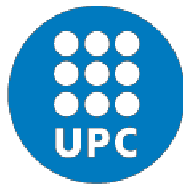


UNIVERSITAT POLITÈCNICA
DE CATALUNYA
BARCELONATECH

Low-power accelerators for cognitive computing

Marc Riera Villanueva

ADVERTIMENT La consulta d'aquesta tesi queda condicionada a l'acceptació de les següents condicions d'ús: La difusió d'aquesta tesi per mitjà del repositori institucional UPCommons (http://upcommons.upc.edu/tesis) i el repositori cooperatiu TDX (http://www.tdx.cat/) ha estat autoritzada pels titulars dels drets de propietat intel·lectual **únicament per a usos privats** emmarcats en activitats d'investigació i docència. No s'autoritza la seva reproducció amb finalitats de lucre ni la seva difusió i posada a disposició des d'un lloc aliè al servei UPCommons o TDX. No s'autoritza la presentació del seu contingut en una finestra o marc aliè a UPCommons (*framing*). Aquesta reserva de drets afecta tant al resum de presentació de la tesi com als seus continguts. En la utilització o cita de parts de la tesi és obligat indicar el nom de la persona autora.

ADVERTENCIA La consulta de esta tesis queda condicionada a la aceptación de las siguientes condiciones de uso: La difusión de esta tesis por medio del repositorio institucional UPCommons (http://upcommons.upc.edu/tesis) y el repositorio cooperativo TDR (http://www.tdx.cat/?locale-attribute=es) ha sido autorizada por los titulares de los derechos de propiedad intelectual **únicamente para usos privados enmarcados** en actividades de investigación y docencia. No se autoriza su reproducción con finalidades de lucro ni su difusión y puesta a disposición desde un sitio ajeno al servicio UPCommons No se autoriza la presentación de su contenido en una ventana o marco ajeno a UPCommons (*framing*). Esta reserva de derechos afecta tanto al resumen de presentación de la tesis como a sus contenidos. En la utilización o cita de partes de la tesis es obligado indicar el nombre de la persona autora.

WARNING On having consulted this thesis you're accepting the following use conditions: Spreading this thesis by the institutional repository UPCommons (http://upcommons.upc.edu/tesis) and the cooperative repository TDX (http://www.tdx.cat/?locale- attribute=en) has been authorized by the titular of the intellectual property rights **only for private uses** placed in investigation and teaching activities. Reproduction with lucrative aims is not authorized neither its spreading nor availability from a site foreign to the UPCommons service. Introducing its content in a window or frame foreign to the UPCommons service is not authorized (framing). These rights affect to the presentation summary of the thesis as well as to its contents. In the using or citation of parts of the thesis it's obliged to indicate the name of the author.

Low-Power Accelerators for Cognitive Computing

Marc Riera Villanueva

Doctor of Philosophy

Department of Computer Architecture
Universitat Politècnica de Catalunya

Advisors: Jose-Maria Arnau, Antonio González

June, 2020

Barcelona, Spain

Abstract

Deep Neural Networks (DNNs) have achieved tremendous success for cognitive applications, and are specially efficient in classification and decision making problems such as speech recognition or machine translation. Mobile and embedded devices increasingly rely on DNNs to understand the world. Smartphones, smartwatches and cars perform discriminative tasks, such as face or object recognition, on a daily basis. Despite the increasing popularity of DNNs, running them on mobile and embedded systems comes with several main challenges: delivering high accuracy and performance with a small memory and energy budget. Modern DNN models consist of billions of parameters requiring huge computational and memory resources and, hence, they cannot be directly deployed on low-power systems with limited resources. The objective of this thesis is to address these issues and propose novel solutions in order to design highly efficient custom accelerators for DNN-based cognitive computing systems.

As a first step, we characterize popular DNNs for different applications on General-Purpose-Architectures (GPAs) such as CPU and GPGPU. DNNs are composed of multiple layers of neurons whose core operation is the dot product between the inputs of each neuron and its weights. The weights of a DNN model are learned during the training procedure and used at inference. We observe that the main performance and energy bottleneck of GPA systems is the memory hierarchy due to the huge number of inputs, weights, and partial products, resulting in numerous data movements. Thus, we develop a baseline architecture for accelerating DNN inference based on a state-of-the-art DNN accelerator. After identifying the memory subsystem as the main bottleneck, we introduce an on-chip eDRAM memory to store the weights, and an SRAM buffer to store inputs and temporal partial results. The on-chip memories favor the reuse of data among different executions of the DNN, reducing the accesses to main memory, and improving the performance and energy efficiency of the accelerator.

Efficient support for DNNs on CPUs, GPUs and accelerators has become a prolific area of research, resulting in a plethora of techniques for energy-efficient DNN inference. However, previous proposals focus on a single execution of a DNN. Popular sequence processing applications, such as speech recognition or video classification, require many back-to-back executions of the same DNN to process a sequence of inputs (e.g., audio frames, images). In the first contribution of this thesis, we show that consecutive inputs exhibit a high degree of similarity, causing the inputs/outputs of the different layers to be extremely similar for successive frames of speech or images of a video.

Based on the high degree of input similarity observed, we propose DISC, a hardware accelerator implementing a Differential Input Similarity Computation technique to reuse some results of the previous execution, instead of computing the entire DNN. Computations related to inputs with negligible changes can be avoided with minor impact on accuracy, saving a large percentage of computations and memory accesses. Our results show that, on average, more than 60% of the inputs of any neural network layer tested exhibit negligible changes with respect to the previous execution. Avoiding the memory accesses and computations for these inputs results in 63% energy savings on average.

On the other hand, DNN quantization is a common optimization that reduces the precision of the inputs and weights, saving storage and simplifying the computations. Prior works exploit

the weight/input repetition that arises due to quantization to avoid redundant computations in Convolutional Neural Networks (CNNs). However, in the second contribution of this thesis we show that their effectiveness is severely limited when applied to Fully-Connected (FC) layers, which are commonly used in state-of-the-art DNNs, as it is the case of modern Recurrent Neural Networks (RNNs) and Transformers.

To improve energy-efficiency of FC computation we present CREW, a hardware accelerator that implements Computation Reuse and an Efficient Weight Storage mechanism to exploit the large number of repeated weights in FC layers. CREW first performs the multiplications of the unique weights by their respective inputs and stores the results in an on-chip buffer. The storage requirements are modest due to the small number of unique weights and the relatively small size of the input compared to convolutional layers. Next, CREW computes each output by fetching and adding its required products. To this end, each weight is replaced offline by an index in the buffer of unique products. Indices are typically smaller than the quantized weights, since the number of unique weights for each input tends to be much lower than the range of quantized weights, which reduces storage and memory bandwidth requirements. Overall, CREW greatly reduces the number of multiplications and provides significant savings in model memory footprint and memory bandwidth usage. We evaluate CREW on a diverse set of modern DNNs. On average, CREW provides $2.61x$ speedup and $2.42x$ energy savings over a TPU-like accelerator. Compared to UCNN, a state-of-art computation reuse technique, CREW achieves $2.10x$ speedup and $2.08x$ energy savings on average.

Activation pruning has been previously proposed in Convolutional Neural Networks (CNNs) and Multi-Layer Perceptrons (MLPs) to avoid computations by dynamically removing connections whose input value is (close to) zero. However, fine-grained activation pruning results in sparse computations, reducing the efficiency of the hardware implementation, and heavily relies on ReLU activation functions that are unpopular in Recurrent Neural Networks (RNNs). RNN cells perform element-wise multiplications across the activations of different single-layer fully-connected networks, a.k.a. gates, sigmoid and tanh being the common activation functions. We observe that a significant percentage of activations are saturated towards either zero or one in different gates, which results in a large percentage of neuron outputs being multiplied by a value close to zero. We propose Coarse-Grained Pruning of Activations (CGPA) to avoid the computation of entire neurons, rather than pruning individual connections, based on the activation values of the gates. CGPA results in much less sparse computation and memory access patterns than previous proposals and, hence, it can be easily implemented on top of conventional accelerators such as TPU with negligible area overhead, resulting in 12% speedup and 12% energy savings on average for a set of widely used RNNs.

