POLYTECHNIQUE MONTRÉAL

affiliée à l'Université de Montréal

Energy-Efficient, Flexible and Fast Architectures for Deep Convolutional Neural Network Acceleration

MEHDI AHMADI

Département de génie informatique et génie logiciel

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

Décembre 2020

POLYTECHNIQUE MONTRÉAL

affiliée à l'Université de Montréal

Cette thèse intitulée:

Energy-Efficient, Flexible and Fast Architectures for Deep Convolutional Neural Network Acceleration

présentée par **Mehdi AHMADI**

en vue de l'obtention du diplôme de *Philosophiæ Doctor*

a été dûment acceptée par le jury d'examen constitué de :

Guy BOIS, président

Pierre LANGLOIS, membre et directeur de recherche

Yvon SAVARIA, membre

Claude THIBEAULT, membre externe

DEDICATION

For my Mom, Dad and sisters

ACKNOWLEDGEMENTS

I would like to express my sincere and greatest gratitude to my supervisor, Dr. Pierre Langlois. I am thankful for his constant support and encouragement and for his constructive advice and comments.

I am also thankful for Dr. Shervin Vakili, for the good times, mutual support and encouragement that we shared.

I would also like to thank the faculty, staff and students of the department who helped me to expand my knowledge and expertise. I would also like to thank them for every bit of help they provided, each in their own way.

My eternal gratitude to my family and friends for their unconditional encouragement and support.

RÉSUMÉ

Les méthodes basées sur l'apprentissage profond, et en particulier les réseaux de neurones convolutifs (CNN), ont révolutionné le domaine de la vision par ordinateur. Alors que jusqu'en 2012, les méthodes de traitement d'image traditionnelles les plus précises pouvaient atteindre 26% d'erreurs dans la reconnaissance d'images sur l'étalon normalisé et bien connu ImageNet, une méthode basée sur un CNN a considérablement réduit l'erreur à 16%. En faisant évoluer la structure des CNN, les méthodes actuelles basées sur des CNN atteignent désormais couramment des taux d'erreur inférieurs à 3%, dépassant souvent la précision humaine.

Les CNN se composent de nombreuses couches convolutives, chacune effectuant des opérations de convolution complexes de haute dimension. Pour obtenir une précision élevée en reconnaissance d'images, les CNN modernes empilent de nombreuses couches convolutives, ce qui augmente considérablement la diversité des motifs de calcul entre les couches. Ce haut niveau de complexité dans les CNN implique un nombre massif de paramètres et de calculs.

Étant donné que les processeurs mobiles ne sont pas conçus pour effectuer des calculs massifs, le déploiement de CNN sur des appareils portables et mobiles est un défi. Ainsi, une unité coprocesseur appelée accélérateur doit être placée à proximité du processeur mobile pour effectuer efficacement des calculs CNN. La structure profonde des CNN, cependant, impose des contraintes strictes pour la conception d'accélérateurs CNN.

La grande variété de calculs d'une couche à l'autre d'un CNN nécessite un accélérateur capable de prendre en charge diverses configurations de convolution. L'allocation inefficace des données sur les unités de calcul de l'accélérateur entraîne une latence de calcul élevée et rend l'accélérateur inadapté à de nombreuses applications de vision nécessitant un temps de traitement court. De plus, de nombreux appareils portables sont alimentés par des batteries. Un nombre massif d'accès à la mémoire et de calculs dans les CNN, en revanche, entraîne une consommation d'énergie élevée et une décharge rapide de la batterie. Par conséquent, une considération de conception importante est d'éviter les opérations gourmandes en énergie pendant les calculs de convolution pour économiser la batterie. En outre, la consommation d'énergie de l'accélérateur CNN doit correspondre au budget de puissance de l'appareil mobile.

Dans cette thèse, nous proposons plusieurs nouveaux flux de données pour assigner de manière efficace divers calculs de couches convolutives à des ressources matérielles fixes. Ces flux de données partitionnent les convolutions à haute dimension en parties plus petites pour s'adapter aux ressources matérielles réalisables et offrent les schémas de mouvement de données les plus efficaces pour fournir des données valides pour les unités de calcul à chaque cycle d'horloge. Cela permet de maximiser l'utilisation efficace des unités matérielles lors des calculs qui permettront d'atteindre des débits élevés et de surmonter les limitations de la majorité des flux de données existants qui limitent leur débit réel à moins de la moitié du débit annoncé.

La thèse propose également une nouvelle architecture reconfigurable avec plusieurs modes de fonctionnement pour traiter les flux de données avec une faible surcharge matérielle. Cela inclut des méthodologies de conception originales pour exploiter le parallélisme de calcul, le partage de données communes entre les unités de calcul et des stratégies de réutilisation des données sur puce pour réduire au strict minimum les accès DRAM hors puce gourmands en énergie.

En outre, la thèse propose une nouvelle configuration SRAM sur puce pour réduire le coût de conception et améliorer l'efficacité énergétique. Dans les méthodes proposées, les SRAM individuelles qui sont assignées pour stocker les résultats finaux sont fusionnées entre des unités de calcul parallèles adjacentes pour rendre les SRAM plus étroites et unifiées.

Enfin, la thèse propose une nouvelle méthode d'élagage pour réduire la complexité des modèles CNN tout en ayant un impact insignifiant sur la précision de la classification. La méthode proposée élague au hasard les groupes de paramètres dans les CNN concernant les schémas de calcul des flux de données dans l'architecture conçue. Cela se traduit par une réduction significative du nombre d'accès DRAM et de la latence de calcul.

ABSTRACT

Deep learning-based methods, and specifically Convolutional Neural Networks (CNNs), have revolutionized the field of computer vision. While until 2012, the most accurate traditional image processing methods could reach 26% errors in recognizing images on the standardized and well-known ImageNet benchmark, a CNN-based method dramatically reduced the error to 16%. By evolving CNNs structures, current CNN-based methods now routinely achieve error rates below 3%, often outperforming human level accuracy.

CNNs consist of many convolutional layers each performing high dimensional complex convolution operations. To achieve high image recognition accuracy, modern CNNs stack many convolutional layers which dramatically increases computation pattern diversity across layers. This high level of complexity in CNNs implies massive numbers of parameters and computations.

Since mobile processors are not designed to perform massive computations, deploying CNNs on portable and mobile devices is challenging. Thus, a co-processor unit called accelerator must be placed close to the mobile processor to efficiently perform CNN computations. The deep structure of CNNs, however, imposes stringent constraints for CNN accelerator design.

The high computation variations across CNN layers require an accelerator to support various convolution configurations. Inefficient allocation of data onto the computational units of the accelerator results in high computation latency and makes the accelerator unsuitable for many vision applications with short processing time demand. In addition, many portable devices rely on batteries to provide energy. Massive numbers of memory accesses and computations in CNNs, on the other hand, result in high energy consumption and fast battery drain. Therefore, an important design consideration is to avoid energy-hungry operations during convolution computations to save battery life. Furthermore, the power consumption of the CNN accelerator must fit within the power budget of the mobile device.

In this thesis, we propose several novel dataflows to map efficiently diverse convolutional layer computations onto fixed hardware resources. These dataflows partition high dimensional convolutions into smaller parts to fit within the feasible hardware resources and offer the most efficient data movement schemes to provide valid data for computational units in every clock cycle. This helps to maximize the effective utilization of the hardware units during computations that will

enable reaching high throughputs and will overcome the limitations in majority of existing dataflows that restrict their actual throughput to less than half of their advertised numbers.

The thesis also proposes a novel reconfigurable architecture with several operation modes to support the dataflows with low hardware overhead. This includes original design methodologies to exploit computation parallelism, sharing common data among computational units and on-chip data reuse strategies to reduce energy-hungry off-chip DRAM accesses to their bare minimum.

In addition, the thesis proposes a new on-chip SRAM configuration to decrease the design cost and to enhance energy-efficiency. In the proposed method, individual SRAMs which are assigned to store final results are merged among adjacent parallel computational units to make narrower and unified SRAMs.

Finally, the thesis proposes a new pruning method to reduce the complexity of CNN models while having insignificant impact on classification accuracy. The proposed method randomly prunes groups of parameters in CNNs regarding the computational patterns of dataflows in the designed architecture.

TABLE OF CONTENTS

DEDICATION III

ACKNOWLEDGEMENTS IV

RÉSUMÉ V

ABSTRACT VII

TABLE OF CONTENTS IX

LIST OF TABLES XII

LIST OF FIGURES XIII

LIST OF SYMBOLS AND ABBREVIATIONS XV

CHAPTER 1 INTRODUCTION 1

1.1 Overview and motivation 1

1.2 Problem statement 2

1.3 Research objectives 3

1.4 Summary of contributions 5

1.5 Thesis organization 7

CHAPTER 2 LITERATURE REVIEW 8

2.1 Convolutional neural networks 8

2.1.1 Overview of CNN operation 8

2.1.2 Image recognition benchmarks 10

2.1.3 Popular CNN models 11

2.2 Energy and power consumption 13

2.3 CNN model compression techniques 15

2.4 CNN accelerators and low-energy architectures 18

2.5 Conclusion 21

CHAPTER 3 CONVOLUTION UNIT ARCHITECTURE DESIGN WITH SERIAL ACCUMULATION DATAFLOW 23

3.1 Overview of CNN accelerator hardware design 23

3.1.1 Architecture design challenges 23

3.1.2 General functionality of CNN accelerators 24

3.2 Proposed CNN accelerator 25

3.2.1 Row-wise 3×3 convolution in the proposed architecture 26

3.2.2 Description of CU functionality for 3×3 convolution using an example 27

3.2.3 Controller design in the proposed architecture 30

3.3 3×3 convolution architecture performance analysis 32

3.4 Design verification methodologies 33

3.5 Results and discussions 34

3.6 Conclusion 36

CHAPTER 4 HETEROGENEOUS DISTRIBUTED SRAM CONFIGURATION FOR ENERGY-EFFICIENT DEEP CNN ACCELERATORS 37

4.1 SRAM configurations: existing methods and issues 37

4.2 Impacts of SRAM sizing on the number of DRAM accesses 39

4.3 Proposed CNN accelerator memory configuration 42

4.4 Results and discussions 44

4.5 Conclusion 46

CHAPTER 5 CARLA: A CONVOLUTION ACCELERATOR WITH A RECONFIGURABLE AND LOW-ENERGY ARCHITECTURE 47

5.1 The CARLA architecture 47

5.1.1 1×1 convolution mode 52

5.1.2 General operation of 1×1 convolution 53
5.1.3 Description of CU functionality for 1×1 convolution using an example 54
5.1.4 1×1 convolution architecture performance analysis 55
5.1.5 1×1 convolution mode for very small size in-fmaps 56
5.1.6 7×7 convolution mode and others 57
5.2 Results and discussion 58
5.2.1 The structure of ResNet-50 58
5.2.2 Performance and DRAM access results 58
5.2.3 Comparison with state-of-the-art CNN implementations 62
5.3 Conclusion 66
CHAPTER 6 SEMI-STRUCTURED RANDOM ROW-WISE PRUNING FOR ENERGY-EFFICIENT AND FAST CONVOLUTION ACCELERATOR 67
6.1 Overview of existing pruning methods and their challenges 67
6.2 Proposed semi-structured random row-wise pruning 68
6.3 Experimental results 70
6.4 Compatibility of CARLA with the proposed pruning method 71
6.5 Conclusion 73
CHAPTER 7 CONCLUSION AND FUTURE WORK 74
7.1 Summary of the work 74
7.2 Thesis limitations and future works 75
7.2.1 Word-length optimization 75
7.2.2 Supporting other CNN families 76
7.2.3 On-device CNN training 77
REFERENCES 78

LIST OF TABLES

Table 2.1 Number of operations and parameters for different CNNs. 13

Table 2.2 Energy consumption of computations and memory accesses in 45 nm ([8] ,[26]). 14

Table 2.3 Typical power consumption for GPUs and mobile devices. 14

Table 2.4 Implementation results for low-energy designs. 21

Table 3.1 Implementation results for low-energy designs benchmarked on VGGNet-16 (© 2020 IEEE from [74]). 35

Table 4.1 Number of computation partitions for convolutional layers of VGGNet-16. 40

Table 4.2 Implementation results (© 2020 IEEE adapted from [75]). 45

Table 5.1 Structure of ResNet-50 convolutional layers 58

Table 5.2 Comparison of the proposed method with the state-of-the-art. 63

Table 6.1 Comparison of accuracy of the proposed method with the state-of-the-art. 70

LIST OF FIGURES

Figure 2.1 Computation of a convolutional layer. 9

Figure 2.2 Max-pooling function (© 2018 IEEE from [73]). 10

Figure 2.3 Example images for CIFAR-10, MNIST, and SVHN datasets. 11

Figure 2.4 An example image for ImageNet datasets [16]. 11

Figure 2.5 Top-5 Classification error of different CNNs. 12

Figure 3.1 Partitioning convolution computations [76]. 24

Figure 3.2 A convolution engine with the proposed CU architecture (© 2020 IEEE from [74]). .25

Figure 3.3 Row-wise convolution to generate a sub-out-fmap. 27

Figure 3.4 Proposed dataflow for 3×3 convolution to generate a sub-out-fmap (adapted from [76]). 28

Figure 4.1 Baseline architecture with a ping–pong mechanism (© 2020 IEEE adapted from [75]). 38

Figure 4.2 Portion of utilized silicon area for MACs and SRAMs inside a CU. 39

Figure 4.3 Number of DRAM accesses vs. SRAM depth. 41

Figure 4.4 Silicon area for different depths of TSMC dual port SRAMs with 32-bit words. 41

Figure 4.5 Power vs. area consumption for different configurations of SRAMs (© 2020 IEEE from [75]). 43

Figure 4.6 Proposed heterogeneous SRAM configuration (© 2020 IEEE from [75]). 44

Figure 5.1 CARLA architecture [76]. 48

Figure 5.2 Configuration of buses in CARLA for 3×3 convolution. 49

Figure 5.3 Configuration of buses in CARLA for 1×1 convolution. 49

Figure 5.4 Common input features in different convolution steps of 3×3 convolutions. 50

Figure 5.5 Configuration of a CU for 3×3 convolution [76]. 51

Figure 5.6 Configuration of a CU for 1×1 convolution [76].....51

Figure 5.7 Computing sub-out-fmaps in 1×1 convolution.....53

Figure 5.8 Proposed dataflow for 1×1 convolution [76].....55

Figure 5.9 Splitting each plane of a 7×7 filter into 21 pieces [76].....57

Figure 5.10 PUF for convolutional layers in ResNet-50.....59

Figure 5.11 Computation time for convolutional layers in ResNet-50.....59

Figure 5.12 DRAM access numbers of in-fmaps and filter weights for ResNet-50.....59

Figure 5.13 DRAM access numbers of out-fmaps and total for ResNet-50.....60

Figure 5.14 Number of DRAM access for CARLA compared to FID for VGGNet-16 [76].....64

Figure 5.15 PUF for CARLA compared to ZASCAD for ResNet-50 [76].....65

Figure 5.16 PUF for CARLA compared to ZASCAD for ResNet-50 [76].....65

Figure 5.17 Number of DRAM access for CARLA compared to ZASCAD for ResNet-50 [76]. 66

Figure 6.1 Proposed random row-wise pruning method (adapted from [104]).....69

Figure 6.2 Number of DRAM access in CARLA for the proposed sparse model compared to baseline model.....72

Figure 6.3 Computation time of CARLA for the proposed method compared to baseline model.72

LIST OF SYMBOLS AND ABBREVIATIONS

ASIC	Application-Specific Integrated Circuit
CNN	Convolutional Neural Network
CU	Convolution Unit
DNN	Deep Neural Network
DRAM	Dynamic Random-Access Memory
FC	Fully Connected
FPGA	Field Programmable Gate Array
GPU	Graphics Processing Unit
ILSVRC	ImageNet Large Scale Visual Recognition Challenge
In-fmap	Input feature map
LFSR	Linear Feedback Shift Register
MAC	Multiply-Accumulate
MCR	Misclassification Rate
Out-fmap	Output feature map
PE	Processing Element
PUF	Processing Element Utilization Factor
RELU	Rectified Linear Unit
ResNet	Residual Network

SRAM	Static Random-Access Memory
TPU	Tensor Processing Unit

CHAPTER 1 INTRODUCTION

1.1 Overview and motivation

Convolutional Neural Networks (CNNs) have demonstrated breakthrough results for several computer vision tasks such as image classification [1], object detection [2], segmentation[3], and activity recognition [4]. The success of CNNs mainly came to be known in the ImageNet Large Scale Visual Recognition Challenge (ILSVRC), which has been held annually since 2010 with the goal of finding the image classification methods. The image classification problem consists of identifying the class to which an object inside an image belongs to. In 2010, a CNN model known as AlexNet showed only 16% misclassification rate (MCR) while the MCR of the runner up using traditional approaches was 26%. This great improvement had a profound effect on the computer vision community and since then all the winners of the ILSVRC have used CNNs in their works. In 2015, the Residual Network (ResNet) achieved an MCR of 3.6%, which outperformed the human level accuracy [9].

CNNs are specific types of Deep Neural Networks (DNNs) constructed by stacking multiple layers of neurons, each transforming the representation at one level into a more complex representation at a higher level [5]. The feature extraction in deep CNNs is performed using a series of convolutional layers, each one applying multiple filters on its input. This structure is largely inspired from the animal visual cortex [6]. The first layers extract low-level features, e.g. edges from a raw image, while later layers obtain motifs and more complex structures. Typically, the convolutional layers are followed by Fully Connected (FC) layers where the neurons are connected to each output from the previous layer. At the end of all these layers, a classifier computes class scores. A CNN is first trained using a training dataset, such as ImageNet. Then, the trained parameters are used in inference for recognizing the test images.

In parallel with recent improvements in image recognition, real-word smart applications such as virtual reality [7] and smart wearable devices have experienced many advancements and gained considerable attention. This motivates the deployment of state-of-the-art CNNs on smart portable devices. However, the massive data movements and computational complexity of CNNs incur significant challenges to power efficiency and performance. Graphics Processing Units (GPUs) are widely used devices to perform classification tasks with complex CNN models. The high power

and energy consumption of GPUs, however, makes them unfit for computing CNN inference in portable and embedded devices. Typically, GPUs performing CNN computations consume over 200 watts [10], which is well beyond the power envelope of typical mobile devices. In addition, running complex CNNs requires massive computations and memory accesses which results in fast battery drain. Furthermore, many vision applications are latency-sensitive, demanding fast CNN inference, typically within hundreds of milliseconds for each image sample. For mobile applications, however, CPUs are not commonly fast enough to perform inference of large CNNs within the requested time frame. Therefore, in a mobile device, utilizing a hardware accelerator for convolutions can help to speed up CNN inference.

This work proposes architectures and techniques to facilitate the massive deployment of CNNs on portable devices. This in turn could result in improved quality of life, from high performance and availability of smart portable devices and embedded systems in applications such as health care, communications, entertainment, and the management of our profiles in the digital society.

1.2 Problem statement

Designers of CNN accelerators for mobile applications face several challenges regarding computation complexity, classification latency, power envelope of the mobile devices and their energy consumption.

A CNN typically consists of many cascaded layers, most of them being convolutional layers. It is well accepted that, in most cases, increasing the number of convolutional layers can improve the image classification accuracy [9]. Such an increase, however, raises the computational complexity of CNNs and the number of parameters. For instance, in ResNet-152, the number of convolutional layers increases to 152 layers, 80 M parameters and 11.3 G multiply-accumulate (MAC) operations.

In addition, in state-of-the art deep CNNs, the diversity of computations varies significantly from one convolutional layer to the next. For instance, the convolutional layers in ResNet models perform high-dimensional convolution operations using up to 2048 filters of different sizes, including 7×7, 3×3, and 1×1, and different strides of 1 and 2. In addition to these diversities, the

dimension of the input image varies in different layers from 224×224×3 in the first layer to 7×7×2048 in the last one.

Moreover, many computer vision applications such as action recognition in video surveillance, body-worn cameras and pedestrian and obstacle detection in self-driving vehicles are latency-sensitive and require real-time classification. In other words, it is crucial to compute the CNN inference quickly to allow completion of the classification task and related decision making to be performed within few milliseconds.

Furthermore, the power consumption of CNN accelerators must remain within the order of a few hundred milliwatts to meet the power budget of mobile devices. Additionally, mobile devices are battery constrained, while running deep CNNs consumes considerable energy. Energy consumption depends on the on-chip power consumption over time plus the energy consumption caused by external memory access. Due to the high complexity of deep CNNs, the network parameters i.e. the filter weights, and a large part of computation results have to be stored in an off-chip memory, e.g. DRAM, rather than on-chip storages such as SRAMs and registers. While an SRAM access consumes only 5 pJ in 45 nm technology, a DRAM access requires 120× more energy, i.e. 640 pJ, which results in faster battery drain [8].

Such tight power, energy and latency requirements make it difficult to run complex CNN models on mobile devices. In addition, it is required to have a flexible design that allows a single hardware to support all the diversities in convolutional layers.

1.3 Research objectives

In this work, we aim to design efficient CNN accelerators amenable to portable device implementation. We focus on CNN inference, since the training process is not expected to be performed on resource-constrained devices at this time. We thus propose highly efficient inference engines to compute convolutional layers of state-of-the-art CNNs such as VGGNet and ResNet. These CNNs contain massive numbers of parameters and operations with high variations in the number of filters, filter size and stride, and input size across layers. The proposed accelerators therefore have high flexibility to support a wide array of convolution configurations, they consume less energy than state-of-the-art designs, and they meet tight latency constraints. To meet portable

device power constraints, the design will be implemented on Application Specific Integrated Circuit (ASIC).

The research objectives of this project are as follows:

O1- Proposing new efficient dataflows to perform convolution. Several architectures have been introduced in the literature for CNN computation. CNN architectures utilize specific computational dataflows for fetching information from an off-chip memory, perform computations and write back the results. A major challenge for CNN architecture design is to effectively map the diverse convolution layers on the inherently fixed computing structure of the hardware. The vast majority of existing architectures, therefore, are plagued with low resource utilization and rarely maintain their advertised peak throughput. To accelerate the convolutional computations on deep CNNs, we will propose new dataflows for different convolution configurations that efficiently map the diverse computations on hardware resources.

O2- Proposing new reconfigurable and scalable CNN architectures. Diversity across convolutional layers requires well-engineered dataflows for efficient mapping of computations to hardware. For a specific architecture to support different dataflows with low overhead, it must support several operation modes through reconfiguration. Reconfigurable architectures should also exploit parallelism and pipelining to achieve high throughput, and they should maximize data reuse to minimize the number of memory transfers. The operation modes must then determine the function of parallel units, the portion of the computations that each unit must perform, and the data movement schemes.

O3- Proposing new on-chip memory configurations to limit off-chip memory accesses. CNNs are complex networks with a large number of parameters. These parameters must be stored in an off-chip DRAM and are fetched prior to computation. Off-chip DRAM is also used to store computation results. DRAM accesses consume orders of magnitude more energy than the computations themselves. Having a proper memory configuration and efficient dataflow increases data reuse inside the accelerator and reduces the number of DRAM accesses. We will thus propose new memory configurations with on-chip SRAMs of the proper size, configuration and integration in the computation process to improve energy-efficiency and to reduce design cost.

O4- Decreasing network complexity using sparsity. A well-known approach to reduce the number of DRAM accesses is to prune the CNN by removing unimportant weights. However, many existing weight pruning approaches have demonstrated a poor fit with custom hardware implementations. In fact, the pruned networks lose the regular structure of dense matrices while pruned connection addresses must still be recorded in memory. This results in latency overhead for computing unpruned weight indices, while the irregular pruned weight distribution impedes computation acceleration. Therefore, we will propose new pruning strategies tuned for our custom hardware architecture. The proposed strategies prune the weights based on the computation patterns of the proposed architecture without requiring additional memory to keep the pruned indices, thus affecting less the computation throughput.

1.4 Summary of contributions

An Energy-Efficient Accelerator Architecture with Serial Accumulation Dataflow for Deep CNNs. This work proposes a new architecture for accelerating computation of convolutional layers. A major advantage of this architecture is its capability to efficiently utilize all the hardware computational units in all clock cycles. The proposed architecture includes several parallel units, each comprised of cascaded MAC operators whose outputs are accumulated in a serial fashion. The architecture is evaluated using the convolutional layers of VGGNet-16, a widely used benchmark model. The results demonstrate that the proposed architecture can significantly reduce the computation latency and the number of DRAM accesses compared to state-of-the-art designs. This work received a best student paper award at NEWCAS 2020 [74].

Heterogeneous Distributed SRAM Configuration for Energy-Efficient Deep CNN Accelerators. This work proposes a new memory configuration for CNN accelerators to facilitate the data transfer from internal SRAMs to the external DRAM. Compared to the well-known ping-pong mechanism, which uses identical SRAMs to overlap data communication with convolution computations, the proposed method utilizes SRAMs with different sizes and assigns different roles to them. While shallow SRAMs with large word-length are used by processing elements (PEs) locally, deeper but narrower SRAMs are shared among adjacent parallel units. The shared SRAMs store the final results at the end of each computation iteration. These results are then gradually transferred to the off-chip DRAM. Compared to a ping-pong mechanism architecture, the proposed configuration

achieves 18% higher energy efficiency while reducing the memory footprint and the utilized silicon area. This work was presented in NEWCAS 2020 [75].

CARLA: A Convolution Accelerator with a Reconfigurable and Low-Energy Architecture. This work proposes CARLA, a convolution accelerator architecture with a reconfigurable and low-energy architecture, which supports efficient computation of diverse convolutional layers. CARLA uses several homogenous hardware units to perform computations in parallel. It handles the computational diversity of different layers using several operating modes with distinct computational dataflows. CARLA utilizes a new dataflow architecture for 1×1 convolutions that is integrated with the architecture of 3×3 convolutions to achieve low reconfiguration cost. When evaluated with ResNet-50, this architecture achieves a high processing unit utilization factor of 98% across the majority of 1×1 and 3×3 convolutions and supports other convolution structures such as 7×7. This results in a latency of only 92.7 ms and total DRAM accesses of 124.0 MB [76]. This work has been submitted to the IEEE Transactions on Circuits and Systems I: TCAS-I, Special Issue on CASS Conferences.

