MVU array can process up to 8 convolutions and fully-connected layers all at once. Each MVU can be configured to use different precisions. In cases where a DNN model contains more than 8 layers, the MVU array can be programmed to process the entire model by dividing it into subsets of up to 8 layers each. Each MVU can be loaded with weights from layers in each subset, either all from the start of processing if there is sufficient weight memory available in each MVU or on-the-fly from external memory if not. Output activations from the last MVU in the chain can also be stored temporarily in off-chip memory and fetched later in the case where the first MVU is still processing data from the current lap. In the Distributed mode, to minimize latency, the objective is to process single batch inputs as fast as possible. As can be seen in Figure 6.5.b, in this mode, the computation of a single layer is broken into 8 independent computation regions. All MVUs will be programmed to share the same set of weights. To make sure an MVU computation is independent of those performed on other MVUs, the user might need to copy the input regions that are shared between computation units. The programmability of BARVINN allows the user to mix and match these execution modes for different layers and models to achieve highest performance.

6.4.2 PITO

To make use of MVUs for neural networks, a control unit is required. The controller is a barrel RISC-V processor designed to control the 8 MVUs using separate but communicating hardware threads (harts) that each manage their respective MVUs. DNN layers are executed either in distributed or in a pipelined fashion, depending on whether the DNN is compiled to maximize throughput or minimize latency. This design allows MVUs to complete tensor operations independently of each other. The drawback is that it requires 8 microprocessors to execute the 8 programs. We instead amortized the fixed costs of the processor by adopting barrel processing. With a 8-way threaded processor, we may assign one thread to control each of the MVUs. Because every thread comes up for execution only every 8 clock cycles, the five pipeline stages (fetch, decode, execute, data read & writes and commit) can be completely hidden. Branch prediction units are unnecessary. Since tensor operations can require hundreds of cycles to execute on a MVU, the barrel processor can fully turn over dozens of times in the interim, allowing each thread to issue the next command to its MVU in a few instructions.

We adopted a Harvard architecture and divided the instruction and data RAM, 8KB each, and shared between all harts. This gives a 1K word space to store data and instructions to control each MVU. The processor executes instructions following compilation order and without any further scheduling. A hart scheduler provides access to the required resources for the hart at each stage. In the fetch



a. Pipelined mode

b. Distributed mode

Figure 6.5 Execution flow of a DNN on the MVU array in Pipelined (a) and Distributed (b) modes. In Pipelined mode each MVU processes one layer at a time. In distributed mode, the computation of a single layer is distributed among multiple MVUs

stage, each hart loads instructions from the instruction RAM. The program counter (PC) and register file for each hart is different and the hart scheduler indicates which register should be accessed at a given time. The Decode stage decodes instruction and loads source registers or an immediate operand. Our RISC-V controller is compatible with RV32I RISC-V ISA with minimal support for privilege specification to make Control and Status Registers (CSRs) and Interrupts available to interface with the MVU array. In addition to the base CSRs, we have added 74 MVU-specific CSRs to allow software to control the processing element array. These CSRs control different settings within an MVU such as weight and activation precision, AGU's jump settings, input, weight and output memory address and pipeline module selection as described in 6.4.1.

6.4.3 Code Generator

BARVINN performs GEMM/GEMV, Convolutions, Maxpooling and activation (ReLU). However, it is up to the user to sequence the operations within a DNN with software. To facilitate this, we developed a code generator that takes a DNN described in ONNX [217] and configuration settings (weight/input/output precision), and generates RISC-V code for each operation. The code generator exports weights to the bit-transposed format described in section 6.4.1. Since each MVU works on 64-bit words, the code generator tiles each weight tensor in blocks of 64×64. When this cannot be done (either tensor input channel or output channel is not a multiple of 64), we pad the corresponding tile. Currently, our code generator does not apply graph optimization techniques. Also, for now, our code generator supports Pipelined mode execution. In the following section, we used our code generator to map PyTorch models to micro kernel codes which can then be directly used by BARVINN.

6.5 Performance Analysis and Results

6.5.1 Experimental Setup

To illustrate the performance of BARVINN, we chose the ResNet9 image classifier model for the CIFAR10 dataset. We trained and quantized a ResNet9 model on CIFAR10 using LSQ [2] and used the residual distillation [218] technique to remove shortcut connections (Plain CNN models). In many image classification DNN models such as ResNet [6], the input to the first layer typically consists of less than 64 channels. Furthermore, due to sensitivity of the first and last layer to information loss, most state-of-the-art compression and quantization methods do not apply optimization on input and output layers [2], hence keeping these layers untouched and in full precision. We have adopted the same technique to compute first and last layers on the host or on the RISC-V controller.

Table 6.2 shows the performance of ResNet9 on CIFAR10 in the PyTorch framework. Once we were satisfied with the performance of our quantized model, we exported the trained model to ONNX and then used our code generator. Table 6.3 illustrates the per layer computation cost of running ResNet9 on BARVINN with 2-bit activations and weights. All convolutions use a padding of 1. As discussed before, we skipped running the first and last layer on BARVINN and we kept them in their original format. The overall computation takes 194,688 cycles to complete.

Our design was written in Verilog and synthesized using Xilinx Vivado 2021.1 for the Xilinx Alveo U250 accelerator card. Synthesis results for the RISC-V controller, the processing array, and the accelerator are presented in Table 6.4. Power consumption was estimated using the software tools in Vivado.

Table 6.2 ResNet9 with different bit precision on CIFAR10

ResNet9 Model	Precision	Accuracy	Size (Bytes)
Original	Fp32	90.8%	19605141
Plain-CNN	Fp32	91.1%	18912487
Quantized Plain-CNN	Int2	89.2%	1181360

Table 6.3 ResNet9 layers for CIFAR10 dataset and computation cost. All layers are quantized to 2-bit for activation and weights, except for the first and last layers

Layer	Input	Kernel	Output	Cycles
conv0	[3, 32, 32]	[64, 3, 3, 3]	[64, 32, 32]	N/A
conv1	[64, 32, 32]	[64, 64, 3, 3]	[64, 32, 32]	34560
conv2	[64, 32, 32]	[64, 64, 3, 3]	[64, 32, 32]	34560
conv3	[64, 32, 32]	[128, 64, 3, 3]	[128, 16, 16]	17280
conv4	[128, 16, 16]	[128, 128, 3, 3]	[128, 8, 8]	32256
conv5	[128, 8, 8]	[256, 128, 3, 3]	[128, 8, 8]	16128
conv6	[128, 8, 8]	[256, 256, 3, 3]	[256, 4, 4]	27648
conv7	[256, 4, 4]	[512, 256, 3, 3]	[256, 4, 4]	13824
conv8	[256, 4, 4]	[512, 512, 3, 3]	[512, 4, 4]	18432
fc	[512, 4, 4]	[10, 512]	[10]	N/A
			Total:	194688

6.5.2 Discussion

We compared BARVINN with FINN [213], which is a templated Vivado HLS C++ library of common DNN layers. Like BARVINN, FINN can generate hardware for arbitrary precision, but is not software programmable. Hence, once the FINN hardware is generated, the user cannot change the computation data stream. We attempted to compare the performance of BARVINN with FINN using the ResNet9 model we used earlier. However, at the time of writing, FINN supports simple linear topologies and we were not able to get performance metrics for our model. Instead, we used the available CIFAR10-CNV model from the FINN repository that was tuned for the FINN dataflow for our comparison. Table 6.5 shows the performance of BARVINN and FINN. For this experiment, we used different precisions for weights and activation. For both tools, we used the performance estimation numbers for frames per second (FPS). For FINN, we used the default folding configurations publicly available in FINN-example repository [219]. As illustrated in Table 6.5, we provide 7-15 times better throughput albeit with higher LUT usage. On the other hand, for higher bit precisions, FINN provides a better FPS/LUT, suggesting a scalable solution for bigger models.

We also compared the performance on a ResNet-50 model. Table 6.6 shows our estimated FPS for BARVINN executing in Pipelined mode along with reported performance for FINN [219] synthesized

for the Xilinx U250 and for FILM-QNN [100] synthesized for the Xilinx ZCU102 FPGA. While FINN has the highest FPS, BARVINN shows the best performance per Watt. According to the FINN-example repository [219], a fine-tuned ResNet50 model, requires more than 87% of Alveo U250 accelerator's resources. This shows the limits of FINN dealing with bigger models. BARVINN requires the same LUT usage regardless of the model size and bit-width.

Table 6.4 Post-synthesis resource utilization of BARVINN

Resource	PITO RISC-V	MVU Array	Overall
LUT	10454	190625	201079
BRAM	15	1312	1327
DSP	0	512	512
Dynamic Power	0.410 W	21.066 W	21.504 W
Frequency	250 MHz	250 MHz	250 MHz

Table 6.5 Estimated performance of running CNV model on CIFAR10 on Alveo U250 when different bit precision is used

	Bits (W/A)	kLUT	BRAM	DSP	FPS	FPS/ kLUT
Ours	1/1	201.1 (15.0%)	1327	512	61035	303.5
	1/2	201.1 (15.0%)	1327	512	30517	151.7
	2/2	201.1 (15.0%)	1327	512	15258	75.8
FINN	1/1	28.2 (2.1%)	150	0	7716	273.6
	1/2	19.8(1.47%)	103	0	2170	109.6
	2/2	24.3(1.81%)	202	0	2170	89.3

Table 6.6 Performance for ResNet-50 model on ImageNet

	Bits (W/A)	Clock Freq.	FPS	FPS/Watt
Ours	1/2	250 MHz	2296	106.8
FINN-R [219] [103]	1/2	178 MHz	2873	41.0
FILM-QNN [100]	4(8)/5	150 MHz	109	8.4

6.6 Conclusion

In this paper, we presented an FPGA-based DNN accelerator that supports arbitrary bit precision computations. We tested the performance of BARVINN over different DNN kernels and models with different bit precision. For model deployment, we developed a code generator tool that takes in a model in ONNX format and generates RISC-V assembly code for the controller. Compared

to other low precision accelerators, we provide a programmable solution which makes BARVINN more flexible. With the programmable MVUs, the user can run different models regardless of their size. BARVINN allows trading off throughput and latency by running DNN layers either in distributed or in pipeline modes. Unlike other low precision accelerators, our proposed solution offers implementing various trade-offs through software and the end user can control them for each individual layer without FPGA reconfiguration at run time. Compared to programmable accelerators, BARVINN was shown to provide a better throughput per Watt performance.

CHAPTER 7 ARTICLE 5 : QUARK: AN INTEGER RISC-V VECTOR PROCESSOR FOR SUB-BYTE QUANTIZED DNN INFERENCE

7.1 Prologue

7.1.1 Article Details

Quark: An Integer RISC-V Vector Processor for Sub-byte Quantized DNN Inference
MohammadHossein AskariHemmat, Theo Dupuis, Yoan Fournier, Nizar El Zarif, Matheus Cavalcante, Matteo Perotti, Frank Gurkaynak, Luca Benini, Francois Leduc-Primeau, Yvon Savaria, Jean-Pierre David.
Published at 58th International Symposium on Circuits and Systems conference (ISCAS) , May 25 2023.

7.1.2 Context

Quark was proposed as a part of "RISC-V Vector Processor for High-throughput Multidimensional Sensor Data Processing & Machine Learning Acceleration at the Edge" project. This project was a collaboration between Ecole Polytechnique Montreal, OpenHW, ETH Zurich and CMC Microsystems. The initial goal was, to make Ara [220] more efficient for machine learning tasks at the edge. As our first project milestone and to get an understanding of how different components within Ara work, MohammadHossein AskariHemmat helped the ETH team by adding fixed-point instructions defined in RISC-V vector ISA [221] that were not supported by Ara at the time. To make Ara more efficient for DNN inference at the edge, MohamamdHossein AskariHemmat proposed to add support for sub-byte computation to Ara. Since quantized neural networks do not require floating point operations, we removed floating point unit from the vector engine. With all the aforementioned modifications, compared to Ara, each quark lane is 2 times smaller and 1.9 times more power efficient compared to the ones of Ara.

7.1.3 Contributions

Based on his experience in BARVINN project, MohammadHossein AskariHemmat proposed to create a custom RISC-V vector processor tailored to run sub-byte quantized neural network models efficiently. Compared to BARVINN, Quark is a general purpose vector processor, designed to accelerate computation by performing arithmetic operations on multiple data elements simultaneously.

Since there are no support for sub-byte computation in RISC-V vector ISA [221], MohammadHossein AskariHemmat proposed to add custom instructions to accelerate sub-byte computation for bit-serial algorithm. In addition to custom instructions, Quark has no floating-point unit on the vector processor. For this project, Yoan Fournier with the help from Matheus Cavalcante Matteo Perotti and Frank Gurkaynak implemented Quark in Globalfoundries 22FDX FD-SOI technology. Theo Dupuis wrote DNN kernels with custom instructions. Throughout the project, Luca Benini, Yvon Savaria, and Jean-Pierre David provided supervision and valuable feedback, which greatly contributed to the final form of the paper and the project.

Quark: An Integer RISC-V Vector Processor for Sub-byte Quantized DNN Inference

7.2 Introduction

Vector processors are gaining traction for machine learning computations, thanks to their efficiency when executing tensor operations, combined with the flexibility of fully programmable instruction processing. Many Deep Neural Network (DNN) tasks can perform well with far lower precision than is typically used for operands of generic digital signal processing computations [2, 135, 222, 223]. Quantized models operate on much smaller processing and power budgets than their full precision counterparts, making them appealing options for edge applications. Even if using sub-byte networks can help speeding up computation in neural networks, to the best of our knowledge, there are currently no other vector processors that can efficiently store, load, and operate on sub-byte operands.

This paper makes the following contributions: first, we modified Ara [220], a RISC-V V compliant vector processor to support operations on sub-byte operands. To achieve this, we added custom instructions to accelerate low-bitwidth operations. Indeed, the majority of quantized DNN inference is performed only in the integer domain, hence, we removed the floating-point unit from Ara. This resulted in each lane of Quark being $2\times$ smaller and $1.9\times$ more power efficient than the ones in Ara. On the other hand, in quantized DNNs, to recover accuracy, there is a re-scaling step after each convolution or linear layer that requires floating-point operations. In Quark, higher precision operations can be performed in the CVA6 [224] scalar processor.

Second, we developed a set of custom instructions, specifically designed for running sub-byte DNNs. Then, using these custom instructions, we developed a small vector DNN runtime that provides common DNN kernels operating on sub-byte data. We show that compared to `Int8` and `FP32` kernels, the quantized kernels achieve higher performance. Finally, we have implemented Quark in GlobalFoundries' 22FDX Fully-Depleted Silicon-on-Insolator (FD-SOI) technology and provide its detailed power and area usage report.

The rest of the paper is organized as follows. In Section 7.3, we review the related works. In Section 7.4, we describe the architectural changes made to Ara to make it more efficient for running quantized neural networks. In Section 7.5, we provide performance comparisons between Quark and Ara, and an analysis of Quark's implementation results. Finally, Section 7.6 concludes this work.

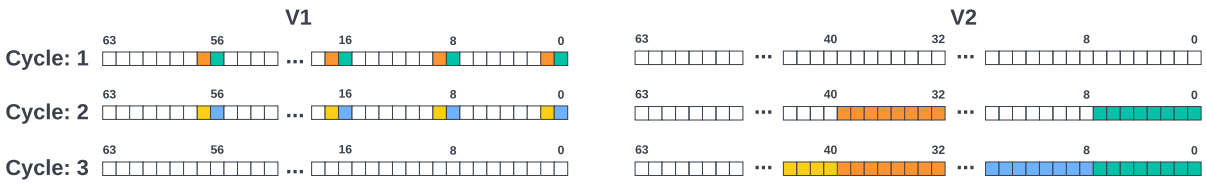


Figure 7.1 This figure shows the content of 8 vector elements of `v1` and `v2` when `vbitpack` instructions are executed consecutively on the same input and output vector registers. In this example, we are packing the content of `v1` into `v2` when 2 bit precision is used. ■ ■ ■ and ■ show different bit slices

7.3 Background and Related Works

Quantized neural networks are a well-studied research topic [102]. However, supporting sub-byte precision has not been as popular. In the following, we have divided the related works for these types of quantized neural networks into three different groups depending on how they are implemented: on custom accelerators, commodity CPUs and custom RISC-V Microprocessors.

7.3.1 Custom Accelerators

Since the inception of quantized neural networks, many custom accelerators based on ASIC and FPGA were proposed. Loom [145] supports sub-byte computation through bit-serial computation. Loom accelerates the inherent computation cost of bit-serial by using a large number of parallel bit-serial units that perform a 1-bit by 1-bit multiplication. Loom expects the input data to these parallel units to be in a bit-stream format. Because of that, Loom transposes data into a bit-stream format at each output unit, with a significant amount of overhead in area and power consumption. In [225] a DNN accelerator based on Loom was proposed and implemented in 65 nm technology, achieving 805 GOPS when operating on 8-bit precision operands. BitFusion [214] and BitBlade [215] use a set of parallel 2-bit by 2-bit multipliers followed by a shift accumulation unit to support sub-byte computation. To further optimize for area and power, BitBlade [215] replaces the shift accumulation units with bitwise summation. RaPiD [146] is an ultra low precision accelerator for AI training and inference. RaPiD supports 16 and 8-bit floating-point and 4 and 2-bit fixed-point and was implemented in 7 nm technology. It delivers a peak performance of 3.5 TOPS/W in its 8-bit floating-point mode and 16.5 TOPS/W in its 4-bit fixed-point mode.