In spite of significantly reducing the size of the models and the number of computations for several DNNs, by using the proposed computation reuse schemes and optimizations, the memory hierarchy is still the main bottleneck of our accelerators. In the last contribution of this thesis, we explore different pruning methods. DNN pruning reduces memory footprint and computational work of DNN-based solutions to improve performance and energy-efficiency. An effective pruning scheme should be able to systematically remove connections and/or neurons that are unnecessary or redundant, reducing the DNN size without any loss in accuracy. We show that prior pruning schemes require an extremely time-consuming iterative process that requires retraining the DNN

many times to tune the pruning hyperparameters. Then, we propose a DNN pruning scheme based on Principal Component Analysis and relative importance of each neuron's connection that automatically finds the optimized DNN in one shot without requiring hand-tuning of multiple parameters.

Keywords

Machine Learning, Deep Neural Network (DNN), Hardware Accelerator, Low-Power Architecture, Computation Reuse, Input Similarity, Weight Repetition, Quantization, Pruning.

Acknowledgement

Firstly, I would like to express my sincere gratitude to my two advisors, Jose-Maria Arnau and Antonio González, for their continuous support of my PhD study and related research, for their patience, motivation, and immense knowledge. I am very grateful to Prof. Antonio González for offering me the opportunity to start a research career at ARCO in UPC and supporting me during these years. I am always indebted to Prof. Jose-Maria for all that he taught me, the daily meetings and helpful discussions that I had with him, and also for giving me the fruitful ideas to help my work gets mature enough. The guidance of my two advisors helped me in all the time of research and writing of this thesis and the several publications. I could not have imagined having better advisors and mentors for my PhD study. It was, is and will be a great pleasure to work with them.

Besides my advisors, I would like to thank the rest of my thesis committee: Andreas Moshovos, David Brooks, and Jordi Tubella, for their valuable feedback and comments, but also for their questions which incented me to widen my research from various perspectives. I also thank the staff in the Department of Computer Architecture (DAC), human resources and other units of UPC for their kind help and support during these years.

I would also like to thank the various institutions that have granted me with the financial support to finish this thesis. This work has been supported by the the CoCoUnit ERC Advanced Grant of the EU's Horizon 2020 program (grant No 833057), the Spanish State Research Agency under grant TIN2016-75344-R (AEI/FEDER, EU), and the Spanish Ministry of Education under grant FPU15/02294.

I am also very thankful to all the members of ARCO, and specially to my fellow labmates ("The D6ers"). I was lucky enough to be part of a large, well-known and enriching research group. Special thanks to Reza Yazdani with whom I have collaborated, Hamid Tabani, Martí Anglada, Martí Torrents, Gem Dot, Enrique de Lucas, Franyell Silfa, Albert Segura, Dennis Pinto, Diya Joseph, Raúl Taranco, Pedro Exenberger, Jorge Sierra, and Josue Quiroga, for the stimulating discussions, and for all the good moments that we had at UPC in the last years.

Finally, last but by no means least, I would like to thank all my family, and especially my parents and my little brother for their endless love, continuous encouragement and support. I owe my deepest gratitude to my family and I would like to dedicate this thesis to them. I have been extremely fortunate in my life to have parents who have shown me unconditional love and support. The relationships and bonds that I have with my parents hold an enormous amount of meaning to me. I admire them for all of their accomplishments in life, for their independence and for all of the knowledge and wisdom that they have passed on to me over the years. My parents have played a key role in the development of my identity and shaping the individual that I am today.

This thesis is dedicated to my family
for their endless love, support and encouragement.

Contents

1 **Introduction** **19**
1.1 Motivation 19
1.2 Problem Statement, Objectives and Contributions 22
1.2.1 Temporal Reuse 23
1.2.2 Spatial Reuse 25
1.2.3 Dynamic Pruning 27
1.2.4 Static Pruning 29
1.3 Thesis Organization 31

2 **Background and Related Work** **33**
2.1 Deep Neural Networks 33
2.1.1 Fully-Connected Layers 36
2.1.2 Convolutional Layers 36
2.1.3 Recurrent Layers 37
2.2 DNN Acccelerators 38
2.2.1 DaDianNao 39
2.2.2 TPU 41
2.3 DNN Optimizations 41
2.3.1 Linear Quantization 43
2.3.2 Pruning and Sparse Accelerators 43
2.4 Main DNN Pruning Schemes 45

2.4.1 Near Zero Weights Pruning 45
2.4.2 Node Pruning 46
2.4.3 Similarity Pruning 47
2.4.4 Scalpel 47
2.4.5 PCA Pruning 48
2.5 Computation Reuse 49

3 Experimental Methodology 51
3.1 Hardware Acceleration Modeling and Evaluation 51
3.2 DNN Software Implementations 53
3.3 DNN Models and Datasets 54

4 DISC: Differential Input Similarity Computation 57
4.1 Input Similarity and Temporal Reuse Analysis 57
4.2 DISC Accelerator 60
4.2.1 DISC Architecture 61
4.2.2 DISC for FC Layers 61
4.2.3 DISC for CONV Layers 63
4.2.4 DISC for Recurrent Layers 64
4.3 Evaluation Methodology 64
4.4 Experimental Results 65
4.4.1 Reduced-precision Accelerator 69

5 CREW: Computation Reuse & Efficient Weight Storage 71
5.1 Unique Weights in FC Layers 71
5.2 Partial Product Reuse 73
5.2.1 Partial Product Memoization 73
5.2.2 Partial Product Approximation 75
5.3 CREW Accelerator 78

5.3.1 Architecture . . . 78
5.3.2 Dataflow . . . 79
5.3.3 Design Flexibility . . . 81
5.4 Evaluation Methodology . . . 81
5.5 Experimental Results . . . 83
5.5.1 CREW Evaluation . . . 83
5.5.2 CREW-PPA Evaluation . . . 84
5.5.3 Overheads . . . 86

6 CGPA: Coarse-Grained Pruning of Activations 87
6.1 Analysis of RNN Activations . . . 87
6.2 CGPA Accelerator . . . 92
6.2.1 CGPA Architecture . . . 92
6.2.2 CGPA for GRU Layers . . . 92
6.2.3 CGPA for LSTM Layers . . . 93
6.2.4 Design Flexibility . . . 93
6.3 Evaluation Methodology . . . 94
6.4 Experimental Results . . . 95

7 PCA+UC: The (Pen-) Ultimate DNN Pruning 97
7.1 Weaknesses of Previous Static Pruning Schemes . . . 97
7.2 PCA+UC Pruning Method . . . 101
7.2.1 Node Pruning Through PCA . . . 101
7.2.2 Pruning Unimportant Connections . . . 103
7.3 Evaluation Methodology . . . 104
7.4 Experimental Results . . . 105
7.4.1 Sensitivity Analysis . . . 105
7.4.2 PCA+UC Evaluation . . . 106

8 Conclusions and Future Work **109**

8.1 Conclusions 109

8.2 Contributions 111

8.3 Open-Research Areas 112

List of Figures

1.1 Diagram of a Virtual Personal Assistant (VPA) Pipeline [29]. 20

1.2 Average speedup and energy reduction of several DNNs running on an accelerator based on DaDianNao [14]. Baseline CPU and GPU are the Intel i7 7700K and NVIDIA Geforce GTX 1080 respectively. 21

1.3 In speech recognition, the audio signal is split in frames of 10 ms. The DNN is executed multiple times to classify the frames in phonemes. Each DNN execution takes as input a sliding window of several frames. 24

1.4 Cumulative distribution of the unique weights per input neuron from all the FC layers of DeepSpeech2 [4], GNMT [97], Transformer [93], Kaldi [75] and PTBLM [103]. 26

1.5 Standard (a) vs CREW (b) DNN inference for a fully-connected (FC) layer. For each input, CREW only performs the multiplications with unique weights, avoiding redundant computations and memory fetches. Furthermore, it replaces weights by indexes in the buffer of unique multiplications. In this example there are only two unique weights per input, so the 8-bit quantized weights can be replaced by 1-bit indexes. 27

1.6 Fine-grained (a) vs coarse-grained (b) activation pruning. Connections/neurons shown with red dashed lines are dynamically pruned. 29