Semi-structured random row-wise pruning. This work proposes a semi-structured random row-wise pruning to achieve a high pruning ratio. While weight pruning methods often result in computation irregularity, the proposed method maintains computation regularity by removing the weights in a row-wise fashion based on the hardware dataflow. This results in fewer parameters and decreased computational complexity. For this purpose, instead of removing individual weights, a group of filter rows are removed together. This discards the corresponding computations in the accelerator parallel units and allows to proceed to the next group of computation. To avoid hardware overhead, the pruned row indices are determined by using Linear Feedback Shift Registers (LFSRs). Experimental results with the ImageNet dataset show that the proposed method can remove 50% of parameters in convolutional layers with only 1.45% accuracy degradation accuracy while speeding up the image classification. This work is protected as a provisional patent by Huawei [77].

In addition to the mentioned works on efficient CNN accelerator design, the following work on the topic of pooling layers was presented in NEWCAS 2018.

Power Reduction in CNN Pooling Layers with a Preliminary Partial Computation Strategy. This work proposes a new method to reduce CNN power consumption by simplifying computations before the max-pooling layers. The proposed method estimates the max-pooling layer outputs by approximating the value of the preceding convolutional layer with a preliminary partial computation. Then, the method performs a complementary computation to generate an exact convolution output only for the selected feature. We also present an analysis of the approximation parameters. Simulation results show that the proposed method reduces the power consumption by 21% and the silicon area by 19% with negligible degradation in classification accuracy for the CIFAR-10 dataset [73].

1.5 Thesis organization

This thesis is organized as follows. Chapter 2 reviews the literature by presenting an overview of CNN structures, popular benchmarks to evaluate the classification accuracy, and widely used CNN models. Then, the main sources of the energy and power consumption in CNNs are characterized and current approach to reduce the complexity of CNNs as well as state-of-the-art hardware designs are reviewed. In chapter 3, an accelerator with serial accumulation dataflow is presented and the implementations results are compared with existing designs for the VGGNET-16 model. In chapter 4, on-chip memory sizing and configuration are analyzed and a heterogeneous distributed SRAM configuration to achieve more power and area efficiency in hardware is proposed. In chapter 5, a new reconfigurable and low-energy accelerator to support more convolution configurations is presented. The implementation results in this chapter are based on the ResNet-50 model. Chapter 6 presents a new pruning method which is compatible with the proposed accelerator to reduce both the number of parameters and computations. Finally, the conclusion is drawn in chapter 7.

CHAPTER 2 LITERATURE REVIEW

In this chapter, we review the current state-of-the-art with respect to current knowledge of CNN layers, classification benchmarks and CNN models. We then investigate current CNN complexity reduction trends that exploit model compression techniques, including sparsity. Finally, we review current CNN hardware implementations in ASICs and summarize implementation results.

2.1 Convolutional neural networks

2.1.1 Overview of CNN operation

Figure 2.1 shows the convolution operation in a convolutional layer. A convolutional layer takes as input a 3-D matrix of size *IL*×*IL*×*IC*, convolves it with *M* filters of size *FL*×*FL*×*FC*, and produces a 3-D output of size *OL*×*OL*×*OC*. The quantities *IL*, *FL*, and *OL* denote the length, and *IC*, *FC* and *OC* indicate the depth, also called the number of channels, of the corresponding matrices. Each input or output matrix element is called a feature, and each filter element is called a weight. In the convolution operation, the number of input channels equals the number of filter channels, i.e., *IC* = *FC*. In addition, the number of filters is equal to the number of output channels, i.e., *M* = *OC*. The convolution operation is given by

$$y_k(m,n) = b^k + \sum_{c=0}^{IC-1} \sum_{j=0}^{FH-1} \sum_{i=0}^{FL-1} x_c(m \times s + j, n \times s + i) \times w_c^k(j,i)$$

$$0 \le m,n \le OL, 0 \le k \le OC, OL = (IL - FL + 2z)/s + 1 \quad (2.1)$$

where *y*, *b*, *x*, and *w* denote the elements in output, bias, input and filter matrices, respectively. The *z* index denotes the number of zero pads used to preserve the spatial size of the output features and *s* indicates the filter stride [12]. In (2.1), for an element in a matrix, $x_c(r,q)$, *c* represents the channel index in the matrix, while *r* and *q* indicate the row and column of the element, respectively. Similarly, for the filter weights, $w_c^k(j,i)$, *c*, *j* and *i* denote the channel index, row number and column number, respectively, while *k* represents the filter index.

Each convolutional layer is followed by a non-linearity function. The Rectified Linear Unit (RELU) is commonly used as the non-linear function [5]. RELU can be represented as follows:

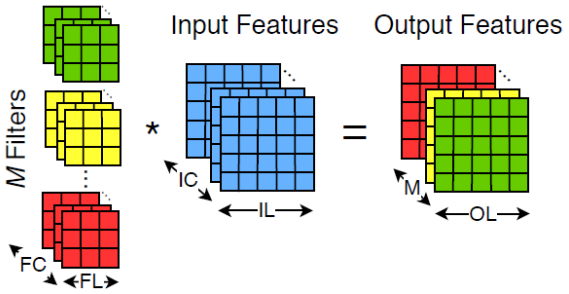


Figure 2.1 Computation of a convolutional layer.

$$f(x) = max(0, x) \tag{2.2}$$

Based on (2.2), the positive activations remain unchanged while the negative ones are set to zero. The RELU layer does not modify the data size or the number of dimensions.

Some CNN models include pooling layers between groups of convolutional layers to reduce the spatial dimensions ($IL \times IL$) of the input values for the following layers. This dimension reduction can lead to information loss but it reduces computational cost [12]. Figure 2.2 shows a max pooling layer with filter size 2×2 and stride of 2 which selects the maximum value among four adjacent inputs and moves by 2 over the input, discarding 75% of the inputs.

In a CNN model, the output of the last layer is delivered to FC layers in which all the input nodes are fully connected with the output nodes. The output of an FC layer is computed with a matrix multiplication followed by a bias offset as:

$$x(m) = f(W^t x(m-1) + b) \tag{2.3}$$

where W, b and $f(.)$ are a weight matrix, a bias vector and an activation function, respectively. The fully connected layers are usually placed last, just before the classifier layer in CNNs, to construct the desired number of outputs for the network. One of the well-known classifiers is softmax, which represents the following function:

$$f(x_i) = \frac{exp(x_i)}{\sum_{j=0}^{n-1} exp(x_j)} \quad (i = 0, \ldots, n-1) \tag{2.4}$$

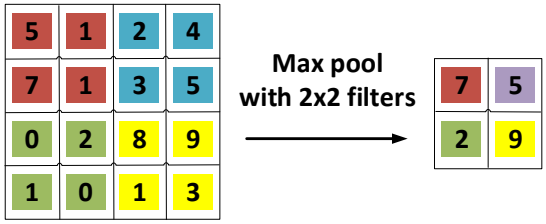


Figure 2.2 Max-pooling function (© 2018 IEEE from [73]).

where x is the classifier input and n is the number of classes [101]. The classifier function assigns a score to each input for each of the output classes. Then, the index corresponding to the maximum value among the outputs is returned as the classification result.

2.1.2 Image recognition benchmarks

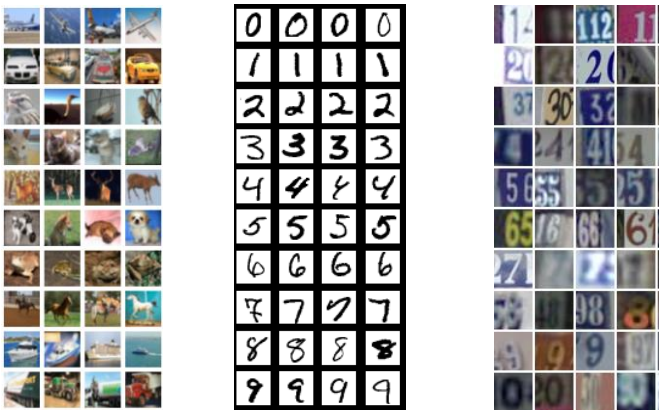
Common image recognition benchmarks are essential to accurately assess the performance of CNN architectures, models, algorithms and implementations. The standard benchmarks for image classification can be divided according to their datasets of tiny or large images. Figure 2.3 shows typical images from three datasets with tiny images: CIFAR-10, MNIST and SVHN.

CIFAR-10: The CIFAR-10 image dataset [13] contains 60,000 images of 32×32 pixels, RGB images of natural scenes which are categorized in 10 different classes. The dataset is divided into 50,000 training images, and 10,000 test images.

MNIST: The MNIST image dataset [6] has 50,000 training images and 10,000 test images of handwritten digits. Its grey value image of size 28×28 pixels are categorized in 10 classes.

SVHN: The SVHN dataset includes 32×32 pixels RGB images of house numbers [14]. It consists of more than 600,000 training images and 26,032 test images.

ImageNet: The most common large-scale benchmark for the image recognition task is ImageNet [15]. ImageNet has more than 1,200,000 training images and 50,000 validation and test images which are categorized in 1000 classes. The images in this dataset are natural images with higher resolution, e.g., 256×256×3, compared to the ones in tiny image datasets. Since 2010, the ImageNet Large Scale Visual Recognition Challenge (ILSVRC) has been held annually where research



a) CIFAR-10 [13] b) MNIST [97] c) SVHN [14]

Figure 2.3 Example images for CIFAR-10, MNIST, and SVHN datasets.

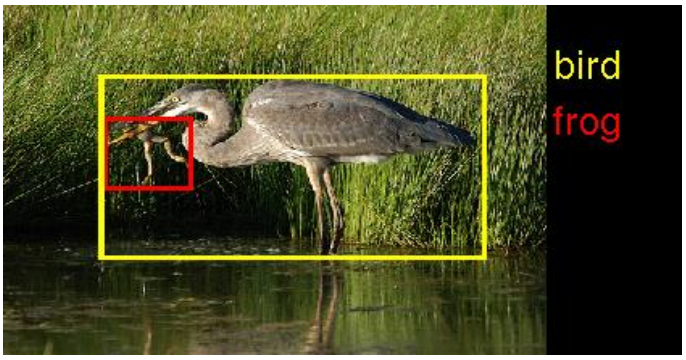


Figure 2.4 An example image for ImageNet datasets [16].

groups submit their methods for classifying and detecting objects in the ImageNet dataset [16]. Since 2012, all the winners of ILSVCR have used CNNs as their recognition structure. Figure 2.4 shows an example image of the ImageNet dataset.

2.1.3 Popular CNN models

CNN models are widely used for image classification. The most popular CNN models tend to be winners of the ImageNet challenge and they are widely used for image classification. Typically, the image classification results of CNN models are reported in terms of top-1 and top-5 numbers. The top-1 number indicates the number of times the correct label predicted by the CNN model with highest probability. The top-5 number, on the other hand, denotes the number of times the correct label is among the top-5 predicted classes by the CNN model.

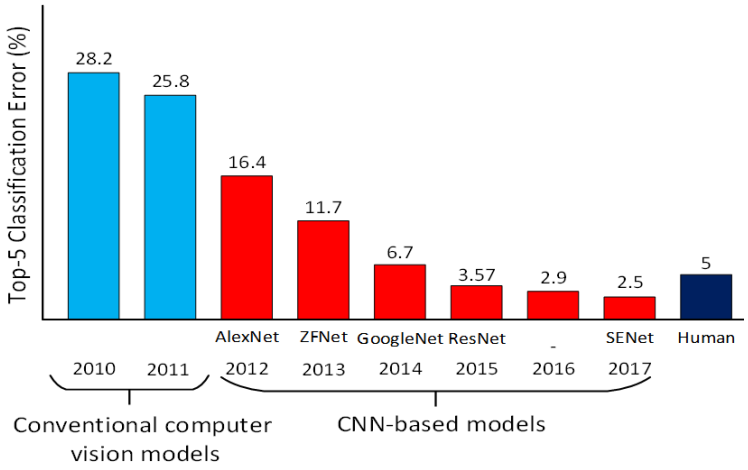


Figure 2.5 Top-5 Classification error of different CNNs.

As shown in Figure 2.5, by introducing the CNN models for image classification task the top-5 classification error of ImageNet dataset has been reduced from 28.2% to 16.4% in 2012 and then it continuously decreased each year to 2.5% in 2017. The most popular state-of the-art CNN models are as follows:

AlexNet [1]: It was the first model that popularized CNNs for image recognition task by winning ILSVRC 2012. The model was inspired from LeNet [6], but revealed a deeper and larger structure. It was the first deep CNN model that has been implemented on GPUs.

ZFNet [17]: It was the winner of ILSVRC 2013. It improved AlexNet by means of optimizing the model parameters and using smaller filter sizes and strides on the first layer.

GoogLeNet [18]: The winner of ILSVRC 2014 was a CNN proposed by Google, introducing an Inception module that leads to a significant reduction in the number of network parameters. It also used an average pooling layer instead of fully connected layers.

VGGNet [19]: It was the second ranked in ILSVRC 2014. It gained attention with its regular structure including only 3×3 filters and 2×2 pooling windows for feature extraction. The main drawback of VGGNet is its high number of parameters (140 M) which require a huge amount of storage.

ResNet [9]: Residual Network (ResNet) was the winner of ILSVRC 2015. It was developed by Microsoft Research Asia. It also won the 1st place on several competitions including ImageNet

Table 2.1 Number of operations and parameters for different CNNs.

Network Name	# Floating Point Operations	# Parameters
AlexNet	0.725 G	60 M (240 MB)
VGGNet	19.6 G	138 M (552 MB)
ResNet-18	1.8 G	14.5 M (58 MB)
ResNet-152	11.3 G	82 M (328.8 MB)

detection, ImageNet localization, COCO detection, and COCO segmentation. The ResNet model utilizes identity shortcuts (ISs) which directly connect the input of each layer to the output. This allows to stack more layers and gain more accuracy by increasing depth[9]. Compared to VGGNet, the ResNet structure includes 7×7, 3×3 and 1×1 filters and has more variations in computations across convolutional layers. ResNet has been the default CNN network for image recognition since 2016.

The winner of ILSVRC 2016, used multi-scale ensembling of Inception, Inception-ResNet, ResNet and wide ResNet models [12]. The SENet model is the winner in 2017, which improved ResNet by adding a feature recalibration module [20]. The annual ImageNet competition stopped after 2017 and it is moved to Kaggle which is an online data science community.

The ResNet is substantially deeper and more accurate than the previous CNN models such as VGGNet. For instance, the ResNet-152 (including 152 layers) reduces the top-5 classification error to only 3.57%, outperforming human level accuracy. Among different ResNet models, ResNet-50 is widely used as a benchmark in the literature and we evaluate our accelerator with this model.

2.2 Energy and power consumption

CNNs are computationally and memory intensive, making it difficult to run them on mobile devices with limited computational resources and constrained battery life. The network size is proportional to the number of weights stored in memory, while the computational complexity is proportional to the number of arithmetic operations necessary to classify an image. Table 2.1 shows the required number of arithmetic operations and parameters of several CNNs for image classification.

Energy efficiency is one of the most demanding issues for CNN implementation in mobile devices. Horowitz et al. reported the energy consumption of computations and memory accesses in 45 nm

Table 2.2 Energy consumption of computations and memory accesses in 45 nm ([8] ,[26]).

Operation	Add	Mult	SRAM	Off-chip DRAM
32-bit integer	0.1 pJ	3.1 pJ	5 pJ	640 pJ
32-bit floating point	0.9 pJ	3.7 pJ	-	-

Table 2.3 Typical power consumption for GPUs and mobile devices.

Device	Power (W)
GPU (Tesla K40) [22]	235
Mobile GPU (Tegra K1) [24]	11
Tablet [11]	10
Smartphone [11]	5
Video Recording with Google Glass [23]	3
Video Recording with Smartphone [11]	1.5
Video Recording with GoPro (Wi-Fi off) [24]	1.5

technology [8] as shown in Table 2.2. The authors report that the energy cost of a DRAM access is 200× greater than a MAC operation. Even with emerging HBM (High Bandwidth Memory) DRAM solutions [103], DRAMs still considerably consume more energy compared to on-chip components. Unfortunately, the large number of input features, filter weights and output features in CNNs requires using an off-chip DRAM to store data. This makes the reduction of the number of DRAM accesses a prime target, which favors data reuse in on-chip memories. The on-chip energy usage is equal to the on-chip power consumption multiplied by the total execution time. The total energy is obtained as,

$$\text{Energy} = \#\text{DRAM Access} \times \text{Energy}_{\text{DRAM}} + \text{Power}_{\text{on-chip}} \times \text{Time} \tag{2.5}$$

A valid solution should also meet power consumption and performance constraints. Performing computation dissipates power that should be less than the power envelope of the mobile devices on which they are implemented.

Table 2.3 illustrates the typical power consumption of typical GPUs, several mobile devices and applications. Low power solutions for CNNs on portable devices should aim to consume less power than a few hundred mW [21].

One important metric in CNN hardware design is Processing element Utilization Factor (PUF), which indicates the proportion of time that the processing elements are actively contributing to computations.

$$\text{PUF} = \frac{\text{Operations}}{\#\text{PEs} \times \text{Clock cycles}} \times 100 \tag{2.6}$$

The PUF depends on the effectiveness of the employed dataflow to map the computations to available resources. Due to the high variations in the number of filters, filter sizes and strides, and feature map sizes across different layers in CNNs, finding a dataflow that achieves a high PUF across different layers is a significant challenge [68]. Most of the existing designs cannot reach their advertised peak performance with a fixed architecture when applied to state-of the-art CNNs. Therefore, among different implementation metrics the number of DRAM accesses and PUF are of profound importance when comparing low-energy accelerators. Increasing the PUF improves hardware efficiency and reduces computation time. According to the data in Table 2.2, reducing the number of off-chip DRAM accesses has a significant positive impact on energy efficiency.

2.3 CNN model compression techniques

During recent years, several model compression techniques have been proposed to reduce the number of parameters and operations in CNNs [25]-[47], [80]-[96]. There are three main techniques in CNN model compression : compact models [36]-[39], low-bit quantization [40]-[43], [87]-[92], and pruning [25]-[34], [93]-[96].

Compact models aim to reform the complex standard convolution operation and replace it with simpler ones. For instance, the structure of the MobileNet families mainly consists of 1×1 point-wise convolutions and 3×3 depth-wise convolutions [38], [39]. The 1×1 point-wise convolution is the same as 1×1 standard convolution, in which filters have a single weight in their spatial dimensions and perform convolutions along input channels. In depth-wise convolution, the filters have only one channel [38]. This reduction in the number of channels noticeably reduces the number of parameters and of computations. In depth-wise convolutions, different filters operate on different input channels with no sharing of data. This requires a very high bandwidth for DRAM to provide a high volume of independent data for computational units, which can be a costly solution. for mobile devices.

Low-bit quantization reduces CNN complexity by reducing data word-length. In the extreme case, network values are represented by either -1 or 1, and this is called binarization. Courbariaux et al. proposed a heuristic method called BinaryConnect (BC) to constrain the weights to two values instead of floating-point numbers of 32 bits [40]. BinaryConnect achieved classification results close to the state-of-the-art for the CIFAR-10, MNIST and SVHN datasets. Courbariaux et al. extended the binarization idea to activation maps and proposed Binarized Neural Network (BNN) [41]. Since both weights and activations have binary values, the 32-bit floating-point MACs are changed to 1-bit XNOR-count operations. Similarly, Rastegari et al. proposed the BWN and XNOR-Net methods for the ImageNet dataset [42]. Although BWN shows promising results for networks such as AlexNet, the gap in classification accuracy between the binarized and full precision networks is rather high in deeper networks such as ResNet. Using ternary weights whose values are constrained to {-1, 0, 1} is another way to reduce CNN complexity while achieving better accuracy than binary networks [43]. These techniques have generated a lot of interest in the recent years; however, a recent survey highlights the fact that binarization and ternarization still cannot achieve high accuracy in large CNN models [35].

From a hardware perspective, using binary values does not always result in an efficient design. For instance, Andri et al. proposed a low-power accelerator for binary-weight CNNs called YodaNN[21], but its PUF is limited to 28% when running VGGNET on CIFAR-10 [44]. Therefore, it is necessary to have an efficient dataflow to properly map computations into hardware in a way that all the processing units are active during computational cycles. In other words, by improving the PUF, the hardware efficiency is increased and the total computation time is reduced.

Pruning consists of setting the value of some weights to zero, therefore eliminating the need to fetch them from memory and to perform their associated computation. These removed weights must be selected carefully in order not to negatively impact the classification accuracy of CNNs. Very early in the research on neural networks, it was in fact observed by LeCun et al. that pruning can improve the accuracy [79]. Han et al. present a three-stage pipeline including pruning, trained quantization and Huffman coding in order to decrease the number of weights while maintaining the original accuracy of the neural network [25]. The first step is to prune unimportant connections by setting weights whose value is below a given threshold to 0. Then, the remaining weights are quantized to a reduced shared set with small values. Next, Huffman coding is applied to achieve

more compression. The proposed method results in up to 91% sparsity in the fully connected layers of AlexNet and achieves a reduced storage of 6.9 MB instead of the initial 240 MB, without loss of accuracy. For VGGNet-16, the resulting sparsity is up to 96%, or 11 MB storage compared to the original 552 MB. The same pruning method was later used by the same authors to design an energy efficient inference engine (EIE) for the inference phase of the fully connected layers of AlexNet and VGGNet [26]. Nurvidathi et al. showed that applying pruning can achieve up to 85% and 65% sparsity for the AlexNet and GoogleNet convolutional layers, respectively, with only 1% and 0.2 % degradation in accuracy, respectively [27].

Yu et al. implemented weight pruning on different hardware platforms and showed that network sparsity often causes a reduction in DNN computation throughput [28]. Network sparsity breaks matrix structure regularity and incurs extra address computations to access the weights.

Since the pruned weights can be located randomly in the network, the indices of unpruned connections could be stored instead of being computed. Shafiee et al. proposed an alternative method by randomly removing up to 61% connections in the fully-connected and convolution layers of DNNs before training [29]. From a hardware implementation perspective, such a method is more desirable since generating pseudo random connections is much easier than storing random locations. Ardakani et al. proposed a model called sparsely-connected network which uses linear-feedback shift registers (LFSRs) to generate random connection masks [30]. The proposed method improves network accuracy for small datasets by 2.73% compared to the baseline unpruned network while removing up to 90% of the connections and saving up to 84% energy compared to the conventional architectures. The latency of the network is unaffected.

In order to achieve speed up it is necessary to preserve computation regularity in the pruned sparse matrix. Removing individual weights in a fine-grained pruning approach, however, tends to result in irregular computation structures. Coarse-grained sparsity, on the other hand, is more suitable for a CNN accelerator because of its potential in the reduction of index storage. Mao et al. explored different sparsity granularities including weight, vector, kernel and filter pruning using a magnitude-based pruning method [31]. They showed that the resulting classification accuracies after moderate-grained sparsity, i.e., vector (row) and kernel (channel) pruning, can be similar to the fine-grained sparsity, i.e. by individual weights. Their method to prune the whole filter (entire weights in a filter), however, results in noticeable drop in accuracy.

Compared to other sparsity-levels, pruning filters incur less indexing overhead, preserves the structure of the CNN model with fewer parameters, and results in a compact model without requiring special library. Since removing a filter is equal to removal of one output channel, this approach is also called channel pruning [34]. Liu et al. propose a method called network sliming that automatically detect the insignificant channels during the training and prunes them [32]. Network sliming uses l1 regularization on scaling factors of the channels in batch normalization layers and prune the channels with small scaling factors. Several studies has shown pre-trained models are better choices for pruning compared to pruning models from scratch since the pre-trained models includes learning task information [33], [40], [34]. For instance, Luo et al.[34] propose a method called Thinet to remove the filters in convolutional layers after training based on the statistic information form it succeeding layers [34]. In general, structured sparsity faces more issues to preserve the model accuracy [35].

In this thesis, we will propose a semi-structured pruning method that uses a low-complexity process to remove filter weights and achieves high pruning rate while it is designed based on the computational patterns of CNN accelerators.

2.4 CNN accelerators and low-energy architectures

Deep learning algorithms such as CNNs are typically trained and executed on powerful GPUs [45], [48]. The most remarkable work on GPU implementation was first proposed by Krizhevsky et al. after a CNN was successfully implemented on two NVIDIA GPUs for ILSVRC [1]. In addition, many deep learning frameworks such as CAFFE [57], Tensorflow [58] and Pytorch [59] use GPUs for acceleration. Although GPUs offer tremendous computation speed up, they are power hungry devices. For instance, the NVIDIA Tesla K-40 consumes 235 watts [10].

As an alterative to GPUs, a large number of customized neural network processors have been introduced for accelerating deep learning tasks on servers and datacenters [50], [98]-[99][50] where strict energy and power consumption constraints are not normally demanded. For example, the Google Tensor Processing Unit (TPU), which is a custom reconfigurable accelerator for computations in FC and convolutional layers, consumes 40 W [50].

Field Programmable Gate Arrays (FPGAs) are also used for CNN accelerations. Zhang et al. implemented convolutional layers of AlexNet on a Virtex-7 Xilinx FPGA and they achieved a frame rate of 46 images/s with a power budget of 18.61 W [51]. Their implementation outperforms the previous works (Cadambi et al. [52], Farbet et al. [53], [54], Lacey et al. [55]). Ovtcharov et al. from Microsoft achieved a performance of 134 images/s operating at 25 W on a Stratix V Altera FPGA for AlexNet [10]. Qui et al. implemented VGGNet-16 on a Xilinx Zynq ZC706 board and achieved a frame rate of 4.45 frames/s at 9.63 W. Overall, FPGA-based CNN accelerators typically consume power in the order of tens of watts [56].

Another approach is to design accelerators that only consume a few hundreds of milliwatts to fit within the power budget of mobile devices [21]. This class of accelerators can be used by host mobile processors to perform the convolution computations. Several works have proposed energy efficient accelerators on Application-Specific Integrated Circuit (ASIC). Early works in this class of accelerators benchmarked their implementations using small CNNs [60]-[62]. With emerging large CNNs in the ImageNet challenge, the image size and number of filter weights become large, demanding off-chip DRAM to store data. Therefore, in state of the-art-deep CNNs, one of the most important factors dictating the system energy efficiency is the number of DRAM accesses.