There are examples of FPGA-based quantized neural network accelerators as well. FINN [213] is a framework that was originally designed for the acceleration of binary neural networks. It has now evolved to a data-flow style framework with support for low precision operations for DNN acceleration. FINN can process quantized neural networks in ONNX format. Then, FINN's front-end

compiler optimizes the computational graph for the FINN's back-end compiler, which uses an HLS hardware library to generate hardware supporting input quantized models. However, FINN requires the entire model to be implemented all at once on the FPGA, which limits its flexibility. BARVINN [22] is an FPGA based neural network accelerator that was designed to support arbitrary precision operations. It consists of a barrel RISC-V processor [21] used as a controller and an array of Matrix Vector Units (MVU) [98]. Each MVU has a set of configuration registers that can be programmed by the RISC-V processor through RISC-V's control status registers. This allows the MVU to perform a set of neural network specific operations in a pipelined fashion with any precision. The MVUs are connected to a crossbar that allows them to send the results of one MVU to another. Furthermore, BARVINN has a Python code generator library that can consume a quantized DNN in ONNX format and generates RISC-V code.

7.3.2 Sub-byte Precision on Commodity CPUs

In addition to custom hardware accelerators, there have been efforts to support low-precision inference on commodity CPUs. In [101], the authors added support for bit-serial convolution in TVM to generate code for Arm devices. They exploit the vector instructions of Arm devices to overcome the computationally expensive task of formatting tensors in bit-stream format. They show that on a Cortex-A72 CPU in a Raspberry Pi 4B device, the relative speedup over 32-bit floating-point kernels for 1-bit and 2-bit precision is 6.2× and 1.7×, respectively. With ULPPACK [226], the authors propose a novel packing method that allows a commodity CPU to operate on sub-byte precision. In this packing method, the quantized inputs are packed into CPU's registers with enough space to allow for overflow. In [226], the authors show that on a Cortex-A72 in the Raspberry Pi 4B CPU, compared to [101], ULPPACK has a better performance for 3-bit or higher precision inputs.

7.3.3 Custom RISC-V MPUs

Dustin [227] is a 16-core RISC-V processor based on OpenHW Group's CV32E40P [228, 229] cores. Each core has a special dot product unit that can be programmed through RISC-V's control status registers (CSRs) to perform a dot product operation with different precision (2, 4, 8 or 16-bit operations are supported). Dustin was taped-out in 65 nm CMOS technology and achieves a peak performance of 58 GOPS and a peak efficiency of 1.15 TOPS/W. Darkside [230] is a heterogeneous RISC-V compute cluster specifically designed for DNN inference at the edge. It has 8 RISC-V cores enhanced with 2-bit to 32-bit mixed-precision integer arithmetic. To improve performance, Darkside is equipped with a specialized datamover, a 16 bit floating point Tensor Product Engine and an accelerator for computing depthwise convolution. Darkside was implemented in 65 nm CMOS technology and achieves a peak integer performance of 65 GOPS on 2-bit integer kernels.

Compared to the existing works, Quark is a vector processor that extends the RISC-V ISA by adding new instructions to accelerate packing methods for bit-serial computation. As we will discuss in Section 7.4, compared to Ara, Quark is a much slimmer processor since it does not require any vectorized floating-point operations. Compared to ASIC or FPGA based accelerators and due to the nature of general processors, Quark is very flexible and more likely to support any type of neural network that might come up in the literature in the future. Finally, none of the previous works used a vector RISC-V to provide sub-byte support for quantized DNNs.

7.4 Architecture

Quark's architecture is based on Ara, a modular open-source vector processor recently updated to the RISC-V V 1.0 vector ISA that is able to reach almost its theoretical peak performance when running matrix multiplication between "large enough" matrices [231]. The Ara system is a decoupled vector architecture composed of a CVA6, the scalar RV64GC core, and Ara, its vector engine. Each unit has independent access to the upper level of the memory hierarchy via a shared AXI bus.

CVA6, being the only unit capable of accessing the instruction side of the memory, fetches both the scalar and vector instructions from its private L1 instruction cache, executes the former ones, and dispatches the latter ones to Ara. Since CVA6 is an in-order issue/commit processor that can still execute instructions out-of-order by means of a scoreboard, the dispatch of vector instructions to Ara happens non-speculatively as soon as the instruction reaches the top of the scoreboard. Then, CVA6 waits for Ara's acknowledgment and answer, before committing the instruction. This is a fire-and-forget dispatch since Ara, if there is no result to return, acknowledges CVA6 immediately after checking that no exceptions can occur. Ara's architecture matured through time following the RISC-V V specifications, from RVV0.5 to RVV1.0.

Ara has been designed to accelerate a wide range of computations by means of vectorization. However, it only supports standard data types. In quantized neural networks, we try to avoid floating-point operations since they are computationally costly. Therefore, our design focuses on maximizing the throughput of integer computations, paying only a negligible degradation in accuracy. This allowed us to remove Ara's floating-point units and to save precious area to make room for our custom bit-serial computation instructions. In the following sub-sections, we briefly discuss these modifications.

7.4.1 Bit-Serial Computation

Figure 7.2 illustrates a typical computation flow for quantized neural network inference. One way to accelerate the inference of neural networks is to accelerate convolutions or matrix multiplications by

using low precision operations. However, as it can be seen, the rest of the computation has to happen in full precision. Since the computation complexity of convolution or matrix multiplication kernels is much higher than the rest of the operations in Figure 7.2, it makes sense to accelerate them first. As discussed in Section 7.3, there are many ways to support low precision and particularly sub-byte operations. With Quark, this is achieved by exploiting its vector processing capacity.

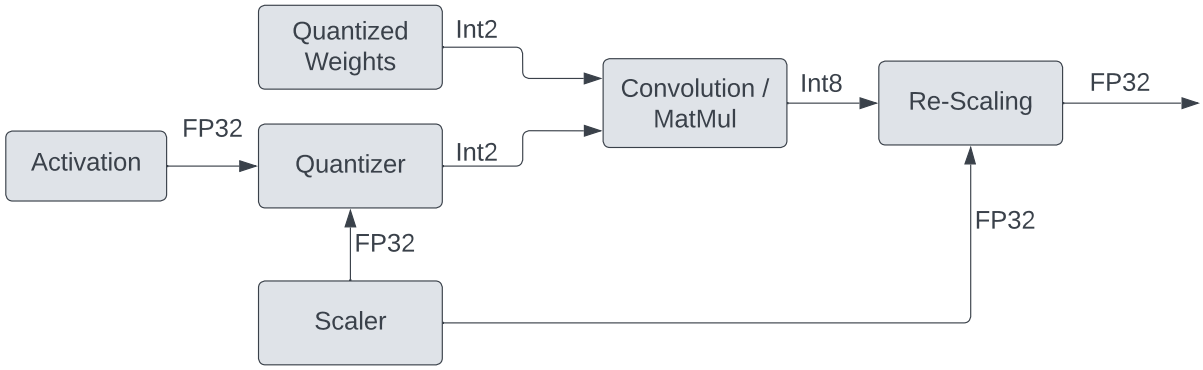


Figure 7.2 Typical computation graph in a forward path of a quantized neural network. Here, we show the computation flow for 2 bit inference

As mentioned earlier, Quark extended the RISC-V vector ISA with custom instructions to speed up bit-serial computation and packing. To understand how these bit-serial computations operate, let us consider a vector-vector multiplication:

$$w \cdot a = \sum_{n=0}^{N-1} \sum_{m=0}^{M-1} 2^{n+m} \text{popcnt}(w_m \wedge a_n) \tag{7.1}$$

Equation (7.1) describes a vector-vector multiplication in bit-serial format between two input vectors w and a, where m and n are the precision used for w and a, respectively. To perform this operation in a vector processor, we need logical AND, popcount, and shift-and-accumulate operators. Moreover, Equation (7.1) expects the data to be in bit-stream format, and since this data transposition has to happen for each input of a convolution or linear layer, this data transformation should be fast to avoid making it a bottleneck to the whole computation.

Although the base RISC-V vector ISA includes vectorized AND, it does not support the other operators needed. To accelerate the bit-serial computation, we added three custom instructions as follows: `vpopcnt`, `vshacc` and `vbitpack`. Our popcount operator requires an instruction able to count the number of bits at 1 for every element of the input vector; however, the base RISC-V vector ISA can only count the global number of 1s over the entire vector length. Our special `vpopcnt`

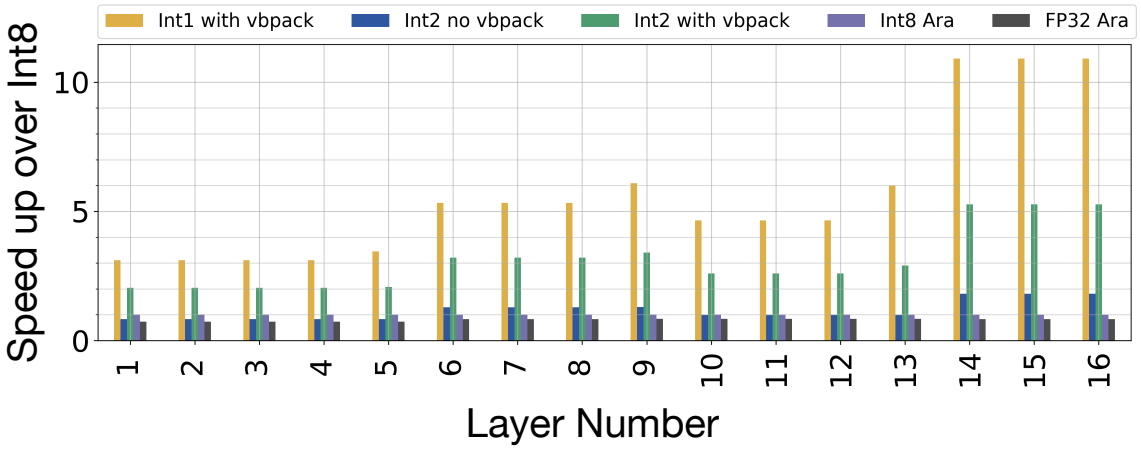


Figure 7.3 Per layer relative speed up when running ResNet18 on CIFAR100 with batch size of 1, on Quark with `Int1` and `Int2` over Ara with `Int8`

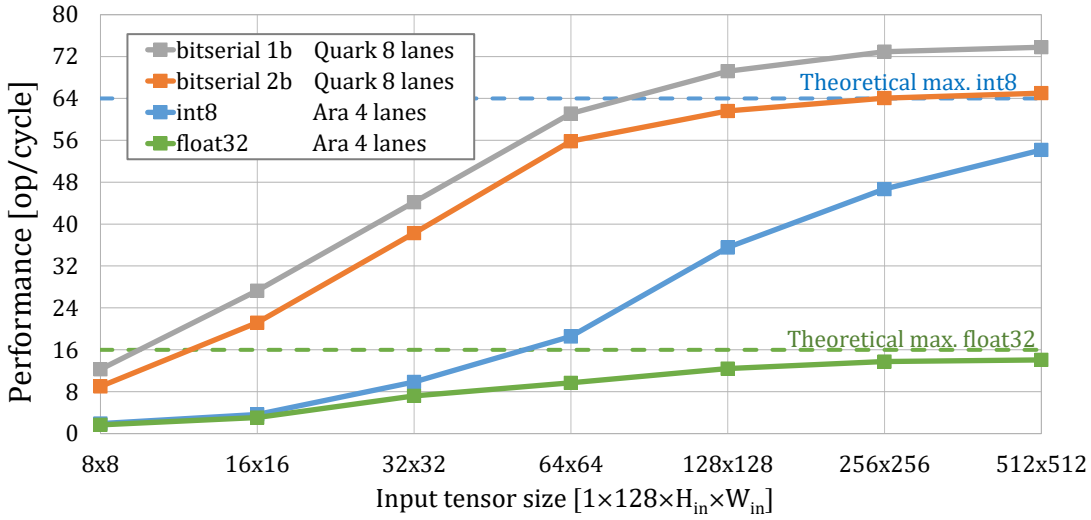


Figure 7.4 Roofline plot for Quark with 8 lanes and Ara with 4 lanes on Conv2d with a 3×3 kernel

instruction performs popcount on each vector element. We also developed `vshacc` to fuse the shift and accumulation operations in a single instruction. Finally, we developed a custom instruction to accelerate bit-packing operations. `vbitpack` slices the elements of a vector register into bits and then packs these bits into an output register. To accumulate the bit slices of previous elements, each call to `vbitpack` shifts the target register to the left and then performs the packing operations. Figure 7.1 illustrates this operation in detail.

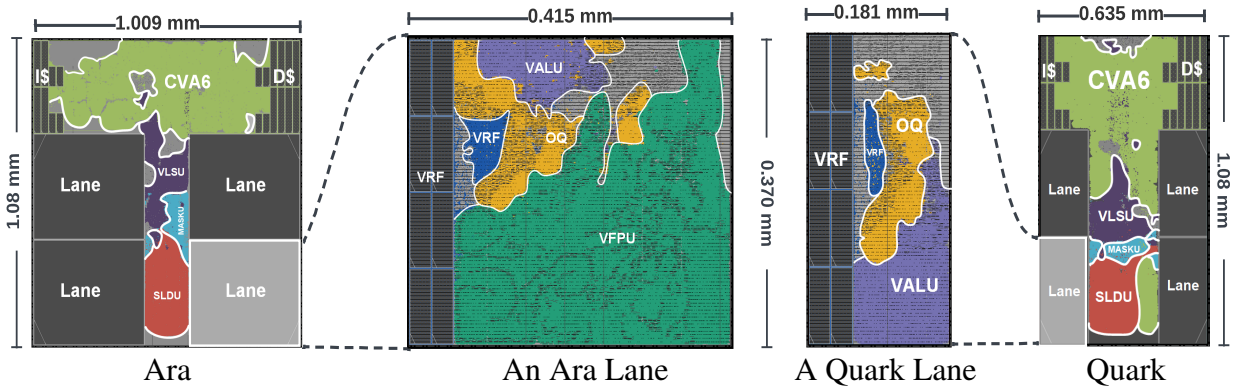


Figure 7.5 Ara and Quark system placed-and-routed designs. ■ is vector register file related , ■ is operand queue, ■ is the vector floating-point unit, ■ is the vector register file and ■ is the vector ALU

Table 7.1 Quantization of ResNet18 Using LSQ [2] Quantization method

Dataset	Model	Precision (W/A)	Accuracy	Size (MB)
CIFAR 100	ResNet18	LSQ(1/1)	57.32	1.45
		LSQ(2/2)	76.81	2.89
		LSQ(8/8)	78.45	10.87
		FP32	76.82	42.80

7.5 Results and Performance Analysis

7.5.1 Performance Analysis

To demonstrate the benefits of adding our custom instructions, we benchmarked Ara and Quark with different kernel sizes and shapes. To do this, we developed customized bit-serial programs for conv2d, matrix multiplication, and other common kernels needed to run typical neural networks on Quark.

We benchmarked the system performance by running ResNet18 (with CIFAR100 dataset) on both Ara and Quark. To simulate, we benchmarked both Ara and Quark with Questasim 10.7c. For each experiment, we used CVA6's cycle CSRs to know exactly how many clock cycles each kernel takes. To preserve the accuracy of the model, we used full precision data type for input and output layers. Figure 7.3 illustrates the per-layer performance from running ResNet18 on Quark and Ara under different precisions. We ran ResNet18 on Ara with `Int8` and `FP32` data types and on Quark with `Int1`, `Int2` data types. To show the importance of using a specialized packing operation, we ran the `Int2` experiment once with RISC-V vector instructions and then with the custom `vbitpack` instruction. As Figure 7.3 shows, running ResNet18 with `Int1` precision on Quark outperforms

`Int8` on Ara in each layer. However, as Table 7.1 shows, quantization of ResNet18 with 1-bit precision comes with a significant accuracy degradation. On the other hand, with 2-bit quantization, the accuracy drop is negligible, but the performance boost over `Int8` is significant. For 2-bit quantization, we measured the performance of each layer with and without `vbitpack` instruction. Finally, although `Int2` without the `vbitpack` instruction performs better than `Int8` on average, the improvement is not significant. On the other hand, Figure 7.3 shows that `Int2` with `vbitpack` instruction outperforms `Int8` by an average of $5.67\times$.