2.1 Example of DNN training vs DNN inference. 34

2.2 Example of a DNN from the (a) MLP category composed of (b) FC layers. 35

2.3 Convolutional 2D layer. 36

2.4 (a) Bidirectional LSTM layer, architecture of an LSTM cell, and (b) LSTM cell Computations. $\odot$, ϕ and σ are element-wise multiplication, hyperbolic tangent and sigmoid function respectively. 37

2.5 (a) Architecture of a GRU cell, and (b) Computations performed in a GRU cell. $\odot$, ϕ and σ are element-wise multiplication, hyperbolic tangent and sigmoid function respectively. 38

2.6 Architecture of the baseline DaDianNao-based accelerator showing the (a) tile organization, and the main hardware components of the (b) Processing Unit (PU) and the (c) Compute Engine (CE). 40

2.7 Architecture of (a) the baseline TPU-based accelerator and (b) a Processing Element (PE). 41

2.8 Example of static DNN Pruning. 44

3.1 The SoC architecture includes a CPU, GPU and a DNN accelerator. The CPU preprocesses the inputs of the DNN. Then, the DNN accelerator performs the inference of the DNN model. Finally, the CPU receives the results of the inference and executes a post-processing step to decode the final outputs. In addition, the GPU is used for DNN training. 52

3.2 Methodology flowchart to model and evaluate the DNN accelerators. The inputs to the simulation tools are marked in yellow, the tools and platforms in red, the partial results from the tools are marked in blue, and the final evaluation summaries in green. 52

4.1 Relative difference defined as the Euclidean distance between current and previous input vectors, divided by the magnitude of the input vector in the previous execution. 58

4.2 Input similarity and computation reuse for various DNNs. 60

4.3 FC execution in the DISC Accelerator. 62

4.4 Execution of a convolutional layer in the DISC accelerator. IB_b means Input Block from input feature map b, whereas OB_o means Output Block from output feature map o. 63

4.5 Speedups achieved by DISC for each DNN. Baseline configuration is the DaDianNao-based DNN accelerator without any computation reuse technique. 66

4.6 Normalized energy of DISC for each DNN. Baseline configuration is the DaDianNao-based DNN accelerator without any computation reuse technique. 66

4.7 Energy breakdown for the baseline DaDianNao-based DNN accelerator and the DISC accelerator using the computation reuse scheme. 67

4.8 Speedup and energy reduction of DISC compared to a high-end GPU. Baseline configuration is the Intel i7 7700K CPU. 68

5.1 Histograms of the unique weights per input neuron from all the FC layers of DS2, GNMT, Transformer, Kaldi and PTBLM. 72

5.2 Example of partial product memoization (a) vs partial product approximation (b) using a small FC layer. 74

5.3 Usage frequency histograms of the unique weights per input neuron for all the FC layers of DS2, GNMT, Transformer, Kaldi and PTBLM. The frequency of use is the number of times each unique weight is repeated divided by the total number of weights for each input neuron. . . . 76

5.4 Accuracy loss versus compression ratio over partial product memoization (CREW without approximation) for different thresholds of the heuristic for unique weight approximation. Thresholds tested go up to 20% in steps of 5%, where the initial 0% threshold means no weight approximation. . . . 77

5.5 Architecture of (a) the CREW accelerator and (b) a Processing Element (PE). The PE components shaded in gray represent the required extra hardware for CREW. The partial product buffer is shared among all or a subset of PEs of the same row of the array. . . . 78

5.6 CREW Execution Dataflow. PE_{row} is the number of PEs per row in the systolic array, and BS_{row} and BS_{col} determine the block size, i.e. the number of indexes per block. BS_{row} refers to the number of indexes relative to different input neurons, while BS_{col} refers to indexes associated to the same input neuron but used in the computation of different output neurons. . . . 79

5.7 FC Execution in the CREW Accelerator. . . . 80

5.8 Speedups achieved by CREW and UCNN for each DNN. Baseline configuration is the TPU-like DNN accelerator without any computation reuse mechanism. . . . 84

5.9 Normalized energy savings for each DNN. Baseline configuration is the TPU-like DNN accelerator without the computation reuse technique. . . . 84

5.10 Speedup of partial product approximation with less than 1% accuracy loss. Baseline configuration is the CREW accelerator. . . . 85

5.11 Normalized energy of partial product approximation with less than 1% accuracy loss. Baseline configuration is the CREW accelerator. . . . 85

5.12 Area overhead of UCNN and CREW. Baseline configuration is the TPU-like DNN accelerator without any computation reuse technique. . . . 86

6.1 Computations performed in (a) LSTM and (b) GRU cells. $\odot$, ϕ and σ are element-wise multiplication, hyperbolic tangent and sigmoid function respectively. . . . 88

6.2 Histogram of activations of the Update Gate (Z) for DS2-L and DS2-T. . . . 89

6.3 Input Gate (I) (a), Generate Gate (G) and Cell State (C) (b) histogram of activations for the GNMT and PTBLM RNNs. . . . 90

6.4 Computation savings for each RNN. . . . 91

6.5 Speedups achieved for each RNN. Baseline configuration is the TPU-like accelerator without CGPA. 96

6.6 Normalized energy for each RNN. Baseline configuration is the TPU-like accelerator without CGPA 96

7.1 Comparison between near-zero and random pruning of the Kaldi DNN for different percentages of global pruning. 99

7.2 Comparison between i-norm, similarity and random pruning of the Kaldi DNN for different percentages of global pruning. 99

7.3 Comparison between near-zero and random pruning of LeNet5 for different percentages of global pruning. 100

7.4 Comparison between i-norm, similarity and random pruning of LeNet5 for different percentages of global pruning. 100

7.5 Comparison between near-zero and random pruning of AlexNet for different percentages of global pruning. 101

7.6 Cumulative variance of a sample layer of AlexNet, Kaldi and LeNet5. 103

7.7 Example of the weights distribution translation pruning problem. 104

List of Tables

3.1 Parameters of the CPU and GPU employed to evaluate the performance and energy consumption of the DNN software implementations. 54

3.2 DNNs used in the experimental evaluation of the different proposals of this thesis. . 54

4.1 Deep Neural Networks employed for the analysis of computation reuse. The table only includes Fully-Connected (FC), Convolutional (CONV) and Bidirectional LSTM (BiLSTM) layers, as these layers take up the bulk of computations in DNNs. Other layers, such as ReLU or Pooling, are not shown in the table for the sake of simplicity. 58

4.2 Parameters for the DISC accelerator. 65

4.3 Memory overheads of the DISC accelerator implementing the computation reuse scheme. 68

5.1 **UW/I** shows the average number of unique weights per input neuron. **MULs** is the percentage of multiplications of inputs by unique weights with respect to the total number of multiplications in the original model. 73

5.2 Reduction in multiplications and storage. 75

5.3 Parameters for the accelerators. 82

5.4 DNNs employed for the experimental evaluation of CREW. The model size accounts for the original FP parameters of the FC layers where CREW is applied. 83

6.1 RNNs employed for the evaluation. Only LSTM and GRU layers are included since these layers take up the bulk of computations in RNNs. 88

7.1 DNNs employed for the pruning study. Kaldi is an MLP for acoustic scoring, AlexNet is a CNN for image classification and LeNet5 is a CNN for digit classification. The table only includes Fully-Connected (FC) and Convolutional (CONV) layers, as these layers take up the bulk of computations in DNNs. Other layers, such as ReLU or Pooling, are not shown for the sake of simplicity. 98

7.2 Kaldi results after the PCA step (first step) of the proposed pruning scheme using different coefficients of variance. (Baseline WER=10.04%) . . . 106

7.3 Kaldi results after the unimportant connections pruning step (second step) of the proposed pruning scheme for different thresholds of the mean. (Baseline WER=10.04%) 106

7.4 Accuracy (WER) and percentage of weights and computations removed by different pruning schemes for the Kaldi DNN. . . . 107

7.5 Accuracy (Top-1) and percentage of weights and computations removed by different pruning schemes for LeNet5. . . . 108

7.6 Accuracy (Top-1) and percentage of weights and computations removed by different pruning schemes for AlexNet. . . . 108

7.7 Link Pruning Comparison for the Kaldi DNN. (Baseline WER=10.04%) . . . 108

1 Introduction

This chapter describes the motivation and objectives of this work, explaining the challenges for deploying cognitive computing applications in mobile and embedded devices, and presents a summary of the main proposals and contributions of this thesis in order to perform efficient DNN inference.