Chen et al. proposed a convolutional accelerator called Eyeriss fabricated in 65 nm CMOS technology [63]. Eyeriss performs convolutional computations of AlexNet and VGGNet-16 in 115.3 ms and 4.3 s while requiring 15.4 MB and 321.1 MB memory accesses and using batch size of 4 and 3, respectively. The proposed architecture consists of an off-chip memory, buffer and a spatial array. Using a 108 KB SRAM as the buffer helps reusing intermediate data and reducing access to off-chip DRAM memory. Data compression reduces DRAM memory bandwidth by a factor of 2×. In addition, zero skipping and clock gating prevent unnecessary computations and result in 45% power saving in PEs. In Eyeriss v2 [64] , design flexibility is increased by integrating PEs and global buffers into 16 clusters. Eyeriss v2 was benchmarked on a small CNN model from the MobileNet family with the same accuracy as AlexNet and occupies 2× more silicon area compared to Eyeriss.

Moons and Verhelst presented a low-power scalable processor for the convolutional layers of AlexNet [65]. In contrast to the work by Chen et al [63], which keeps the computational precision constant, the design by Moons et al. uses dynamic precision scalability for different layers. The

scaled computations decrease the switching activity and critical path and result in a possible lower supply voltage. The proposed design consists of a scalable power domain for convolutional operations and a fixed supply voltage domain including memories, controller and non-scalable arithmetic. In addition, the chip is clock-gated and the computations for zero-valued weights or pixels are eliminated to save dynamic power. The fabricated processor in 40 nm technology dissipates 76 mW for 47 fps. Moons et al. then achieved a higher reduction in power consumption with the Envision accelerator fabricated in FD-SOI 28 nm technology [66]. This architecture consumes 26 mW with a frame rate of 1.67 fps to implement VGGNet-16.

Wang et al. proposed a 1D chain architecture to accelerate convolutional layers, implemented in 28 nm CMOS technology [67]. This architecture operates on a batch of 128 images and achieves a throughput of 326.2 fps, a latency of 393.36 ms, and it consumes 567.5 mW on AlexNet.

Ardakani et al. proposed a dataflow called fully-connected inspired dataflow (FID) which is customized for VGGNet-16 and VGG-like networks based on computational cores of fully-connected layers [69]. The work shows that the convolutional layer operations can be treated as a special case of fully connected ones where a set of weights is shared among all neurons. The architecture utilizes computational units called Tiles, each consisting of sets of neurons and a weight generator unit that provides neuron weights using a shift register. The proposed architecture achieves a power consumption of 260 mW with total latency of 453.3 ms while demanding 331.7 MB of DRAM accesses. ZASCA is built on FID to support a larger number of filter sizes [71]. Compared to FID, ZASCA requires 2× larger silicon area and it can perform the convolutional layers of complex CNN models such as ResNet-50. The ZASCA configuration for computing the original CNN model (dense activations) is called ZASCAD in [71]. Table 2.4 illustrates the implementation results of the state-of-the-art designs for CNNs benchmarked on VGGNet-16 and ResNet-50.

As shown in Table 2.4, even though Envision is fabricated in 28 nm technology, its latency for VGGNet is high and its PUF is low. This means that its computational dataflow in different convolutional layer cannot efficiently map the computations onto PEs. Similarly, Eyeriss suffers from a high computational latency of 4 seconds to execute VGGNet due to processing batches of images. Although image batching is suitable for training neural networks on CPUs or GPUs, it is impractical for real-time applications. The PUF of Eyeriss is also low at 26% for VGGNet.

Table 2.4 Implementation results for low-energy designs.

	Eyeriss [63]	Envision [66]	FID [69]	ZASCAD [71]	
Technology (nm)	65	28	65	65	
On-chip SRAM (KB)	181.5	86	86	36.9	
Core Area (mm2)	12.25	1.87	3.5	6	
Gate Count (NAND2)	1852 k	1950 k	1117 k	1036 k	
Frequency (MHz)	200	200	200	200	
Batch size	3	N/A	1	1	
Bit precision (bits)	16	1-16	16	16	
# PEs	168	256-1024	192	192	
CNN model	VGG-16	VGG-16	VGG-16	VGG-16	ResNet-50
PUF% (filter size)	26% (3×3)	32% (3×3)	89 % (3×3)	94% (3×3)	88% (total)
Latency (ms)	4309.5	598.8	453.3	421.8	103.6
Performance (Gops)	21.4	51.3	67.7	72.5	74.5
Power (mW)	236	26	260	301	248
Efficiency (Gops/W)	90.7	1973	260.4	240.9	300.4
# DRAM access/batch (MB)	321.1	N/A	331.7	375.5	154.6

The proposed architecture in FID leads to a PUF of 89% which in turn results in lower latency and better performance efficiency with smaller area compared to Eyeriss. Despite these advantages, FID only supports 3×3 filters and suffers from weight passing issues in which some clock cycles are wasted to correctly order the weights in FID weight generator unit when computing new output rows. ZASCAD supports more filter sizes and can perform the convolution computations in ResNet-50. While its PUFs for convolutional layers with 3×3 filters are similar to FID, they drop significantly in the 1×1 convolutional layers [70]. Thus, the total PUF of ZASCAD drops to 88 % compared to FID. FID and ZASCAD do not support any method to handle the transfer of the generated results from the on-chip storages to off-chip DRAM after completion of each computation cycle. Therefore, convolution computations in those designs must be interrupted while freeing up the SRAMs resulting in performance inefficiency. In the present work, we will address the major issues of these implementations including low PUF, high number of DRAM accesses, poor performance and large silicon area.

2.5 Conclusion

In this chapter, we discussed the CNN as the dominant approach in computer vision for image recognition tasks. The CNN structure includes a succession of layers that automatically extract

input image features and determine which class the image belongs to. Usually, CNNs consist of convolutional, max-pooling, RELU, fully connected and classifier layers. Next, benchmarks for measuring classification accuracy were presented. Popular small-scale datasets include MNIST, SVHN and CIFAR-10, while ImageNet is a large-scale dataset. Subsequently, state-of-the-art CNNs were introduced. Among them, ResNet shows promising classification accuracy and it is a popular model for image recognition. Then, we introduced the energy consumption of different operations in CNN, by focusing on off-chip DRAM accesses as the main source of energy consumption. Next, several model compression approaches including low-bit quantization, using compact models and pruning weights were introduced to address energy-related issues. Finally, we reviewed recent works on low-energy implementations.

The upcoming chapters will present the process of designing CNN accelerators step by step. First, we propose an efficient computation unit with a proposed serial accumulation dataflow to perform 3×3 convolution. Since on-chip memory makes a significant contribution to power and area, we optimize the size and configuration to reduce these metrics as well as design cost. Next, we provide reconfigurability to the design to support 1×1, 7×7 and other convolutions to support convolutional layers of ResNet, which is one of the most complex CNN models to date. After completing the architectural design, we propose a new pruning method that fits with the dataflows in the architecture and reduces both the number of external memory accesses and the classification latency.

CHAPTER 3 CONVOLUTION UNIT ARCHITECTURE DESIGN WITH SERIAL ACCUMULATION DATAFLOW

In this chapter, we propose a new architecture for accelerating the computation of convolutional layers. A major advantage of this architecture is its capability to efficiently utilize all the computational units in all clock cycles. The proposed architecture includes several parallel units, each comprised of cascaded MAC operators whose outputs are accumulated in a serial fashion. In addition, it provides a mechanism to save clock cycles when reaching the column borders of inputs by avoiding unnecessary fetches of padded zeros. We evaluate the architecture using the VGGNet-16 convolutional layers as a widely used benchmark model. Portions of the material in this chapter were presented at NEWCAS 2020 and are part of the conference proceedings [74].

3.1 Overview of CNN accelerator hardware design

3.1.1 Architecture design challenges

When computing deep CNN inference, direct mapping of all computations on hardware resources of a single chip is not feasible due to the massive complexity of the associated computations. Existing designs commonly use a processing engine to perform a portion of computations at a time [68]. The engine is reutilized in a serial fashion to complete the entire computations.

Figure 3.1 shows the required computations in one convolutional layer using an example partitioning solution. In this example, each output channel is divided into P sub-out-fmaps. Each sub-out-fmap is the result of convolving one filter with the associated portion of the input features, called sub-in-fmaps.

The large number of weights, input pixels and intermediate data cannot fit within internal memories and thus, utilizing an external DRAM is inevitable. As discussed in section 2.2, among all the required operations in hardware acceleration, the DRAM access is the most expensive one in terms of energy consumption. Hence, it is proportionally beneficial to maximize data reuse inside the chip to avoid DRAM accesses.

A straightforward solution is to utilize larger on-chip SRAMs to reduce the costly off-chip DRAM accesses at the expense of additional silicon area. In resource-limited ASIC designs, however, an

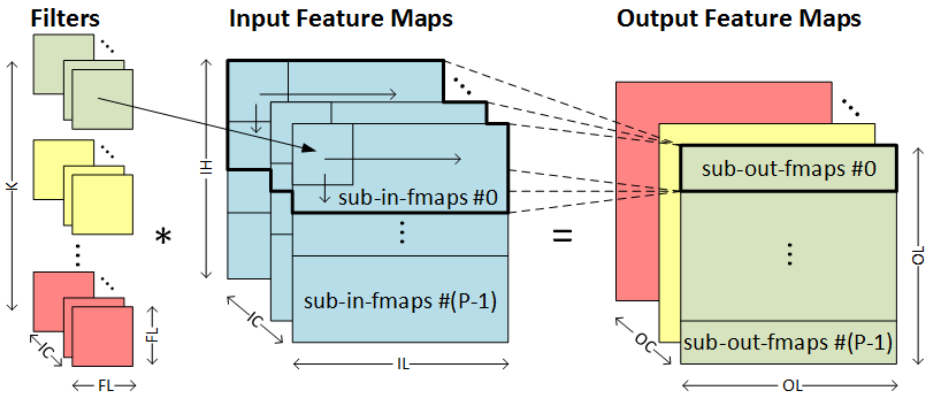


Figure 3.1 Partitioning convolution computations [76].

optimized dataflow is necessary to minimize data movements between off-chip and on-chip memories while respecting resource constraints. When increasing the number of MACs, another challenge is to effectively map the computations onto the available MAC resources. A significant issue in several existing designs is that they are not able to reach their advertised peak performance with a fixed architecture when executing modern CNNs [70].

3.1.2 General functionality of CNN accelerators

A typical acceleration system consists of an inference engine, an off-chip DRAM, and a host processor [68]. When the execution reaches convolution operations, the processor uses the inference engine to speed up computations. Due to the large data requirement in deep CNNs, it is typical to use an off-chip DRAM to store the filter weights, input features and generated output features. The convolution process is started by fetching filter weights and input features from DRAM to the convolution inference engine. Inside the engine, there are several parallel units, also called Convolution Units (CU). Each unit consists of some PEs, each one equipped with a MAC operator, to perform dot product computations. In each clock cycle, partial results are generated by MACs and are stored in on-chip SRAMs. These partial results are accumulated and stored in the same memory locations in the next clock cycles. After completing the computations, the obtained output features are transferred from the on-chip SRAM to the off-chip DRAM. Due to massive data requirements and the limitations of SRAM size, the convolution engine cannot typically generate the entire output features of a convolutional layer in a single round of operation. In other

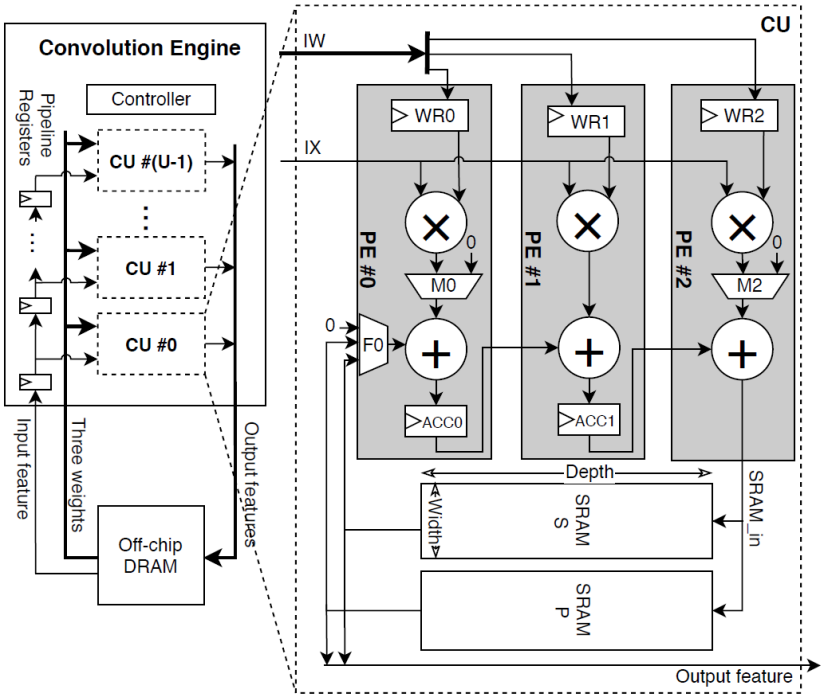


Figure 3.2 A convolution engine with the proposed CU architecture (© 2020 IEEE from [74]).

words, the engine operates on a part of the input data to generate a part of the output features in each iteration. The engine works sequentially to generate the entire output features.

Properly partitioning of the filter and input feature matrices enables assignment of independent computations to parallel units, which consequently enhances the performance of the engine. Many existing architectures exploit the inherent data level parallelism in the convolutional layers by assigning the weights from each filter (output channel) to a distinct parallel unit [63], [64], [69]. The input features, on the other hand, can be shared among the parallel units through a global buffer [64] or pipelining [69]. In this paper, we propose a new architecture that falls into this group of parallel architectures and utilizes pipelined buffers.

3.2 Proposed CNN accelerator

Figure 3.2 shows a convolution engine with the proposed CU architecture. The engine consists of U parallel units, and each CU includes N multiply-accumulators (MACs) and N registers to keep the weight values. Since the number of filters in VGGNet is divisible to 64 and 3×3 filters are used in its convolutional layers, U is set to 64 and N is set to 3.

In the convolution engine, CUs #0 to #63 each compute one output channel of the out-fmaps. Here, we explain the functionality of the first CU, i.e. CU #0, while the same process is performed in other parallel CUs which have the weights of other filters. The proposed architecture performs convolutions in a row-wise fashion. At the beginning of the convolution operation, the weights for the first filter row are fetched from DRAM and through the input IW, they are placed in registers WR0, WR1, and WR2. These weights are fixed in the registers, each of them provides an input of one multiplier. The other CU input is common to all three multipliers and receives the value of an input feature. The input feature is provided to each CU by pipeline registers through the input IX. In each clock cycle, a new input feature is fed to the pipeline registers while the previous input features advance one stage forward. When an input features arrives to the input IX of the CU, the dot product between filter weights and the input feature is performed by multipliers. The outputs of the multipliers are passed through multiplexers, added to the partial results from previous clock cycles and stored in accumulator registers, except for the last PE which writes the computed partial results directly into an SRAM. In other words, the PEs are connected together serially thorough accumulators and the content of accumulators are added from left to right until the result of three multiply and accumulations is stored in an SRAM by PE #2. The content of the SRAM can be fetched to PE #0 later for further processing.

As shown in Figure 3.2, the proposed architecture utilizes duplicated SRAM units, i.e., SRAM M and SRAM P, and a ping-pong mechanism to enable overlap of the convolution computations with data transfer from the on-chip SRAMs to the off-chip DRAM. In Figure 3.2, the multiplexers M0 and M2 facilitate the handling of zero padding in the borders. These multiplexers replace the output of the multipliers by zeros in the borders.

3.2.1 Row-wise 3×3 convolution in the proposed architecture

Figure 3.3 shows how to perform a row-wise convolution of a 3×3 filter with 6-row sub-in-fmaps, resulting in a 4-row sub-out-fmap. In step #0, the first filter row is convolved with the first four rows of the sub-in-fmap (both in the first channel), producing the corresponding partial results. Next, in step #1, the second filter row is convolved with the corresponding portion of the sub-in-fmap and the partial results are accumulated to the results of step #0. In step #2, the third row of the filter slides over the last four rows of sub-in-fmaps in the first input channel, and the partial

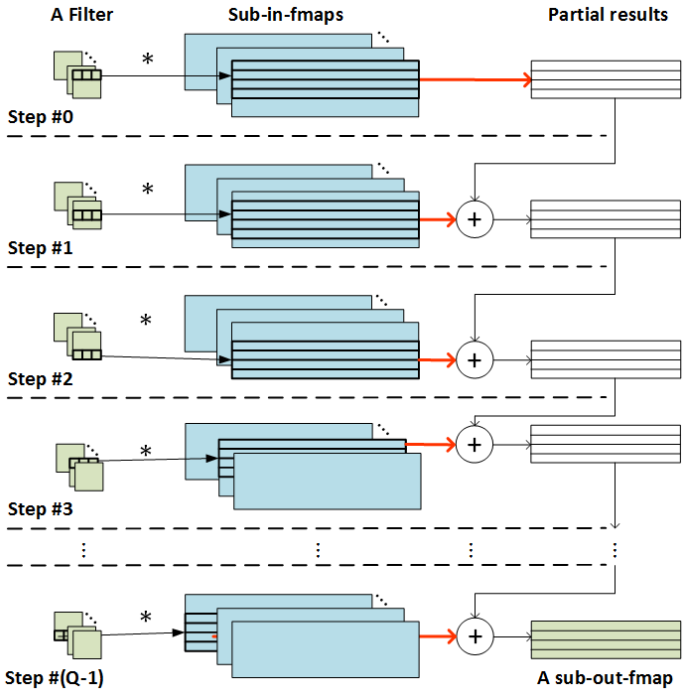


Figure 3.3 Row-wise convolution to generate a sub-out-fmap.

results are accumulated again. This process is repeated for the other input channels of the filter and the sub-in-fmaps.

During these steps, a row of a filter is convolved with a subset of input features. Therefore, to minimize DRAM accesses, three filter weights can be loaded once, reused for numerous times, then discarded. After $Q = (3 \times IC)$ steps consisting of many clock cycles, where IC denotes the number of input channels, a complete sub-out-fmap is computed and later transferred to the off-chip DRAM.

3.2.2 Description of CU functionality for 3×3 convolution using an example

This subsection describes the functionality of CU #0 using an example, where a matrix of in-fmaps of size 56×56×64, with zero padding of 1, is convolved with 64 filters of size 3×3×64 with stride of 1. The out-fmaps matrix has a size of 56×56×64. The computation of the output channel n is performed in CU #n. In this example, there are 56×56= 3136 output features per output channel. In the proposed architecture, each CU has a pair of 448 SRAM locations. Therefore, the out-fmap

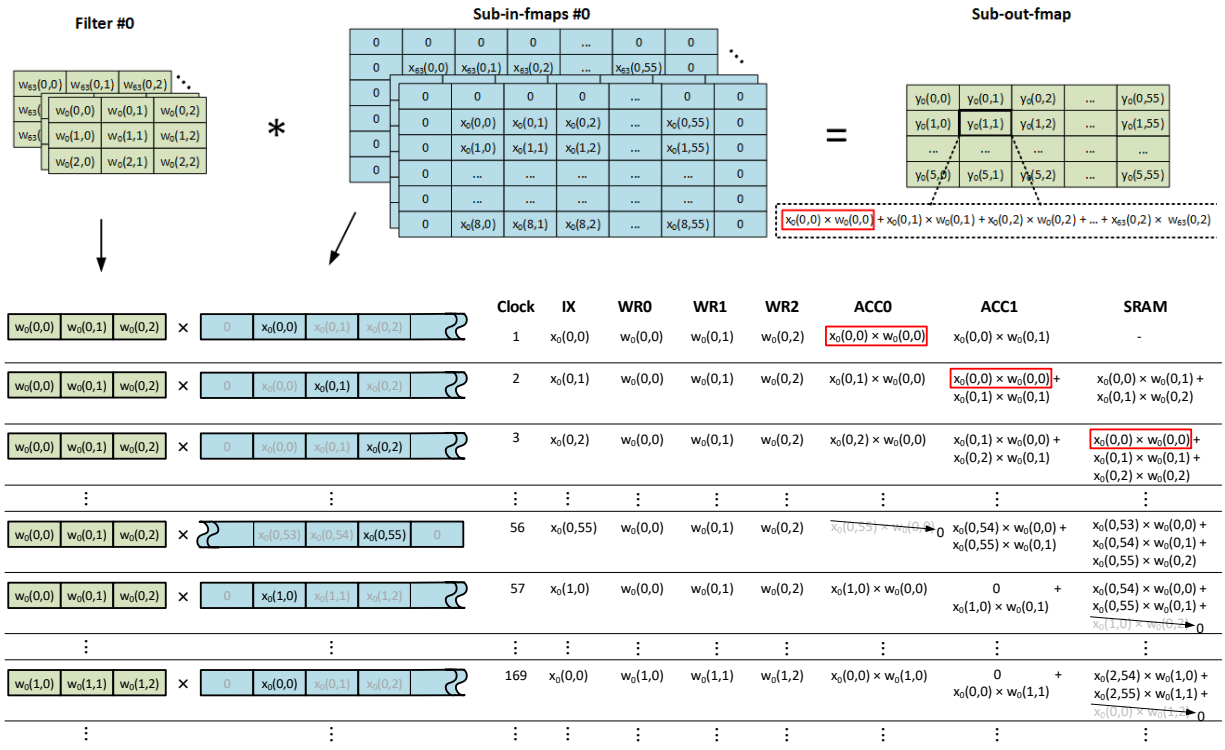


Figure 3.4 Proposed dataflow for 3×3 convolution to generate a sub-out-fmap (adapted from [76]).

is divided into 3136/448= 7 sub-out-fmaps of size 8×56. As shown in Figure 3.3, to produce 8 sub-out-fmaps, the sub-in-fmaps are divided into blocks of 10 rows. To summarize, CUs convolve sub-in-fmaps of size 10×56×64 with filters of size 3×3×64 to compute sub-out-fmaps of size 8×56. By repeating this process 7 times, the entire out-fmap of one output channel is computed by the corresponding CU.

Figure 3.4 shows the proposed dataflow to perform such a convolution in CU #0. Due to zero padding of 1, the convolution of the first filter row with the first row of the sub-in-fmaps is not required. In clock cycle #0, the registers WR0, WR1 and WR2 are loaded with weights of the first filter row. The following clock cycles are separated into three categories: routine computing cycles, in-fmap boundary handling, and row filter and input channel change.

Routine computing cycles: In clock cycle #1, the first input feature, $x_0(0,0)$, is fed to the input IX. This input feature is multiplied to the filter weights of the first row, $w_0(0,0)$, $w_0(0,1)$ and $w_0(0,2)$, and the results are stored in the accumulators ACC0 and ACC1. The third adder computes $x_0(0,0)$

× $w_0(0,2)$, but this value does not contribute in any required computation. Therefore, it is discarded and replaced by zero using the multiplexer MUX M2.

In clock cycle #2, the second input feature, $x_0(0,1)$, appears in IX, while the registers maintain their previous values. As a result, the third adder sums the contents of ACC1 (from the previous cycle) with the output of the multiplier, i.e., $x_0(0,1) \times w_0(0,2)$ and the result is stored in the corresponding SRAM address. Similarly, the contents of ACC0 (from the previous cycle) are added to the output of the second adder, containing $x_0(0,1) \times w_0(0,1)$, and the result is stored in ACC1. In this cycle, MUX M0 passes the value 0 and consequently, the result of the first multiplier, $x_0(0,1) \times w_0(0,0)$ is directly stored in ACC0.

As shown with a red box in Figure 3.4, the movement of $x_0(0,0) \times w_0(0,0)$ is completed in clock cycle #3 and the partial result of $x_0(0,0) \times w_0(0,0) + x_0(0,1) \times w_0(0,1) + x_0(0,2) \times w_0(0,2)$, is stored in SRAM. This is exactly the convolution of the first filter row with the first three elements of the sub-in-fmaps. This computation flow continues, as one input feature is fetched in each clock cycle, its element-wise multiplication with the weights of one filter row is performed, and the generated partial sums move across accumulator registers to finally be stored in the on-chip SRAM. As a result, all PEs actively contribute in computations in all clock cycles.

In-fmap boundary handling: In clock cycle #56, the input reaches the end of the first line, i.e., $x_0(0,55)$. Since there is no overlap between $x_0(0,55)$ and $w_0(0,0)$ their product does not need to be computed. Therefore, MUX M0 replaces this output with zero. The next input row starts with zero due to zero-padding. The produced zero in PE #0 plays the role of that zero-pad element of the next row in subsequent cycles. Without this mechanism, the architecture would have to fetch an additional zero as input, which would waste one clock cycle.

In clock cycle #57, $x_0(1,0)$ from the second row of the sub-in-fmaps, is fetched to start convolution with the first filter row. Since the product of $x_0(1,0)$ and $w_0(0,2)$ is not useful, MUX M2 substitutes the output of third multiplier with zero. This zero plays the role of the zero-pad element of the last row, which saves another clock cycle. This zero-pad substitution mechanism, applied in clock cycles #56 and #57, save two clock cycles each time the computation reaches the end of an input row.

Row filter and input channel change: In clock cycle #169, the convolution of the first filter row with the first portion of sub-in-fmaps has been completed. Thus, the weights of the second filter row are stored in WR0 to WR2 registers and its convolution with the corresponding portion is started in a process similar to the first filter row. Without any stall, the computations are continued for other filter rows and along the input channels.

In the end, the corresponding output features are obtained and stored in the SRAM. The full SRAM then starts to transfer its contents to the off-chip DRAM. In the proposed architecture, during this SRAM-DRAM data transfer, the second SRAM is utilized by PEs for the new round of computational cycles. The procedure of generating output features in the CU #0 is performed in other CUs, in parallel, for the other filters.

Due to limited SRAM size, the convolution engine cannot normally generate all the output features in a single round of operation. Therefore, several iterations must be performed serially to complete the computation of the entire output features. In each iteration, all the filter weights are re-fetched from the off-chip DRAM. Increasing SRAM size enables storage of more partial results, and consequently increases the number of generated output rows in each iteration. This reduces the total number of required iterations that results in fewer re-fetches of weights. Increasing size of SRAMs, however, increases the on-chip cost and power consumption. Considering the trade-off between the number of DRAM accesses and the area and power consumption, we selected SRAMs of size 448 words in the proposed architecture. The approach to select the SRAM size is discussed in more details in chapter 4.

3.2.3 Controller design in the proposed architecture

The controller is responsible to provide values for all the required control signals for different components in the proposed architecture, including the reset and enable signals of registers, multiplexer selectors, read/write enable signals and addresses for memories, etc. Since in the proposed architecture CUs are arranged in parallel, most of their control signals are common and they are given to CUs thorough pipeline.