7.5.2 Physical Implementation Results

We implemented Quark and Ara with GlobalFoundries' 22FDX FD-SOI technology. For both Quark and Ara, we used 4 lanes, a vector length of 4096 bits (i.e., 16 KiB of Vector Register File), and an equally-configured CVA6. We used Synopsys Design Compiler 21.06 for synthesis and Cadence Innovus 21.14 for physical implementation. In Table 7.2, we report Innovus's static power analysis for both Quark and Ara, in typical operating conditions. Figure 7.5 illustrates the placed and routed design for both Ara and Quark. Based on the results in Figure 7.5 and Table 7.2, each Quark lane is about $2.3\times$ smaller and consumes $1.9\times$ less power than Ara. Nevertheless, both cores can operate at the same 1 GHz clock frequency. Finally, Figure 7.4 illustrates the roofline plot for Quark and Ara on `conv2d` with a 3×3 kernel. In this graph, considering the same power and area budget for Ara and Quark as shown in Table 7.2, Quark outperforms Ara in all the input tensor sizes.

Table 7.2 Physical Implementation of Ara and Quark

	Ara	Quark	
Number of Lanes	4	4	8
VRF Size [KiB]	16	16	32
Lane Cell Area [mm^2]	0.120	0.051	0.046
Die Area [mm^2]	1.09	0.69	1.09
TT Frequency [GHz]	1.05	1.05	1.00
Core power per lane [mW]	229	119	97

7.6 Conclusion

This paper introduced Quark, an integer RISC-V vector processor based on Ara. It extends the RISC-V vector ISA by adding custom instructions for sub-byte computations in DNNs. We modified Ara by removing the floating-point unit and adding custom instructions to accelerate bit-serial computations. Our simulation shows that on ResNet18 with 1-bit and 2-bit precision, Quark is 5.7x

and 3.5x faster than Ara with 8-bit precision on average for each kerenel. Finally, our implementation results of Quark in GlobalFoundries' 22 nm technology show that Quark's lanes are 2.3x smaller than the corresponding ones in Ara.

CHAPTER 8 GENERAL DISCUSSION

As we discussed in Chapter 2, our research question in this thesis is "*What is the effect of quantization on neural network learning and how to run quantized neural networks efficiently?*". To this end, throughout this dissertation, we have presented five articles that build upon each other to provide an answer to our research quesiton.

In the first article, we introduced a novel fixed-point quantization methodology aimed at the medical image segmentation tasks. We demonstrated the viability of quantization in such critical applications by evaluating our technique on a challenging task of segmenting grey matter in the spinal cord. Our results showed that the accuracy loss incurred by the quantization process was negligible, indicating the practical utility of our proposed method in critical medical applications.

In addition to [166] that was discussed in Chapter 3, there have been other research papers put forth that enhance our investigation. In [109, 110, 150, 151] authors provide a study on accelerating computation for image segmentation for edge computing using quantization.

MedQ [109] proposes a lossless ultra-low-bit CNN quantization framework for volumetric medical image segmentation. They design an adaptive quantizer and propose an architectural method called radical residual connection to overcome information loss and gradient mismatch caused by the non-differentiable quantization function. Compared to our study, MedQ's focus is on 3D segmentation and ultra-low precision quantization.

Like MedQ, in [151] authors focused on ultra-low bit quantization. They proposed a binarized encoder-decoder network (BEDN) and a binarized deconvolution engine (BiDE) to accelerate semantic segmentation in low-power and real-time environments. The approach uses custom hardware to implement a binarized segmentation network with zero-aware binarized deconvolution and batch normalization embedded binary activation. The BEDN has a simple structure and full-binarization, resulting in reduced computational and memory demands while maintaining acceptable accuracy. The approach was implemented on Xilinx ZU7EV FPGA and achieved a performance of 1.682 Tera operations per second (TOPS). However, they did not provide any results on medical datasets.

Unlike other methods discussed, in [110], authors propose a Mixed-Precision quantization scheme that uses different quantization levels for different layers of U-Net model. In their propose method, they utilizes skip-supervised quantization aware training for quantizing U-Net model. This scheme addresses the problem of not sufficiently exploiting the skip connection between the encoder and decoder structures in existing quantization methods. Split convolution enables fine-grained

bit-width allocation for mixed-precision quantization, while skip-supervised quantization aware training enhances the skip features in finetuning. Experimental results on four medical image datasets show that this proposed scheme outperforms existing U-Net quantization strategy and recent general-purpose mixed-precision quantization methods in medical image segmentation. Compared to our proposed method, they achieve better accuracy performance with 1.9 /3.1 (weights/activation) Mixed-Precision.

Although the quantization algorithm proposed in Chapter 3 worked well under multiple datasets, we only tested this algorithm on U-Net architecture. We did not explore the effect of quantization on more efficient segementation models. For instance, unlike U-Net, DeepLab [232] uses atrous convolutions (also known as dilated convolutions) to increase the receptive field without increasing the number of parameters. This can be more efficient than U-Net for large images. Also, as described, since the publication of our methodology, there have been other methods that improved the results we got initially. However, throughout the course of this project, two important realizations emerged. Firstly, certain quantization levels appeared to induce a form of regularization, leading to improved accuracy when compared to full-precision models. Secondly, the use of quantization to achieve greater computational efficiency was found to be limited by the hardware available. These findings highlight the potential benefits of quantization for machine learning tasks, as well as the importance of hardware considerations when implementing such techniques.

In the second article, we built on the findings from our initial project and proposed a hypothesis that quantization provides a regularization effect. To test our hypothesis, we conducted a series of experiments. To isolate the effect of quantization, we trained full precision models from scratch without any other form of regularization. We then utilized these full-precision models to train quantized models, while varying the quantization levels, models, tasks, and datasets. To measure the regularization effect, we applied various augmentations to our datasets and calculated the accuracy of vanilla and augmented datasets for both full precision and quantized models. We then calculated the generalization by subtracting the accuracy of the vanilla dataset from the accuracy of the augmented dataset for both types of models. Our results showed that 8-bit quantization exhibited a consistent regularization effect across various models, datasets, and tasks. However, lower quantization levels were found to be less stable than 8-bit quantization.

To investigate the effectiveness of quantization as a regularization technique, we assessed whether quantized models outperform full-precision models on corrupted data. Although we observed that quantized models exhibited lower generalization gaps on corrupted data, the versatility of this effect remains unclear. One approach to studying this effect would be to use as many models as possible, which would be extremely resource-intensive. An alternative method involves utilizing various generalization metrics proposed in literature [233–235]. Although not commonly used, some of

these metrics establish causal relationships between the metrics and generalization [233]. In the case of quantization, in addition to the experiments conducted in Chapter 4, we could employ these metrics to examine the generalization of quantized models compared to base models.

Another aspect of our work that could be improved is regarding the way we modeled quantization as noise. In Equation 4.4, for simplification, we assumed that the induced noise generated by quantization comes from a normal distribution $\delta \sim \mathcal{N}(0, \sigma I)$. To test our assumptions, using LSQ [2] quantization and using Equation 2.16, we quantized Resnet50 model on Imagenet dataset with 8, 4 and 2 bit precisions and calculated the quantization error as below:

$$\begin{aligned} \bar{t} &= \lfloor clip(\frac{t}{s}, -Q_N, Q_P) \rceil \\ \hat{t} &= \bar{t} \times s \\ error_q &= t - \hat{t} \end{aligned}$$

By using the equation presented above, we computed the quantization error for every layer of Resnet50 and observed that, for 8, 4, and 2-bit quantization, the quantization error deviated from a normal distribution in 89.12%, 39.3%, and 1.9% of all layers. Which means that our assumptions is not accurate for all quantization levels and we need a better way of modelling quantization noise.

In the last three articles of our dissertation, we proposed three custom hardware designs aimed at accelerating computation for sub-byte quantized models. In Chapter 5, we introduced PITO, which is a custom RISC-V processor designed for optimal area usage. PITO comprises eight RISC-V hardware threads, which were used in Chapter 6 to control eight Matrix Vector Units (MVUs) inside BARVINN, a DNN accelerator for quantized neural networks with arbitrary precision support. In BARVINN, each MVU is capable of performing Convolution, Matrix Multiplication, Pooling/ReLU, and quantization in arbitrary precision. Jobs can be assigned to any processing element by programming the associated hardware thread on PITO. To facilitate programming for each processing element, we developed a code generator tool that accepts a quantized model defined in ONNX format and generates RISC-V code to be executed on PITO.

In Chapter 7, we explored how a commodity vector processor can be optimized to run quantized neural networks with sub-byte precision. We took a RISC-V vector processor called Ara and customized it for accelerating computation for quantized neural networks. By adding specialized instructions and removing vector floating-point units, we optimized Ara for both area and speed, and named it Quark. Quark has shown to be efficient in running quantized models at sub-byte precision, particularly for 1-bit and 2-bit quantized models, by accelerating the computation of Conv2d over various ranges of inputs and kernel sizes.

BARVINN and Quark were designed for different workloads and power budgets. BARVINN has

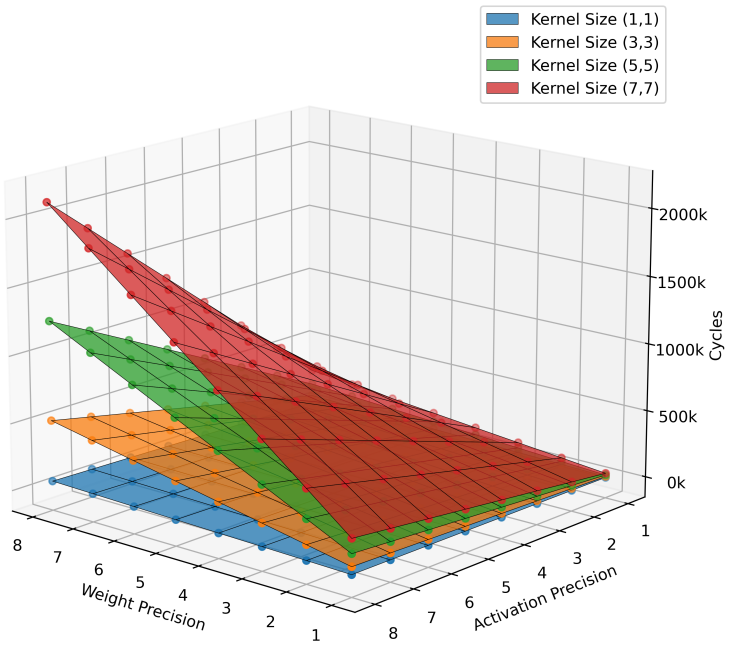


Figure 8.1 Number of cycles to compute a 2D convolution over different weight and activation precision. Each surface corresponds to kernel size.The input tensor size is $1\times64\times32\times32$. Note that only one MVU was used

a customized data flow and can perform arbitrary precision computation with minimal overhead, while Quark is a vector processor capable of running modern operating systems in addition to supporting sub-byte computation. However, both devices are most efficient when performing 2-3 bit precision computation, as the bit-serial computation algorithm used for arbitrary precision computation is a limiting factor. Figure 8.1 demonstrate this performance bottleneck where we show the performance of running convolutions on BARVINN using different precision. The input tensor size is $1\times64\times32\times32$, with padding of 1 and a stride of 1. For a 3×3 kernel, a typical scalar processor requires 580,608 multiplications and 516,096 additions to compute this convolution. Assuming a modern processor performs a multiply accumulation in a single instruction and requires only one cycle to complete, a scalar processor requires 580,608 cycles to complete this task. In BARVINN, a single MVU performs this task in only 32,400 clock cycles when 2 bit precision is used for both weights and activations. For lower latency applications, the MVU array can be programmed to compute this kernel in only 4050 clock cycles. While this analysis omits other factors such as memory transfers, cache miss rates, etc., it shows that BARVINN performs low bit-count operations in much lower number of cycles. Another factor to take into consideration is that these results are applicable to each MVU. With a proper job scheduling, all the MVUs can work simultaneously on bigger kernels or can be pipelined back-to-back to increase throughput.

However, as the required precision for computation increases, BARVINN's performance deteriorates.

Other methods exist, such as those discussed in [236,237], that enable arbitrary precision computation with different memory and computation bounds. As shown in [237] paper, the bit-serial algorithm has significant speed advantages when it comes to running binary and low-precision models on Raspberry Pi4 devices. Specifically, bit-serial method is found to be 2.7 times faster than ULPPACK for binary models (using both binary weights and binary activations), and 1.13 times faster for models with 2-bit precision (again, for both weights and activations). Furthermore, bit-serial is also faster than ULPPACK when only 1 bit is used for either weight or activation, and this speed advantage persists up to 5 bits for the other parameter (activation or weight). However, for all other precision combination, ULPPACK is shown to be better. These results are consistent with data provided in Figure 8.1. Future work on BARVINN and Quark could explore such methods to provide additional computation capabilities.

CHAPTER 9 CONCLUSIONS AND RECOMMENDATIONS

Deep learning models have achieved remarkable success in various fields such as image and speech recognition and natural language processing. However, running these models on edge devices such as smartphones, cameras, and drones is still a challenging task due to their computation and power limitations. Several methods have been proposed in the literature to address this issue, including pruning and quantization. In this dissertation, we focused on quantization as a method to accelerate computation in neural networks. Through Chapter 3 to 7, we discussed our contributions to advancing the understanding of the impact of quantization on deep learning models and how it can be effectively utilized in real-world applications using custom hardware. Our contributions can be summarized as follows:

- We proposed a fixed-point quantization technique to perform medical image segmentation on edge devices.
- We showed that the regularization effect of quantization is more pronounced in 8-bit quantization levels. This conclusion was obtained by conducting empirical studies to demonstrate that 8-bit quantization can improve the generalization performance of deep learning models.
- We proposed and developed a custom hardware platform that supports sub-byte quantization levels to accelerate computation in neural networks.
- Finally, we discussed the necessary changes for a general-purpose vector processor to facilitate sub-byte quantization and subsequently constructed such a processor.

While our study made several contributions to the field of quantization, it also has several limitations. One of these limitations is that we were unable to provide a detailed theoretical analysis of the mechanics underlying how quantization improves model generalization. While we empirically demonstrated that 8-bit quantization is a reliable form of regularization, we were not able to show analytically why this is the case. This is an area that requires further investigation and exploration. Another limitation is that our custom hardware solutions focused on sub-byte quantization levels. As described in Section 8, the bit-serial algorithm used in our custom hardware is also a limiting factor to achieve better performance for higher bit precision.

Despite these limitations, the topic of quantization, including sub-byte quantization, remains a prominent area of research. Open-source platforms such as BARVINN can facilitate the testing and validation of quantization methods on real hardware, enabling researchers to gain new insights and

advance the field. Future research can explore the mechanisms underlying the regularization effect of quantization and investigate the potential of sub-byte quantization to improve the performance of deep learning models.

In conclusion, our research focused on advancing the understanding of the impact of quantization on deep learning models and how it can be effectively utilized in real-world applications using custom hardware. We developed a custom hardware platform that supports sub-byte quantization levels and conducted empirical studies to demonstrate that 8-bit quantization can improve the generalization performance of deep learning models. While our study has limitations, it provides a foundation for future research to explore the mechanisms underlying the regularization effect of quantization and investigate the potential of sub-byte quantization to improve the performance of deep learning models.

REFERENCES

[1] M. Horowitz, "1.1 computing's energy problem (and what we can do about it)," *2014 IEEE International Solid-State Circuits Conference Digest of Technical Papers (ISSCC)*, pp. 10–14, 2014.

[2] 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. [Online]. Available: https://openreview.net/forum?id=rkgO66VKDS

[3] S. J. Kwon, D. Lee, B. Kim, P. Kapoor, B. Park, and G.-Y. Wei, "Structured compression by weight encryption for unstructured pruning and quantization," in *2020 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, 2020, pp. 1906–1915.

[4] T. Brown, B. Mann, N. Ryder, M. Subbiah, J. D. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, S. Agarwal, A. Herbert-Voss, G. Krueger, T. Henighan, R. Child, A. Ramesh, D. Ziegler, J. Wu, C. Winter, C. Hesse, M. Chen, E. Sigler, M. Litwin, S. Gray, B. Chess, J. Clark, C. Berner, S. McCandlish, A. Radford, I. Sutskever, and D. Amodei, "Language models are few-shot learners," in *Advances in Neural Information Processing Systems*, H. Larochelle, M. Ranzato, R. Hadsell, M. Balcan, and H. Lin, Eds., vol. 33. Curran Associates, Inc., 2020, pp. 1877–1901. [Online]. Available: https://proceedings.neurips.cc/paper/2020/file/1457c0d6bfcb4967418bfb8ac142f64a-Paper.pdf

[5] O. Russakovsky, J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma, Z. Huang, A. Karpathy, A. Khosla, M. Bernstein, A. C. Berg, and L. Fei-Fei, "ImageNet Large Scale Visual Recognition Challenge," *International Journal of Computer Vision (IJCV)*, vol. 115, no. 3, pp. 211–252, 2015.

[6] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2016.