1.1 Motivation

Deep learning has transformed how mobile and embedded devices interpret and respond to many different types of information/data, as well as the way in which we interact with our devices. The interaction with smart devices in our homes, offices, cars, and pockets is changing and evolving by leaps and bounds. The traditional keyboard and mouse, and even the currently popular touch-screens are being left behind in favor of more intuitive and yet sophisticated interfaces based on cognitive applications such as image, text and speech recognition. Commercial examples of such applications include Virtual Personal Assistants (VPA) [36, 58, 45] like Google's Assistant, Apple's Siri, Microsoft's Cortana and Amazon's Alexa. Figure 1.1 shows an example of such an application. VPAs are meant to interact with an end user in a natural way (i.e. voice, text or images), to answer questions, follow a conversation and accomplish different tasks. These mobile applications are used on a daily basis by millions of people and are powered by efficient deep learning models.

Nowadays, deep learning, and machine learning in general, has expanded its use cases and can be found not only in most of our daily life devices but also in our cities and environment despite not being fully aware of it. Examples range from simple classification and perceptual tasks to control algorithms for autonomous systems [9] (ranging from home robots, to drones and cars) or medical health care systems [65] (e.g. medical diagnose or drug discovery and development). Moreover, we

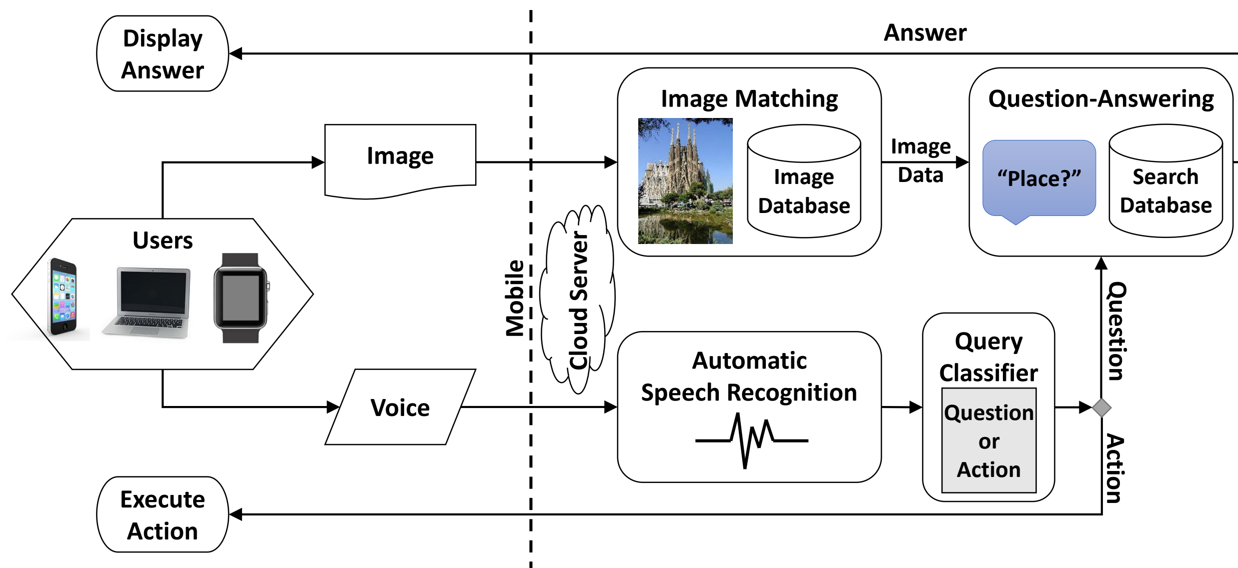


Figure 1.1: Diagram of a Virtual Personal Assistant (VPA) Pipeline [29].

are witnessing how even mature system components and algorithms can have their performance and functionality improved by deep learning techniques. For example, computer architecture researchers are exploring the integration of neural networks for branch prediction [42], and also in computer system areas such as network protocols, data compression or encryption.

On the other hand, substantial amount of computation power is necessary for enabling cognitive computing applications, which usually comes at the expense of high energy consumption. These requirements make solutions for smart devices to rely on the internet to offload most computations to servers in the cloud. For instance, Apple uses a small local neural network to detect keywords such as "Hey Siri" [88], whereas the bulk of the speech recognition is done in the cloud. Although this approach is functional, the end-user experience is negatively impacted by the limited response time in situations of slow internet connection and the impossibility of interacting with the device if the user is in a place with no internet coverage. Moreover, there is a large number of applications which cannot depend on an internet connection, either because requiring more reliability (e.g. autonomous systems), or due to security and privacy concerns (e.g. health care systems), making it undesirable to entrust our data to a remote server.

Furthermore, users of smart devices are increasingly demanding greater functionality of the applications they use while requiring fast and highly accurate responses. In consequence, leading companies such as Google are actively improving both their software and hardware to efficiently support on-device deep learning applications. In recent years, there is a growing interest in machine learning techniques such as neural networks, and computer systems that are specialized for cognitive applications. Accelerators offer better performance and energy efficiency than general purpose systems, which makes them especially attractive in portable devices such as smartphones and smartwatches, for which battery lifetime has a direct impact on the market share.

Deep Neural Networks (DNN) [82] represent a machine learning approach that delivers the most effective solution to a broad range of applications such as speech and image recognition [75, 48],

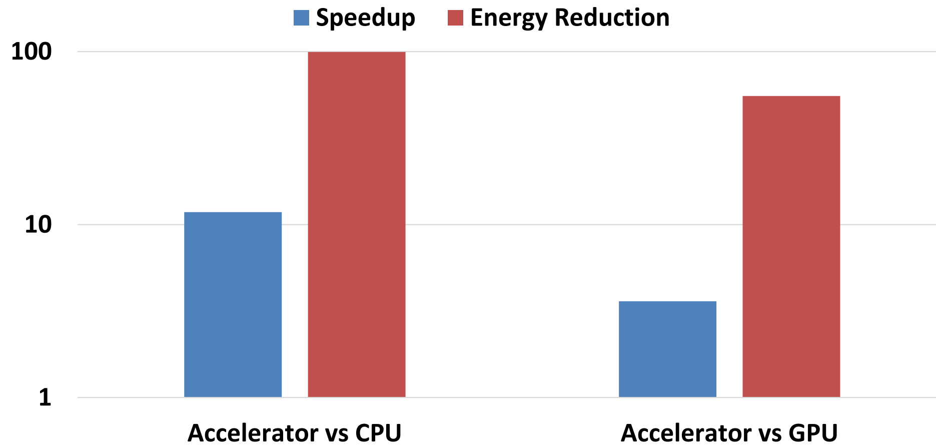


Figure 1.2: Average speedup and energy reduction of several DNNs running on an accelerator based on DaDianNao [14]. Baseline CPU and GPU are the Intel i7 7700K and NVIDIA Geforce GTX 1080 respectively.

language modeling [103] or machine translation [97]. Similarly to the human brain, artificial neural networks [56] are formed by different nodes (neurons) and connections (synapses). DNNs are composed of hundreds of layers of interconnected nodes according to the degree of relevance among them. A single execution of a DNN can demand the evaluation of hundreds of millions of model parameters (i.e. synaptic weights). Through a process of learning, these networks are able to adapt the weights in the synapses to accurately infer the sequences of phonemes of an audio signal or the objects in an image. DNN-based systems are ubiquitous and, thus, providing energy-efficient inference of DNNs is a key feature for current and future computing devices.