In the proposed controller, the values of control signals are determined based on several conditions. These conditions are set according to the proposed dataflow and in most cases, they occur when

filter weights or input features reach their borders. Therefore, these conditions depend on data movements. In the controller, for each data movement a counter is defined. For instance, there are counters for filter rows, channels and filter numbers. In order to fetch a new filter row, for instance, the value of its corresponding counter will be changed and in this new condition, the controller changes the values of related signals. In addition, we define a main counter which is active during the entire computation cycles of a partition.

In the controller, the write and read operations are separated and implemented using state-machines. Read operations are referred to the required operations for fetching inputs from off-chip DRAM until feeding them to pipeline registers and loading them into CU registers, WR0, WR1, and WR2. The write operations, on the other hand, include the operations related to accumulate the partial results, writing them into on-chip SRAMs, and fetching them from SRAMs to PEs.

To implement the read operations in the controller, we employed three Verilog modules; the read_position, CU_registers_A, and CU_registers_B modules. In the read_position module, different conditions on counter values are checked and proper signals for input feature and filter counters are set. For instance, after reading all the filter weights in a channel, the filter channel counter is enabled to increment its value and to start reading data for the next filter channel. To load data in CU registers, in the CU_registers_A module, we defined an intermediate control signal whose values are determined based on conditions on main counter values. This intermediate signal is used in the CU_registers_B module to reset or enable CU registers signals. In other words, the CU_registers_B module includes case statements to determine if the WR0, WR1, WR2 registers have to be reset, to remain unchanged, or to be loaded with new values while selection among those cases is performed using an intermediate signal from the CU_registers_A module.

The write operations are implemented using two modules; write_position and mux_m_selection modules. In the write_position module, similarly to the discussion about read_position module, values of write control signals are changed based on conditions on counters values. In this module, values for control signals such as read/write enables of SRAMs, SRAMs read/write addresses, and selectors of the multiplexer F0 in Figure 3.2 are set. In addition, in this module an intermediate signal is defined which is used by the mux_m_selection module. This module includes case statements to change the selectors values of multiplexers M0 and M2.

In addition, the controller has two modules to generate DRAM fetching addresses for the input features and filter weights. These addresses are generated by the combination of counters.

To transfer the contents of on-chip SRAMs to the off-chip DRAM, the controller uses read enable signals to select SRAMs among different CUs. In the proposed architecture, when the read address is generated by the controller, a set of four SRAMs in adjacent CUs are selected to provide a 4×16 = 64-bit output data. Then, a counter is incremented and it activates read enable signals of the next set of four SRAMS. If PEs use more than one SRAM to store partial results, another counter is employed by the controller to select among SRAMs in a CU.

3.3 3×3 convolution architecture performance analysis

The computation time and the number of DRAM accesses for fetching input features and filter weights can be analyzed for the 3×3 convolution case. These deterministic metrics depend on the architecture and CNN characteristics such as number of CUs, number of filters, filter size, feature map size and number of input channels.

In the proposed architecture, in each clock cycle, an input feature is read from the off-chip DRAM and a partial result is generated. Thus, each step in Figure 3.3 requires OL^2/P clock cycles. Q steps are necessary to calculate one sub-out-fmap. This process is repeated for the P partitions to generate the entire out-fmap. No clock cycle is spent for zero pad rows, which saves $2Z \times OL$ clock cycles. Moreover, the proposed boundary mechanism introduces no timing overhead for the zero pad columns.

The proposed architecture uses U parallel CUs, each computing one output channel in parallel. Therefore, for K filters (i.e., K output channels), the computations are repeated $\left\lceil \frac{K}{U} \right\rceil$ times. Hence, the total number of clock cycles required to generate the entire out-fmaps is

$$\#Clock\ Cycles = (Q \times OL^2 - 2Z \times OL \times IC) \times \left\lceil \frac{K}{U} \right\rceil = (3 \times OL^2 - 2Z \times OL) \times IC \times \left\lceil \frac{K}{U} \right\rceil \quad (3.1)$$

The total number of DRAM accesses is composed of the accesses required to fetch the elements of in-fmaps and the filter weights and to store the out-fmaps.

$$\#DRAM\_Access = \#DRAM\_Access_{filter} + \#DRAM\_Access_{in\text{-}fmaps} + \#DRAM\_Access_{out\text{-}fmaps} \quad (3.2)$$

Since in each clock cycle one input feature is fetched from the off-chip DRAM, the total number of memory accesses for fetching input features, #DRAM_Access$_{\text{in-fmaps}}$, is the same as (3.1). The number of DRAM accesses required to store the results, #DRAM_Access$_{\text{out-fmaps}}$, is equal to the size of the out-fmaps, i.e., $OL^2 \times OC$. On the other hand, the architecture fetches three weights of a filter row in each cycle. This process is repeated for all Q steps required to compute one sub-out-fmap. Having U convolution units, the number of DRAM accesses for fetching weights of K filters is increased by $\left\lceil \frac{K}{U} \right\rceil$ times. This process is repeated for the other partitions P times. Hence, the total number of required DRAM access for fetching weights is

$$\#DRAM\_Access\_filter = 3 \times U \times Q \times \left\lceil \frac{K}{U} \right\rceil \times P \tag{3.3}$$

The number of operations in each convolutional layer is the total number of MAC operations required to compute out-fmaps excluding the operations associated to zero-pads. Thus, the total number of operations is

$$\#Operations = IC \times K \times (FL^2 \times OL^2 - 2Z(2 \times FL \times OL - 2Z)) \tag{3.4}$$

Replacing the number of clock cycles with (3.1) and the number of operations with (3.4), the PUF in (2.6) is obtained as $\frac{K}{(U+1)\times\lceil K/U \rceil}$. In the proposed design, the U is set to 64 and K is divisible to U. Thus, the PUF of the 3×3 convolution is 98.46%.

3.4 Design verification methodologies

In this work, we validate our implementations in Verilog HDL using collected data from pre-defined CNN models in MatConvnet [102] . MatConvNet is a deep learning library developed for MATLAB that provides required functions to run CNN models. By feeding an example input image to a target CNN model in MatConvNet, we can obtain input and output data as well as the filter weights for each convolutional layer. For a specific target convolutional layer, we store input features, filter weights and output features values in 16-bits hex numbers in three separate files. Next, in ModelSim, a testbench reads input features and filter weights from these files and feeds them to the RTL model of the proposed architecture. The outputs of the architecture are then compared with the expected values in the output files generated in MATLAB. This validation process verifies the functionality of the parallel units, pipeline registers, MAC operators, on-chip

SRAM contents, etc. In addition, it helps to validate the performance analysis equations described in section 3.3 for calculating the number of clock cycles and DRAM accesses for a convolutional layer. Our approach is consistent with similar validation strategies described by others [69][71].

After validating the proposed architecture, the RTL model was synthesized using Cadence Genus at 200 MHz frequency with 1 V supply voltage to obtain power and area consumption. In this phase, the timing report is also checked to prevent any timing violations. We also perform post synthesis simulations to ensure that all the modules are synthesized correctly.

To further process the design for fabrication, a necessary step would be to implement the proposed architecture on FPGA and to measure the number of clock cycles and DRAM accesses on real hardware. This would provide more precise measurements of these metrics since running the entire CNN model on software platforms such as ModelSim is time consuming and for large CNN models it is infeasible. Moreover, after drawing the design layout, post-layout simulations would be required to consider the effect of parasitic capacitances for proper power estimations. Fabrication of the proposed design in an ASIC is not an aim of this work.

3.5 Results and discussions

The proposed architecture was modeled in Verilog HDL and implemented in TSMC 65 nm LP CMOS technology at 200 MHz clock frequency using Cadence Genus. The word lengths of filter weights, input features and output features were set to 16 bits and the bit-width of SRAMs was set to 32 bits. Table 3.1 shows the implementation results for the proposed architecture and its comparison with the state-of-the-arts. We selected the designs that are benchmarked on VGGNet-16 since this model consists of 3×3 filters in all its convolutional layers.

As shown in Table 3.1, our architecture consumes 140 mW of power and 6.2 mm^2 of silicon area. In addition, it achieves a latency of 393 ms and requires 263.7 MB memory accesses to classify an image using VGGNet-16.

The FID architecture in [69] performs convolutions based on the computation patterns of fully connected (FC) layers. In that design, each MAC operator is considered as a neuron and its output is connected to an SRAM. The weights for the MAC operators are provided by a weight generator unit, which consists of five registers and two multiplexers. Using a separate SRAM unit to each

Table 3.1 Implementation results for low-energy designs benchmarked on VGGNet-16 (© 2020 IEEE from [74]).

	FID [69]	Envision [66]	Eyeriss [63]	Ours
Technology (nm)	65	28	65	65
On-chip SRAM (KB)	86	144	181.5	224
Frequency (MHz)	200	200	200	200
Bit precision (bits)	16	1..16	16	16
#PEs	192	256..1024	168	192
Core Area (mm2)	3.5	1.87	12.25	6.2
NAND2 Gate (K)	1117	1950	1852	937
Power (mW)	260	26	236	140
Latency (ms)	453.3	598.8	4309.5	393.0
Performance (Gops)	67.7	51.3	21.4	78.1
Efficiency (Gops/W)	260.4	1973	90.7	557.9
#DRAM access (MB)/batch	331.7	NA	321.1	263.7

MAC operator, however, introduces area and power consumption overhead compared to our design which uses unified SRAMs for all the PEs of each CU. In addition, FID was not actually implemented in an SRAM library from TSMC and the reported area and power consumption results were estimated. Furthermore, the paper does not describe any mechanism to allow computations to proceed during SRAM-DRAM data transfers. In the absence of such a mechanism, the convolution engine must be stalled for many clock cycles to first empty all parallel units SRAMs of their contents before starting new computations. Our proposed architecture outperforms FID by 13.3% in latency and by 24.1% in DRAM accesses.

Eyeriss [63] uses a NOC-based architecture utilizing a large global SRAM of size 108KB to buffer filter weights, input features and intermediate results. Eyeriss reduces the number of DRAM accesses by performing image classification on batches of three images instead of a single image. This results in a long computational latency of 4.3 seconds, which is prohibitive for many real-time applications. In addition, latency-sensitive applications normally require to process an image frame quickly and individually, before the next image frame arrives. Therefore, image batching is not a proper solution for a wide range of real-time applications. Compared to Eyeriss, our proposed method reduces the latency by 10.9× and the number of DRAM accesses per batch by 21%.

Envision [66] is a CNN accelerator implemented in the smaller 28 nm UTBB FD-SOI technology. From an architecture perspective, the MAC units can be dynamically configured to different word

lengths for different convolutional layers according to the accuracy requirements of each layer. Our proposed method outperforms Envision by offering 1.5× lower latency and 1.5× higher performance efficiency while utilizing 52% fewer NAND2 gate equivalents – a more equitable comparison than strict silicon area, given the different technologies. It is worth noting that Envision achieves its high on-chip energy efficiency thanks to using a more recent fabrication technology and a low-power DVFS circuit design technique. Although this technique could be employed in our implementation to further reduce power consumption, this was considered out of scope of this work. The number of DRAM accesses to perform image classification was not reported for Envision.

3.6 Conclusion

In this chapter, we proposed an energy efficient architecture to perform convolutions in deep CNNs. The proposed architecture includes several parallel units in which the processing elements are connected serially via accumulator registers. Therefore, the partial results are accumulated from the output of the first PE to the next PE until the last serial PE writes them to an SRAM. This dataflow maximally utilizes all the processing elements, which results in low-latency computations. In addition, properly selecting the SRAM size in the proposed architecture significantly decreases the number of DRAM accesses while introducing low hardware overheads. The performance of the proposed architecture on VGGNet-16 shows that it outperforms the existing implementations on both latency and the number of memory accesses.

CHAPTER 4 HETEROGENEOUS DISTRIBUTED SRAM CONFIGURATION FOR ENERGY-EFFICIENT DEEP CNN ACCELERATORS

The memory configuration of CNN accelerators highly impacts their area and energy efficiency and employing on-chip memories such as SRAMs is unavoidable. In this chapter, we review SRAM configurations in existing CNN accelerators and their effects on accelerator performance. Then, we analyze the impacts of SRAM sizing on the number of DRAM accesses. Next, we propose a new on-chip memory configuration which divides on-chip SRAMs into two groups with different assigned roles. Simulation results show that the proposed configuration can achieve higher energy efficiency while requiring less storage and less silicon area compared to the baseline model. Portions of the material in this chapter were presented at NEWCAS 2020 and are part of the conference proceedings [75].

4.1 SRAM configurations: existing methods and issues

During the past few years, CNN accelerators have employed different SRAM configurations in their architecture. For instance, Eyeriss [63], which has an NOC-like structure with an array of parallel PEs, utilizes a large global SRAM of size 108 KB to store and reuse the generated partial results. Utilizing a global buffer considerably reduces the number of DRAM accesses. However, it also degrades design flexibility for the computational flow. This limits the ability of the design to effectively benefit from the PEs in the computations. As a result, a high latency of 4.3 seconds was reported for performing the 3×3 convolutions of VGGNet-16.

In contrast to using a large global buffer, another approach is to split it over a number of smaller SRAMs distributed in parallel CUs [69]-[71]. For instance, Ardakani et al. proposed a system called FID where each PE communicates with its local SRAM directly [69]. This flexibility considerably improves the performance of the design when performing 3×3 convolutions. However, the described computational flow does not include any mechanism for transferring the previously computed output features from the parallel on-chip SRAMs to the off-chip low-bandwidth DRAM. The required number of clock cycles to free up all SRAM contents is given by

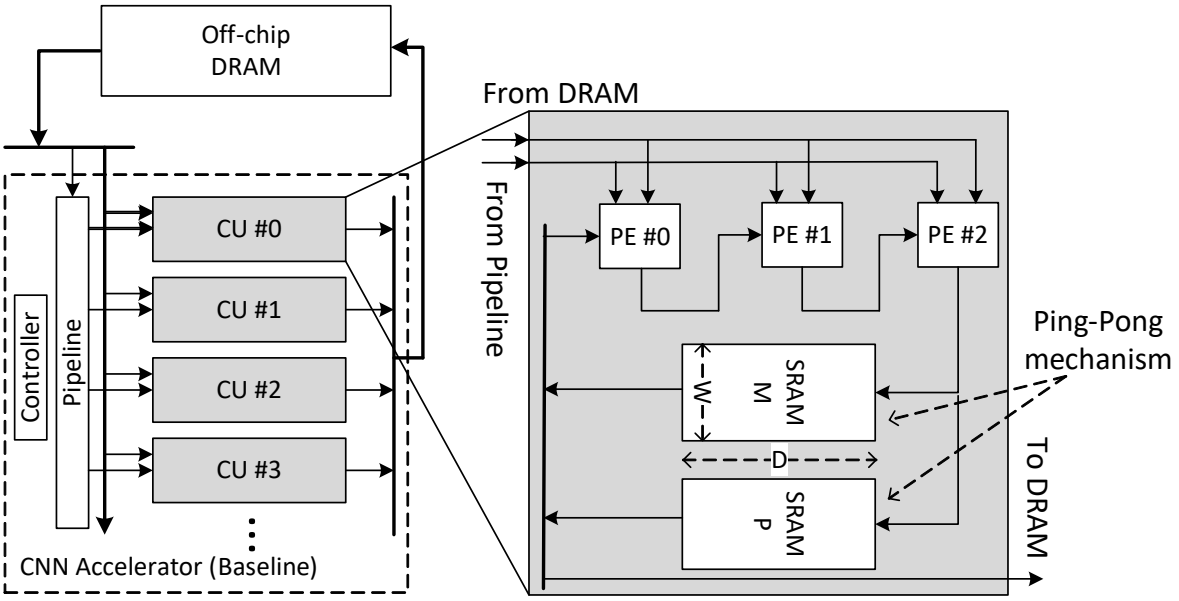


Figure 4.1 Baseline architecture with a ping–pong mechanism (© 2020 IEEE adapted from [75]).

$$\#\text{Clock Cycles} = \#\text{PE} \times D \times \frac{\text{bit\_width}_{\text{output\_feature}}}{\text{bit\_width}_{\text{DRAM\_data\_bus}}} \tag{4.1}$$

where #PE is the number of processing elements, D is the SRAM depth, $\text{bit\_width}_{\text{output\_feature}}$ is the required number of bits to represent the value of output features, and $\text{bit\_width}_{\text{DRAM\_data\_bus}}$ is the DRAM data bus width.

Since the DRAM data bus width is commonly limited to 64 bits in existing low-energy accelerators, increasing the number of PEs and the SRAM depth to improve throughput results in longer latency for SRAM-to-DRAM data transfers. Thus, without an efficient mechanism to handle these data transfers, the convolution engine will need to stall for several clock cycles to free up the SRAMs after computing each portion of the results. This can severely degrade the overall throughput. For instance, the MMIE design [70] suffers from a noticeable degradation in PUF when performing early convolutional layers of VGGNet-16, e.g., ~35% performance efficiency for the first layer. In other words, MMIE only achieve 35% of its peak performance for that layer since its PEs have to wait and cannot write into SRAMs when transferring SRAMs contents to the off-chip DRAM.

The proposed architecture in Chapter 3 uses distributed SRAMs in parallel units and adopts a ping-pong mechanism to address the crucial SRAM-DRAM data transfer issue. This architecture is used

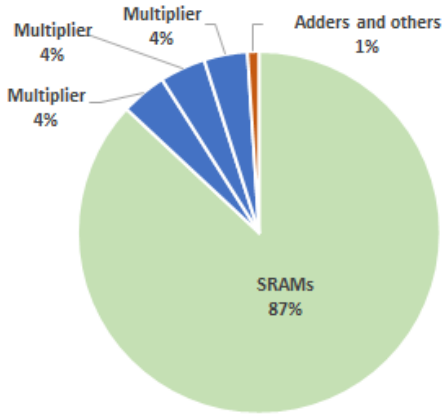


Figure 4.2 Portion of utilized silicon area for MACs and SRAMs inside a CU.

as the baseline architecture through this chapter. Figure 4.1 shows the baseline architecture with U = 64 parallel CUs. To perform 3×3 convolution inside a CU, three PEs are cascaded since there are three weights in a filter row. Figure 4.1 also shows the ping-pong-enabled configuration of SRAMs to overlap computations with data transfer to DRAM. In this configuration, another set of identical SRAMs allows the architecture to continue computations without interruption. Since the convolution engine can only generate a portion of the output features in each iteration, a CU initially stores the partial results into SRAM M. After completing the computations and filling this SRAM with output features, these results must be transferred to the off-chip DRAM. To avoid any interruption in the computation flow during this transfer, the CU uses SRAM P to store the new partial results. This swap of SRAM sets is performed repeatedly each time the CUs complete the computation of one set of output features.

4.2 Impacts of SRAM sizing on the number of DRAM accesses

A major trend in designing energy-efficient CNN accelerators is to reduce the number of DRAM accesses by increasing local data reuse and storing it in on-chip SRAMs [63]. As shown in Figure 4.1, SRAMs act as buffers to keep a portion of partial results close to the PEs during computations. Utilizing larger SRAMs is an obvious solution to reduce the number of DRAM accesses. However, a tradeoff must be made because SRAMs are area-intensive modules and their power and energy consumption are higher than that of other on-chip units [26].

Table 4.1 Number of computation partitions for convolutional layers of VGGNet-16.

Layer number	out-fmap size	SRAM Depth									
		112	224	448	672	896	1120	1344	1568	1792	2016
1	224×224	448	224	112	75	56	45	38	28	25	23
2	224×224	448	224	112	75	56	45	38	28	25	23
3	112×112	112	56	28	19	14	12	10	7	7	6
4	112×112	112	56	28	19	14	12	10	7	7	6
5	56×56	28	14	7	5	4	3	3	2	2	2
6	56×56	28	14	7	5	4	3	3	2	2	2
7	56×56	28	14	7	5	4	3	3	2	2	2
8	28×28	7	4	2	2	1	1	1	1	1	1
9	28×28	7	4	2	2	1	1	1	1	1	1
10	28×28	7	4	2	2	1	1	1	1	1	1
11	14×14	2	1	1	1	1	1	1	1	1	1
12	14×14	2	1	1	1	1	1	1	1	1	1
13	14×14	2	1	1	1	1	1	1	1	1	1

Figure 4.2 compares the proportion of silicon area of a CU utilized by the computational units and SRAMs in the baseline architecture. For this figure, the SRAM size is set to 448×32, i.e., 448 words (depth) of 32-bit width and 16-bit MAC units. Each CU has two SRAM units which occupy most of the area (87%) while MAC operators are much less bulky. This indicates the importance of proper sizing and configurations for SRAM units. Through this chapter, we assume a bit-width of 32 for each word in SRAMs and therefore, SRAM size changes by changing SRAM depth.

Since CNN models require massive number of computations and parameters, the convolution computations must be divided into smaller partitions. The accelerator, thus, serially performs the computations of each partition. The number of generated output features in each parallel unit is equal or less than the SRAM depth. The number of partitions in a convolution is

$$P = \frac{OL \times OL}{SRAM\_\text{Depth}} \tag{4.2}$$

Based on (4.2), the number of partitions is inversely proportional to the size of SRAMs. Utilizing larger SRAMs enables the accelerator to store more partial results and consequently generating more output features in each round of operations. In other words, the convolution computations are divided to fewer partitions with larger sizes.

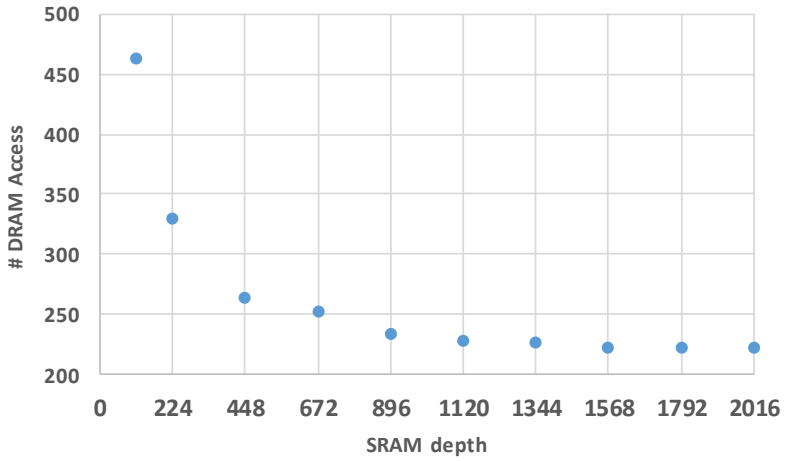


Figure 4.3 Number of DRAM accesses vs. SRAM depth.

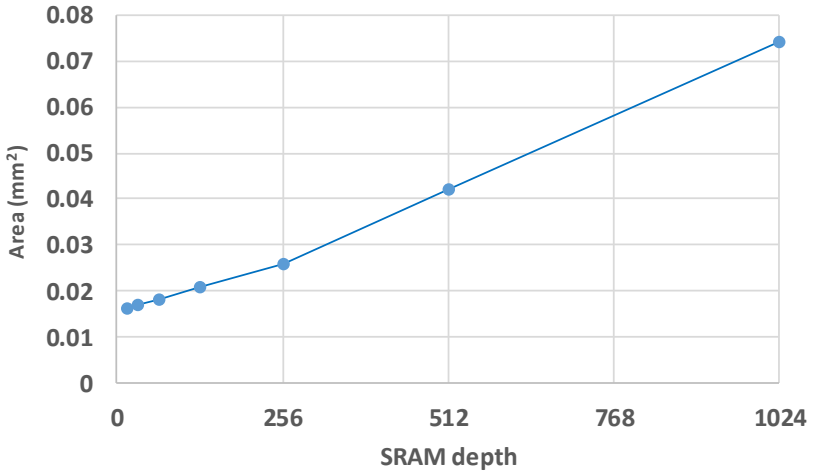


Figure 4.4 Silicon area for different depths of TSMC dual port SRAMs with 32-bit words.

Table 4.1 illustrates the number of partitions for convolutional layers of VGGNet-16, using various SRAM sizes when they are performed by the baseline architecture. Since the larger out-fmap size in VGGNet-16 is 224×224, most of SRAM depths in the Table II, are selected to be equal or greater than 224. This enables the accelerator to generate one output row or more in each output channel during each round of computation. As the results show, by increasing the size of SRAMs, fewer partitions are required. However, when the SRAM size is large enough, the number of partitions in most of the layers either remains unchanged or changes insignificantly.

When dividing convolution computations into partitions, the same filter weights must be re-fetched several times by the accelerator for different partitions. Therefore, dividing computations into fewer partitions, reduces the number of filter weight re-fetches and consequently decreases the number of DRAM accesses. The relation between the number of DRAM accesses and the SRAM size can be obtained by replacing (4.2) into (3.2). Therefore, the number of DRAM access is obtained as

$$\#\text{DRAM\_Access} = \left(9 \times U \times IC \times \left\lceil \frac{K}{U} \right\rceil \times \frac{OL^2}{\text{SRAM\_Depth}}\right) + \left((3 \times OL^2 - 2Z \times OL) \times IC \times \left\lceil \frac{K}{U} \right\rceil\right) + (OL^2 \times OC) \quad (4.3)$$

where the first term in the parenthesis is the number of DRAM access for fetching the filter weights. By increasing the SRAM depth, the first term become smaller and the two other terms will have higher impact on the total number of DRAM accesses.

Figure 4.3 shows the number of DRAM access vs. the SRAM depth. As the results show, increasing the SRAM depth leads to reducing the number of DRAM accesses. This reduction is noticeable when the SRAM depth is at the size of output row or more. However, as per the discussion on partitioning, the impact on DRAM access is not significant for larger SRAM sizes.

Figure 4.4 shows the area of several 65 nm TSMC dual port SRAMs for different SRAM depths with 32-bit word length. As expected, larger SRAMs occupy more silicon area. Considering the tradeoff between the number of DRAM accesses (Figure 4.3) and the silicon area (Figure 4.4), we selected an SRAM size of 448×32 for the baseline model.