[7] A. Canziani, A. Paszke, and E. Culurciello, "An analysis of deep neural network models for practical applications," *CoRR*, vol. abs/1605.07678, 2016. [Online]. Available: http://arxiv.org/abs/1605.07678

[8] I. Hubara *et al.*, "Quantized neural networks: Training neural networks with low precision weights and activations," *JMLR*, vol. 18, no. 187, pp. 1–30, 2018. [Online]. Available: http://jmlr.org/papers/v18/16-456.html

[9] 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 *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2018.

[10] 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.

[11] E. Frantar and D. Alistarh, "SPDY: Accurate pruning with speedup guarantees," in *Proceedings of the 39th International Conference on Machine Learning*, ser. Proceedings of Machine Learning Research, K. Chaudhuri, S. Jegelka, L. Song, C. Szepesvari, G. Niu, and S. Sabato, Eds., vol. 162. PMLR, 17–23 Jul 2022, pp. 6726–6743. [Online]. Available: https://proceedings.mlr.press/v162/frantar22a.html

[12] M. Nonnenmacher, T. Pfeil, I. Steinwart, and D. Reeb, "SOSP: efficiently capturing global correlations by second-order structured pruning," *CoRR*, vol. abs/2110.11395, 2021. [Online]. Available: https://arxiv.org/abs/2110.11395

[13] N. P. Jouppi *et al.*, "In-datacenter performance analysis of a tensor processing unit," in *Proceedings of the 44th Annual International Symposium on Computer Architecture*, ser. ISCA '17. New York, NY, USA: ACM, 2017, pp. 1–12. [Online]. Available: http://doi.acm.org/10.1145/3079856.3080246

[14] Y.-H. Chen, J. Emer, and V. Sze, "Eyeriss: A spatial architecture for energy-efficient dataflow for convolutional neural networks," in *Proceedings of the 43rd International Symposium on Computer Architecture*, ser. ISCA '16. Piscataway, NJ, USA: IEEE Press, 2016, pp. 367–379. [Online]. Available: https://doi.org/10.1109/ISCA.2016.40

[15] Y. Chen, T. Luo, S. Liu, S. Zhang, L. He, J. Wang, L. Li, T. Chen, Z. Xu, N. Sun, and O. Temam, "Dadiannao: A machine-learning supercomputer," in *2014 47th Annual IEEE/ACM International Symposium on Microarchitecture*, Dec 2014, pp. 609–622.

[16] K. Abdelouahab, M. Pelcat, J. Sérot, and F. Berry, "Accelerating CNN inference on fpgas: A survey," *CoRR*, vol. abs/1806.01683, 2018. [Online]. Available: http://arxiv.org/abs/1806.01683

[17] V. Gokhale, A. Zaidy, A. X. M. Chang, and E. Culurciello, "Snowflake: A model agnostic accelerator for deep convolutional neural networks," *CoRR*, vol. abs/1708.02579, 2017. [Online]. Available: http://arxiv.org/abs/1708.02579

[18] S. I. Venieris and C. Bouganis, "fpgaconvnet: Mapping regular and irregular convolutional neural networks on fpgas," *IEEE Transactions on Neural Networks and Learning Systems*, vol. 30, no. 2, pp. 326–342, 2019.

[19] M. AskariHemmat, S. Honari, L. Rouhier, C. S. Perone, J. Cohen-Adad, Y. Savaria, and J.-P. David, "U-net fixed-point quantization for medical image segmentation," in *Medical Imaging and Computer Assisted Intervention (MICCAI), Hardware Aware Learning Workshop (HAL-MICCAI) 2019*. Springer, 2019.

[20] M. AskariHemmat, R. A. Hemmat, A. Hoffman, I. Lazarevich, E. Saboori, O. Mastropietro, S. Sah, Y. Savaria, and J.-P. David, "Qreg: On regularization effects of quantization," 2022. [Online]. Available: https://arxiv.org/abs/2206.12372

[21] M. AskariHemmat, O. Bilaniuk, S. Wagner, Y. Savaria, and J.-P. David, "RISC-V barrel processor for deep neural network acceleration," in *2021 IEEE International Symposium on Circuits and Systems (ISCAS)*, 2021, pp. 1–5.

[22] M. AskariHemmat, O. Bilaniuk, S. Wagner, Y. Hariri, Y. Savaria, and J.-P. David, "BARVINN: Arbitrary precision DNN accelerator controlled by a RISC-V CPU," in *28th Asia and South Pacific Design Automation Conference, ASP-DAC 2023*, 2022.

[23] M. AskariHemmat, T. Dupuis, Y. Fournier, N. E. Zarif, M. Cavalcante, M. Perotti, F. Gurkaynak, L. Benini, F. Leduc-Primeau, Y. Savaria, and J.-P. David, "Quark: An integer risc-v vector processor for sub-byte quantized dnn inference," in *2023 IEEE International Symposium on Circuits and Systems (ISCAS)*, 2023.

[24] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural computation*, vol. 9, no. 8, pp. 1735–1780, 1997.

[25] D. Deuermeyer *et al.*, "Release notes for intel distribution of openvino toolkit 2019," accessed on Jun 13th 2019.

[26] K. Hornik, M. Stinchcombe, and H. White, "Multilayer feedforward networks are universal approximators," *Neural networks*, vol. 2, no. 5, pp. 359–366, 1989.

[27] 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.

[28] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. MIT Press, 2016, http://www.deeplearningbook.org.

[29] P. Swietojanski, A. Ghoshal, and S. Renals, "Convolutional neural networks for distant speech recognition," *IEEE Signal Processing Letters*, vol. 21, no. 9, pp. 1120–1124, 2014.

[30] O. Ronneberger, P. Fischer, and T. Brox, "U-net: Convolutional networks for biomedical image segmentation," in *International Conference on Medical image computing and computer-assisted intervention*. Springer, 2015, pp. 234–241.

[31] V. Badrinarayanan *et al.*, "Segnet: A deep convolutional encoder-decoder architecture for image segmentation," *TPAMI*, vol. 39, no. 12, pp. 2481–2495, Dec 2017.

[32] J. Redmon, S. K. Divvala, R. B. Girshick, and A. Farhadi, "You only look once: Unified, real-time object detection," *CoRR*, vol. abs/1506.02640, 2015. [Online]. Available: http://arxiv.org/abs/1506.02640

[33] Y. LeCun and Y. Bengio, *Convolutional Networks for Images, Speech, and Time Series*. Cambridge, MA, USA: MIT Press, 1998, p. 255–258.

[34] G. Hegde, Siddhartha, N. Ramasamy, and N. Kapre, "Caffepresso: An optimized library for deep learning on embedded accelerator-based platforms," 2016.

[35] A. Krizhevsky *et al.*, "Imagenet classification with deep convolutional neural networks," in *Advances in Neural Information Processing Systems 25*. Curran Associates, Inc., 2012, pp. 1097–1105. [Online]. Available: http://papers.nips.cc/paper/4824-imagenet-classification-with-deep-convolutional-neural-networks.pdf

[36] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Kopf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala, "Pytorch: An imperative style, high-performance deep learning library," in *Advances in Neural Information Processing Systems 32*, H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, Eds. Curran Associates, Inc., 2019, pp. 8024–8035. [Online]. Available: http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf

[37] S. Chetlur, C. Woolley, P. Vandermersch, J. Cohen, J. Tran, B. Catanzaro, and E. Shelhamer, "cudnn: Efficient primitives for deep learning," *CoRR*, vol. abs/1410.0759, 2014. [Online]. Available: http://arxiv.org/abs/1410.0759

[38] L. Du, Y. Du, Y. Li, J. Su, Y. Kuan, C. Liu, and M. 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.

[39] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," in *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*, Y. Bengio and Y. LeCun, Eds., 2015. [Online]. Available: http://arxiv.org/abs/1409.1556

[40] Y. LeCun and C. Cortes, "MNIST handwritten digit database," 2010. [Online]. Available: http://yann.lecun.com/exdb/mnist/

[41] 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," *CoRR*, vol. abs/1704.04861, 2017. [Online]. Available: http://arxiv.org/abs/1704.04861

[42] B. Zoph, V. Vasudevan, J. Shlens, and Q. V. Le, "Learning transferable architectures for scalable image recognition," *CoRR*, vol. abs/1707.07012, 2017. [Online]. Available: http://arxiv.org/abs/1707.07012

[43] E. Real, S. Moore, A. Selle, S. Saxena, Y. L. Suematsu, Q. V. Le, and A. Kurakin, "Large-scale evolution of image classifiers," *CoRR*, vol. abs/1703.01041, 2017. [Online]. Available: http://arxiv.org/abs/1703.01041

[44] T. Elsken, J. H. Metzen, and F. Hutter, "Neural architecture search: A survey," *J. Mach. Learn. Res.*, vol. 20, pp. 55:1–55:21, 2019.

[45] Y. LeCun, J. S. Denker, and S. A. Solla, "Optimal brain damage," in *Advances in Neural Information Processing Systems 2*, D. S. Touretzky, Ed. Morgan-Kaufmann, 1990, pp. 598–605. [Online]. Available: http://papers.nips.cc/paper/250-optimal-brain-damage.pdf

[46] M. Shen, H. Yin, P. Molchanov, L. Mao, J. Liu, and J. Alvarez, "Structural pruning via latency-saliency knapsack," in *Advances in Neural Information Processing Systems*, 2022.

[47] H. Li, A. Kadav, I. Durdanovic, H. Samet, and H. P. Graf, "Pruning filters for efficient convnets," *ArXiv*, vol. abs/1608.08710, 2016.

[48] T. Chen, L. Liang, D. Tianyu, Z. Zhu, and I. Zharkov, "Otov2: Automatic, generic, user-friendly," in *International Conference on Learning Representations*, 2023.

[49] P. Molchanov, S. Tyree, T. Karras, T. Aila, and J. Kautz, "Pruning convolutional neural networks for resource efficient transfer learning," *CoRR*, vol. abs/1611.06440, 2016. [Online]. Available: http://arxiv.org/abs/1611.06440

[50] Y. He, G. Kang, X. Dong, Y. Fu, and Y. Yang, "Soft filter pruning for accelerating deep convolutional neural networks," in *International Joint Conference on Artificial Intelligence (IJCAI)*, 2018, pp. 2234–2240.

[51] G. Shaopeng, W. Yujie, L. Quanquan, and J. Yan, "Dmcp: Differentiable markov channel pruning for neural networks," in *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2020.

[52] J. Rachwan, D. Zügner, B. Charpentier, S. Geisler, M. Ayle, and S. Günnemann, "Winning the lottery ahead of time: Efficient early network pruning," in *Proceedings of the 39th International Conference on Machine Learning*, ser. Proceedings of Machine Learning Research, vol. 162. PMLR, 2022.

[53] Z. Hou, M. Qin, F. Sun, X. Ma, K. Yuan, Y. Xu, Y.-K. Chen, R. Jin, Y. Xie, and S.-Y. Kung, "Chex: Channel exploration for cnn model compression," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2022, pp. 12 287–12 298.

[54] Y. Sui, M. Yin, Y. Xie, H. Phan, S. Aliari Zonouz, and B. Yuan, "Chip: Channel independence-based pruning for compact neural networks," in *Advances in Neural Information Processing Systems*, M. Ranzato, A. Beygelzimer, Y. Dauphin, P. Liang, and J. W. Vaughan, Eds., vol. 34. Curran Associates, Inc., 2021, pp. 24 604–24 616. [Online]. Available: https://proceedings.neurips.cc/paper_files/paper/2021/file/ce6babd060aa46c61a5777902cca78af-Paper.pdf

[55] A. Ganjdanesh, S. Gao, and H. Huang, "Interpretations steered network pruning via amortized inferred saliency maps," in *Proceedings of the European Conference on Computer Vision (ECCV)*, 2022.

[56] Y. Guo, A. Yao, and Y. Chen, "Dynamic network surgery for efficient dnns," in *Advances in Neural Information Processing Systems*, D. Lee, M. Sugiyama, U. Luxburg, I. Guyon, and R. Garnett, Eds., vol. 29. Curran Associates, Inc., 2016. [Online]. Available: https://proceedings.neurips.cc/paper_files/paper/2016/file/2823f4797102ce1a1aec05359cc16dd9-Paper.pdf

[57] Z. Zhuang, M. Tan, B. Zhuang, J. Liu, Y. Guo, Q. Wu, J. Huang, and J. Zhu, "Discrimination-aware channel pruning for deep neural networks," in *Advances in Neural Information Processing Systems 31*, S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett, Eds. Curran Associates, Inc., 2018, pp. 881–892. [Online].

Available: http://papers.nips.cc/paper/7367-discrimination-aware-channel-pruning-for-deep-neural-networks.pdf

[58] Y. He, P. Liu, Z. Wang, Z. Hu, and Y. Yang, "Filter pruning via geometric median for deep convolutional neural networks acceleration," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2019.

[59] L. Liu, S. Zhang, Z. Kuang, A. Zhou, J.-H. Xue, X. Wang, Y. Chen, W. Yang, Q. Liao, and W. Zhang, "Group fisher pruning for practical network compression," in *Proceedings of the 38th International Conference on Machine Learning*, ser. Proceedings of Machine Learning Research. PMLR, 18–24 Jul 2021.

[60] M. Lin, R. Ji, Y. Wang, Y. Zhang, B. Zhang, Y. Tian, and L. Shao, "Hrank: Filter pruning using high-rank feature map," in *IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2020.

[61] J.-H. Luo, J. Wu, and W. Lin, "Thinet: A filter level pruning method for deep neural network compression," in *2017 IEEE International Conference on Computer Vision (ICCV)*, 2017, pp. 5068–5076.

[62] Y. Tang, Y. Wang, Y. Xu, D. Tao, C. Xu, C. Xu, and C. Xu, "Scop: Scientific control for reliable neural network pruning," *Advances in Neural Information Processing Systems*, vol. 33, 2020.

[63] S. Anwar, K. Hwang, and W. Sung, "Structured pruning of deep convolutional neural networks," *J. Emerg. Technol. Comput. Syst.*, vol. 13, no. 3, feb 2017. [Online]. Available: https://doi.org/10.1145/3005348

[64] F. Meng, H. Cheng, K. Li, H. Luo, X. Guo, G. Lu, and X. Sun, "Pruning filter in filter," in *Advances in Neural Information Processing Systems*, H. Larochelle, M. Ranzato, R. Hadsell, M. F. Balcan, and H. Lin, Eds., vol. 33. Curran Associates, Inc., 2020, pp. 17 629–17 640. [Online]. Available: https://proceedings.neurips.cc/paper/2020/file/ccb1d45fb76f7c5a0bf619f979c6cf36-Paper.pdf

[65] J. Frankle and M. Carbin, "The lottery ticket hypothesis: Finding sparse, trainable neural networks," in *7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA, May 6-9, 2019*. OpenReview.net, 2019. [Online]. Available: https://openreview.net/forum?id=rJl-b3RcF7

[66] S. Fridovich-Keil, B. R. Bartoldson, J. Diffenderfer, B. Kailkhura, and P.-T. Bremer, "Models out of line: A fourier lens on distribution shift robustness," in *NeurIPS*, 2022.

[67] M. E. Halabi, S. Srinivas, and S. Lacoste-Julien, "Data-efficient structured pruning via submodular optimization," in *Advances in Neural Information Processing Systems*, 2022.

[68] T.-W. Chin, R. Ding, C. Zhang, and D. Marculescu, "Towards efficient model compression via learned global ranking," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2020.

[69] X. Yu, T. Serra, S. Ramalingam, and S. Zhe, "The combinatorial brain surgeon: Pruning weights that cancel one another in neural networks," in *Proceedings of the 39th International Conference on Machine Learning*, 2022, pp. 25 668–25 683.

[70] S. Han, J. Pool, J. Tran, and W. Dally, "Learning both weights and connections for efficient neural network," in *Advances in Neural Information Processing Systems*, C. Cortes, N. Lawrence, D. Lee, M. Sugiyama, and R. Garnett, Eds., vol. 28. Curran Associates, Inc., 2015. [Online]. Available: https://proceedings.neurips.cc/paper_files/paper/2015/file/ae0eb3eed39d2bcef4622b2499a05fe6-Paper.pdf

[71] Z. Liu, H. Mu, X. Zhang, Z. Guo, X. Yang, K.-T. Cheng, and J. Sun, "Metapruning: Meta learning for automatic neural network channel pruning," in *Proceedings of the IEEE International Conference on Computer Vision*, 2019, pp. 3296–3305.

[72] S. Han, H. Mao, and W. J. Dally, "Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding," in *International Conference on Learning Representations*, 2016.

[73] Z. Wang and C. Li, "Channel pruning via lookahead search guided reinforcement learning," in *Proceedings of the IEEE/CVF Winter Conference on Applications of Computer Vision (WACV)*, January 2022, pp. 2029–2040.

[74] A. Krizhevsky, V. Nair, and G. Hinton, "Cifar-10 (canadian institute for advanced research)," 2009. [Online]. Available: http://www.cs.toronto.edu/~kriz/cifar.html

[75] W. Balzer, M. Takahashi, J. Ohta, and K. Kyuma, "Weight quantization in boltzmann machines." *Neural Networks*, vol. 4, 1991.