Accelerators for cognitive computing can deliver accurate and robust DNNs in smart devices. DNN accelerators [14, 43, 25] consist mainly of arithmetic and logical units, a memory system and control units that are integrated into the chip to perform cognitive computing much more efficiently than CPUs and GPUs. We have evaluated several DNNs on a CPU, a GPU and a baseline DNN accelerator. Figure 1.2 shows the average performance speedup and energy savings achieved by a DaDianNao-based DNN accelerator. As can be seen, the accelerator performs faster than both CPU ($12x$) and GPU ($4x$), while consuming much less energy (i.e. $100x$ and $55x$ compared to the CPU and GPU respectively). The memory system is a key component due to the huge storage capacity required to completely store the parameters of the neural networks, and the high bandwidth requirements.

Moreover, since the main bottleneck in cognitive accelerators is still the memory system, alternative architectural mechanisms for saving bandwidth to the off-chip memory, such as data compression, computation reuse and removal of ineffectual nodes, are effective ways to improve performance and reduce the energy consumption of these accelerators, while maintaining the accuracy of the DNN models. The efficiency of these mechanisms lies in the characteristics of cognitive algorithms, such as the level of parallelism, data reuse and redundancy, or the deterministic memory access patterns.

In summary, we focus on improving the performance and energy efficiency of DNN inference

for mobile and embedded devices. Deep learning techniques, and in particular DNNs, are already an essential component of cognitive computing applications and will have an even more pivotal role in the evolution of smart devices. Therefore, the design of efficient cognitive accelerators is fundamental. In this thesis, we propose novel accelerator architectures and optimizations based on the characteristics of cognitive applications with special emphasis on low-power designs.

1.2 Problem Statement, Objectives and Contributions

Recent advances in deep learning techniques have achieved human parity in cognitive applications such as speech recognition or machine translation [98, 97]. In order to achieve levels of accuracy similar to those of humans, DNNs have evolved and grown in complexity. From small Multilayer Perceptrons (MLPs) for simple tasks like recognizing written digits or characters, passing by large Convolutional Neural Networks (CNNs) for recognizing objects in images, to complex Recurrent Neural Networks (RNNs) and Transformers to reach the aforementioned human parity in speech recognition and machine translation. Each type of DNN and application has its own characteristics and challenges to be effectively deployed on mobile and embedded devices. In consequence, one of the objectives of this thesis is the study, characterization and analysis of the main cognitive algorithms and applications.

In parallel with the progress in accuracy, DNNs have become much more complex, increasing their size both in number of layers (i.e. deeper), and in amplitude (i.e. number of parameters per layer). Modern DNN models consist of hundreds of layers and billions of parameters with their respective computations and memory accesses. Therefore, the inference of DNNs requires huge computational and memory resources to be efficiently executed. Even popular commercial DNN accelerators, such as DaDianNao [14] and TPU [43], which have proven to be efficient hardware implementations for DNN inference, struggle to effectively run the latest models. Although state-of-the-art DNN accelerators can achieve high levels of parallelism and perform much faster than general purpose architectures, they still have to perform too many computations and memory accesses to be suitable for low-power devices. On the other hand, DNNs are known to be very fault tolerant [14]. Due to the difficulty to create and train DNNs from scratch, DNN models are usually oversized and include a high degree of redundancy. A popular line of research is to exploit this fault tolerance to statically and/or dynamically change the precision of the operations, prune redundant neurons and/or connections, or reuse previous computations. Although many works have been recently proposed in this regard, they focus on optimizing specific layers, DNNs or applications, still leaving plenty of room and potential for improvement. Another objective of this thesis is to propose different techniques to exploit the DNNs redundancy in order to reduce the number of computations and memory accesses, improving the energy efficiency of the DNN accelerators.

We devote Chapter 2 for presenting a detailed background on the most relevant DNN algorithms for a variety of applications. In addition, we describe common optimizations such as linear quantization to lessen the hardware requirements. Furthermore, we provide a summary of related work on DNN accelerators, pruning methods and computation reuse techniques.

The main objective of this thesis is to propose novel hardware architecture designs to improve the performance and energy efficiency of cognitive accelerators for deep learning applications. To

this end, we have evaluated the most important cognitive algorithms for a diverse set of applications on different platforms including CPUs, GPUs, and state-of-the-art DNN accelerators. After discovering that the main bottleneck of all these systems is the memory hierarchy, we propose several optimizations and architectural mechanisms to relax the huge amount of computations, memory storage and memory bandwidth that are required to perform DNN inference.

First, we propose DISC, a novel computation reuse scheme with its corresponding hardware implementation to accelerate DNNs that processes temporal sequences of data such as voice or video. Despite sequence processing applications are very popular, little research has been done to exploit their unique characteristics. Second, we propose CREW, an accelerator implementing a spatial reuse mechanism to efficiently execute fully-connected (FC) layers. Modern MLPs, RNNs and Transformers are mainly composed of FC layers. CREW reduces the amount of computations and memory accesses required to perform inference of the latest FC-based DNN models, improving the performance and energy consumption with respect to state-of-the-art accelerators. Third, we propose CGPA, a technique to further accelerate the execution of RNNs by exploiting the dynamic pruning of activations at a coarse granularity. Finally, prior works have proposed methods to statically prune ineffectual neurons and connections, but we show that prior schemes are ineffective for modern DNNs, and propose a new mechanism to effectively apply pruning.

The following sections outline the problems we target, describe the approach we take to solve each problem and highlight the novel contributions of this thesis.

1.2.1 Temporal Reuse

The first technique proposed in this thesis aims to improve the execution of sequence processing applications. Sequence to sequence learning is a broad area with numerous and important applications such as speech recognition, machine translation, video description or language modeling. According to numbers published by Google [43], at least 29% of Google's datacenter workloads are sequence processing. Despite the relevance of these applications, hardware-accelerated DNN systems [14, 25, 3, 76] are commonly optimized for an isolated execution of the DNN. Our research is motivated by the observation that many computations and memory accesses are redundant if we take into account successive executions of a DNN, especially for applications that process a temporal sequence of inputs (e.g., speech, video).

Figure 1.3 illustrates the case of speech recognition, where a DNN is executed many times to classify a sequence of audio frames in phonemes. Consecutive DNN executions have extremely similar inputs due to two main reasons. First, the length of these frames is in the order of several milliseconds. The speech signal is quasi stationary for such a short interval and, therefore, consecutive frames exhibit a high degree of similarity. Second, the DNN uses context information (neighbor frames) to classify each audio frame and, hence, successive executions operate on overlapping windows of frames. Note that other applications such as video processing exhibit similar behavior, as consecutive images in a video tend to be very similar too.

Despite the high degree of similarity, floating point computations are not exactly the same in two successive DNN executions. Nevertheless, DNNs are known to be error tolerant [14]. We leverage this property to boost the potential of reuse across successive executions of the DNN. In

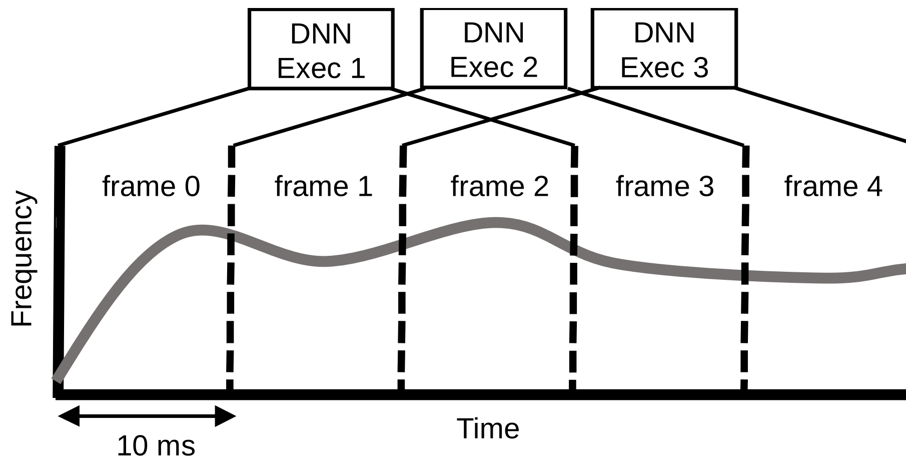


Figure 1.3: In speech recognition, the audio signal is split in frames of 10 ms. The DNN is executed multiple times to classify the frames in phonemes. Each DNN execution takes as input a sliding window of several frames.

particular, we found that linear quantization is a very effective mechanism to increase the ability of the technique to exploit redundancy. Linear quantization [76, 43] is a highly popular technique used to compress the weights of the model and reduce the complexity of the computations. Quantization maps a continuous set of values to a discrete set, favoring the appearance of repeated weights and/or inputs in consecutive executions of the DNN. We observed that by applying quantization to the inputs of various DNNs for speech recognition [107, 101, 87, 63], video classification [92] and self-driving cars [9], more than 50% of the inputs of fully-connected, convolutional and recurrent layers remain unmodified with respect to the previous DNN execution, whereas quantization has negligible impact in accuracy.