4.3 Proposed CNN accelerator memory configuration

Figure 4.5 shows the power and area for several configurations of dual port SRAMs in 65 nm TSMC technology. It can be observed that a single SRAM unit with 896 words of 32 bits requires 1.14× less silicon area and consumes 1.77× less power than two SRAMs with 448 words of the same width. In other words, while splitting SRAMs into smaller units improves the efficiency of the computational dataflow by increasing the flexibility of data movement between the PEs and the SRAMs, it introduces significant area and power overhead. Therefore, the energy and area efficiency can be further enhanced by merging small SRAMs into larger ones while preserving the flexibility of the dataflow. In addition, as Figure 4.5 illustrates, reducing the bit-width of SRAMs decreases their power and area proportionally. From the configurations presented in Figure 4.5, the

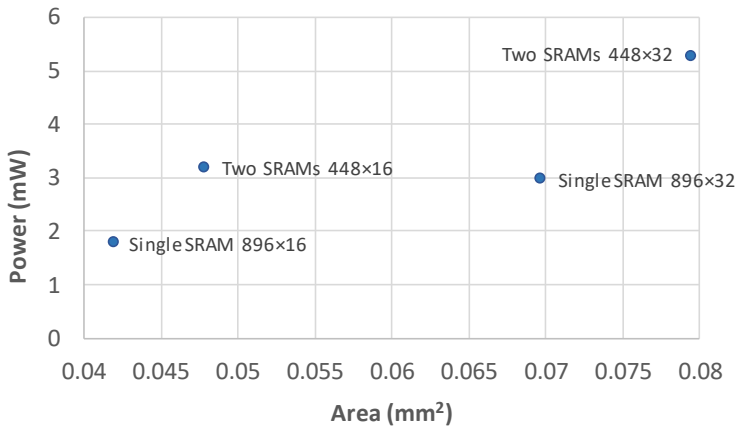


Figure 4.5 Power vs. area consumption for different configurations of SRAMs (© 2020 IEEE from [75]).

best result is achieved by 896×16 SRAM which requires 1.92× less area and consumes 2.95× less power compared to two 448×32 SRAMs.

Thus, we propose a heterogeneous configuration of distributed SRAMs in CUs. Figure 4.6 shows the proposed SRAM configuration for CNN accelerators. In this configuration, one set of SRAMs is still assigned to the CUs to store their partial results. The paired SRAMs in two adjacent CUs, however, are replaced with shared SRAMs which are 2× deeper but narrower, i.e., Z < W, compared to each of removed small SRAMs. The shared SRAMs store the resulting output features before their transfer to the off-chip DRAM. They do not store intermediate partial results.

When performing convolutions, the CUs use their private SRAMs M to store partial results. However, when the computation of the output features is completed, the final results are stored in the larger shared SRAMs, P. Since two different PEs from two neighbor CUs need to write into the shared SRAMs, both of their ports are set to the write mode. After storing all the generated features in the shared SRAMs, the CUs again use the small local SRAMs M for the new partial results for the next operating iteration. During that time, the SRAMs P transfer their contents to the off-chip DRAM. For this purpose, the two ports of the shared SRAMs are set to read mode. Having narrower and unified SRAMs rather than wider and divided ones, required in ordinary ping-pong scheme, reduces the area and power overheads and results in higher energy efficiency.

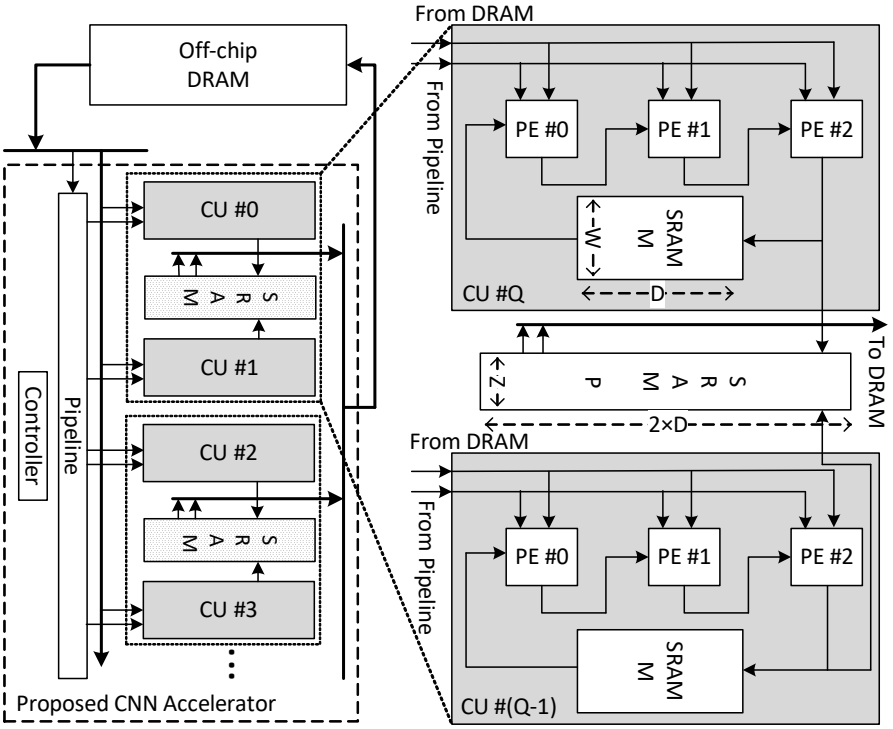


Figure 4.6 Proposed heterogeneous SRAM configuration (© 2020 IEEE from [75]).

In the proposed memory configuration, sharing SRAMs among neighbor CUs is feasible since standard TSMC SRAMs have dual ports. Each port of the shared memory is controlled by a different CU. The fact that CNNs are stacks of convolutional layers where the output of one layer is the input of the next layer makes it possible to narrow down the shared SRAMs width. Performing convolution within a layer requires large accumulators to preserve the partial computation precision (e.g., 32 bits). However, the final required precision for the values of the outputs (and inputs) is much lower (e.g., 16 bits or less).

4.4 Results and discussions

The proposed memory configuration was modeled in Verilog HDL and implemented in TSMC 65 nm LP CMOS technology using Cadence Genus for a 200 MHz clock frequency. The SRAM units were generated using TSMC 65 nm low-power dual-port SRAM Compiler, and power and area consumptions were obtained from synthesis results. The proposed memory configuration was applied to accelerate the inference of VGGNet as a widely used benchmark model. The evaluated

Table 4.2 Implementation results (© 2020 IEEE adapted from [75]).

	Eyeriss [63]	Eyeriss v2 [64]	Baseline Architecture	Proposed Architecture
CNN model	VGGNet	MobileNet	VGGNet	VGGNet
# Conv. Layers	13	27	13	13
CNN Error (Top-1)	24.4	29.4	24.4	24.4
On-chip SRAM (KB)	181.5	246	224	168
Core Area (mm2)	12.25	-	6.2	4.9
Gate Count (NAND2)	1852 K	2695 K	937 K	742.42 K
Frequency (MHz)	200	200	200	200
Bit precision (bits)	16	8	16	16
#PEs	168	384	192	192
Latency (ms)	4309.5	0.7	393.0	393.0
Performance (Gops)	21.4	-	78.1	78.1
Power (mW)	236	-	140	115
Efficiency (Gops/W)	90.7	193.7	557.9	679.1

design consists of 64 parallel CUs. Inside the CUs, the size of the SRAMs M was set to 448×32. Since these SRAMs are accumulators in the computation process, their bit-widths are set to 32, which is the same as the numerical precision of the CNN model in the training platform. Shared memories, on the other hand, only store the output features which requires fewer bits. Therefore, the sizes of the SRAMs P was set to 896×16.

Table 4.2 illustrates the implementation results for the proposed architecture and compares it with selected state-of-the-art solutions. The proposed method achieves an energy efficiency of 679.1 GOPs/W while occupying 4.9 mm^2.

When comparing with Eyeriss [63] on the VGGNet benchmark, our proposed architecture achieves 7.5× higher energy efficiency compared to Eyeriss while occupying 60% less area. Eyeriss v2 [64] is a modified version of Eyeriss with improved flexibility for handling variations in CNNs. In Eyeriss v2, the single global buffer is split into several smaller buffers grouped into 16 clusters. Inside each buffer cluster, four SRAM banks are assigned to store the partial sums (psums). The width of the psum buffers is 20 bits. When the accumulation is completed, the generated output features are converted to 8-bit numbers and transferred to the external DRAM. Compared to Eyeriss v2, our proposed design requires 3.6× smaller area, in terms of gate counts, and its energy efficiency is 3.5× greater.

Tu et al. [68] introduced a customized architecture to accelerate AlexNet inference. That work employed duplicated SRAM units and ping-pong mechanism to handle SRAM-DRAM data transfer. The large SRAM utilization in that design (280 KB overall) results in a large area footprint consuming 16 mm^2 in 65 nm technology.

We also made a direct comparison of the proposed architecture including the proposed memory configuration (Figure 4.6) with the baseline architecture that uses the ordinary ping-pong mechanism to handle SRAM-DRAM data communication (Figure 4.1). The proposed memory configuration in this chapter, therefore, is built on the baseline architecture except for the configuration of SRAMs. Using shared memories with narrower bit-widths compared to the wide SRAMs of the ping-pong approach results in 18% improvement in energy efficiency and 21% reduction in the silicon area.

4.5 Conclusion

In this chapter, we proposed a new memory configuration for CNN accelerators to facilitate the data transfer from internal SRAMs to the external DRAM. Compared to the well-known ping-pong mechanism, which uses identical SRAMs to overlap data communication with convolution computation, the proposed method utilizes SRAMs with different sizes and assigns different roles to them. While shallow SRAMs with large word length are used by processing elements locally, deeper but narrower SRAMs are shared among adjacent parallel units. The shared SRAMs store the final results at the end of each computation iteration and then transfer them to the DRAM. Compared to the architecture with a ping-pong mechanism, the proposed configuration achieves 18% higher energy efficiency while requiring less storage and less silicon area

CHAPTER 5 CARLA: A CONVOLUTION ACCELERATOR WITH A RECONFIGURABLE AND LOW-ENERGY ARCHITECTURE

Many recent low-energy accelerators have targeted the VGGNet-16 model, which has a regular structure using only 3×3 filters in all convolutional layers [63], [66], [69]. However, the current default choice for image recognition benchmarks is ResNet-50 which is a much deeper CNN model with 49 convolutional layers compared to 16 layers in VGGNet-16. The ResNet-50 model has very diverse specifications in the input data size (from 224×224 to 7×7), filter size (7×7, 3×3, and 1×1), number of filters (from 64 to 2048), number of input channels (64 to 2048) and filter strides (1 and 2) [9]. This requires an accelerator with a high-level of reconfigurability to support the large number of computation configurations across the convolutional layers.

This chapter describes CARLA, a convolutional accelerator with a reconfigurable and low-energy architecture, which supports efficient computation of various convolutional layer configurations. CARLA uses homogenous hardware units to perform computations in parallel. It handles the computational diversity of different layers using different operating modes with distinct computational dataflows. The chapter presents performance and computational cost results for CARLA when it is benchmarked on the convolutional layers of VGGNet-16 and Res-Net-50. Compared to the state-of-the-arts which fail to reach their claiming peak performance, CARLA achieve a near maximum PUF of 98% for majority of the convolutional layers and performs faster image recognition with fewer DRAM accesses.

5.1 The CARLA architecture

In a CNN inference accelerator, the computation of each layer starts with fetching the weights and input pixels from the external memory, e.g., DRAM. Then, convolutional operations are performed, and the generated output features are stored back in the DRAM, then fed to the next layer as inputs.

CARLA is composed of a set of U+1 cascaded convolution units (CUs) and a controller. The controller handles the data flow and assigns computations to the CUs. Each CU contains N

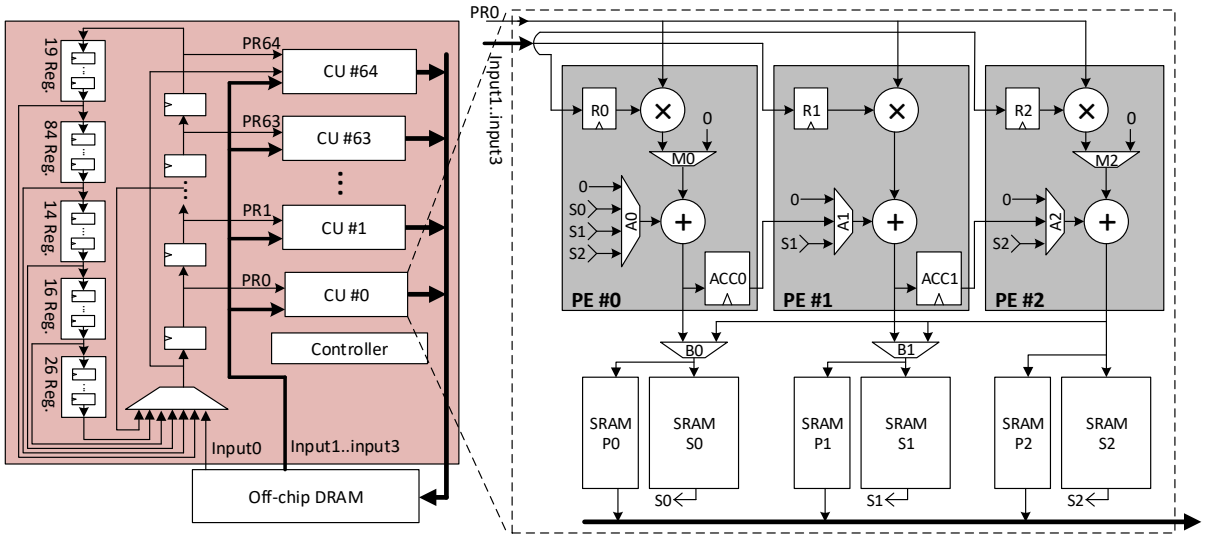


Figure 5.1 CARLA architecture [76].

processing elements (PEs) to perform convolution operations, except the last one, CU #*U*, which can contain more PEs.

Figure 5.1 shows the CARLA architecture configured for the ResNet models, for which *U* is set to 64 and *N* is set to 3. In this case, CU #64 contains four PEs and all other CUs contain three PEs. CARLA has four input buses, Input #0 to Input #3, coming from the external DRAM. Input #0 is buffered in a set of *U*+1 pipelined registers, each feeding one CU. Buses Input #1 to Input #3 are connected to the inputs of all the CUs. The four input buses may carry weights or input features, interchangeably.

Figure 5.2 and Figure 5.3 show the bus configuration of CARLA for 3×3 convolution and 1×1 convolution, respectively. For the sake of readability, in each CUs, only the contents of registers are shown. In the 3×3 convolution mode, the process starts by loading three filter weights from one filter row, $w_0^0(0,0)$, $w_0^0(0,1)$, $w_0^0(0,2,)$ and storing them in registers R0, R1 and R2 of the first CU, CU #0. In the same clock cycle, one input feature, $x_0(0,0)$, is loaded in the pipelined registers. For the following clock cycles, three new weights from a different filter and a new input feature are read from the off-chip DRAM. The new filter weights are placed in the registers of the next CU while the input feature is fed to pipeline registers. The input features are passed in the pipelined registers to be used by other CUs to compute their respective out-fmap channels. When registers

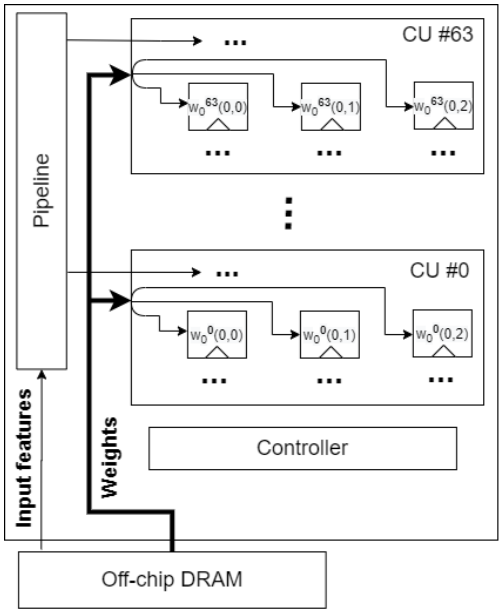


Figure 5.2 Configuration of buses in CARLA for 3×3 convolution.

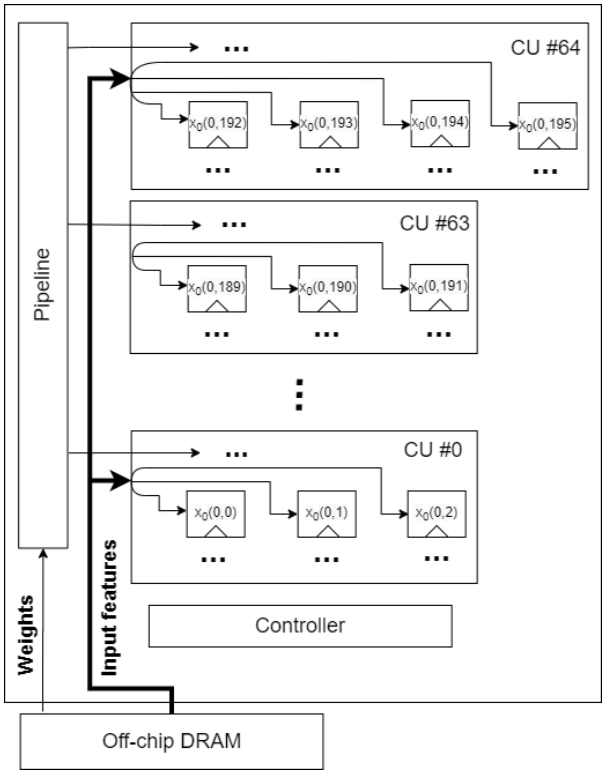


Figure 5.3 Configuration of buses in CARLA for 1×1 convolution.

R0, R1 and R2 in all the CUs are filled with weights from different filters, fetching new weights is stopped and CARLA only fetches input features. The weights are kept inside the CU and reused until computations have been performed for all the elements of the sub-in-fmap.

In the 1×1 convolution mode, the buses Input #1 to Input #3 are loaded with input features while filter weights are written on Input #0. The computation process starts by loading the first filter weight, $w_0^0(0,0)$, into the pipelined registers and loading the three first input features, $x_0(0,0)$, $x_0(0,1)$, x_0 (0,2,), into the registers R0, R1 and R2 of the first CU. In the following clock cycles, the three next input features are placed in CU registers while a new weight from a different filter is loaded into pipeline registers. In this mode, exceptionally for CU #64, all four input buses are assigned to load four input features into CU registers.

The pipelined input features are shared between parallel units [29], [34]. The proposed pipeline has several feedback paths that allow the existing data inside the pipe to return into the input pipeline.

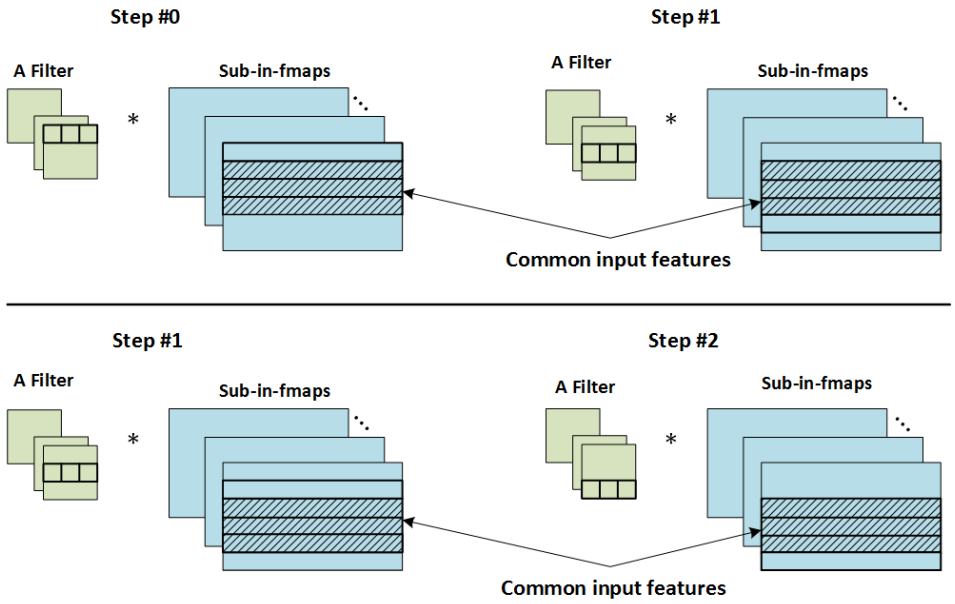


Figure 5.4 Common input features in different convolution steps of 3×3 convolutions.

For 3×3 convolutions, some input feature rows are used in different computation steps. Figure 5.4 shows the common input features in three steps of row-wise convolution which was previously described in Figure 3.3 of section 3.2.1. For instance, in step #0 and step #1 which are respectively the convolution of the first and the second filter row with the corresponding portion of input features, the input features in row#1, row#2 and row#3 of sub-in-fmaps are common. Similarly, the input features in row#2, row#3 and row#4 of sub-in-fmaps are common between convolution computations in step #1 and step #2. The feedback paths enable multiple reuse of these common data after the first fetch, instead of repeated re-fetching from the off-chip DRAM. This feedback mechanism is not used in 1×1 convolution.

The left side of Figure 5.1 illustrates the pipelining structure designed for CARLA. Pipelining input features is an approach to share input features among parallel units [29], [34]. The proposed pipelining has several feedback paths that allow the existing data inside the pipe to return back into the pipeline input. In 3×3 convolution, some of the input feature rows are used in different computation steps. Figure 5.4 shows the common input features in three steps of row-wise convolution which is previously described in Figure 3.3 of section 3.2.1. For instance, in step #0 and step #1 which are respectively the convolution of the first and the second filter row with the

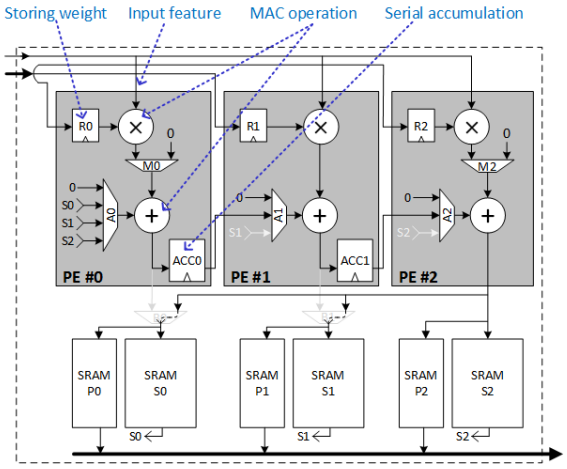


Figure 5.5 Configuration of a CU for 3×3 convolution [76].

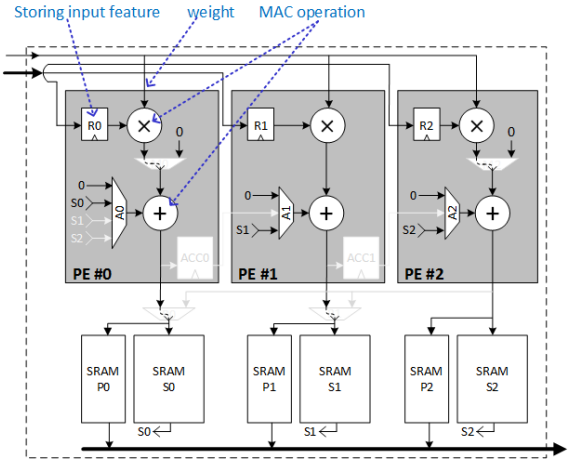


Figure 5.6 Configuration of a CU for 1×1 convolution [76].

corresponding portion of input features, the input features in row#1, row#2 and row#3 of sub-in-fmaps are common. Similarly, the input features in row#2, row#3 and row#4 of sub-in-fmaps are common between convolution computations in step #1 and step #2. The feedback paths enable multiple reuse of these common data after the first fetch, instead of repeated re-fetching from the off-chip DRAM. In 1×1 convolution, this feedback mechanism is not used.

Figure 5.1 also illustrates the internal structure of one CU when N=3. Inside a CU, each PE can perform one Multiply-and-Accumulate (MAC) operation in each clock cycle. Each PE contains one internal register (e.g. R0 in PE #0) that provides the first operand of the multiply operation. These registers can be loaded with values from the CU inputs. The second multiplier operand is shared among all PEs of a given CU. This operand is provided from the pipelined registers.

Figure 5.5 and Figure 5.6 show the configuration of CARLA for 3×3 and 1×1 convolutions, respectively. Unused resources for each mode are shown with greyed-out lines. We point out to different parts of PE #0 by arrows.

The PEs can be connected in series to compute the convolution with filter sizes greater than 1×1 (Figure 5.5), or they can process independent data for 1×1 filters (Figure 5.6). The MAC operation result is either stored in an accumulator (ACC0 and ACC1) when PEs are connected in series, or directly stored in SRAMs when the PEs process independent data.

Several multiplexers route the signals inside a PE and between the PEs of a CU based on the operating mode and the working state. The first and the last PE have special multiplexers, MUX M0 and MUX M2, to facilitate zero-pad handling at in-fmaps spatial borders. The three multiplexers MUX A0, MUX A1 and MUX A2 provide flexibility to handle various convolution configurations required in different modes.

As mentioned in Chapter 4, in the case of deep CNNs, on-chip SRAMs are generally not large enough to store entire out-fmaps. Thus, after computing a part of the out-fmaps, i.e., sub-out-fmaps, the architecture must free up the SRAMs by transferring the contents to the off-chip DRAM. To enable the architecture to operate uninterruptedly, we use a pair of SRAMs. While the PEs store partial results in one set of wider SRAMs After completing each computation cycle, the contents of the narrower SRAMs are gradually transferred to the off-chip DRAM.

In CARLA, while PE #0 and PE #1 only have access to one pair of SRAMs, PE #2 can access all SRAMs. As will be discussed in section 5.1.1, each PE writes its produced partial sum directly into its allocated SRAMs when computing 1×1 convolution. However, in 3×3 convolution, the partial sums of the three PEs are added together and the partial result is obtained in PE #2 and stored in the SRAMs. Therefore, PE #2 has a write access to all SRAM pairs to enable utilization of all SRAM resources in 3×3 convolution. The two multiplexers MUX B0 and MUX B1 handle the write access modes. The controller for CARLA was implemented following an approach similar to the descriptions in section 3.2.3. However, the state machines are more complex, and include more conditions for filter sizes such as 1×1 convolutions. In CARLA, the 3×3 convolution mode is realized based on the serial accumulation dataflow described in Chapter 3. The following subsections describe how CARLA realizes the 1×1, 7×7 and other convolutions in the ResNet models.