[76] Y. Dong, R. Ni, J. Li, Y. Chen, J. Zhu, and H. Su, "Learning accurate low-bit deep neural networks with stochastic quantization," *CoRR*, vol. abs/1708.01001, 2017. [Online]. Available: http://arxiv.org/abs/1708.01001

[77] A. Polino, R. Pascanu, and D. Alistarh, "Model compression via distillation and quantization," in *International Conference on Learning Representations*, 2018. [Online]. Available: https://openreview.net/forum?id=S1XolQbRW

[78] X. Xu *et al.*, "Quantization of fully convolutional networks for accurate biomedical image segmentation," *CVPR*, pp. 8300–8308, 2018.

[79] A. Zhou *et al.*, "Incremental network quantization: Towards lossless cnns with low-precision weights," *CoRR*, vol. abs/1702.03044, 2017.

[80] D. Lin, S. Talathi, and S. Annapureddy, "Fixed point quantization of deep convolutional networks," in *Proceedings of The 33rd International Conference on Machine Learning*, ser. Proceedings of Machine Learning Research, M. F. Balcan and K. Q. Weinberger, Eds., vol. 48. New York, New York, USA: PMLR, 20–22 Jun 2016, pp. 2849–2858. [Online]. Available: http://proceedings.mlr.press/v48/linb16.html

[81] Y. Guo, "A survey on methods and theories of quantized neural networks," *CoRR*, vol. abs/1808.04752, 2018. [Online]. Available: http://arxiv.org/abs/1808.04752

[82] R. Krishnamoorthi, "Quantizing deep convolutional networks for efficient inference: A whitepaper," *CoRR*, vol. abs/1806.08342, 2018. [Online]. Available: http://arxiv.org/abs/1806.08342

[83] C. Zhu, S. Han, H. Mao, and W. J. Dally, "Trained ternary quantization," *CoRR*, vol. abs/1612.01064, 2016. [Online]. Available: http://arxiv.org/abs/1612.01064

[84] O. Bilaniuk, S. Wagner, Y. Savaria, and J. David, "Bit-slicing FPGA accelerator for quantized neural networks," in *2019 IEEE International Symposium on Circuits and Systems (ISCAS)*, May 2019, pp. 1–5.

[85] J. Hennessey and D. Patterson, *Computer Architecture: A Quantitative Approach*, 5th ed. Morgan Kofmann Publishers, Inc., 2011.

[86] B. Wu, Y. Wang, P. Zhang, Y. Tian, P. Vajda, and K. Keutzer, "Mixed precision quantization of convnets via differentiable neural architecture search," *CoRR*, vol. abs/1812.00090, 2018. [Online]. Available: http://arxiv.org/abs/1812.00090

[87] X. Xu, Q. Lu, L. Yang, S. Hu, D. Chen, Y. Hu, and Y. Shi, "Quantization of fully convolutional networks for accurate biomedical image segmentation," *Preprint at https://arxiv.org/abs/1803.04907*, 2018.

[88] A. Abdolrashidi, L. Wang, S. Agrawal, J. Malmaud, O. Rybakov, C. Leichner, and L. Lew, "Pareto-optimal quantized resnet is mostly 4-bit," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2021, pp. 3091–3099.

[89] I. Hubara, M. Courbariaux, D. Soudry, R. El-Yaniv, and Y. Bengio, "Quantized neural networks: Training neural networks with low precision weights and activations," *J. Mach. Learn. Res.*, vol. 18, no. 1, p. 6869–6898, jan 2017.

[90] C. Xu, J. Yao, Z. Lin, W. Ou, Y. Cao, Z. Wang, and H. Zha, "Alternating multi-bit quantization for recurrent neural networks," in *6th International Conference on Learning Representations, ICLR 2018, Vancouver, BC, Canada, April 30 - May 3, 2018, Conference Track Proceedings*. OpenReview.net, 2018. [Online]. Available: https://openreview.net/forum?id=S19dR9x0b

[91] C. Louizos, M. Reisser, T. Blankevoort, E. Gavves, and M. Welling, "Relaxed quantization for discretized neural networks," in *International Conference on Learning Representations*, 2019. [Online]. Available: https://openreview.net/forum?id=HkxjYoCqKX

[92] W. Chen, H. Qiu, J. Zhuang, C. Zhang, Y. Hu, Q. Lu, T. Wang, Y. Shi, M. Huang, and X. Xu, "Quantization of deep neural networks for accurate edge computing," *J. Emerg. Technol. Comput. Syst.*, vol. 17, no. 4, jun 2021. [Online]. Available: https://doi.org/10.1145/3451211

[93] Y. Mishchenko, Y. Goren, M. Sun, C. Beauchene, S. Matsoukas, O. Rybakov, and S. N. P. Vitaladevuni, "Low-bit quantization and quantization-aware training for small-footprint keyword spotting," in *2019 18th IEEE International Conference On Machine Learning And Applications (ICMLA)*, 2019, pp. 706–711.

[94] M. Alizadeh, A. Behboodi, M. van Baalen, C. Louizos, T. Blankevoort, and M. Welling, "Gradient ℓ_1 regularization for quantization robustness," in *International Conference on Learning Representations*, 2020. [Online]. Available: https://openreview.net/forum?id=ryxK0JBtPr

[95] L. Hou and J. T. Kwok, "Loss-aware weight quantization of deep networks," in *International Conference on Learning Representations*, 2018.

[96] J. Choi, Z. Wang, S. Venkataramani, P. I.-J. Chuang, V. Srinivasan, and K. Gopalakrishnan, "PACT: Parameterized clipping activation for quantized neural networks," 2018. [Online]. Available: https://openreview.net/forum?id=By5ugjyCb

[97] K. Zhang, M. Yin, and Y.-X. Wang, "Why quantization improves generalization: Ntk of binary weight neural networks," 2022. [Online]. Available: https://arxiv.org/abs/2206.05916

[98] O. Bilaniuk, S. Wagner, Y. Savaria, and J.-P. David, "Bit-slicing FPGA accelerator for quantized neural networks," in *2019 IEEE International Symposium on Circuits and Systems (ISCAS)*, 2019, pp. 1–5.

[99] M. Cowan, T. Moreau, T. Chen, J. Bornholt, and L. Ceze, "Automatic generation of high-performance quantized machine learning kernels," in *Proceedings of the 18th ACM/IEEE International Symposium on Code Generation and Optimization*, ser. CGO 2020. New York, NY, USA: Association for Computing Machinery, 2020, p. 305–316. [Online]. Available: https://doi.org/10.1145/3368826.3377912

[100] M. Sun *et al.*, "Film-qnn: Efficient fpga acceleration of deep neural networks with intra-layer, mixed-precision quantization," in *Proceedings of the 2022 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, 2022, pp. 134–145.

[101] M. Cowan, T. Moreau, T. Chen, J. Bornholt, and L. Ceze, "Automatic generation of high-performance quantized machine learning kernels," in *Proceedings of the 18th ACM/IEEE International Symposium on Code Generation and Optimization*, 2020, pp. 305–316.

[102] A. Gholami, S. Kim, Z. Dong, Z. Yao, M. W. Mahoney, and K. Keutzer, "A survey of quantization methods for efficient neural network inference," 2021.

[103] M. Blott, T. B. Preußer, N. J. Fraser, G. Gambardella, K. O'brien, Y. Umuroglu, M. Leeser, and K. Vissers, "Finn-r: An end-to-end deep-learning framework for fast exploration of quantized neural networks," *ACM Trans. Reconfigurable Technol. Syst.*, vol. 11, no. 3, dec 2018. [Online]. Available: https://doi.org/10.1145/3242897

[104] O. Bilaniuk, S. Wagner, Y. Savaria, and J.-P. David, "Bit-slicing FPGA accelerator for quantized neural networks," in *2019 IEEE International Symposium on Circuits and Systems (ISCAS)*, 2019, pp. 1–5.

[105] B. H. J. Lee, D. Kim, "Network quantization with element-wise gradient scaling," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2021.

[106] K. Wang, Z. Liu, Y. Lin, J. Lin, and S. Han, "HAQ: Hardware-aware automated quantization with mixed precision," in *2019 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, 2019, pp. 8604–8612.

[107] J. Choi, S. Venkataramani, V. Srinivasan, K. Gopalakrishnan, Z. Wang, and P. Chuang, "Accurate and efficient 2-bit quantized neural networks," in *Proceedings of the 2nd SysML Conference*, 2019.

[108] G. B. Hacene, V. Gripon, M. Arzel, N. Farrugia, and Y. Bengio, "Quantized guided pruning for efficient hardware implementations of convolutional neural networks," *arXiv preprint 1812.11337*, 2018.

[109] R. Zhang and A. C. Chung, "Medq: Lossless ultra-low-bit neural network quantization for medical image segmentation," *Medical Image Analysis*, vol. 73, p. 102200, 2021. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1361841521002450

[110] L. Guo, W. Fei, W. Dai, C. Li, J. Zou, and H. Xiong, "Mixed-precision quantization of u-net for medical image segmentation," in *2022 IEEE International Symposium on Circuits and Systems (ISCAS)*, 2022, pp. 2871–2875.

[111] E. Fiesler, C. Amar, and H. J. Caulfield., "Weight discretization paradigm for optical neural networks," *International Society for Optics and Photonics*, pp. 164–173, 1990.

[112] C. Z. Tang and H. K. Kwan, "Multilayer feedforward neural networks with single powers-of-two weights." *IEEE Transactions on Signal Processing,*, vol. 41(8):, 1993.

[113] Intel, *Intel® Core™i7-4770 Processor*, https://ark.intel.com/content/www/us/en/ark/products/75122/intel-core-i7-4770-processor-8m-cache-up-to-3-90-ghz.html Accessed on April 10th 2020.

[114] Z. Yuan, C. Xue, Y. Chen, Q. Wu, and G. a. Sun, "Ptq4vit: Post-training quantization framework for vision transformers," *arXiv preprint arXiv:2111.12293*, 2022.

[115] H. Jiang, Q. Li, and Y. Li, "Post training quantization after neural network," in *2022 14th International Conference on Computer Research and Development (ICCRD)*, 2022, pp. 1–6.

[116] D. Zheng, Y. Liu, L. Li *et al.*, "Leveraging inter-layer dependency for post-training quantization," *Advances in Neural Information Processing Systems*, vol. 35, pp. 6666–6679, 2022.

[117] H. Wu, P. Judd, X. Zhang, M. Isaev, and P. Micikevicius, "Integer quantization for deep learning inference: Principles and empirical evaluation," *ArXiv*, vol. abs/2004.09602, 2020.

[118] Y. Bengio, N. Léonard, and A. C. Courville, "Estimating or propagating gradients through stochastic neurons for conditional computation," *CoRR*, vol. abs/1308.3432, 2013. [Online]. Available: http://arxiv.org/abs/1308.3432

[119] A. L. Friesen and P. M. Domingos, "Deep learning as a mixed convex-combinatorial optimization problem," *ArXiv*, vol. abs/1710.11573, 2017.

[120] D.-H. Lee, S. Zhang, A. Fischer, and Y. Bengio, "Difference target propagation," in *Machine Learning and Knowledge Discovery in Databases*, A. Appice, P. P. Rodrigues, V. Santos Costa, C. Soares, J. Gama, and A. Jorge, Eds. Cham: Springer International Publishing, 2015, pp. 498–515.

[121] E. Jang, S. Gu, and B. Poole, "Categorical reparameterization with gumbel-softmax," 2017. [Online]. Available: https://arxiv.org/abs/1611.01144

[122] 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 *2019 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, 2019, pp. 4345–4354.

[123] Y. Bhalgat, J. Lee, M. Nagel, T. Blankevoort, and N. Kwak, "Lsq+: Improving low-bit quantization through learnable offsets and better initialization," in *2020 IEEE/CVF Conference on Computer Vision and Pattern Recognition Workshops (CVPRW)*, 2020, pp. 2978–2985.

[124] K. Yamamoto, "Learnable companding quantization for accurate low-bit neural networks," in *2021 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, 2021, pp. 5027–5036.

[125] M. Courbariaux, Y. Bengio, and J.-P. David, "Binaryconnect: Training deep neural networks with binary weights during propagations," in *Advances in Neural Information Processing Systems 28*, C. Cortes, N. D. Lawrence, D. D. Lee, M. Sugiyama, and R. Garnett, Eds. Curran Associates, Inc., 2015, pp. 3123–3131. [Online]. Available: http://papers.nips.cc/paper/5647-binaryconnect-training-deep-neural-networks-with-binary-weights-during-propagations.pdf

[126] M. Rastegari *et al.*, "Xnor-net: Imagenet classification using binary convolutional neural networks," in *ECCV*. Springer, 2016, pp. 525–542.

[127] H. Qin, X. Ma, Y. Ding, X. Li, Y. Zhang, Z. Ma, J. Wang, J. Luo, and X. Liu, "Bifsmnv2: Pushing binary neural networks for keyword spotting to real-network performance," in *IEEE Transactions on Neural Networks and Learning Systems (TNNLS)*, 2023.

[128] H. Qin, Y. Ding, M. Zhang, Q. Yan, A. Liu, Q. Dang, Z. Liu, and X. Liu, "Bibert: Accurate fully binarized bert," in *International Conference on Learning Representations (ICLR)*, 2022.

[129] Z. Liu, B. Oguz, A. Pappu, L. Xiao, S. Yih, M. Li, R. Krishnamoorthi, and Y. Mehdad, "Bit: Robustly binarized multi-distilled transformer," *arXiv preprint arXiv:2205.13016*, 2022.

[130] H. Qin, X. Ma, Y. Ding, X. Li, Y. Zhang, Y. Tian, Z. Ma, J. Luo, and X. Liu, "Bifsmn: Binary neural network for keyword spotting," in *Proceedings of the Thirty-First International Joint Conference on Artificial Intelligence, IJCAI-22*, 2022, pp. 4346–4352.

[131] H. Qin, Z. Cai, M. Zhang, Y. Ding, H. Zhao, S. Yi, X. Liu, and H. Su, "Bipointnet: Binary neural network for point clouds," in *ICLR*, 2021.

[132] Z. Liu, Z. Shen, S. Li, K. Helwegen, D. Huang, and K.-T. Cheng, "How do adam and training strategies help bnns optimization?" in *International Conference on Machine Learning*. PMLR, 2021.

[133] Z. Shen, Z. Liu, J. Qin, L. Huang, K.-T. Cheng, and M. Savvides, "S2-bnn: Bridging the gap between self-supervised real and 1-bit neural networks via guided distribution calibration," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, 2021.

[134] A. Bulat, B. Martinez, and G. Tzimiropoulos, "High-capacity expert binary networks," in *International Conference on Learning Representations (ICLR)*, 2021.

[135] Z. Xu, M. Lin, J. Liu, J. Chen, L. Shao, Y. Gao, Y. Tian, and R. Ji, "Recu: Reviving the dead weights in binary neural networks," in *Proceedings of the International Conference on Computer Vision (ICCV)*, 2021, pp. 5198–5208.

[136] H. Qin, R. Gong, X. Liu, M. Shen, Z. Wei, F. Yu, and J. Song, "Forward and backward information retention for accurate binary neural networks," in *IEEE CVPR*, 2020.

[137] D. Kim, K. P. Singh, and J. Choi, "Learning architectures for binary networks," in *ECCV*, 2020.

[138] G.-J. P., "Padmavati parallel associative development machine as a vehicle for artificial intelligence," *Commission of the European Communities, Directorate-General Telecommunications Information Industries and Innovation (eds) Esprit '89*, 1989.

[139] K. Asanovic, K. B., M. N., and W. J., "Hipnet-1: A highly pipelined architecture for neural network training," 1990.

[140] N. P. Jouppi, G. Kurian, S. Li, P. Ma, R. Nagarajan, L. Nai, N. Patil, S. Subramanian, A. Swing, B. Towles, C. Young, X. Zhou, Z. Zhou, and D. Patterson, "Tpu v4: An optically reconfigurable supercomputer for machine learning with hardware support for embeddings," 2023.

[141] H. Kung and C. E. Leiserson, "Systolic arrays (for vlsi)," *Sparse Matrix Proceedings*, vol. 1, pp. 256–282, 1978.

[142] (2020) RISC-V Foundation. Accessed on Jan 7th 2020. [Online]. Available: https://riscv.org

[143] S. Davidson, S. Xie, C. Torng, K. Al-Hawai, A. Rovinski, T. Ajayi, L. Vega, C. Zhao, R. Zhao, S. Dai, A. Amarnath, B. Veluri, P. Gao, A. Rao, G. Liu, R. K. Gupta, Z. Zhang, R. Dreslinski, C. Batten, and M. Taylor, "The celerity open-source 511-core risc-v tiered accelerator fabric: Fast architectures and design methodologies for fast chips," *IEEE Micro*, vol. 38, no. 02, pp. 30–41, March 2018.

[144] *The RISC-V Instruction Set Manual*, December 2019. [Online]. Available: https://riscv.org/specifications/

[145] S. Sharify, A. D. Lascorz, K. Siu, P. Judd, and A. Moshovos, "Loom: Exploiting weight and activation precisions to accelerate convolutional neural networks," in *2018 55th ACM/ESDA/IEEE Design Automation Conference (DAC)*, 2018, pp. 1–6.