Based on the high degree of redundancy for the inputs of consecutive DNN executions, we propose a mechanism that computes the outputs of each DNN layer by reusing the buffered results of the previous execution. A simple example helps to illustrate this proposal. Let us consider a neuron of a fully-connected layer with three inputs. For the first execution of the DNN, the output z^1 is computed as follows: $z^1 = i_1^1 w_1 + i_2^1 w_2 + i_3^1 w_3 + b$, where i, w and b are the inputs, weights and bias respectively. In a similar way, the output of this neuron in the second DNN execution is $z^2 = i_1^2 w_1 + i_2^2 w_2 + i_3^2 w_3 + b$. However, if the first two inputs are the same in both executions, the output can be computed more efficiently as: $z^2 = z^1 + (i_3^2 - i_3^1) w_3$. In other words, we just need to subtract the old inputs that are different and add the new ones (multiplied by their respective weights). Note that in this case only one weight has to be fetched from memory instead of three, the bias is not required and only three computations are performed instead of six. Moreover, the subtraction of the two inputs can be reused for all the neurons in the same layer, so its cost is practically negligible and, in practice, for the above example the total cost approaches to just two operations. Therefore, for DNNs that exhibit some degree of input similarity across multiple executions, it is more efficient to compute the current output in this way rather than evaluating again the entire DNN from scratch. This mechanism reduces the number of memory accesses and computations by 66% on average.

In this work, we propose DISC, a hardware implementation of the Differential Input Similarity Computation reuse-based DNN inference mechanism. We extend the architecture of a state-of-the-art DNN accelerator to buffer the outputs of the different layers and reuse them for the next execution. The accelerator quantizes the inputs of each fully-connected, convolutional and recurrent layer, and then it computes the output by using only the inputs that have changed with respect to the previous execution, avoiding the corresponding memory accesses and computations for the inputs that remain unmodified. The extra hardware required for the technique is modest, as the components already available for DNN computation can also be employed for most of the operations, such as quantization. In addition to small changes in the control unit, DISC only requires extra storage for the outputs of each layer. The extra memory represents a small increase in the on-chip storage of the accelerator. Our experimental results show that the overheads are minimal compared to the savings in memory fetches and computations, so the proposed scheme improves performance by $3.5x$ and reduces energy consumption by 63% on average for several DNNs.

To summarize, the first work of this thesis focuses on energy-efficient, real-time DNN inference. A computation reuse scheme for any sequence processing neural network is proposed. Due to the small overheads, only a small degree of input similarity across DNN executions is required to achieve savings in computations and energy. Chapter 4 presents the analysis of input similarity of several DNNs, describes the hardware implementation of the reuse-based DNN inference technique, and discusses the experimental results of DISC. This work has been published in the 45th IEEE/ACM International Symposium on Computer Architecture [77].

1.2.2 Spatial Reuse

The second technique proposed in this thesis improves performance and energy-efficiency of the inference of Fully-Connected (FC)-based DNNs. The complexity of the DNN models continues to grow along with its computational cost and memory requirements, making it difficult to support them on conventional computer systems. Accelerators adopting a systolic array architecture such as TPU [43] from Google, have proven to be efficient hardware implementations to perform DNN inference [13, 72, 59]. The systolic array architecture [50, 51], which was designed for massive parallelization and data reuse, is especially effective for Convolutional Neural Networks (CNNs) since the weights of a layer are shared across a large number of sliding windows and can be reused multiple times. In addition, there has been a plethora of recent proposals to further optimize CNN inference [33, 34, 1]. However, CNNs are just a subset of DNNs with very specific characteristics. Therefore, techniques targeting CNNs do not necessarily achieve similar benefits for other DNN architectures. The second proposal is motivated by the fact that state-of-the-art models for sequence-to-sequence problems are either Recurrent Neural Networks (RNNs) [97, 4] or very deep Multi-Layer Perceptrons (MLPs) such as the Transformer [93], which are both composed of fully-connected (FC) layers. FC layers exhibit different characteristics with respect to CNNs: weights are not reused by different neurons and the compute to memory access ratio is significantly smaller, i.e., FC layers are more memory intensive.

FC layer computation mainly consists of dot products between a vector of inputs and a matrix of weights. The inputs are generated dynamically while the weights are static and learned during the training phase. Current DNN model sizes are in the hundreds of megabytes (MB) and require

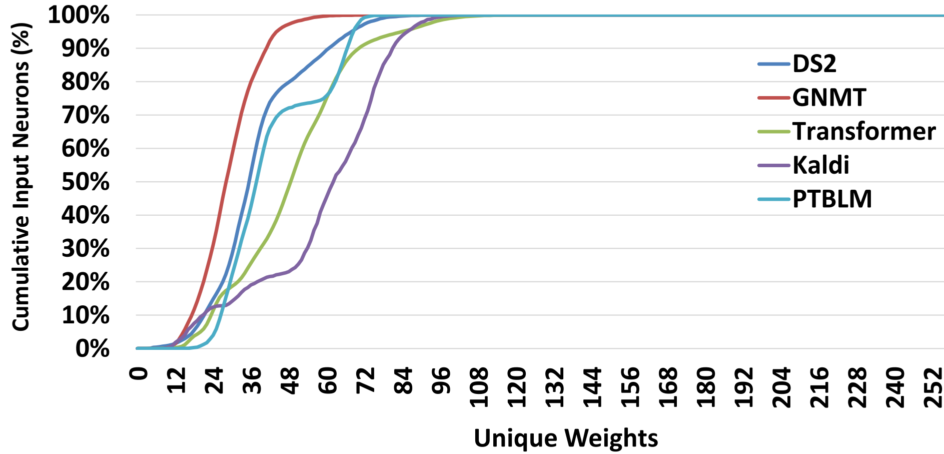
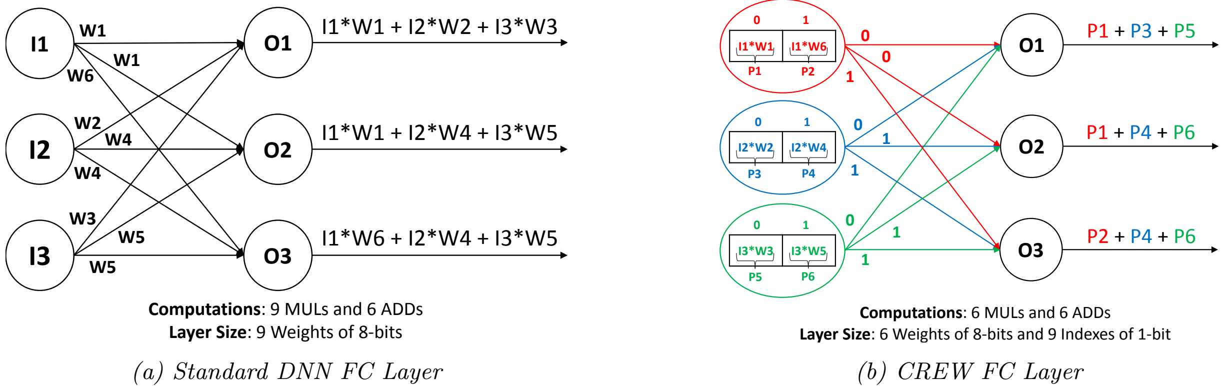


Figure 1.4: Cumulative distribution of the unique weights per input neuron from all the FC layers of DeepSpeech2 [4], GNMT [97], Transformer [93], Kaldi [75] and PTBLM [103].

billions of floating-point (FP) computations to perform inference. Linear quantization is normally applied to compress the weights of the model and reduce the complexity of the computations, which as a side effect favors the appearance of repeated weights. We observed that, on average, more than 80% of the inputs among different FC layers of a set of DNNs are multiplied by less than 64 unique weights. Figure 1.4 shows the cumulative distribution of unique weights per input of DeepSpeech2 (DS2) [4], GNMT [97], Transformer [93], Kaldi [75] and PTBLM [103], which are popular models for speech recognition, machine translation, and language modeling. The average number of unique weights per input is 44 when applying an 8-bit quantization, and can never be higher than 256.