5.1.1 1×1 convolution mode

The proposed serial accumulation architecture in Chapter 3 could be used naively to perform 1×1 convolution. However, this would result in a considerable degradation of the PUF. Each input feature would be convolved with only one weight instead of three per row, and only one PE in each CU would be usable. The other two PEs per CU would be permanently inactive. To avoid this

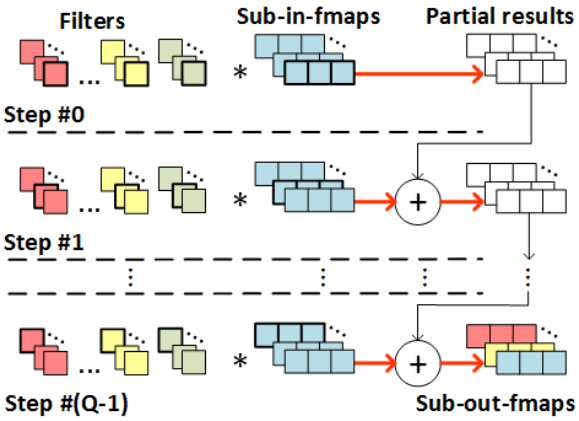


Figure 5.7 Computing sub-out-fmaps in 1×1 convolution.

waste of resources, CARLA exploits the parallelism available in the 1×1 convolution case using a distinct dataflow and hardware configuration.

In 1×1 convolution, the filter row size shrinks to 1. Thus, the computed partial sum in each PE does not have to be summed with the adjacent PEs. In other words, in 1×1 convolution, all PEs in CUs operate independently. On the other hand, in many convolutional layers, the number of features in each input channel is larger than the number of PEs. This provides the following possibility to parallelize the computations.

Each filter weight is convolved with all input features of the same channel in 1×1 convolution. A solution to improve PUF is to fill the PE registers with input features and slide the filter weights over them (which is a reverse from the 3×3 convolution mode). With this change, the PE registers R0, R1 and R2 can each store one input feature. The weights pass through the pipelined registers shown in Figure 5.1 and appear at the PRn input of each PEs. In every cycle, each PE participates in computations by multiplying the available weight at its input and its registered feature and accumulates the result.

5.1.2 General operation of 1×1 convolution

In contrast to the 3×3 convolution where each CU computes one sub-out-fmap, in 1×1 convolution, each CU computes a portion of sub-out-fmap in all output channels. For 1×1 convolutions, each sub-in-fmap is divided into small groups each containing N features. K different filters slide over

each group of input features and generate *K* output channels each containing *N* output features. For instance, for *K*= 64 filters and a group size of *N* = 3 features, 3 partial results are produced for 64 output channels. Figure 5.7 shows the case for group size *N*=3. All processing is performed for the same input channel for both the filters and the sub-in-fmaps.

In Step #0, channel #0 of all filters are convolved with the *N* features of the first input channel. In Step #1, the process is repeated for the second channel and the generated partial sums are accumulated. This continues by performing computations for one new channel in each step. After *Q=IC* steps, the convolution results of all channels have been performed and the corresponding portion of the sub-out-fmaps is obtained.

5.1.3 Description of CU functionality for 1×1 convolution using an example

In this section, we use an example from the ResNet models to describe CU functionality when computing 1×1 convolutions. In this example, the input feature map has size 56×56×256 with no zero padding. There are 64 filters of size 1×1×256, the filter stride is 1, and the output size is 56×56×64. The architecture has 65 CUs (*U*+1=65) that together provide 196 PEs, because the first 64 CUs contain three PEs while the last CU has four PEs. The out-fmaps are partitioned into 56×56/196=16 sub-out-fmaps, each containing 196 features. For 1×1 convolutions, the number of features in sub-in-fmaps is the same as sub-out-fmaps, i.e. 196, and they are placed in the registers inside the 65 CUs while the filter weights are fed through the pipeline.

Figure 5.8 shows the proposed dataflow to perform 1×1 convolution in CU #0. There are two types of working clock cycles in the architecture: routine computing cycles and exceptional cycles for the last input feature.

Routine computing cycles: In each routine computing cycle, three input features and one filter weight are read from the off-chip DRAM. The input features are loaded into three registers of a CU and the weight is given to the pipeline input. In clock cycle #1 in CU #0, the input PR0 is fed by the first weight of the first filter $w_0^0(0,0)$. This weight is multiplied by three input features, $x_0(0,0)$ and $x_0(0,1)$ and $x_0(0,2)$, which have been already loaded to the registers. The results are stored in the SRAM-S0, SRAM-S1 and SRAM-S2. In clock cycle #2, the first weight of the second filter, $w_0^1(0,0)$, is given to PR0, while the first input features are still in the PE registers. The dot

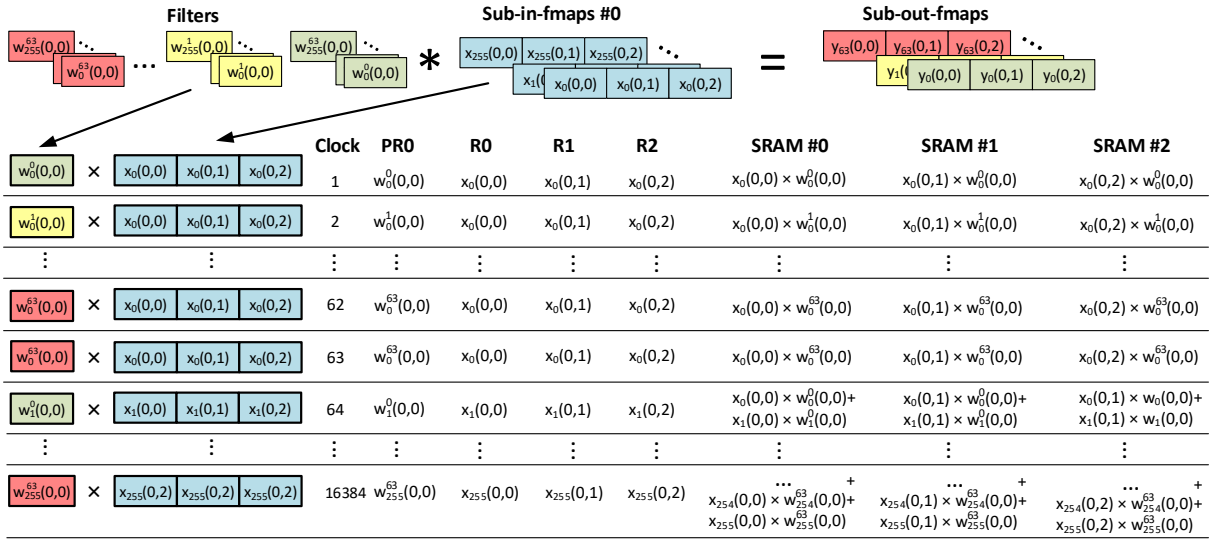


Figure 5.8 Proposed dataflow for 1×1 convolution [76].

product results are stored in different addresses in SRAMs. In the following clock cycles, the weights of the next filters enter the pipelined registers while new input features are stored in the PE registers of next CUs.

Exceptional cycles for the last input feature: The routine computing cycles continue until loading the input features for the last CU, CU #64, which exceptionally contains four PEs. To fill the registers of all these four PEs in a single clock cycle, the architecture uses all four DRAM read buses to fetch four input features rather than three. Consequently, since no DRAM read bus is left to fetch a weight, the pipeline must be stalled in this cycle.

After this clock cycle, the dataflow returns to its normal operation cycle by fetching three features and one weight from the next input channel. The pipeline must be stalled each time the input feature fetch reaches the last CU. After $Q = IC$ steps, each one taking many clock cycles, a portion of sub-out-fmaps, including three output features in each channel, is generated and transferred to the off-chip DRAM.

5.1.4 1×1 convolution architecture performance analysis

In the proposed dataflow, $U+1$ clock cycles are necessary to perform the operations in step #0. Q steps are required to complete computation in CUs. Since there are P partitions, this process is

performed *P* times to generate all entire out-fmaps. For *K* filters, the computations are repeated $\left\lceil \frac{K}{U} \right\rceil$ times. Thus, the total number of clock cycles for 1×1 convolution is

$$\#Clock\ Cycles\ =\ (U+1) \times Q \times P \times \left\lceil \frac{K}{U} \right\rceil =\ (U+1) \times IC \times P \times \left\lceil \frac{K}{U} \right\rceil \tag{5.1}$$

One filter weight is fetched from DRAM on every clock cycle, excluding the exceptional cycles. Thus, the total number of memory accesses for fetching filter weights is

$$\#\ DRAM\_Access_{filter} =\ U \times Q \times P \times \left\lceil \frac{K}{U} \right\rceil \tag{5.2}$$

The architecture must also fetch $\frac{OL^2}{P}$ features of a sub-in-fmaps and store them in CU registers. As this process is repeated for all *P* sub-in-fmaps, the total number of DRAM accesses for fetching input features is

$$\#\ DRAM\_Access_{in-fmaps} =\ OL^2 \times IC \times \left\lceil \frac{K}{U} \right\rceil \tag{5.3}$$

In CARLA, the pipeline must be stalled in exceptional cycles to fetch the last input features. In the example design, one stall cycle occurs in each 65 clock cycles. The PUF in this mode is obtained as $\frac{U}{U+1}$. For *U*= 64, the PUF is 98.46%.

5.1.5 1×1 convolution mode for very small size in-fmaps

The proposed dataflow in section 5.1.3 can perform 1×1 convolutions efficiently and achieves a high PUF when the number of features in each in-fmap is close to or greater than the number of PEs. If the number of features in a channel is much lower than the total number of PEs, however, its effectiveness is degraded. For instance, in the last convolutional layers of the ResNet models, the in-fmaps size shrinks to 7×7 with only 49 features in each input channel. In this case, only 49 PEs out of 196 would be utilized, resulting in a maximum PUF of 25%.

To improve the PUF in this case, the CARLA architecture employs a distinct dataflow in which the weights reside inside CU registers and input features are fed into the pipeline. Using this approach, one limiting factor is that one filter does not contain enough weights to fill all the PE registers in a CU. To overcome this limitation, CARLA fills the remaining PEs with weights from other filters. This strategy is applicable due to the two following facts. (1) As in 1×1 convolution,

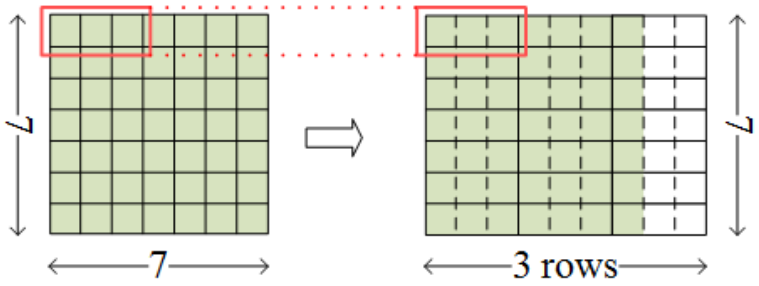


Figure 5.9 Splitting each plane of a 7×7 filter into 21 pieces [76]

PEs inside each CU operate independently, the registers in a CU can store weights from different filters. (2) In state-of-the art CNNs, the in-fmap size typically shrinks by advancing in layers, while the number of filters increases. In this case the total number of clock cycles is

$$\#Clock\ Cycles = U \times IC \times \left\lceil \frac{K}{3U} \right\rceil \tag{5.4}$$

In this dataflow, each filter weight is only fetched once. Therefore, the number of required memory accesses for fetching filter weights is

$$\#\text{DRAM\_Access\_filter} = K \times FL^2 \times IC \tag{5.5}$$

The input features, however, need to be fetched multiple times if the number of weights is more than the number of PEs. Thus, the number of DRAM accesses to fetch input feature is

$$\#\text{DRAM\_Access}_{in-fmaps} = IL^2 \times IC \times \left\lceil \frac{K}{3U} \right\rceil \tag{5.6}$$

5.1.6 7×7 convolution mode and others

The CARLA architecture achieves high PUFs of 98% for 3×3 and 1×1 convolutions, as the most widely-used convolutions in most modern CNNs. However, other convolution sizes are used in exceptional cases. For example, the first ResNet layer performs 7×7 convolutions. To support 7×7 and other larger convolutions, we divide the filter plane into smaller pieces and utilize a row-wise dataflow. As shown in Figure 5.9, the 7×7 filter is split into 21 pieces: 14 of them contain a row with three weights, and the remaining 7 consist of a row with one weight and two empty locations. The convolution of these small pieces can be handled by CARLA based on the row-wise

Table 5.1 Structure of ResNet-50 convolutional layers

Layers	Output size	1×1 filter	3×3 filter	7×7 filter
Conv1	112×112	-	-	1
3 × Conv2	56×56	3 × 2	3 × 1	-
4 × Conv3	28×28	4 × 2	4 × 1	-
6 × Conv4	14×14	6 × 2	6 × 1	-
3 × Conv5	7×7	3 × 2	3 × 1	-
Total	-	32	16	1

convolutions explained in section 3.2.1. To preserve computation flow homogeneity, all 21 pieces with one weight are computed using the 3×3 convolution mode.

5.2 Results and discussion

CARLA was evaluated with the ResNet-50 convolutional layers. It was modeled in Verilog HDL and was implemented in TSMC 65 nm LP CMOS technology at 200 MHz clock frequency with 1 V supply voltage using Cadence Genus. The word lengths of weights, in-fmaps and out-fmaps were set to 16 bits while the accumulators were 32 bits wide. The verification process followed the same process as described in section 3.4.

5.2.1 The structure of ResNet-50

Table 5.1 illustrates the structure of ResNet-50 as one of the widely used ResNet models for benchmarking. In this model, the convolutional layers are grouped into five categories, Conv1 to Conv5, based on the output layer size. All categories except Conv1, which only contains one 7×7 convolutional layer, are repeated several times. Thirty-two out of the 49 convolutional layers require 1×1 convolutions and 16 layers need 3×3 convolutions. In addition to the convolutional layers listed in Table 5.1, there are three other 1×1 convolutional layers in the projection shortcuts of ResNet-50 [9].

5.2.2 Performance and DRAM access results

Figure 5.10 to Figure 5.13 show the results of evaluating different efficiency metrics for CARLA, including PUF, computation time and DRAM memory accesses.

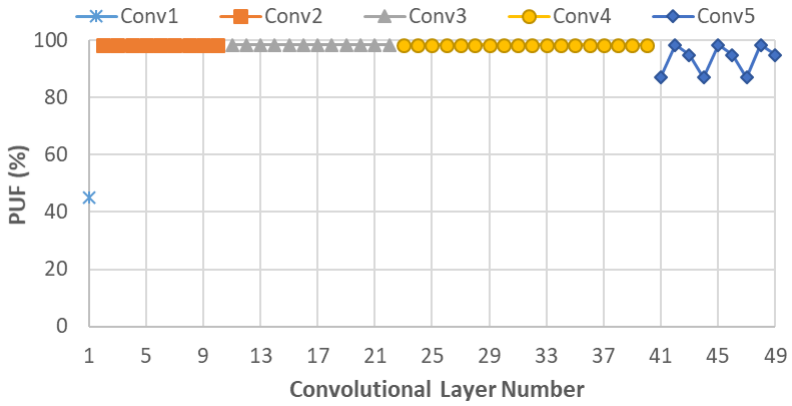


Figure 5.10 PUF for convolutional layers in ResNet-50.

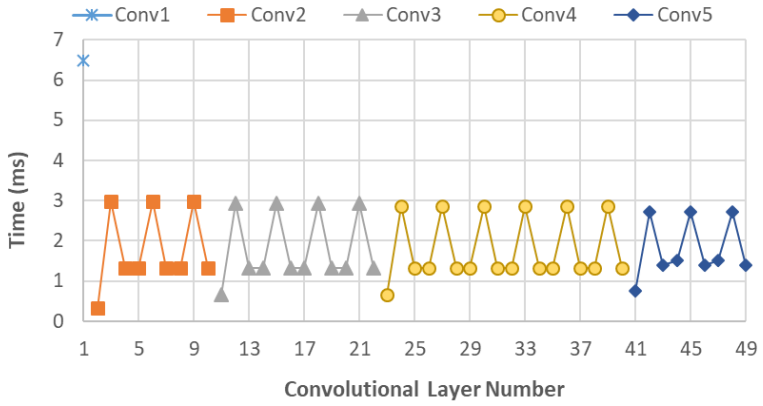


Figure 5.11 Computation time for convolutional layers in ResNet-50.

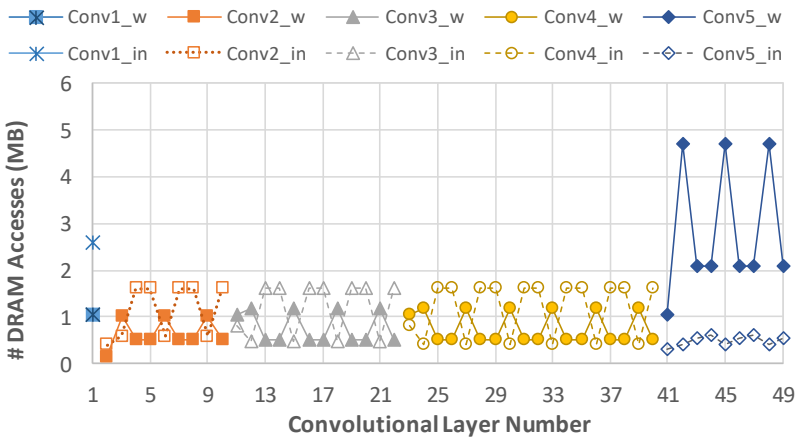


Figure 5.12 DRAM access numbers of in-fmaps and filter weights for ResNet-50.

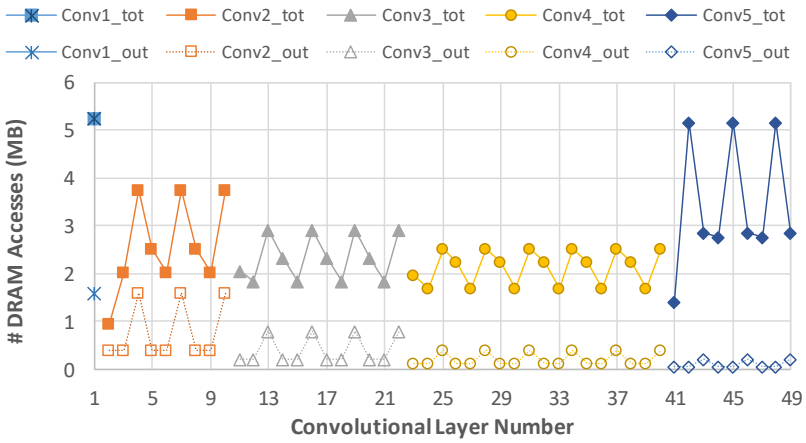


Figure 5.13 DRAM access numbers of out-fmaps and total for ResNet-50.

As shown in Figure 5.10, CARLA achieves a remarkable PUF of 98% for 1×1 and 3×3 convolutions when the output size is larger than or equal to 14×14, i.e., in the Conv2 to Conv4 layers of ResNet-50. In the Conv5 layer, in which the size of in-fmap is very small, CARLA utilizes the dataflow described in section 5.1.5 and reaches a remarkable PUFs of 87.1% and 94.5%. The high PUFs for 1×1 and 3×3 convolutions highlight the effectiveness of the proposed dataflows to map the required computations on the available resources in most convolutional layers. The PUF for Conv1 with one 7×7 convolution layer is only 45%, but since only the first layer of ResNet-50 uses this convolution size, this lower PUF does not have a significant impact on the overall efficiency of the architecture. On the other hand, computing the 7×7 convolution using the described method in section 5.1.6, rather than using an additional customized architecture, prevents considerable hardware overhead.

As shown in Figure 5.11, 3×3 convolutions require more computation time compared to 1×1 convolutions. The computation time for layers in Conv2 to Conv5 follows the same pattern. According to the ResNet-50 structure in [9], inside each group of layers, the number of filters is inversely proportional to the number of channels from one convolutional layer to the other. The reverse changes of these two factors compensate the impacts of each other on the computation time. On the other hand, when moving from one group to the next, the out-fmaps size decreases while the number of filters increases proportionally. Consequently, and as the results in Figure 5.11

illustrate, CARLA obtains similar computation times for the same filter size in various layers. The minimum points in Figure 5.11 represent the computation time in layers of 1×1 convolution and stride of 2. This stride has the role of down sampling in ResNet-50 to decrease the output size.

Figure 5.12 shows the required number of DRAM accesses for fetching the inputs and filter weights. In the convolutional layers of Conv5, the filter weights are fetched only once. Due to the very large number of filter weights, the number of DRAM accesses is higher in these layers compared to the others. For the Conv4 layers, where the output size is 14×14, the number of DRAM accesses is significantly reduced compared to Conv5 layers. Although the number of filters is decreased in Conv4, a reduction in the number of DRAM accesses is obtained by proper selection of SRAM sizes. In fact, there is a trade-off between the number of DRAM accesses and the hardware overhead when selecting SRAM size. Increasing the size of sub-in-fmap reduces the number of partitions, resulting in fewer re-fetches of the same weights. On the other hand, having larger partitions incurs larger SRAMs to store entire partial results, which introduces additional hardware overhead. To select a proper SRAM size, we considered the layers which are most frequently computed. Conv4 is the most often repeated group of layers in ResNet-50; it is computed 6 times (in ResNet-101 and 152, it is computed 23 times and 36 times, respectively). For this reason, we set the SRAM size to 224, which is divisible by all row sizes in ResNet, and greater than the number of generated partial results in the Conv4 layers, i.e. 14×14=196. This allows to maintain all partial results in on-chip memories. For other layers, due to the large size of the outputs, partitioning is unavoidable and the DRAM accesses for fetching filter weights follows the same pattern as for the Conv4 layers.

Figure 5.12 also shows the number of DRAM accesses for fetching input features (Conv1_in to Conv5_in). In layers with 3×3 convolutions, the number of DRAM accesses for input features is almost the same for all the groups. Again, this occurs since the product of the number of filters and input channel size is constant for all the layers of 3×3 convolutions across ResNet-50 layers. The slight variations are caused by the differences in the zero-pad sizes. The number of DRAM accesses also remains the same for 1×1 convolutions in Conv1 to Conv4. In Conv5, however, the proposed dataflow described in section 5.1.5 saves some DRAM accesses for small feature maps by utilizing a case-specific dataflow which reduces unnecessary re-fetch of input features. The minimum point in each convolution group corresponds to 1×1 convolutions when the filter stride is 2. DRAM

access curves show that in Conv1 to Conv4, when the number of DRAM accesses for weights increases, the number of DRAM accesses for input features decreases and vice versa.

Figure 5.13 shows the number of DRAM accesses for transferring output features from the internal SRAMs to the DRAM. In CARLA, the output features are transferred to the DRAM only once. As a result, the number of DRAM accesses for output features is only proportional to the size of the output. This number gradually decreases for the deeper layers.

Figure 5.13 also shows the total number of DRAM accesses. According to the results, the number of DRAM accesses for Conv5 is greater than other layers due to their higher number of filters. In addition, the number of DRAM accesses for Conv2 layers is relatively larger due to its larger output size. The discussion in this section can also be generalized to ResNet-101 and ResNet-152 which have the same groups of convolutions with more iterations of each group.

5.2.3 Comparison with state-of-the-art CNN implementations

Several existing works on hardware accelerator design for CNNs have targeted VGGNet-16 [66], [63], [69]. While CARLA supports all the variations across convolutional layers of ResNet-50 as a widely-used CNN, we also provide the evaluation results on VGGNet-16 to facilitate comparison with existing designs. Table 5.2 compares the implementation results of CARLA with the state-of-the-art. Two implementation metrics are of profound importance when comparing low-energy accelerators. First, the PUF measures dataflow efficiency for mapping the computations onto available PEs. Second, the number of DRAM accesses is critical and must be minimized since DRAMs consume orders of magnitude more energy than other functions. In addition, the power consumption of CARLA falls within the range of other low-energy accelerators. It is worth noting that the power and area numbers of CARLA are from post-synthesis results and they could differ from measurements of fabricated chips.

Eyeriss [63] utilizes an NOC-like structure to compute VGGNet-16 and processes batches of three images at a time to reduce DRAM accesses. Data batching is a suitable technique to train neural networks [26]. When running inference, however, many real-world applications demand to process each image frame individually to make a quick decision instead of waiting for the batch processing. Excessive batch processing latency is often unacceptable for real-time applications [2], [26], [69].

Table 5.2 Comparison of the proposed method with the state-of-the-art.

	Eyeriss [63]	Envision [66]	FID [69]	ZASCAD [71]		CARLA (This work)	
Technology (nm)	65	28	65	65		65	
On-chip SRAM (KB)	181.5	86	86	36.9		85.5	
Core Area (mm^2)	12.25	1.87	3.5	6		6.2	
Gate Count (NAND2)	1852 k	1950 k	1117 k	1036 k		938 k	
Frequency (MHz)	200	200	200	200		200	
Batch size	3	N/A	1	1		1	
Bit precision	16b	1-16b	16b	16b		16b	
#PEs	168	256-1024	192	192		196	
CNN model	VGG-16	VGG-16	VGG-16	VGG-16	ResNet-50	VGG-16	ResNet-50
Power (mW)	236	26	260	301	248	247	247
PUF % (filter size)	3×3: 26%	3×3: 32%	3×3: 89 %	3×3: 94%	total: 88%	3×3: 98%	1×1, 3×3: 98% 7×7: 45%
Latency (ms)	4309.5	598.8	453.3	421.8	103.6	396.9	92.7
Performance (Gops)	21.4	51.3	67.7	72.5	74.5	77.4	75.4 (83.26)†
Efficiency (Gops/W)	90.7	1973	260.4	240.9	300.4	313.4	305.3 (337.1)†
# DRAM access/batch (MB)	321.1	N/A	331.7	375.5	154.6	258.2	124.0

† In [71], the number of operations (Gop) in ResNet-50 is different from our calculations. The numbers in parenthesis are computed when using the same number of operations as [71].

Eyeriss suffers from a high latency of 4.3 s for executing VGGNet-16 and obtains a low PUF of 26% for VGGNet-16. CARLA achieves 11× faster image recognition with 72% higher PUF in 3×3 convolutions while consuming 31% less silicon area. Moreover, CARLA offers 3.6× higher performance, in terms of number of executed operations per second, and 3.5× higher on-chip energy efficiency due to its highly optimized model-specific dataflows.

Envision [66] takes advantage of the more advanced 28 nm UTBB FD-SOI technology and dynamic voltage-accuracy frequency scaling (DVAFS) to improve its energy efficiency. That design offers a relatively low PUF of 32% resulting in a latency of 598.8 ms when running VGGNet-16. CARLA outperforms the Envision architecture by achieving 34% lower latency while requiring 2.1× fewer gates. The number of DRAM accesses was not reported for Envision.