[146] S. Venkataramani, V. Srinivasan, W. Wang, S. Sen *et al.*, "RaPiD: AI accelerator for ultra-low precision training and inference," in *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*, 2021, pp. 153–166.

[147] C. Zhang, Zhenman Fang, Peipei Zhou, Peichen Pan, and Jason Cong, "Caffeine: Towards uniformed representation and acceleration for deep convolutional neural networks," in *2016 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, 2016, pp. 1–8.

[148] A. Zaidy, A. X. M. Chang, V. Gokhale, and E. Culurciello, "A high efficiency accelerator for deep neural networks," in *The 1st Workshop on Energy Efficient Machine Learning and Cognitive Computing*, 2018.

[149] X. Zhang, J. Wang, C. Zhu, Y. Lin, J. Xiong, W.-m. Hwu, and D. Chen, "DNNBuilder: an automated tool for building high-performance DNN hardware accelerators for FPGAs," in *International Conference on Computer Aided Design (ICCAD)*, 2018.

[150] A. Abid, P. Sinha, A. Harpale, J. Gichoya, and S. Purkayastha, "Optimizing medical image classification models for edge devices," in *Distributed Computing and Artificial Intelligence, Volume 1: 18th International Conference*, K. Matsui, S. Omatu, T. Yigitcanlar, and S. R. González, Eds. Cham: Springer International Publishing, pp. 77–87.

[151] H. Kim, J. Kim, J. Choi, J. Lee, and Y. H. Song, "Binarized encoder-decoder network and binarized deconvolution engine for semantic segmentation," *IEEE Access*, vol. 9, pp. 8006–8027, 2021.

[152] R. Miotto *et al.*, "Deep learning for healthcare: review, opportunities and challenges," *Briefings in Bioinformatics*, vol. 19, no. 6, pp. 1236–1246, 05 2017. [Online]. Available: https://doi.org/10.1093/bib/bbx044

[153] T. S. *et al.*, "The role of deep learning in improving healthcare," *Data Science for Healthcare*, 2019.

[154] F. Prados *et al.*, "Spinal cord grey matter segmentation challenge." *NeuroImage*, 2017.

[155] A. Cardona *et al.*, "An integrated micro-and macroarchitectural analysis of the drosophila brain by computer-assisted serial section electron microscopy," *PLoS Biology*, vol. 8, no. 10, p. e1000502, 2010.

[156] H. R. Roth. *et al.*, "Deeporgan: Multi-level deep convolutional networks for automated pancreas segmentation." *MICCAI*, 2015.

[157] D. L. Pham *et al.*, "Current methods in medical image segmentation," *Annual review of biomedical engineering*, vol. 2, no. 1, pp. 315–337, 2000.

[158] G. Litjens *et al.*, "A survey on deep learning in medical image analysis," *Medical Image Analysis*, vol. 42, pp. 60–88, 2017.

[159] D. Shen *et al.*, "Deep learning in medical image analysis," *Annual review of biomedical engineering*, vol. 19, pp. 221–248, 2017.

[160] S. Honari *et al.*, "Recombinator networks: Learning coarse-to-fine feature aggregation," in *CVPR*, 2016, pp. 5743–5752.

[161] H. Noh *et al.*, "Learning deconvolution network for semantic segmentation," in *ICCV*, 2015, pp. 1520–1528.

[162] P. Isola *et al.*, "Image-to-image translation with conditional adversarial networks," in *CVPR*, July 2017.

[163] A. Newell *et al.*, "Stacked hourglass networks for human pose estimation," in *ECCV*. Springer, 2016, pp. 483–499.

[164] Ö. Çiçek *et al.*, "3d u-net: learning dense volumetric segmentation from sparse annotation," in *MICCAI*. Springer, 2016, pp. 424–432.

[165] M. Courbariaux *et al.*, "Binaryconnect: Training deep neural networks with binary weights during propagations," in *NeurIPS*, 2015, pp. 3123–3131.

[166] M. P. Heinrich *et al.*, "Ternarynet: Faster deep model inference without gpus for medical 3d segmentation using sparse and binary convolutions," *CoRR*, vol. abs/1801.09449, 2018. [Online]. Available: http://arxiv.org/abs/1801.09449

[167] G. Hinton *et al.*, "Neural networks for machine learning, video lectures," *Coursera*, 2012.

[168] N. Srivastava *et al.*, "Dropout: a simple way to prevent neural networks from overfitting," *JMLR*, vol. 15, no. 1, pp. 1929–1958, 2014.

[169] W. Tang *et al.*, "How to train a compact binary neural network with high accuracy?" in *AAAI*, 2017.

[170] M. Horowitz, "Computing's energy problem (and what we can do about it)," *Interational Solid State Circuits Conference*, 2014.

[171] M. Courbariaux, Y. Bengio, and J.-P. David, "Binaryconnect: Training deep neural networks with binary weights during propagations," in *Advances in Neural Information Processing Systems*, C. Cortes, N. Lawrence, D. Lee, M. Sugiyama, and R. Garnett, Eds., vol. 28. Curran Associates, Inc., 2015. [Online]. Available: https://proceedings.neurips.cc/paper/2015/file/3e15cc11f979ed25912dff5b0669f2cd-Paper.pdf

[172] R. Andri, L. Cavigelli, D. Rossi, and L. Benini, "Yodann: An architecture for ultralow power binary-weight cnn acceleration," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 37, no. 1, pp. 48–60, 2018.

[173] X. Wang, Z. Cai, D. Gao, and N. Vasconcelos, "Towards universal object detection by domain attention," *CoRR*, vol. abs/1904.04402, 2019. [Online]. Available: http://arxiv.org/abs/1904.04402

[174] J. Liu, J. Zhang, Y. Ding, X. Xu, M. Jiang, and Y. Shi, "Binarizing weights wisely for edge intelligence: Guide for partial binarization of deconvolution-based generators," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 39, no. 12, pp. 4748–4759, 2020.

[175] P. Merolla, R. Appuswamy, J. Arthur, S. K. Esser, and D. Modha, "Deep neural networks are robust to weight binarization and other non-linear distortions," 2016. [Online]. Available: https://arxiv.org/abs/1606.01981

[176] L. Hou, Q. Yao, and J. T. Kwok, "Loss-aware binarization of deep networks," in *International Conference on Learning Representations*, 2017.

[177] I. Goodfellow, Y. Bengio, and A. Courville, *Deep learning*. MIT press, 2016.

[178] S. Hochreiter and J. Schmidhuber, "Simplifying neural nets by discovering flat minima," *Advances in neural information processing systems*, vol. 7, 1994.

[179] 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," *CoRR*, vol. abs/1606.06160, 2016. [Online]. Available: http://arxiv.org/abs/1606.06160

[180] G. Jocher, A. Stoken, J. Borovec, NanoCode012, ChristopherSTAN, L. Changyu, Laughing, tkianai, A. Hogan, lorenzomammana, yxNONG, AlexWang1900, L. Diaconu, Marc, wanghaoyang0106, ml5ah, Doug, F. Ingham, Frederik, Guilhen, Hatovix, J. Poznanski, J. Fang, L. Yu, changyu98, M. Wang, N. Gupta, O. Akhtar, PetrDvoracek, and P. Rai, "ultralytics/yolov5: v3.1 - Bug Fixes and Performance Improvements," Oct. 2020. [Online]. Available: https://doi.org/10.5281/zenodo.4154370

[181] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "Bert: Pre-training of deep bidirectional transformers for language understanding," *arXiv preprint arXiv:1810.04805*, 2018.

[182] J. X. Morris, E. Lifland, J. Y. Yoo, J. Grigsby, D. Jin, and Y. Qi, "Textattack: A framework for adversarial attacks, data augmentation, and adversarial training in nlp," *arXiv preprint arXiv:2005.05909*, 2020.

[183] D. Hendrycks and T. Dietterich, "Benchmarking neural network robustness to common corruptions and perturbations," *Proceedings of the International Conference on Learning Representations*, 2019.

[184] V. Sanh, L. Debut, J. Chaumond, and T. Wolf, "Distilbert, a distilled version of bert: smaller, faster, cheaper and lighter," *arXiv preprint arXiv:1910.01108*, 2019.

[185] T. Wolf, L. Debut, V. Sanh, J. Chaumond, C. Delangue, A. Moi, P. Cistac, T. Rault, R. Louf, M. Funtowicz *et al.*, "Huggingface's transformers: State-of-the-art natural language processing," *arXiv preprint arXiv:1910.03771*, 2019.

[186] R. Alverson, D. Callahan, D. Cummings, B. Koblenz, A. Porterfield, and B. Smith, "The Tera computer system," in *Proceedings of the 4th International Conference on Supercomputing*, ser. ICS '90. New York, NY, USA: Association for Computing Machinery, 1990, p. 1–6. [Online]. Available: https://doi.org/10.1145/77726.255132

[187] P. Kongetira, K. Aingaran, and K. Olukotun, "Niagara: a 32-way multithreaded Sparc processor," *IEEE Micro*, vol. 25, no. 2, pp. 21–29, March 2005.

[188] M. AskariHemmat, O. Bilaniuk, S. Wagner, Y. Savaria, and J.-P. David, "Risc-v barrel processor for accelerator control," in *2020 IEEE 28th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, 2020, pp. 212–212.

[189] A. Cheikh, S. Sordillo, A. Mastrandrea, F. Menichelli, G. Scotti, and M. Olivieri, "Klessydra-t: Designing vector coprocessors for multithreaded edge-computing cores," *IEEE Micro*, vol. 41, no. 2, pp. 64–71, 2021.

[190] G. Lacey, G. W. Taylor, and S. Areibi, "Deep learning on FPGAs: Past, present, and future," *arXiv preprint arXiv:1602.04283*, 2016.

[191] N. J. Fraser, Y. Umuroglu, G. Gambardella, M. Blott, P. Leong, M. Jahre, and K. Vissers, "Scaling binarized neural networks on reconfigurable logic," *Proceedings of the 8th Workshop and 6th Workshop on Parallel Programming and Run-Time Management Techniques for Many-core Architectures and Design Tools and Architectures for Multicore Embedded Computing Platforms - PARMA-DITAM '17*, 2017. [Online]. Available: http://dx.doi.org/10.1145/3029580.3029586

[192] J. Wang, Q. Lou, X. Zhanng, C. Z. Zhu, Y. Lin, and D. Chen, "Design flow of accelerating hybrid extremely low bit-width neural network in embedded FPGA," in *International Conference on Field-Programmable Logic and Applications (FPL)*, 2018.

[193] E. Nurvitadhi, D. Sheffield, Jaewoong Sim, A. Mishra, G. Venkatesh, and D. Marr, "Accelerating binarized neural networks: Comparison of FPGA, CPU, GPU, and ASIC," in *2016 International Conference on Field-Programmable Technology (FPT)*, Dec 2016, pp. 77–84.

[194] Y. Umuroglu, L. Rasnayake, and M. Sjalander, "Bismo: A scalable bit-serial matrix multiplication overlay for reconfigurable computing," in *Field Programmable Logic and Applications (FPL), 2018 28th International Conference on*, ser. FPL '18, 2018.

[195] S. Ryu, H. Kim, W. Yi, and J. Kim, "Bitblade: Area and energy-efficient precision-scalable neural network accelerator with bitwise summation," in *2019 56th ACM/IEEE Design Automation Conference (DAC)*, June 2019, pp. 1–6.

[196] S. Sharify, A. D. Lascorz, K. Siu, P. Judd, and A. Moshovos, "Loom: Exploiting weight and activation precisions to accelerate convolutional neural networks," in *2018 55th ACM/ESDA/IEEE Design Automation Conference (DAC)*, June 2018, pp. 1–6.

[197] A. Kosper and D. Vollrath, "Overview of the next generation Cray XMT," in *Cray User Group 2011 Proceedings*, May 2011.

[198] D. May, *The XMOS XS1 Architecture*, 2009. [Online]. Available: https://www.xmos.com/download/The-XMOS-XS1-Architecture(1.0).pdf

[199] D. Fotland, "Ubicom's MASI wireless network processor," in *HOT Chips 15*, August 2003.

[200] A. Waterman, Y. Lee, R. Avizienis, D. Patterson, and K. Asanovic. (2020) The RISC-V instruction set manual volume II: Privileged architecture. Accessed on Jan 7th 2020. [Online]. Available: https://people.eecs.berkeley.edu/~krste/papers/riscv-privileged-v1.9.1.pdf

[201] T. Strauch, "Using system hyper pipelining (SHP) to improve the performance of a coarse-grained reconfigurable architecture (CGRA) mapped on an FPGA," *CoRR*, vol. abs/1508.07139, 2015. [Online]. Available: http://arxiv.org/abs/1508.07139

[202] J. Gray, "GRVI Phalanx: A massively parallel RISC-V FPGA accelerator accelerator," in *2016 IEEE 24th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, May 2016, pp. 17–20.

[203] C. Wolf. (2020) Picorv32 - a size-optimized risc-v cpu. Accessed on Jan 7th 2020. [Online]. Available: https://github.com/cliffordwolf/picorv32

[204] (2020) This repository hosts unit tests for risc-v processors. Accessed on Jan 7th 2020. [Online]. Available: https://github.com/riscv/riscv-tests

[205] (2020) Virtex-6 FPGA Configurable Logic Block. Accessed on Mar. 19th 2020. [Online]. Available: https://www.xilinx.com/support/documentation/usr_guides/ug364.pdf

[206] T. Chen, T. Moreau, Z. Jiang, L. Zheng, E. Yan, M. Cowan, H. Shen, L. Wang, Y. Hu, L. Ceze, C. Guestrin, and A. Krishnamurthy, "Tvm: An automated end-to-end optimizing compiler for deep learning," in *Proceedings of the 13th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI'18. USA: USENIX Association, 2018, p. 579–594.

[207] F. Foundation. (2023) El correo libre issue 59. [Online]. Available: https://www.fossi-foundation.org/2023/02/14/ecl59

[208] N. Limare, "Floating-point math speed vs precision," 2014. [Online]. Available: http://nicolas.limare.net/pro/notes/2014/12/16_math_speed/

[209] P. Micikevicius *et al.*, "Mixed precision training," *arXiv preprint arXiv:1710.03740*, 2017.

[210] S. Uhlich *et al.*, "Mixed precision DNNs: All you need is a good parametrization," in *8th International Conference on Learning Representations, ICLR 2020, Addis Ababa, Ethiopia, April 26-30, 2020*, 2020.

[211] H. Yu, H. Li, H. Shi, T. S. Huang, and G. Hua, "Any-precision deep neural networks," in *Thirty-Fifth AAAI Conference on Artificial Intelligence*. AAAI Press, 2021, pp. 10 763–10 771. [Online]. Available: https://ojs.aaai.org/index.php/AAAI/article/view/17286

[212] A. Bulat and G. Tzimiropoulos, "Bit-mixer: Mixed-precision networks with runtime bit-width selection," in *Proceedings of the IEEE/CVF International Conference on Computer Vision (ICCV)*, October 2021, pp. 5188–5197.

[213] Y. Umuroglu *et al.*, "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.

[214] H. Sharma *et al.*, "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.

[215] S. Ryu, H. Kim, W. Yi, and J.-J. Kim, "Bitblade: Area and energy-efficient precision-scalable neural network accelerator with bitwise summation," in *Proceedings of the 56th Annual Design Automation Conference 2019*, 2019, pp. 1–6.

[216] "ONNX model zoo," 2021. [Online]. Available: https://github.com/onnx/models

[217] "Open neural network exchange," 2021. [Online]. Available: https://onnx.ai/

[218] G. Li *et al.*, "Residual distillation: Towards portable deep neural networks without shortcuts," in *Advances in Neural Information Processing Systems*, vol. 33, 2020, pp. 8935–8946.

[219] "FINN dataflow accelerator examples," 2021. [Online]. Available: https://github.com/Xilinx/finn-examples

[220] M. Cavalcante, F. Schuiki, F. Zaruba, M. Schaffner, and L. Benini, "Ara: A 1-GHz+ scalable and energy-efficient RISC-V vector processor with multiprecision floating-point support in 22-nm FD-SOI," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 28, no. 2, pp. 530–543, 2020.

[221] (2023) Risc-v v vector extension. [Online]. Available: https://github.com/riscv/riscv-v-spec

[222] M. Courbariaux and Y. Bengio, "Binarynet: Training deep neural networks with weights and activations constrained to +1 or -1," *CoRR*, 2016. [Online]. Available: http://arxiv.org/abs/1602.02830

[223] M. Rastegari, V. Ordonez, J. Redmon, and A. Farhadi, "XNOR-net: ImageNet classification using binary convolutional neural networks," in *Computer Vision – ECCV 2016*, B. Leibe, J. Matas, N. Sebe, and M. Welling, Eds. Cham: Springer International Publishing, 2016, pp. 525–542.