In this work, we show how to efficiently exploit weight repetition on FC layers. We first propose a mechanism (CREW) that dynamically computes and stores the partial products of each input by their associated unique weights in a given layer. This first step removes redundant multiplications due to repeated weights and their associated memory reads. Similar to sparse architectures [25, 72, 106] and their pruning mechanisms [26, 102], the weight repetition patterns are irregular, making it very challenging to achieve net benefits due to the sparse accesses and extra metadata that are required. The partial products stored by this mechanism have to be indexed similar to sparse algorithms. Since the unique weights are known statically, the indexation table can be generated offline. The second step of this mechanism produces the final outputs of a FC layer by accessing the partial products with the associated indices and adding all the required values for each output. The main advantage of this mechanism is that the size of the indices depends on the number of unique weights of each input. As previously described, the number of unique weights is typically lower than 64 so indices will be typically smaller than 7 bits. In consequence, the final size of the model can be further compressed and the memory bandwidth is further reduced. Figure 1.5 shows an example of a small FC layer computing the standard dot product (Figure 1.5a) and the CREW mechanism (Figure 1.5b), saving 33% of multiplications and 20% of storage. On a representative set of modern DNNs, this technique reduces the number of multiplications and memory accesses by 98% and 40% respectively on average. Furthermore, it replaces weights by indices in the buffer of partial results, resulting in 25% reduction of the model size.

Then, we present CREW, a novel accelerator that implements the above computation reuse



(a) Standard DNN FC Layer

(b) CREW FC Layer

Figure 1.5: Standard (a) vs CREW (b) DNN inference for a fully-connected (FC) layer. For each input, CREW only performs the multiplications with unique weights, avoiding redundant computations and memory fetches. Furthermore, it replaces weights by indexes in the buffer of unique multiplications. In this example there are only two unique weights per input, so the 8-bit quantized weights can be replaced by 1-bit indexes.

and efficient weight storage scheme for FC layers. CREW is implemented on top of a TPU-like architecture, but it includes an enhanced weight stationary dataflow. The extra hardware required for the technique is modest since most of the components are already available in the baseline TPU-like architecture. CREW only requires small local buffers to store blocks of indices and some shared buffers to store partial results. The extra memory represents a small increase in the on-chip storage of the accelerator. Our experimental results show that the overheads are minimal compared to the savings in memory fetches and multiplications. CREW improves performance by $2.61x$ and reduces energy consumption by $2.42x$ on average over a TPU-like accelerator. Compared to UCNN [34], a recently proposed mechanism for computation reuse, CREW achieves $2.10x$ speedup and $2.08x$ energy savings on average.

To summarize, the second work of this thesis focuses on energy-efficient FC layer inference. Chapter 5 presents the analysis of unique weights per input of several DNNs, describes in detail the reuse scheme and hardware implementation of CREW, and discusses the experimental results. This work has been submitted for publication and is currently under review.

1.2.3 Dynamic Pruning

After proposing different computation reuse techniques, this thesis explores the pruning of input activations. Dynamic operation pruning [76], a.k.a. activation pruning [5], is an optimization technique to reduce the cost of evaluating MLPs and CNNs. Based on the observation that ReLU activations tend to generate a large number of zero values, activation pruning dynamically avoids computations when the synaptic weight is multiplied by a zero-value input. However, such a fine-grained activation pruning results in very sparse computations and memory accesses, since individual connections are selectively computed or skipped [2]. Sparse computations increase the complexity of a hardware accelerator, as it has to handle data representations based on indexes of non-zero values [72], which results in sparse memory access patterns that are challenging for the

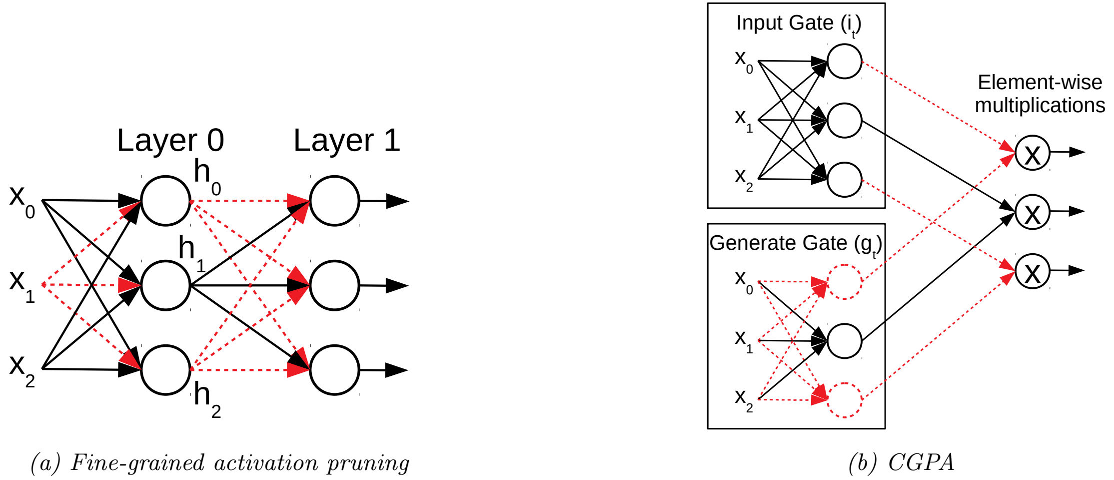
hardware. On the other hand, ReLU activations are rarely used in Recurrent Neural Networks (RNNs) [35], leading to smaller potential for fine-grained activation pruning.

RNNs represent the state-of-the-art solution for sequence processing problems such as machine translation [97], speech recognition [4] or language modeling [103]. Unlike CNNs or MLPs, RNN units store information from past executions to improve the accuracy of future predictions. Deep RNNs consist of multiple RNN layers, a.k.a. cells, stacked on top of each other. The most successful RNN cell architectures are the Long-Short Term Memory (LSTM) [35] and the Gated Recurrent Unit (GRU) [15]. Chapter 2 shows the equations of LSTM and GRU cells. In both cases, each cell consists of multiple single-layer fully-connected networks commonly referred as gates. Furthermore, the evaluation includes element-wise multiplications across the outputs, i.e. activations, of different gates.

Note that RNN activation functions exhibit a very narrow range: sigmoid ranges from 0 to 1, whereas hyperbolic tangent (tanh) is constrained to the interval $[-1, 1]$. We observed that every time a neuron in the input gate (i_t) of an LSTM cell is saturated towards zero, evaluation of its peer neuron in the generate gate (g_t) can be safely avoided. In a similar manner, whenever the tanh activation of the cell state (c_t) is zero the corresponding neuron in the output gate (o_t) can be skipped. Our numbers show that more than 18% of the computations and memory accesses can be avoided by exploiting these zero-value activations in LSTMs. On the other hand, GRU cells exhibit similar behavior: when a neuron is saturated towards one in the update gate (z_t) of a GRU cell, computation of its peer neuron in the generate gate (g_t) can be skipped as the product $(1 - z_t) \times g_t$ is granted to be zero. Our results show that more than 7% of neuron evaluations can be skipped based on this observation.

In this work, we propose to exploit the aforementioned saturations of activation functions to avoid computation of entire neurons in LSTM and GRU networks. We call this method Coarse-Grained Pruning of Activations (CGPA). Figure 1.6 shows the key difference between fine-grained activation pruning and CGPA. Under fine-grained activation pruning (Figure 1.6a), individual connections of a fully-connected or convolutional layer are dynamically skipped if their input value is zero, leading to sparse memory accesses. In this example, we assume x_1, h_0 and h_2 are almost zero. In this case, neurons in layer 0 require accessing their respective first and third weights, whereas neurons in layer 1 only require accessing their respective second weight (pruned synapses are shown in red). Such fine-grained and sparse accesses are challenging for hardware. On the other hand, Figure 1.6b illustrates CGPA. In this example, neurons zero and two in the input gate of an LSTM cell are saturated towards zero. This means that neurons zero and two (peer neurons) in generate gate (g_t) can be safely skipped, avoiding all their computations and memory accesses.