FID [69] uses a dataflow inspired from the computational pattern of FC layers to compute the convolutional layers of VGGNet-16. Figure 5.14 compares the number of DRAM accesses in CARLA with FID for each convolutional layer of VGGNet-16. In the figure, Conv 64-3-224 means a convolutional layer with 64 filters, 3 input channels, and in-fmap size of 224.

As these results show, our proposed pipelining scheme significantly improves data reuse in most convolutional layers and consequently reduces the number of re-fetched input features. A large portion of the total DRAM accesses are associated with fetching the input features. Hence, in

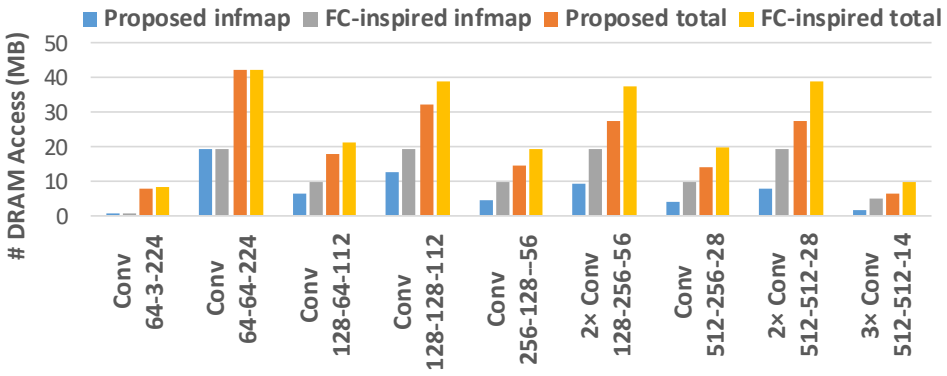


Figure 5.14 Number of DRAM access for CARLA compared to FID for VGGNet-16 [76].

CARLA, the achieved reduction in the number of DRAM accesses for fetching input features reduces the total number of DRAM accesses per layer. Overall, compared to FID, CARLA reduces the latency by 12.4% and the number of DRAM accesses by 22.1%.

ZASCA [71] is built upon FID to support a larger number of filter sizes. Compared to FID, ZASCA requires 2× larger silicon area and can perform the convolutional layers of ResNet-50. The ZASCA configuration for computing the original CNN model (dense activations) is called ZASCAD in [71]. The ZASCAD architecture was previously introduced under the name MMIE and PUFs for each convolutional layer are obtained in [70]. Figure 5.15 and Figure 5.16 compare PUFs obtained by ZASCAD and CARLA when computing 3×3 and 1×1 convolutional layers, respectively.

As shown in Figure 5.15, our proposed dataflow for 3×3 convolutions outperform the FC-inspired dataflow in ZASCAD especially in early convolutional layers. The FC-inspired dataflow in ZASCAD suffers from an issue called weight passing [70], which consists of wasted clock cycles to correct weight positions in the weight generator unit when computing a new output feature row. In addition, in ZASCAD there is no mechanism to enable performing convolution and DRAM data transfers simultaneously. The lack of such a mechanism introduces interrupts in the computations in ZASCAD, reducing the PUF and increasing the computation time.

In CARLA, the serial accumulation dataflow for 3×3 convolutions can utilize all PEs in all computation cycles and takes advantage of a mechanism to handle zero paddings at boundaries.

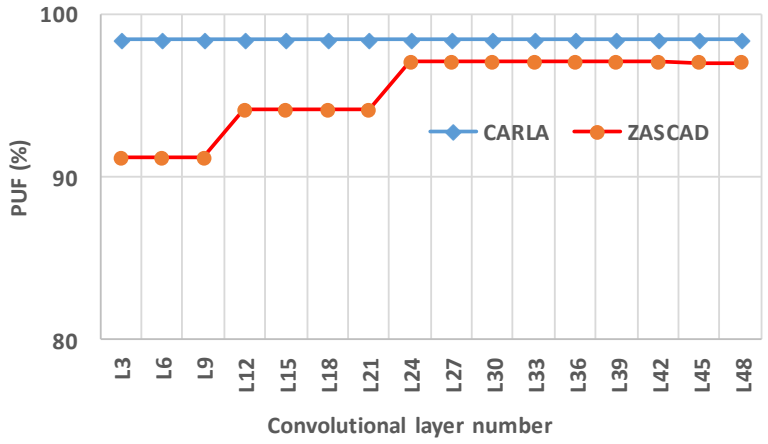


Figure 5.15 PUF for CARLA compared to ZASCAD for ResNet-50 [76].

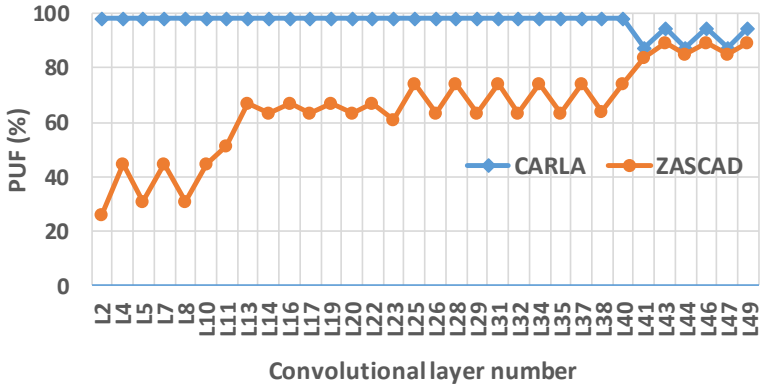


Figure 5.16 PUF for CARLA compared to ZASCAD for ResNet-50 [76].

Thus, it does not waste any clock cycle. In addition, utilizing paired SRAMs enables CARLA to overlap computations with SRAM-DRAM transfers. This results in a near perfect PUF of 98%.

As shown in Figure 5.16, the PUF of ZASCAD for 1×1 convolution is severely degraded in the majority of the convolutional layers. The 1×1 dataflow in ZASCAD does not efficiently map the computations to available PEs and many of them are idle. For instance, in the convolutional layer #2 (L2), 128 out of 192 of PEs do not contribute in computations [70]. The reconfigurable architecture of CARLA, on the other hand, maintains computation efficiency high by using a different dataflow for 1×1 convolution. Therefore, PEs have valid data to perform computations and they remain active in most computational cycles of the 1×1 convolution mode.

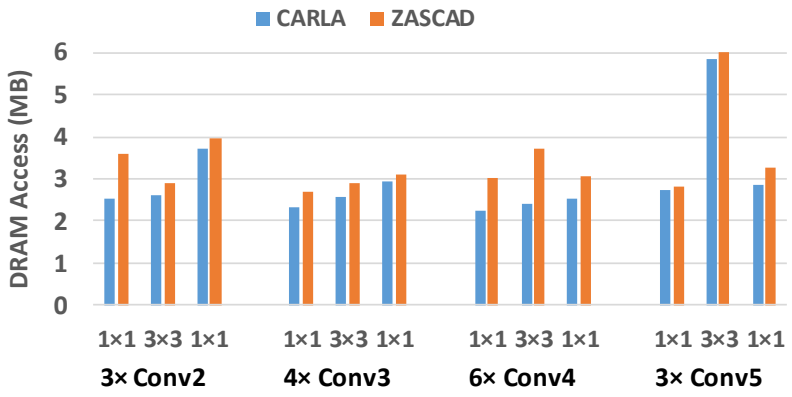


Figure 5.17 Number of DRAM access for CARLA compared to ZASCAD for ResNet-50 [76].

Figure 5.17 shows the number of DRAM accesses for 1×1 and 3×3 convolutions in CARLA and ZASCAD. CARLA demands fewer DRAM accesses for both 3×3 and 1×1 convolutions. In 1×1 convolution, the lower utilization factor of ZASCAD causes a larger number of re-fetches of the same data from the off-chip DRAM. In 3×3 convolution, as discussed for VGGNet-16, CARLA requires fewer DRAM accesses because of its pipeline feedback mechanism. Overall, CARLA requires 19% fewer DRAM accesses compared to ZASCAD, while offering 10.5% lower latency for computing convolutional layers of ResNet-50.

5.3 Conclusion

In this chapter, we described the CARLA architecture which is a convolutional accelerator with several operating modes to efficiently compute the convolutions in deep CNNs such as VGGNet-16 and ResNet-50. For 3×3 convolution, CARLA utilizes a new dataflow to maximally reuse the data fetched from DRAM. In addition, it utilizes an effective solution to avoid latency overhead while switching the rows and channels in boundaries of input feature maps. To support 1×1 convolution, CARLA maximizes data reuse by switching the datapath of the input data and the filter weights. When evaluated with ResNet-50, this architecture achieves a PUF of 98% across all 1×1 and 3×3 convolutions and supports other convolution structures such as 7×7. The CARLA architecture results in a latency of 92.7 ms and total DRAM accesses of 124.0 MB when computing the convolutional layers of ResNet-50 for classifying an input image.

CHAPTER 6 SEMI-STRUCTURED RANDOM ROW-WISE PRUNING FOR ENERGY-EFFICIENT AND FAST CONVOLUTION ACCELERATOR

This chapter first discusses the effect of filter weight sparsity on reducing the computation time and number of DRAM accesses. Next, a new pruning method called semi-structured random row-wise pruning is proposed for accelerating computation computations, and its effectiveness on the classification accuracy of ResNet-50 is evaluated. Finally, it is shown that the structure of CARLA is compatible with the proposed pruning method, and the computation speed up and reduction of the number of DRAM accesses are measured. The proposed pruning method is protected as a provisional patent by Huawei [77] and the experiments for evaluation of the proposed method have been performed at Huawei Montreal's research centre during a MITACS internship program.

6.1 Overview of existing pruning methods and their challenges

Pruning is a widely used model compression method to reduce the computational complexity of deep neural networks. Early applications of pruning focused on removing individual weights from FC layers since those layers contained most of the parameters [25]. However, recent deep CNN models favor more convolutional layers which are responsible for over 90 % of the computation time. Therefore, the objective of pruning has evolved from reducing the number of network parameters to reducing the computation time.

Pruning can be divided into unstructured [25], [31] and structured pruning [32]-[34]. In unstructured methods, the goal is to maximally remove individual weights with the aim of reducing the number of parameters regardless of the computation patterns in CNNs. This, however, introduces irregularity into weight matrices and makes it difficult for a hardware designer to benefit from sparsity for computation speed up and DRAM access reduction. For instance, removing some filter weights results in having zero values in CU registers in CARLA. While this eliminates computations in the corresponding PEs, it does not yield any speed up since CARLA has to wait for other PEs to complete their computations. The unstructured pruned CNNs require smaller memories to preserve weight values. However, pruned weights are removed irregularly and the irregular position of remaining weights has to be stored in a memory. This introduces additional

DRAM accesses to restore the location of weights and then feed the weights to the accelerator. If a randomizer is used to prune filter weights, however, it is not required to store indices. In this case, the irregularity in weight matrices still prevents computation speed up.

Structured filter pruning is one of the promising solutions to reduce CNN model complexity without affecting their computational structures. In contrast to pruning individual weights, in filter pruning entire insignificant filters are removed. Filter pruning, therefore, does not require indexing and keeps CNN model structure unchanged with fewer filters. Since removing each filter leads to removal of one output channel, this approach is also called channel pruning [34]. The proposed dataflows in CARLA operate on filter rows which have smaller granularity compared to filters. Therefore, pruning entire filters is a good fit to the computational dataflow of CARLA and can provide computation speed up. Despite the aforementioned advantages, pruning filters are more complex than pruning individual weights and requires sophisticated methodologies to obtain high pruning rates.

In this chapter, we propose a semi-structured random row-wise pruning with a low-complexity pruning process that achieves a high pruning rate while the obtained sparse model properly fits to the structure of typical CNN accelerators such as CARLA.

6.2 Proposed semi-structured random row-wise pruning

Figure 6.1 shows the proposed pruning method and how it can be applied to the structure of a CNN accelerator, e.g., CARLA. In CARLA, to perform 3×3 convolutions, each parallel unit (CU) accepts one filter row and performs row-wise convolution using three PEs. The proposed pruning method is depicted for 3×3 convolutions. A similar approach is applied for 1×1 convolutions. Since many of existing accelerators [63], [69], [71] also operate on filter rows, the proposed method is applicable to them to achieve computation speed up.

As shown in Figure 6.1, the proposed method is compatible with the structure of CARLA and removes filter rows in all (or groups of) filters randomly. As mentioned earlier, if a single weight is pruned, its value becomes zero and its corresponding computation in a PE is discarded. This, however, cannot results in speed up since other PEs inside that CU still need to perform computations. If all the weights in a filter row are removed, on the other hand, the CU becomes

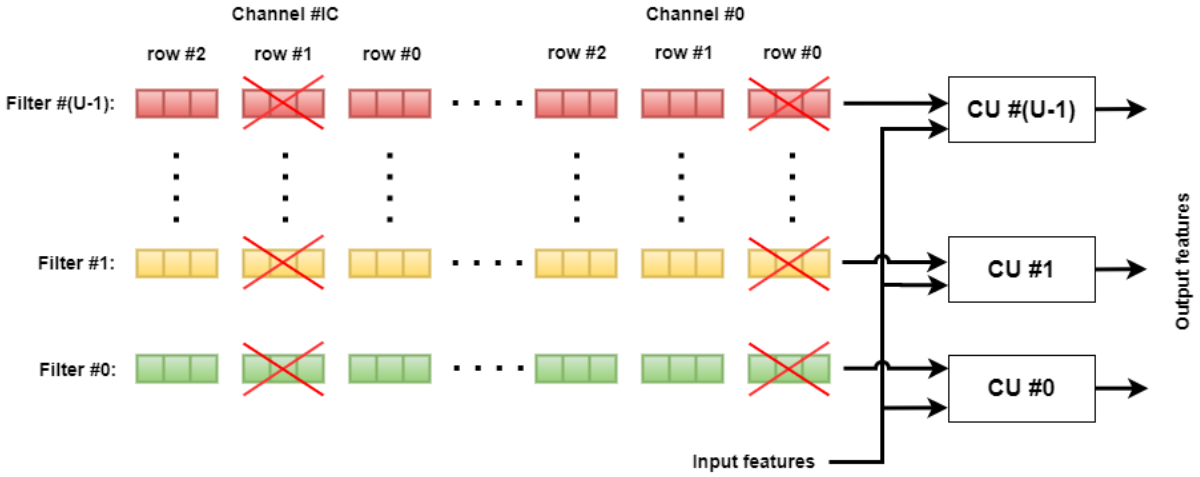


Figure 6.1 Proposed random row-wise pruning method (adapted from [104]).

idle. In other words, removing a filter row instead of a single weight, discards the entire corresponding computations in a CU. Since CUs are arranged in parallel even discarding computations in a CU cannot result in speed up since other parallel CUs are still busy. To simultaneously eliminate all the computations in all the CUs, the same filter row location must be pruned in all the filters. By pruning the same rows in all filters, the corresponding input features are not fetched. Therefore, the corresponding computations are eliminated, and the next filter rows are fed to the parallel units. This way, CARLA can save OL^2 clock cycles on eliminated computations and avoid fetching of $3\times U$ filter weights where OL is the out-fmap size and U is the number of parallel units.

To avoid the need to store the weights indices, the filter rows in the proposed method are pruned according to a sequence generated by a pseudo random number generator. For this purpose, we use a linear-feedback shift register (LFSR) similar to [30]. The LFSR accepts a random seed and for each filter row it generates a pseudo random number. The number is then compared with a threshold value. If it is above the threshold, the row is preserved, otherwise it is pruned. The same LFSR is implemented on hardware and given the same seed will results in generating the same pattern for discarding the filter rows.

Table 6.1 Comparison of accuracy of the proposed method with the state-of-the-art.

Pruning methods	Pruning rate	Top-1	Top-5
Baseline (dense model)	-	76.03	92.97
proposed, all filters	30%	75.6 (↓0.43)	92.64 (↓0.33)
proposed, all filters	50%	73.01 (↓2.94)	91.22 (↓1.75)
proposed, all filters	70%	67.57 (↓10.46)	87.97 (↓4.67)
proposed, g= 64	50%	74.16 (↓1.87)	91.52 (↓1.45)
proposed, g= 8	50%	74.51 (↓1.52)	91.93 (↓1.04)
proposed, g=1	50%	74.44 (↓1.59)	92.10 (↓0.87)
ThiNet [33]	50%	(↓1.87)	(↓1.12)
CP [78]	50%	-	(↓1.40)

6.3 Experimental results

To evaluate the effectiveness of the proposed pruning method on the classification accuracy of the CNN models, several experiments were performed on ResNet-50 using the ImageNet dataset. A pre-trained dense ResNet-50 model (baseline model) was selected for experiments and it is trained for 90 epochs. The learning rate is set to 0.01 and it is scaled by 10 after each 30 epochs.

Table 6.1 show the classification accuracy of the proposed random row-wise pruning method compared to some of well-known filter pruning methods. As the results show, pruning 30% of parameters with the proposed method negligibly degrades the top-5 accuracy of ResNet-50 by only 0.33%. On the other hand, when applying a high pruning rate of 70%, the impact of removing weights on ResNet-50 accuracy degradation is significant. If a moderate pruning rate of 50% is selected, the drop in accuracy is in an acceptable range to the unpruned baseline model.

In addition, the proposed method was examined for different cases where the patterns of pruning varies in different sets of filters. In the proposed method, rows in the same locations of different filters are pruned together. This can be performed in all filters or in groups of filters. For instance, CARLA performs convolutions on groups of 64 filters, i.e., g=64. Thus, instead of pruning filter rows in all filters with the same pattern, it is possible to divide filters into groups of 64 and use different pruning patterns for each group. On the other hand, NEON SIMD units in ARM processors typically consists of 8 parallel cores and pruning can be applied to groups of 8 filters, i.e., g=8.

As Table 6.1 illustrates, the accuracy of the models which use smaller groups of filters is closer to the baseline model. For instance, removing 50% of parameters with pruning rows in the same locations of all filters degrades the top-5 accuracy of ResNet-50 by 1.75% while this degradation is 1.45% and 1.04% for the groups of 64 and 8 filters, respectively. Table 6.1 compares the proposed method with well-known filter pruning methods. Since the classification accuracy of the baseline models in ThiNet and CP are slightly different from ours, only their accuracy degradation is reported in Table 6.1. Generally, in filter pruning methods, first the weak filters (or channels) are recognized, then they are removed and finally the whole CNN model is fine tuned by training for some epochs to recover the effect of damages by filter pruning on classification accuracy [33]. The ThiNet method, prunes filters (or channels) that have small impacts on feature map values in the next layers and iteratively repeats the pruning process. The CP method, on the other hand, solves a LASSO regression optimization problem to select the weak filters. As the results show, the classification accuracy of the proposed method is similar to the filter pruning methods. The proposed method, on the other hand, selects groups of filter rows randomly which is a much simpler strategy for pruning.

ResNet-50 includes 49 convolutional layers in its main body. To reduce the number of parameters, the weights in different layers can be pruned by different rates. For instance, in ThiNet, to prune 50% of parameters, the filters are pruned so that the number of parameters in the layers with 3×3 convolution and 1×1 convolution is reduced by 4× and 2×, respectively. Similarly, we prune the same number of parameters in each layer. Therefore, in the proposed pruning method, to achieve a pruning rate of 50%, the number of filter rows in 3×3 convolutional layers is reduced by 4× while the number of rows in convolutional layers with 1×1 filters is reduced by 2×.

6.4 Compatibility of CARLA with the proposed pruning method

Figure 6.2 shows the number of DRAM accesses in CARLA when running a sparse model with the proposed pruning method compared to the baseline model. As the results show, the number of DRAM accesses continuously decreases in the convolutional layers of the proposed method. This reduction is achieved by avoiding fetching the pruned filter rows. Additionally, since rows in different filters are pruned together, fetching the corresponding input features is also eliminated. In other words, pruning a set of filter rows not only reduces the number of DRAM accesses required

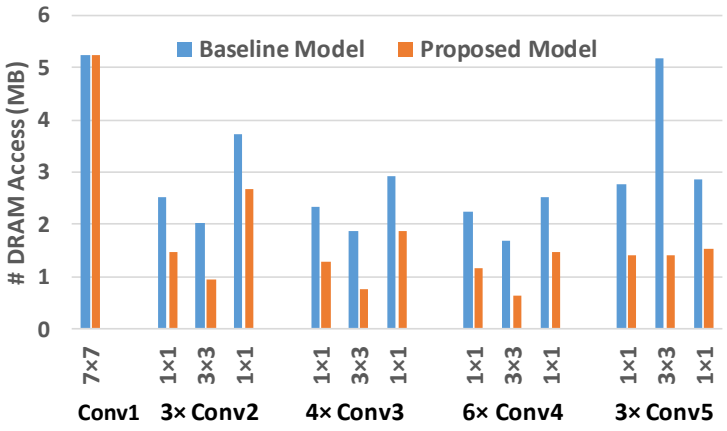


Figure 6.2 Number of DRAM access in CARLA for the proposed sparse model compared to baseline model.

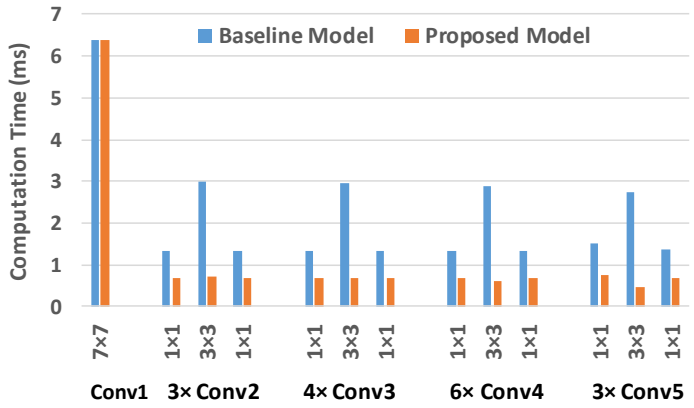


Figure 6.3 Computation time of CARLA for the proposed method compared to baseline model.

for those filter rows but it also avoids fetching the corresponding input features. Therefore, the reduction in the total number of DRAM accesses in CARLA is more than the reduction due to pruning filter weights. Overall, CARLA requires 65.72 MB DRAM accesses when performing the proposed sparse model which is 1.89× fewer than the number of DRAM accesses for the baseline model.

Figure 6.3 shows the computation time in CARLA for each convolutional layer of the proposed sparse model. In all the convolutional layers with 1×1 filters, CARLA achieves 2× speed up when

the number of filter rows, is reduced by half due to pruning. In addition, when the filter rows are pruned by 4× in convolutional layers with 3×3 filters, the speedup of 4× in computation time is achieved. This indicates the prefect match of the proposed semi-structured random row-wise pruning method with the structure of CARLA in which pruning filter rows directly affects the computation time. Overall, CARLA can perform the convolution computations of the proposed sparse model in 36.5 ms which is 2.5× faster than the baseline model.

6.5 Conclusion

In this chapter, we proposed a new pruning method called semi-structured random row-wise pruning which is designed according to computational patterns of CNN accelerators. The proposed pruning method not only reduces the number of parameters but also decrease the computation time of the accelerator. The experimental results show that removing half of the parameters in ResNet-50 by employing the proposed method still maintains the classification accuracy of the sparse model in an acceptable range of the one in the dense model. When comparing the performance of CARLA for the proposed sparse model with the dense model, CARLA requires 1.88× fewer DRAM accesses while achieving 2.5× faster image recognition.

CHAPTER 7 CONCLUSION AND FUTURE WORK

7.1 Summary of the work

In this thesis, we introduced new methodologies and techniques to design fast, flexible and energy efficient CNN accelerators. We provided hardware architectures, design trade-offs, and performance analysis of our designs and showed their superiority over existing methods in terms of the hardware implementation metrics of computation time, number of DRAM accesses and PUF.

Chapter 3 proposed a new architecture that uses serial accumulation dataflow to perform convolution computation with several homogenous parallel units. In the proposed architecture, PEs inside each parallel unit are cascaded together and PE outputs are accumulated serially until they are written in on-chip SRAMs by the last serial PE. The proposed dataflow employs all the PEs for computation and therefore outperforms the existing methods in terms of lowering latency and reducing the number of DRAM access on VGGNet-16 benchmark. For example, the architecture achieves lower latency by 10.9× and requires 21% fewer DRAM accesses per batch compared to Eyeriss when performing image classification.

In chapter 4, the trade-off between the increasing SRAM size on reducing the number of DRAM accesses and increasing the silicon area/power was analyzed. Since efficient CNN accelerators utilize small SRAM units in each parallel unit, the heterogeneous distributed SRAM configuration was proposed to merge SRAMs and improve area and power efficiency. In the proposed method, shallow SRAMs inside parallel units are used to accumulate partial results, with wide-word lengths, while the SRAMs for storing outputs, with shorter word-length, are merged between adjacent parallel units. Implementation results show that the proposed configuration reduces the area by 21% and improves the on-chip energy-efficiency by 18% compared to designs which use an ordinary ping-pong structure for SRAM-DRAM data transfer.

In chapter 5, a reconfigurable and low-energy accelerator called CARLA was proposed. The reconfigurability enables CARLA to support efficient computation of various convolutional layer configurations with different filter sizes. In the 3×3 convolution mode, CARLA uses the serial accumulation dataflow by cascading PEs in each parallel unit while in 1×1 mode, PEs work independently. In addition, CARLA swaps the input feature and filter weight data movement when

performing 1×1 convolution mode compared to the 3×3 mode. This flexibility enables the architecture to take advantage of reusing high volumes of input data in 1×1 convolutions where the spatial size of filters shrinks to only a single weight in each channel. CARLA performs 7×7 convolutions and other filter sizes using 3×3 and 1×1 convolutions with no additional hardware cost. This avoids a significant increase in hardware complexity while the impacts on overall performance remain negligible. In addition, CARLA uses a new pipelining scheme to increase on-chip data reuse by taking advantage of feedbacks paths. CARLA achieves a remarkable PUF of 98% for the majority of 1×1 and 3×3 convolutions on the ResNet-50 benchmark and offers 10.5% lower latency for computing convolutional layers of ResNet-50 while requires 19.8% fewer DRAM accesses compared to the existing method (ZASCAD [71]).