[224] F. Zaruba and L. Benini, "The cost of application-class processing: Energy and performance analysis of a Linux-ready 1.7-GHz 64-bit RISC-V core in 22-nm FDSOI technology," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 27, no. 11, pp. 2629–2640, Nov 2019.

[225] S. Sharify, A. D. Lascorz, M. Mahmoud, M. Nikolic, K. Siu, D. M. Stuart, Z. Poulos, and A. Moshovos, "Laconic deep learning inference acceleration," in *Proceedings of the 46th International Symposium on Computer Architecture*, ser. ISCA '19. New York, NY, USA: Association for Computing Machinery, 2019, p. 304–317. [Online]. Available: https://doi.org/10.1145/3307650.3322255

[226] J. Won, J. Si, S. Son, T. J. Ham, and J. W. Lee, "ULPPACK: Fast sub-8-bit matrix multiply on commodity SIMD hardware," *Proceedings of Machine Learning and Systems*, vol. 4, pp. 52–63, 2022.

[227] G. Ottavi, A. Garofalo, G. Tagliavini, F. Conti, L. Benini, and D. Rossi, "A mixed-precision RISC-V processor for extreme-edge DNN inference," in *2020 IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*, 2020, pp. 512–517.

[228] M. Gautschi, P. D. Schiavone, A. Traber, I. Loi, A. Pullini, D. Rossi, E. Flamand, F. K. Gürkaynak, and L. Benini, "Near-threshold RISC-V core with DSP extensions for scalable iot endpoint devices," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 25, no. 10, pp. 2700–2713, 2017.

[229] P. Davide Schiavone, F. Conti, D. Rossi, M. Gautschi, A. Pullini, E. Flamand, and L. Benini, "Slow and steady wins the race? a comparison of ultra-low-power RISC-V cores for Internet-of-Things applications," in *2017 27th International Symposium on Power and Timing Modeling, Optimization and Simulation (PATMOS)*, 2017, pp. 1–8.

[230] A. Garofalo, Y. Tortorella, M. Perotti, L. Valente, A. Nadalini, L. Benini, D. Rossi, and F. Conti, "Darkside: A heterogeneous RISC-V compute cluster for extreme-edge on-chip DNN inference and training," *IEEE Open Journal of the Solid-State Circuits Society*, pp. 1–1, 2022.

[231] M. Perotti, M. Cavalcante, N. Wistoff, R. Andri, L. Cavigelli, and L. Benini, "A "New Ara" for vector computing: An open source highly efficient RISC-V V 1.0 vector processor design," in *2022 IEEE 33rd International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, 2022, pp. 43–51.

[232] L.-C. Chen, G. Papandreou, I. Kokkinos, K. Murphy, and A. L. Yuille, "Deeplab: Semantic image segmentation with deep convolutional nets, atrous convolution, and fully connected crfs," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 40, no. 4, pp. 834–848, 2018.

[233] Y. Jiang, B. Neyshabur, H. Mobahi, D. Krishnan, and S. Bengio, "Fantastic generalization measures and where to find them," in *8th International Conference on Learning Representations, ICLR 2020, Addis Ababa, Ethiopia, April 26-30, 2020*. OpenReview.net, 2020. [Online]. Available: https://openreview.net/forum?id=SJgIPJBFvH

[234] Y. Jiang, D. Krishnan, H. Mobahi, and S. Bengio, "Predicting the generalization gap in deep networks with margin distributions," in *International Conference on Learning Representations*, 2019. [Online]. Available: https://openreview.net/forum?id=HJlQfnCqKX

[235] P. Foret, A. Kleiner, H. Mobahi, and B. Neyshabur, "Sharpness-aware minimization for efficiently improving generalization," in *International Conference on Learning Representations*, 2021. [Online]. Available: https://openreview.net/forum?id=6Tm1mposlrM

[236] D. C. Ganji, S. Ashfaq, E. Saboori, S. Sah, S. Mitra, M. AskariHemmat, A. Hoffman, A. Hassanien, and M. Léonardon, "Deepgemm: Accelerated ultra low-precision inference on cpu architectures using lookup tables," 2023.

[237] J. Won, J. Si, S. Son, T. J. Ham, and J. W. Lee, "Ulppack: Fast sub-8-bit matrix multiply on commodity simd hardware," in *Proceedings of Machine Learning and Systems*, D. Marculescu, Y. Chi, and C. Wu, Eds., vol. 4, 2022, pp. 52–63. [Online]. Available: https://proceedings.mlsys.org/paper/2022/file/14bfa6bb14875e45bba028a21ed38046-Paper.pdf

APPENDIX A SUPPLEMENTARY MATERIAL FOR U-NET FIXED-POINT QUANTIZATION FOR MEDICAL IMAGE SEGMENTATION

Weight Visualization of Full-Precision U-Net

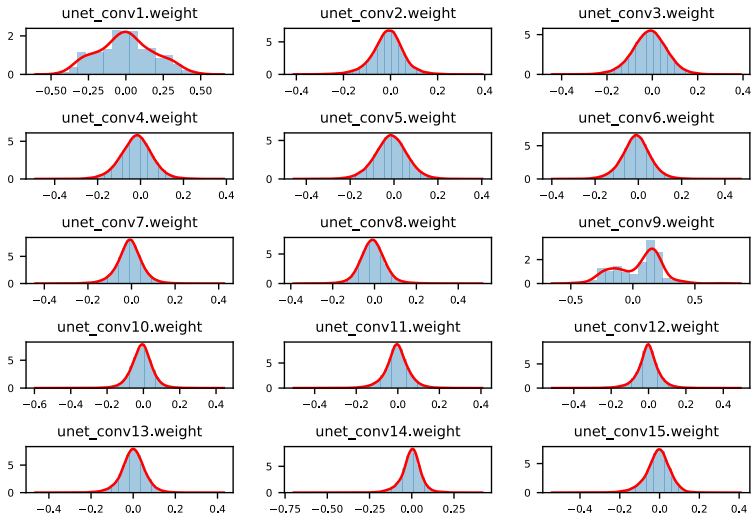


Figure A.1 The weight distribution of U-Net when no quantization is used (full precision). The red curve shows the kernel density estimate (KDE) of the weights and the blue bins show the actual weight values of the distribution. Each plot shows the KDE of one convolutional layer in U-Net. As observed in this figure, in the full precision model almost all weights are distributed in $[-1, +1]$. Hence, no integer part is required in the quantized weights and using higher number of bits for the fractional part of the weights will result in a higher precision

Model Architecture

Layer (type)	Output Shape	Param #
Conv2d-1	[64 , 200, 200]	640
BatchNorm2d-2	[64 , 200, 200]	128
QuantLayer-3	[64 , 200, 200]	0
Conv2d-4	[64 , 200, 200]	36,928
BatchNorm2d-5	[64 , 200, 200]	128
QuantLayer-6	[64 , 200, 200]	0
DownConv-7	[64 , 200, 200]	0
MaxPool2d-8	[64 , 100, 100]	0
Conv2dQuant-9	[128, 100, 100]	73,856
BatchNorm2d-10	[128, 100, 100]	256
QuantLayer-11	[128, 100, 100]	0
Conv2dQuant-12	[128, 100, 100]	147,584
BatchNorm2d-13	[128, 100, 100]	256
QuantLayer-14	[128, 100, 100]	0
DownConv-15	[128, 100, 100]	0
MaxPool2d-16	[128, 50 , 50]	0
Conv2dQuant-17	[256, 50 , 50]	295,168
BatchNorm2d-18	[256, 50 , 50]	512
QuantLayer-19	[256, 50 , 50]	0
Conv2dQuant-20	[256, 50 , 50]	590,080
BatchNorm2d-21	[256, 50 , 50]	512
QuantLayer-22	[256, 50 , 50]	0
DownConv-23	[256, 50 , 50]	0
MaxPool2d-24	[256, 25 , 25]	0
Conv2dQuant-25	[256, 25 , 25]	590,080
BatchNorm2d-26	[256, 25 , 25]	512
QuantLayer-27	[256, 25 , 25]	0
Conv2dQuant-28	[256, 25 , 25]	590,080
BatchNorm2d-29	[256, 25 , 25]	512
QuantLayer-30	[256, 25 , 25]	0
DownConv-31	[256, 25 , 25]	0
Upsample-32	[256, 50 , 50]	0
Conv2dQuant-33	[256, 50 , 50]	1,179,904
BatchNorm2d-34	[256, 50 , 50]	512

```
        QuantLayer-35        [ 256, 50 , 50 ]               0
       Conv2dQuant-36        [ 256, 50 , 50 ]         590,080
       BatchNorm2d-37        [ 256, 50 , 50 ]             512
        QuantLayer-38        [ 256, 50 , 50 ]               0
          DownConv-39        [ 256, 50 , 50 ]               0
            UpConv-40        [ 256, 50 , 50 ]               0
          Upsample-41        [ 256, 100, 100]               0
       Conv2dQuant-42        [ 128, 100, 100]         442,496
       BatchNorm2d-43        [ 128, 100, 100]             256
        QuantLayer-44        [ 128, 100, 100]               0
       Conv2dQuant-45        [ 128, 100, 100]         147,584
       BatchNorm2d-46        [ 128, 100, 100]             256
        QuantLayer-47        [ 128, 100, 100]               0
          DownConv-48        [ 128, 100, 100]               0
            UpConv-49        [ 128, 100, 100]               0
          Upsample-50        [ 128, 200, 200]               0
       Conv2dQuant-51        [ 64 , 200, 200]         110,656
       BatchNorm2d-52        [ 64 , 200, 200]             128
        QuantLayer-53        [ 64 , 200, 200]               0
       Conv2dQuant-54        [ 64 , 200, 200]          36,928
       BatchNorm2d-55        [ 64 , 200, 200]             128
        QuantLayer-56        [ 64 , 200, 200]               0
          DownConv-57        [ 64 , 200, 200]               0
            UpConv-58        [ 64 , 200, 200]               0
            Conv2d-59        [ 1  , 200, 200]             577
================================================================
Total params: 4,837,249
----------------------------------------------------------------
Input size (MB): 0.15
Forward pass size (MB): 593.57
Estimated Total Size (MB): 612.17
----------------------------------------------------------------
```

Dropout In Quantization

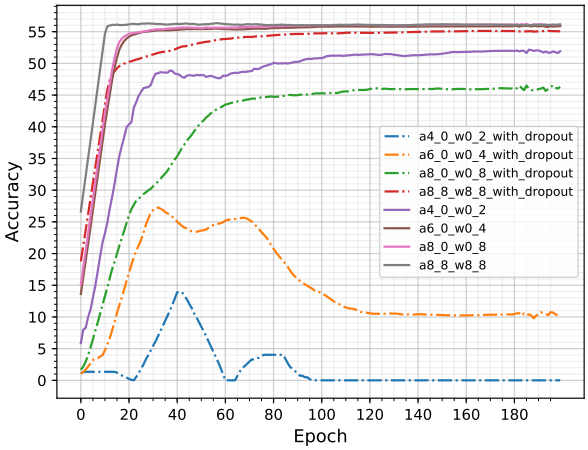


Figure A.2 Dice score of different U-Net quantization precisions over 200 epochs on the Spinal Cord Gray Matter Segmentation dataset. We used our fixed point quantizer to show that using drop out while applying quantization on the model can drastically reduce the dice score. This phenomenon has more severe impact when the model precision is reduced. Curves shown with −− use dropout and quantization

More Experimental Results For EM dataset

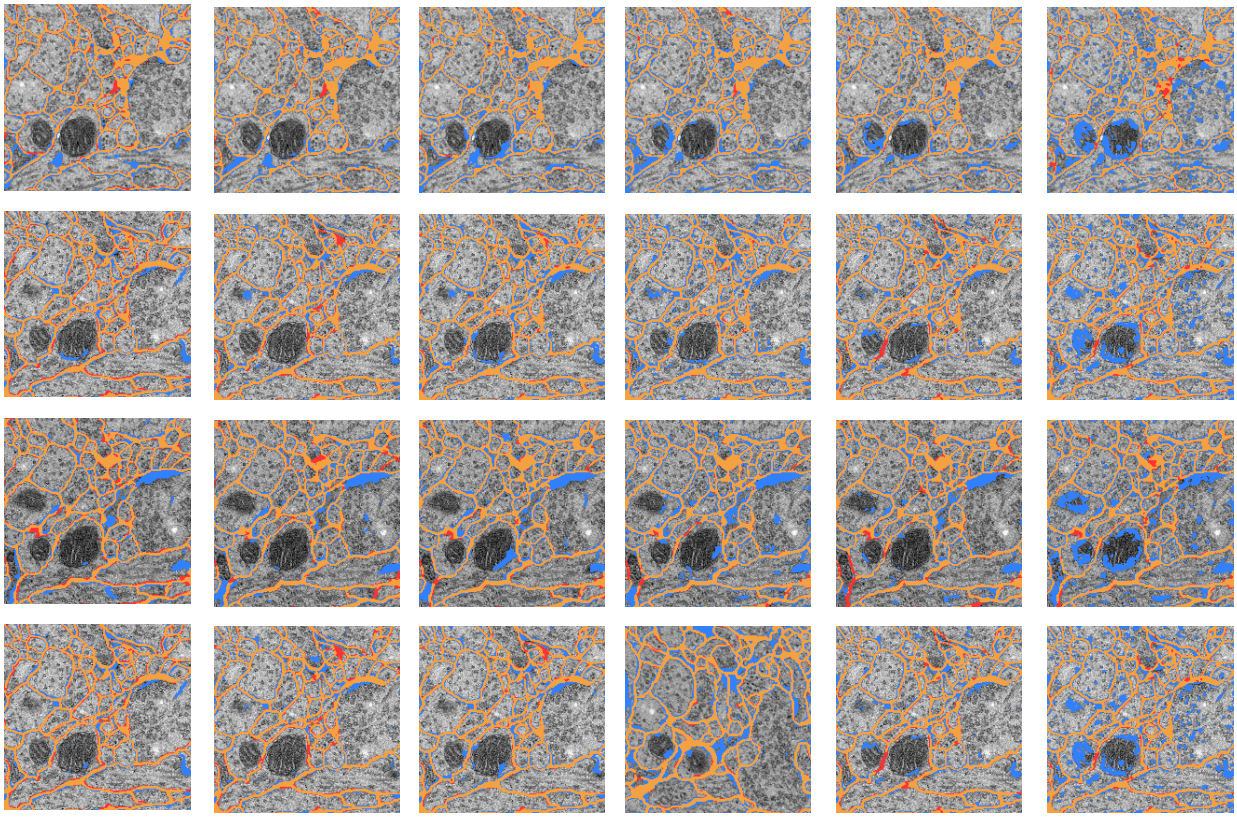


Figure A.3 Sample prediction versus ground truth segmentation results for EM dataset. From left to right, results for different quantization precision are reported. Segments in ■ show false positive, segments in ■ show false negative and segments in ■ show true positive. The quantized model of $Q^a6.0$, $Q^w0.4$ obtains close results compared to the full precision model

More Experimental Results For NIH Pancreas dataset

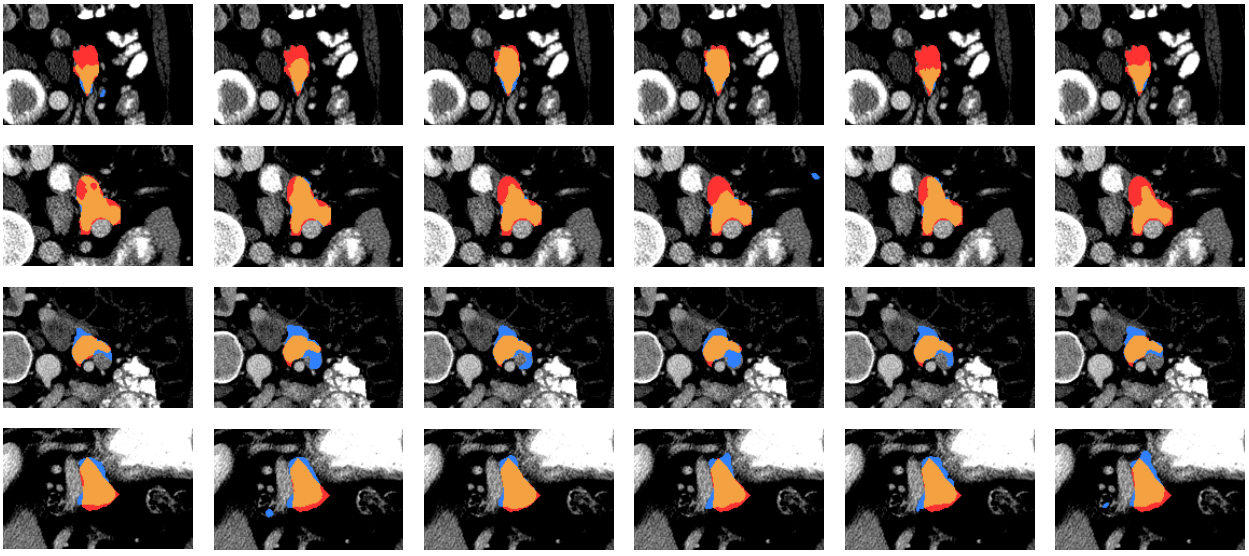


Figure A.4 Sample prediction versus ground truth segmentation results for NIH pancreas dataset. From left to right, results for different quantization methods. Segments in ■ show false positive, segments in ■ show false negative and segments in ■ show true positive. Note that in contrary to GM dataset, NIH pancreas segmentation is much harder since the pancreas has a very high anatomical variability