In chapter 6, we proposed a new pruning method called semi-structured random row-wise pruning which randomly removes rows in the same locations of different filters. The experimental results showed the negligible effect on accuracy but very significant reduction in latency and number of DRAM accesses. For instance, CARLA can perform the convolutional layers of pruned ResNet-50 in only 36.5 ms and requires 65.72 MB DRAM accesses which is 2.5× faster and includes 1.89× fewer access numbers compared to the dense model.

7.2 Thesis limitations and future works

Even though the work described in this thesis has presented multiple contributions in the field of hardware design for efficient CNN accelerators, several improvements could be proposed to the solutions presented and, moreover, many extensions could be provided.

7.2.1 Word-length optimization

In this thesis, the proposed accelerators were compared with the existing work mainly on implementation metrics such as latency, PUF and number of DRAM accesses. In order to make fair comparisons, we limited our bit-width selection for data values and internal computations similar to the selected bit-width by the existing designs. For instance, in CARLA the bit-widths for filter weights, input/output features are set to 16-bits while the bit-width of internal accumulators and partial result SRAMs are set to 24 bits, as done in ZASCAD [71]. On the other hand, the allocated bit-width to data elements has high impact on implementation results such as area usage,

power consumption, and the number of DRAM accesses. Therefore, the future work can be improving the design efficiency by selecting more appropriate bit widths. Proper selection of the data word length requires an error cost model to describe the trade-off between CNN accuracy and hardware cost. The designed accelerator in this thesis can be helpful to obtain the design cost for different range of word length.

7.2.2 Supporting other CNN families

The proposed architectures in this thesis are highly optimized to perform the standard convolution computations in modern CNN models such as ResNet. These models are the first choice for image recognition and are used extensively for many vision applications and benchmarking in the literature. Therefore, it is essential to propose an efficient accelerator for these standard CNN models and this work was devoted for this purpose.

In recent years, other types of CNN models have emerged that employ other convolution structures rather than the standard ones to reduce the complexity of CNN models while maintaining the same accuracy level as of standard CNN models. For instance, the MobileNet family is a category of compact CNN models which utilize 3×3 depth-wise and 1×1 point-wise convolutions in their structures. In contrast to standard convolution in which filters have three dimensions and all of them convolve with the same in-fmaps, in depth-wise convolution filers have two dimensions (only one channel) and each filter only convolves with its corresponding in-fmap, i.e., each filter with a different in-fmap. The 1×1 point-wise convolution is the same as 1×1 standard convolution. Since in depth-wise convolution, different filters convolve with different input features there is no data sharing in computations. Therefore, the parallel units in a CNN accelerator, must be fed with independent data which requires a high bandwidth for fetching data from external DRAM. This requirement limits the efficiency of the accelerators since in the lack of valid data for computations many of the PEs inside the accelerator remain idle.

CARLA can perform MobileNet models more efficiently compared to existing works on standard convolutions. This is due to the fact that, in MobileNet models, the majority of the computations are in 1×1 convolutions (e.g. 94.86% MAC operations in MobileNet v1) and CARLA significantly outperforms the existing works on performing 1×1 convolutions. CARLA can also be used to perform the 3×3 depth-wise convolution while its performance efficiency depends on the

bandwidth of the off-chip DRAM in the mobile device. Since depth-wise convolution represent a small proportion of computations (e.g., only 3.06% in MobileNet v1), its effect on the overall performance is insignificant.

Although CARLA can achieve acceptable results on current MobileNet models, its architecture is not optimized for different convolution types such as depth-wise convolutions which demand high-volumes of data for computations. A promising future work, therefore, would be to develop new dataflow architectures to support emerging convolution types and integrate them into the structure of CARLA to enable supporting both standard and compact convolutions.

7.2.3 On-device CNN training

Most of the existing low-energy CNN accelerators such as CARLA are limited to perform the convolution computations of CNN inference. By popularization of deep CNNs, the demand for on-device training is growing fast while complex process of training forces the developers to use powerful GPUs. In other words, the complexity of backpropagation functions such as computing derivatives and the required storage to preserve weights values and their updates are prohibitive factors to deploy training process on portable devices and embedded systems. Therefore, a future work in this field is to find an acceptable solution to train CNNs with much simpler processes which are compatible with available hardware resources on edge devices.

REFERENCES

[1] A. Krizhevsky, I. Sutskever, and G. Hinton, "ImageNet classification with deep convolutional neural networks," in *Proc. Adv. in Neural Inf. Process. Syst.*, pp. 1097–1105, 2012.

[2] S. Ren, K. He, R. Girshick, J. Sun, "Faster R-CNN: Towards real-time object detection with region proposal networks," *in Proc. Adv. in Neural Inf. Process. Syst.*, pp. 91–99, 2015.

[3] J. Long, E. Shelhamer, T. Darrell, "Fully Convolutional Networks for Semantic Segmentation," *in IEEE Conf. CVPR*, pp. 3431–3440, 2015.

[4] J. Donahue, L. Anne Hendricks, S. Guadarrama, M. Rohrbach, S. Venugopalan, S. Saenko, and T. Darrell, "Long-term recurrent convolutional networks for visual recognition and description," *in IEEE Conf. CVPR*, pp. 2625-2634, 2015.

[5] Y. LeCun, Y. Bengio, and Geoffrey Hinton, "Deep learning," Nature, vol. 521, pp. 436-444, May 2015.

[6] Y. LeCun, et al. "Gradient-based learning applied to document recognition," in *Proc. of the IEEE*, 86(11) 22782324, 1998.

[7] V. Oculus, "Oculus rift-virtual reality headset for 3d gaming," URL: http://www. oculusvr.com, 2012.

[8] M. Horowitz. "Computing's Energy Problem (and what we can do about it)," in *IEEE Int. Solid-State Circuits Conf. (ISSCC)*, pp. 10–14, 2014.

[9] K. He, X. Zhang, S. Ren, and J. Sun, "Deep Residual Learning for Image Recognition," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit.*, pp. 770-778, 2016.

[10] K. Ovtcharov, O. Ruwase, J.-Y. Kim, J. Fowers, K. Strauss, and E. S. Chung, "Accelerating deep convolutional neural networks using specialized hardware," Microsoft Research Whitepaper, 2015.

[11] T. Courtney."How Much Power Do Common Devices Consume?", 2016. [Online]. Available: https://www.gozolt.com/blog/power-devices-consume/.

[12] L. Fei-Fei, et al., "CS231n: Convolutional Neural Networks for Visual Recognition Course Notes," 2016, [Online]. Available: http://cs231n.stanford.edu/.

[13] A. Krizhevsky, G. Hinton, "Learning Multiple Layers of Features from Tiny Images", 2009.

[14] Y. Netzer, T. Wang, A. Coates, A. Bissacco, Bo Wu, and A. Y. Ng "Reading digits in natural images with unsupervised feature learning," in *Adv. in Neural Inf. Process. Syst. Workshops*, 2011.

[15] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei, "ImageNet: A Large-Scale Hierarchical Image Database," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit.*, pp. 1097-1105, 2009.

[16] A. Berg, J. Deng, and L. Fei-Fei. "Large scale visual recognition challenge," URL: www.imagenet.org/challenges, 2010.

[17] Zeiler, M. D. and Fergus, R. "Visualizing and understanding convolutional networks," in *Proc. IEEE European Conf. Comput. Vis.*, pp. 818-833. Springer, Cham, 2014.

[18] Szegedy, C., Liu, W., Jia, Y., Sermanet, P., Reed, S., Anguelov, D., Erhan, D., Vanhoucke, V., Rabinovich, A.," Going deeper with convolutions." in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit.*, pp. 1-9, 2015.

[19] K. Simonyan, A. Zisserman, "Very deep convolutional networks for large-scale image recognition." *arXiv preprint arXiv:1409.1556*, 2014.

[20] J. Hu, L. Shen, G. Sun, "Squeeze-and-Excitation Networks," *arXiv preprint arXiv:1709.01507 [cs.CV]*, 2018.

[21] R. Andri, L. Cavigelli, D. Rossi, and L. Benini, "YodaNN: An Ultra-Low Power Convolutional Neural Network Accelerator Based on Binary Weights," in *Proc. IEEE Symp. on VLSI (ISVLSI)*, pp. 236–241, July 2016.

[22] R. LiKamWa, W. Zhen, A. Carroll, F. Xiaozhu Lin, and L. Zhong "Draining our glass: An energy and heat characterization of google glass," in *Proc. Asia-Pacific Workshop on Systems*, 2014.

[23] Camdo Inc., "Power Consumption By Gopro Camera Model," 2016. [Online]. Available: http://cam-do.com/pages/power-consumption-by-gopro-camera-model/

[24] N. Muralimanohar, R. Balasubramonian, and N. P. Jouppi, "Cacti 6.0: A tool to model large caches," *HP Laboratories*, pp. 22–31, 2009.

[25] S. Han, H. Mao, and W. J. Dally, "Deep compression: Compressing deep neural networks with pruning, trained quantization and Huffman coding," in *Int. Conf. on Learning Representations*, 2016.

[26] S. Han, X. Liu, H. Mao, J. Pu, A. Pedram, M. Horowitz, and W. J. Dally, "EIE: Efficient inference engine on compressed deep neural network," in *Proc. Int. Symp. on Computer Architecture*, pp. 243-254. 2016.

[27] E. Nurvitadhi, et al. "Can FPGAs beat GPUs in accelerating next-generation deep neural networks?" *in Proc. ACM/SIGDA Int. Symp. on Field-Programmable Gate Arrays,* pp. 5-14, 2017.

[28] J. Yu, A. Lukefahr, D. Palframan, G. Dasika, R. Das, and S. Mahlke, "Scalpel: Customizing dnn pruning to the underlying hardware parallelism," *in Proc. of Int. Symp. on Computer Architecture,* pp. 548-560, 2017.

[29] M. J. Shafiee, P. Siva, and A. Wong. "Stochasticnet: Forming deep neural networks via stochastic connectivity," IEEE Access 4 (2016): 1915-1924.

[30] A. Ardakani, C. Condo, and W.J. Gross, "Sparsely-connected neural networks: towards efficient VLSI implementation of deep neural networks," arXiv preprint arXiv:1611.01427, 2016.

[31] H. Mao, S. Han, J. Pool, W. Li, X. Liu, Y. Wang, and W.J. Dally, "Exploring the granularity of sparsity in convolutional neural networks." in *Proc. IEEE Conf. on Comput. Vis. and Pattern Recognition Workshops*, pp. 13-20, 2017.

[32] Z. Liu, J. Li, Z. Shen, G. Huang, S. Yan, and C. Zhang, "Learning efficient convolutional networks through network slimming." in *Proc. IEEE Int. Conf. Comput. Vis.*, pp. 2736-2744,2017.

[33] J. Luo, J. Wu, W. Lin, "ThiNet: A filter level pruning method for deep neural network compression", in *Proc. IEEE Int. Conf. on Comput. Vis.*, pp.5058-5066, 2017.

[34] 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 *Proc. Adv. Neural Inf. Process. Syst.*, pp. 875-886, 2018.

[35] B. L. Deng, G. Li, S. Han, L. Shi and Y. Xie, "Model Compression and Hardware Acceleration for Neural Networks: A Comprehensive Survey," *in Proc. IEEE,* vol. 108, no. 4, pp. 485-532, 2020.

[36] B. Zoph, V. Vasudevan, J. Shlens, and Q. V. Le. "Learning transferable architectures for scalable image recognition," in *Proc. IEEE conf. Comput. Vis. Pattern Recognit.*, pp. 8697-8710. 2018.

[37] X. Zhang, X. Zhou, M. Lin, and J. Sun, "Shufflenet: An extremely efficient convolutional neural network for mobile devices." in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit.*, pp. 6848-6856, 2018.

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

[39] M. Sandler, A. Howard, M. Zhu, A. Zhmoginov, and L.C. Chen, "Mobilenetv2: Inverted residuals and linear bottlenecks," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit.*, pp. 4510-4520, 2018.

[40] M. Courbariaux, Y. Bengio, and J.P. David. "Binaryconnect: Training deep neural networks with binary weights during propagations," in *Proc. Adv. in Neural Inf. Process. Syst.*, pp. 3123-3131. 2015.

[41] M. Courbariaux , et al.,"Binarized Neural Networks: Training Neural Networks with Weights and Activations Constrained to +1 or -1," *in arXiv preprint arXiv:1602.02830v3 [cs.LG]*, 2016.

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

[43] F. Li, and B. Liu. "Ternary Weight Networks." in arXiv preprint arXiv:1605.04711 (2016).

[44] A. Ardakani, C. Condo, and W.J. Gross, "A convolutional accelerator for neural networks with binary weights." in *Proc. IEEE Int. Symp. on Circuits and Syst.*, pp. 1-5, 2018.

[45] M. Rhu, M. O'Connor, N. Chatterjee, J. Pool, Y. Kwon, and S. W. Keckler, "Compressing DMA engine: Leveraging activation sparsity for training deep neural networks," in *Proc. IEEE Int. Symp. High Perform. Comput. Archit.*, pp. 78–91, 2018.

[46] A. Jain, A. Phanishayee, J. Mars, L. Tang, and G. Pekhimenko, "Gist: Efficient data encoding for deep neural network training," in *Proc. Int. Symp. Comput. Archit.*, pp. 776–789, 2018.

[47] E. L. Denton, W. Zaremba, J. Bruna, Y. LeCun, and R. Fergus, "Exploiting Linear Structure Within Convolutional Networks for Efficient Evaluation." *in Proc. Adv. in Neural Inf. Process. Syst.*, pp.1269–1277, 2014.

[48] M. Mathieu, M. Henaff, and Y. LeCun, "Fast Training of Convolutional Networks through FFTs." in *arXiv preprint arXiv:1312.5851*, 2013.

[49] L. Cavigelli, M. Magno, and L. Benini, "Accelerating Real-Time Embedded Scene Labeling with Convolutional Networks," *in Proc. ACM/IEEE Des. Autom. Conf.*, pp. 1-6, 2015.

[50] N.P. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, A. Borchers, and R. Boyle, "In-datacenter performance analysis of a tensor processing unit." *in Proc. Int. Symp. on Comput. Archit.*, pp. 1-12, 2017.

[51] C. Zhang, P. Li, G. Sun, Y. Guan, B. Xiao, and J. Cong. "Optimizing FPGA-based accelerator design for deep convolutional neural networks." *in Proc. Intl. Symp. On Field-Programmable Gate Arrays*, pp. 161–170, 2015.

[52] S. Cadambi, A. Majumdar, M. Becchi, S. Chakradhar, and H. P. Graf. "A programmable parallel accelerator for learning and classification," *in Proc. Int. Conf. on Parallel Archit. and Compilation Techniques*, pp. 273–284, 2010.

[53] C. Farabet, C. Poulet, J. Y. Han, and Y. LeCun. "Cnp: An FPGA-based processor for convolutional networks," *in Int. Conf. on Field Programmable Logic and Applications*, pp. 32–37, 2009.

[54] C. Farabet, et al., "Neuflow: A runtime reconfigurable dataflow processor for vision," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit*, pp. 109-116, 2011.

[55] G. Lacey, G. Taylor, S. Areibi, "Deep Learning on FPGAs: Past, Present, and Future," in *arXiv preprint arXiv:1602.04283 [cs.DC]*, 2016.

[56] J. Qiu, J. Wang, S. Yao, K. Guo, B. Li, E. Zhou, J. Yu, T. Tang, N. Xu, S. Song, and Y. Wang, "Going deeper with embedded FPGA platform for convolutional neural network." in Proc. *Int. Symp. on Field-Programmable Gate Arrays*, 2016, pp. 26-35.

[57] Y. Jia, E. Shelhamer, J. Donahue, S. Karayev, J. Long, R. Girshick, S. Guadarrama, and T. Darrell, "Caffe: Convolutional architecture for fast feature embedding." in *Proc. ACM Int. Conf. on Multimedia*, 2014, pp. 675-678.

[58] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, and M. Kudlur, "Tensorflow: A system for large-scale machine learning," in *Symp. on Operating Syst. Design and implementation*, pp. 265-283, 2016.

[59] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, and A. Desmaison, "Pytorch: An imperative style, high-performance deep learning library," in *Proc. Adv. in Neural Inf. Process. Syst.*, 2019, pp. 8026-8037.

[60] J. Sim, J.-S. Park, M. Kim, D. Bae, Y. Choi, and L.-S. Kim, "A 1.42 TOPS/W deep convolutional neural network recognition processor for intelligent IoE systems," *in Proc. IEEE Int. Solid-State Circuits Conf. (ISSCC)*, pp. 264–265, 2016.

[61] L. Cavigelli, D. Gschwend, C. Mayer, S. Willi, B. Muheim, and L. Benini, "Origami: A convolutional network accelerator," *in Proc. Great Lakes Symp. VLSI*, 2015, pp. 199–204.

[62] S. Park, K. Bong, D. Shin, J. Lee, S. Choi, and H.-J. Yoo, "A 1.93TOPS/W scalable deep learning/inference processor with tetraparallel MIMD architecture for big-data applications," in *Proc. IEEE Int. Solid-State Circuits Conf.*, 2015, pp. 1–3.

[63] Y.-H. Chen, T. Krishna, J. Emer, and V. Sze, "Eyeriss: An energy efficient reconfigurable accelerator for deep convolutional neural networks," in *Proc. IEEE Int. Solid-State Circuits Conf.*, pp. 262–263, 2016.

[64] Y. H. Chen, T. J. Yang, J. Emer, and V. Sze. "Eyeriss v2: A flexible accelerator for emerging deep neural networks on mobile devices," in *IEEE J. Emerging and Selected Topics in Circuits and Syst.*, vol. 9, no. 2, pp. 292-308, 2019.

[65] B. Moons and M. Verhelst, "A 0.3–2.6 TOPS/W precision-scalable processor for real-time large-scale ConvNets," in *Proc. IEEE Symp. VLSI Circuits*, pp. 1–2, June 2016.

[66] B. Moons, R. Uytterhoeven, W. Dehaene, and M. Verhelst, "Envision: A 0.26-to-10 TOPS/W subword-parallel dynamic-voltage-accuracy-frequency-scalable convolutional neural network processor in 28 nm FDSOI," *in IEEE Int. Solid-State Circuits Conf.*, pp. 246–247, 2017.

[67] S. Wang, D. Zhou, X. Han, and T. Yoshimura, "ChainNN: An energy-efficient 1D chain architecture for accelerating deep convolutional neural networks," in *Design, Automation & Test in Europe Conf. & Exhibition*, pp. 1032-1037, 2017.

[68] Tu, F., Yin, S., Ouyang, P., Tang, S., Liu, L. and Wei, S., "Deep convolutional neural network architecture with reconfigurable computation patterns," *IEEE Trans. on Very Large Scale Integration (VLSI) Systems*, 25(8), pp. 2220-2233, 2017.

[69] A. Ardakani, C. Condo, M. Ahmadi, W.J. and Gross, "An Architecture to Accelerate Convolution in Deep Neural Networks." *in IEEE Trans. on Circuits and Systems I: Regular Papers*, pp. 1349-1362, 2017.

[70] A. Ardakani, C. Condo, and W.J. Gross, "Multi-mode inference engine for convolutional neural networks," in arXiv preprint arXiv:1712.03994, 2017.

[71] A. Ardakani, C. Condo, and W.J. Gross, "Fast and efficient convolutional accelerator for edge computing" in *IEEE Trans. on Computers*, vol. 69, no. 1, 2020, pp. 138-152.

[72] T. Ujiie, M. Hiromoto, and T. Sato, "Approximated Prediction Strategy for Reducing Power Consumption of Convolutional Neural Network Processor," *in Proc. IEEE Conf. on Comput. Vis. and Pattern Recognition Workshops*, pp. 870–876, 2016.

[73] M. Ahmadi, S. Vakili, P. Langlois, "Power reduction in CNN pooling layers with a preliminary partial computation strategy," in *Proc. IEEE North-East Workshop on Circuits and Systems*, pp. 125-129, 2018.

[74] M. Ahmadi, S. Vakili, P. Langlois, "An Energy-Efficient Accelerator Architecture with Serial Accumulation Dataflow for Deep CNNs," in *proc. IEEE North-East Workshop on Circuits and Systems*, pp. 214-217, 2020.

[75] M. Ahmadi, S. Vakili, P. Langlois, "Heterogeneous Distributed SRAM Configuration for Energy-Efficient Deep CNN Accelerators," in *proc. IEEE North-East Workshop on Circuits and Systems*, pp. 287-290, 2020.

[76] M. Ahmadi, S. Vakili, P. Langlois, "CARLA: A Convolution Accelerator with a Reconfigurable and Low-energy Architecture," in *arXiv preprint arXiv:2010.00627[cs.AR]*, [Online]. Available: https://arxiv.org/abs/2010.00627, 2020.

[77] Vanessa Courville, M. Ahmadi, M. Zolnouri, "Methods, Systems and Media for Random Semi-Structured Row-Wise Pruning in Neural Networks," *U.S. Patent Application No. 16943573*, 2020.

[78] Y. He, X. Zhang, and J. Sun, "Channel pruning for accelerating very deep neural networks," in *IEEE Int. Conf. on Comput. Vis.*, pages 1389–1397, 2017.

[79] Y. LeCun, J. S. Denker, and Sara A. Solla. "Optimal brain damage," in *Proc. Adv. in Neural Inf. Process. Syst.*, pp. 598-605. 1990.

[80] Y. Liu, S. Yang, P. Wu, C. Li, and M. Yang,"L1-norm low-rank matrix decomposition by neural networks and mollifiers," in *IEEE Trans. Neural Netw. Learn. Syst.*, vol. 27, no. 2, pp. 273–283, 2016.

[81] C. Tai, T. Xiao, Y. Zhang, and X. Wang, "Convolutional neural networks with low-rank regularization," in *Int. Conf. on Learning Representations*, 2016.

[82] M. Masana, J. V. D. Weijer, L. Herranz, A. D. Bagdanov, and J. M. Alvarez, "Domain-adaptive deep network compression," in *Proc. IEEE Int. Conf. Comput. Vis.*, pp. 4289–4297, 2017.

[83] X. Zhang, J. Zou, X. Ming, K. He, and J. Sun, "Efficient and accurate approximations of nonlinear convolutional networks," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit.* pp. 1984–1992, 2015.

[84] Y.-D. Kim, E. Park, S. Yoo, T. Choi, L. Yang, and D. Shin, "Compression of deep convolutional neural networks for fast and low power mobile applications," in in *Int. Conf. on Learning Representations*, 2016.

[85] M. Astrid and S.-I. Lee, "CP-decomposition with tensor power method for convolutional neural networks compression," in *Proc. IEEE Int. Conf. Big Data Smart Comput.*, pp. 115–118, 2017.

[86] M. Zhou, Y. Liu, Z. Long, L. Chen, and C. Zhu, "Tensor rank learning in CP decomposition via convolutional neural network," in *Signal Process., Image Commun.*, vol. 73, pp. 12–21, 2019.

[87] K. Wang, Z. Liu, Y. Lin, J. Lin, and S. Han, "HAQ: hardware-aware automated quantization with mixed precision," in *Proc. Conf. Comput. Vis. Pattern Recognit.*, pp. 8612–8620, 2019.

[88] D. Zhang, J. Yang, D. Ye, and G. Hua, "LQ-Nets: Learned quantization for highly accurate and compact deep neural networks," in *Proc. Eur. Conf. Comput. Vis.*, pp. 365–382, 2018.

[89] S. Jung et al., "Learning to quantize deep networks by optimizing quantization intervals with task loss," in *Proc. IEEE/CVF Conf. Comput. Vis. Pattern Recognit.*, pp. 4350–4359, 2019.

[90] L. Deng, P. Jiao, J. Pei, Z. Wu, and G. Li,"GXNOR-Net: Training deep neural networks with ternary weights and activations without full-precision memory under a unified discretization framework," in *Neural Netw.*, vol. 100, pp. 49–58, 2018.

[91] P. Yin, S. Zhang, J. Lyu, S. Osher, Y. Qi, and J. Xin,"BinaryRelax: A relaxation approach for training deep neural networks with quantized weights," in *SIAM J. Imag. Sci.*, vol. 11, no. 4, pp. 2205–2223, 2018.

[92] C. Leng, Z. Dou, H. Li, S. Zhu, and R. Jin, "Extremely low bit neural network: Squeeze the last bit out with admm," in *Proc. AAAI*, 2018.

[93] N. Wang, J. Choi, D. Brand, C.-Y. Chen, and K. Gopalakrishnan, "Training deep neural networks with 8-bit floating point numbers," in *Proc. Adv. Neural Inf. Process. Syst.*, pp. 7685–7694, 2018.

[94] M. Zhu and S. Gupta, "To prune, or not to prune: Exploring the efficacy of pruning for model compression," in arXiv:1710.01878, 2017, [Online]. Available: http://arxiv.org/abs/1710.01878.

[95] Z. Liu, J. Xu, X. Peng, and R. Xiong, "Frequency-domain dynamic pruning for convolutional neural networks," in *Proc. Adv. Neural Inf. Process. Syst.*, pp. 1049–1059, 2018.

[96] R. Yu et al., "NISP: Pruning networks using neuron importance score propagation," in *Proc. IEEE/CVF Conf. Comput. Vis. Pattern Recognit.*, pp. 9194–9203, 2018.

[97] -, "MNIST database," in Wikipedia, [Online]. Available: https://en.wikipedia.org/wiki/MNIST_database, 2020.

[98] J. Jo, S. Cha, D. Rho, and I. Park, "DSIP: A scalable inference accelerator for convolutional neural networks," in *IEEE J. Solid-State Circuits*, vol. 53, no. 2, 2017, pp. 605-618.

[99] T. Luo, L. Shaoli, L. Li, Y. Wang, S. Zhang, T. Chen, Z. Xu, O. Temam, and Y. Chen. "Dadiannao: A neural network supercomputer," in *IEEE Trans. on Computers*, vol. 66, no. 1, pp. 73-88, , 2016.

[100] A.M. Caulfield et al., "A Cloud-Scale Acceleration Architecture," in *Proc. IEEE/ACM Int. Symp. Microarchit.*, pp. 1-13, 2016.

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

[102] Vedaldi, Andrea, and Karel Lenc. "Matconvnet: Convolutional neural networks for matlab." *in Proc. ACM Int. Conf. on Multimedia*, pp. 689-692. 2015.

[103] -, High-Bandwidth Memory (HBM) reinventing memory technology, *AMD documents*, [Online]. Available https://www.amd.com/system/files/documents/high-bandwidth-memory-hbm.pdf

[104] M. Ahmadi, "Semi-structured random row-wise pruning," *Technical report*, Huawei Montreal research center, 2019.