More Experimental Results For GM dataset

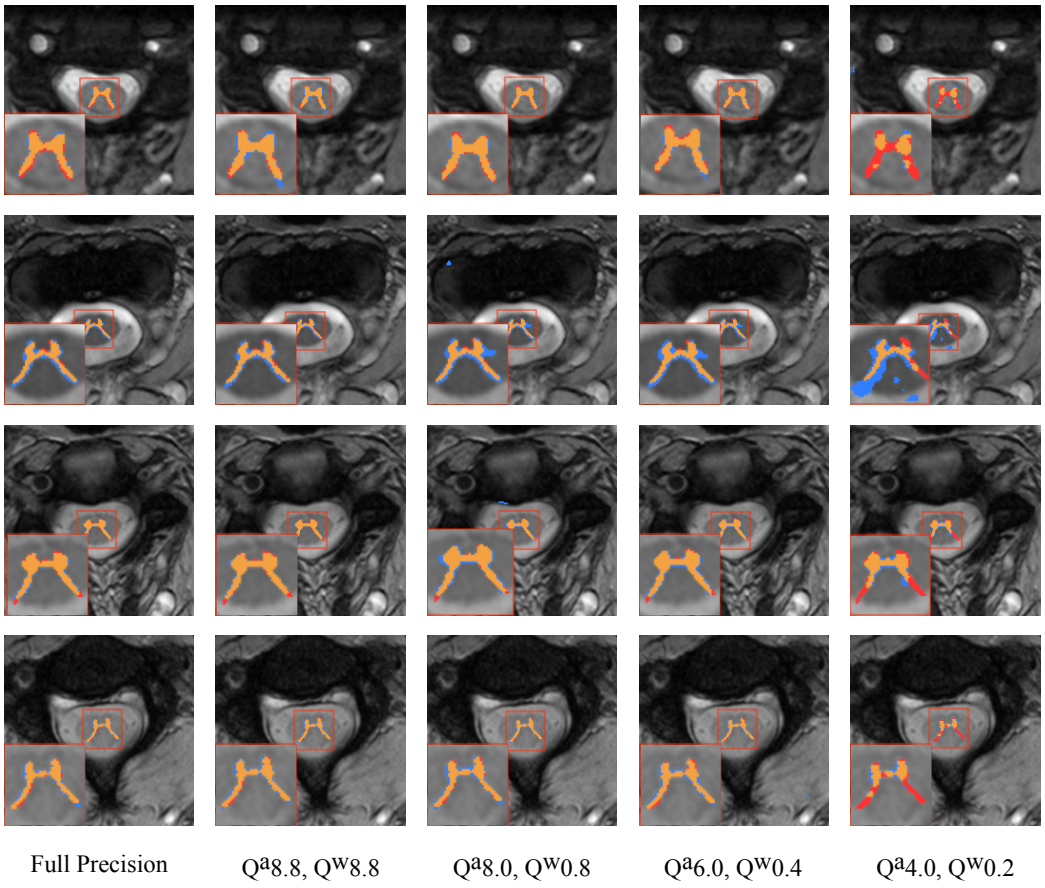


Figure A.5 Sample prediction versus ground truth segmentation results for GM dataset. From left to right, results for different quantization precision are reported. Segments in ■ show false positive, segments in ■ show false negative and segments in ■ show true positive. The quantized model of $Q^a6.0$, $Q^w0.4$ obtains close results compared to the full precision model

APPENDIX B SUPPLEMENTARY MATERIAL FOR QREG: ON REGULARIZATION EFFECTS OF QUANTIZATION

Test Set	Brightness	Contrast	Defocus Blur	Elastic Transform
Fog	Frost	Gaussian Blur	Gaussian Noise	Glass Blur
Impulse Noise	Jpeg Compression	Motion Blur	Pixelate	Saturate
Shot Noise	Snow	Spatter	Speckle Noise	Zoom Blur

(a)

3.60	4.20	3.40	3.80	3.50
3.90	3.10	3.70	-0.40	0.20
2.80	2.60	2.00	3.80	2.20
1.30	3.60	3.10	1.10	2.50

(b)

Figure B.1 Figure B.1(a) illustrates the positional location of augmentation that was applied on the original test dataset. In addition to test images for each dataset, in our experiments, we applied 19 different augmentations on the test dataset. We used the augmentation that is provided in [183]. Figure B.1(b) illustrates the generalization improvement between quantized and full precision ResNet18 on CIFAR100 model when different augmentation is applied. The value in each cell shows the difference and the position of each cell corresponds to the data augmentation technique listed in Figure B.1(a)

Experiments on Classification Task (Weight and Activation Quantization)

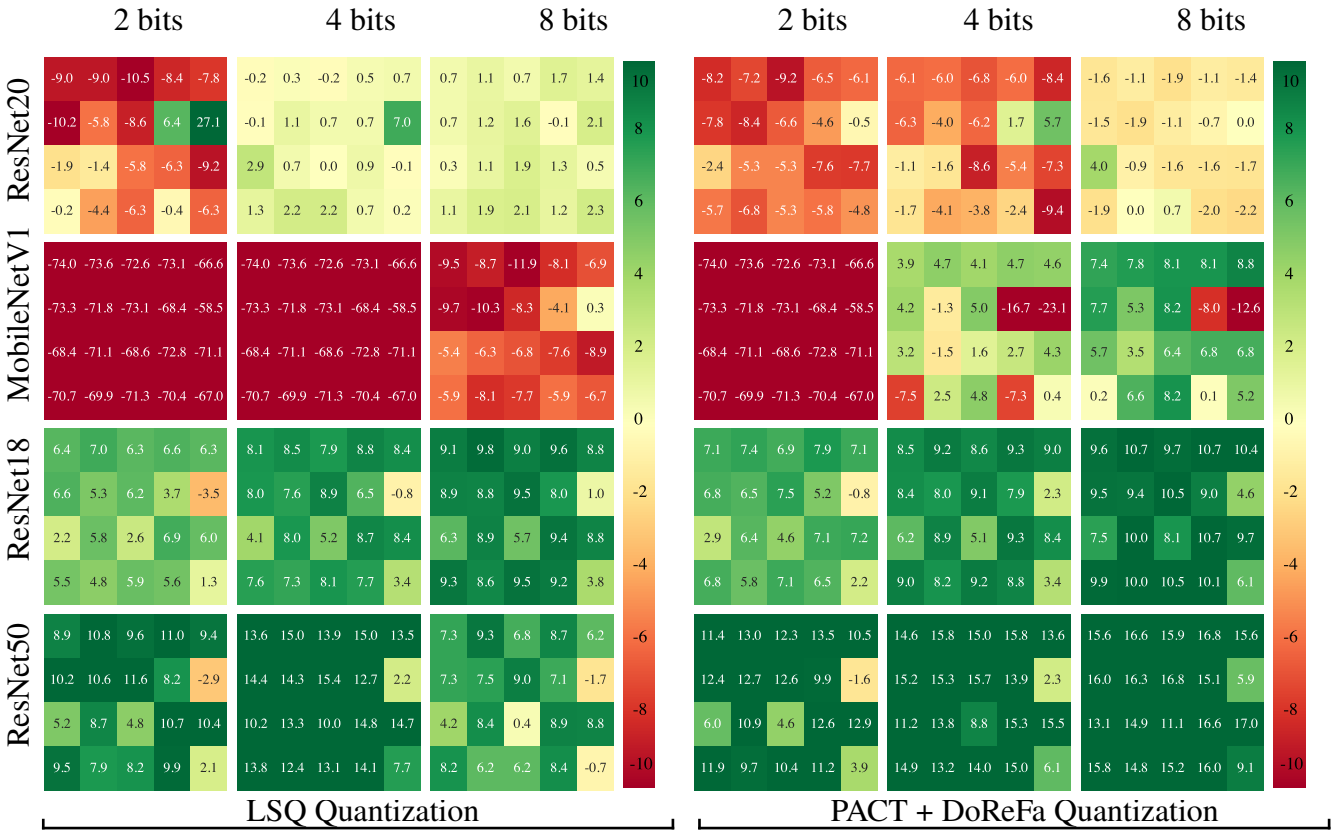


Figure B.2 Regularization effect of quantization in classification task when different quantization methods and levels are used on CIFAR10 dataset.

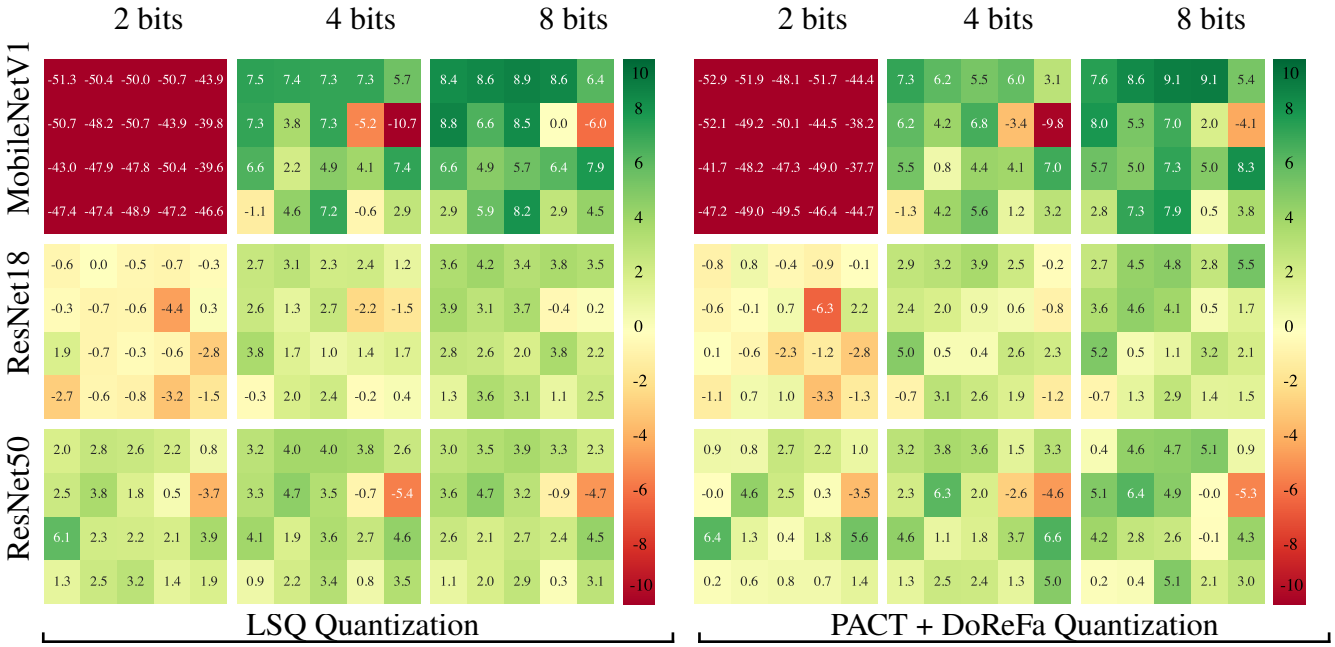


Figure B.3 Regularization effect of quantization in classification task when different quantization methods and levels are used on CIFAR100 dataset

Experiments on Classification Task (Weight Quantization Only)

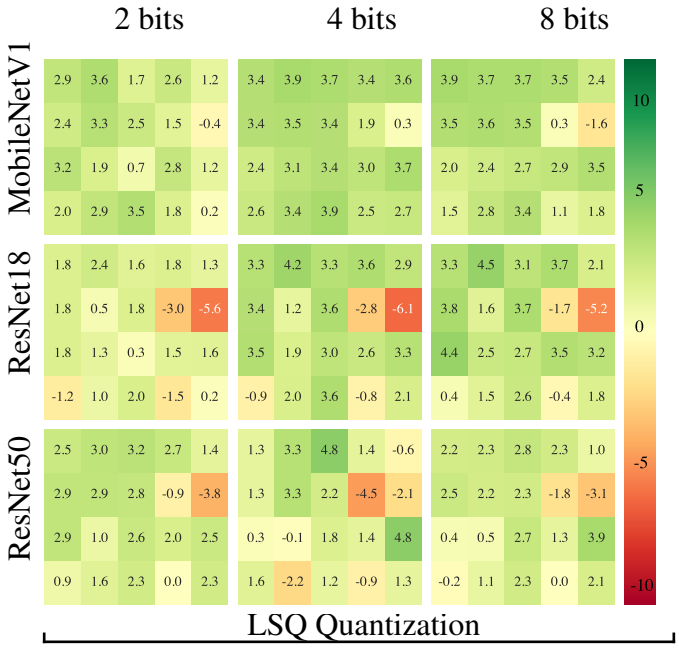


Figure B.4 Regularization effect of quantization in classification task when different quantization levels are used. In this experiment, we only quantized weights and we used CIFAR100 dataset.

Experiments on the Object Detection Task

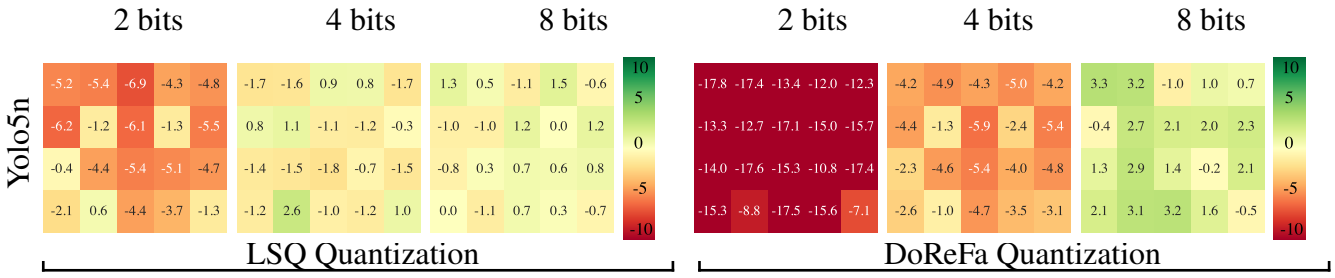


Figure B.5 Regularization effect of quantization in object detection task when different quantization methods and levels are used. NOTE: Unlike previous results we only quantized weights (we used DoReFa and not PACT+DoReFa) for right column results

Experiments on NLP tasks

We used the following set of augmentation recipes provided in the TextAttack framework: SynonymInsertionAugmenter, CharSwapAugmenter, WordNetAugmenter, SwapAugmenter, SwapAugmenter and Back-

TranslationAugmenter (initialized with 3 different language pairs - English-French, English-Spanish and English-Italian). Please see the Figures below for the layout of augmentations and reference accuracy results on 3 tasks.

Test Set	Synonym	Charswap
Wordnet	Swap	Checklist
BackTrans En_Fr	BackTrans En_Esp	BackTrans En_It

Figure B.6 In addition to test images for each dataset, in our experiments, we applied 8 different augmentations on the test dataset. As before, here we are showing the positional location of each augmentation in the subsequent experiments

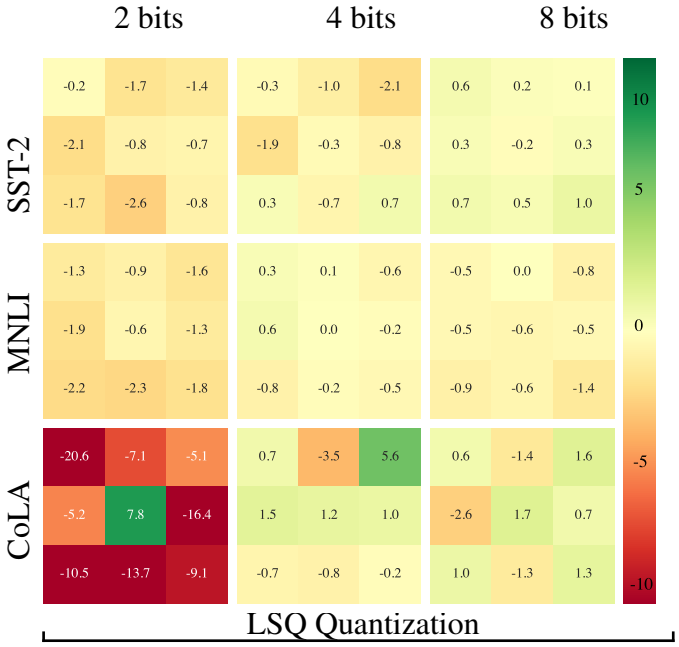


Figure B.7 Regularization effect of quantization in NLP task when different quantization levels are used. We used DistilBERT-base model and tested our hypothesis on three different datasets