Note that CGPA is orthogonal to previous proposals of fine-grained activation pruning and static weight pruning. CGPA is applied at the neuron granularity by exploiting the element-wise operations of RNNs of GRUs and LSTMs and the saturations of the activation functions of the gates. CGPA dynamically prunes entire neurons which generates a compacted pruned model that avoids sparsity, unlike fine-grained activation pruning and static weight pruning which are applied at the connection granularity. Previous works such as Scalpel [102] have shown that in order to exploit the benefits of sparse models in CPU/GPU the degree of fine-grained pruning has to be extremely high, which cannot always be achieved without accuracy loss. In addition, a specific accelerator design to exploit sparse models is required, which in turn increases the complexity



(a) Fine-grained activation pruning

(b) CGPA

Figure 1.6: Fine-grained (a) vs coarse-grained (b) activation pruning. Connections/neurons shown with red dashed lines are dynamically pruned.

of the hardware due to the overheads of managing the sparse data. Both fine-grained activation pruning and static weight pruning can provide high percentages of pruning, but they also require the accelerator to be able to manage sparse data. On the other hand, CGPA can achieve speedups and energy savings with lower pruning degrees, since CGPA requires minor changes to the hardware and can be used in any dense accelerator like TPU.

In this work, we implement CGPA on top of a TPU-like accelerator. We show that the technique has a very small hardware overhead, as it mainly requires a few additional comparators to detect saturation of activation functions and small changes to the control unit to selectively skip entire neurons. In case a neuron is evaluated, all its weights and inputs are fetched from memory and injected into the systolic array, whereas they are completely avoided in case the neuron can be safely skipped, reducing sparsity by a large extent compared to fine-grained activation pruning. CGPA provides, on average, 12% of speedup and 12% of energy savings.

To summarize, the third proposal of this thesis focuses on energy-efficient and high-performance RNN inference. We show that a significant fraction of activations are saturated towards zero or one in popular RNNs, and propose CGPA to avoid the evaluation of entire neurons whenever the outputs of peer neurons are saturated. CGPA significantly reduces the amount of computations and memory accesses while avoiding sparsity by a large extent. Chapter 6 presents the analysis of RNN activations, describes in detail the hardware implementation of CGPA, and discusses the experimental results. This work has been published in the IEEE Micro Journal [78].

1.2.4 Static Pruning

Although increasing the complexity of the hardware accelerators, static DNN pruning has attracted the attention of the research community in recent years [26, 25, 102, 72]. Based on the observation that DNN models tend to be oversized and include a high degree of redundancy, pruning aims at reducing the model size by removing unimportant connections and/or neurons. The

pruned model retains accuracy while requiring significantly less memory storage and computations, resulting in large performance improvements and energy savings.

Pruning requires the pruned model to be retrained; otherwise, the effectiveness of pruning is dramatically reduced. Finding the appropriate amount of pruning on each layer is a key factor that determines the efficiency of the scheme. If the pruning is too aggressive the DNN will not recover its accuracy after retraining, whereas if the pruning is too conservative, an opportunity to further optimize the DNN is lost. Previous DNN pruning schemes set the amount of pruning based on an expensive design space exploration or sensitivity analysis [26]. We argue that these approaches are impractical for very deep neural networks, as they require retraining the DNN a large number of times, and each one may take several hours or days even on a high-end GPU for each retraining. To exacerbate the problem, hyperparameters such as learning rate or weight decay have to be manually tuned [26, 102], further increasing the search space.

In this thesis, we present a novel DNN pruning scheme that does not require such an expensive search to find the percentage of pruning for each layer, and it does not require to tune any hyperparameter. We refer to it as PCA plus Unimportant Connections (PCA+UC) pruning. The PCA+UC scheme first applies node (i.e. neuron) pruning. We consider each layer as a system that produces “information” encoded by an N-dimensional array, where N is the number of neurons. The goal of pruning is to reduce the number of neurons to M (M < N) without loosing “information”. The optimal pruning should find the minimum M for each level.

Principal Component Analysis (PCA) is a well-known statistical procedure that transforms a set of N-dimensional variables to a new coordinate system in which all coordinates are orthogonal and ordered from highest to lowest variance. Since variance can be considered as a good proxy of amount of “information”, the PCA+UC scheme exploits PCA to determine the amount of neurons that can be safely removed. Once the percentage of pruning for a layer is set, the scheme has to select which neurons are removed. We have evaluated prior heuristics for node pruning [32, 85] and found that, if retraining is applied, they achieve the same results than a blind node pruning that randomly selects the nodes to be removed from the model. Our experimental results show that the only relevant parameter is the amount of pruning, i.e. percentage of nodes to be pruned, and not which specific nodes are actually removed, since the topology of the DNN is not affected by that decision and the retraining will adjust the weights of the non-pruned nodes.

Once we determine the minimum number of neurons for each layer, there are still further opportunities to prune at the connection level. The first step, i.e. node pruning, results in layers that are not sparse since full neurons with all their connections are removed or kept. However, for a given non-pruned neuron there may be connections that are unimportant. After the PCA-based node pruning, the PCA+UC scheme measures the relative contribution of each connection with respect to the other connections of the same neuron. Those connections with a low contribution are removed and the final network is retrained. Unlike the node pruning step where the heuristic to select neurons is irrelevant, our results show that the heuristic used to choose the connections to be pruned has a non-negligible impact, achieving up to 30% of additional pruning over randomly choosing the connections. The overall scheme is non-iterative: it consists of only two steps and requires a single retraining after each of these steps. We show that this scheme produces results similar to or better than previously proposed iterative approaches that require an expensive (unfeasible for large networks) search of the parameter space. PCA+UC provides on average 72% of pruning for

a variety of DNNs.

To summarize, the final contribution of this thesis focuses on DNN pruning methodologies. We highlight the weaknesses of current pruning methods and solve them by proposing a more effective and practical scheme. Chapter 7 presents an analysis of prior pruning methods, a discussion about their effectiveness and practicality, and a new pruning method based on PCA and relative connection's importance. Finally, we discuss the evaluation and experimental results. This work has been submitted for publication and is currently under review.

1.3 Thesis Organization

The remainder of this thesis is structured as follows:

Chapter 2 provides background information on state-of-the-art machine learning algorithms and Deep Neural Network accelerators including some common optimizations. We first describe the basics on different types of DNNs including Multilayer Perceptrons, Convolutional Neural Networks, Recurrent Neural Networks and Transformers. Next, we review the architecture of two of the most popular and representative DNN accelerators, DaDianNao and TPU. Finally, we outline prior works on computation reuse techniques and pruning methods.

Chapter 3 describes our experimental methodology. We present the different tools used in order to model the performance, energy consumption, and area of the DNN accelerators. In addition, we describe the various DNN frameworks, models and datasets that are used for the evaluation of the different techniques proposed in this thesis.

In Chapter 4, we improve the inference of DNNs for sequence processing applications. We first present the analysis of similarity in the inputs of a DNN. Then, we describe DISC, a hardware accelerator implementing the reuse-based DNN inference technique based on the high degree of input similarity. Finally, we discuss the experimental results of DISC, and compare them to a CPU, GPU, and a baseline DNN accelerator.

In Chapter 5, we further optimize the inference of DNNs composed of fully-connected (FC) layers. We first present the analysis of weight repetition in FC layers. Then, we describe the mechanism to reuse partial products by leveraging the small number of unique weights of each input neuron on each FC layer. Next, we characterize CREW, an accelerator implementing this reuse-based scheme of partial products as well as an enhanced dataflow to efficiently execute FC layers on a systolic array of processing elements (PEs). Finally, we discuss the evaluation and experimental results of CREW.

In Chapter 6, we first present an analysis of the values of the activation functions that are most commonly used in the gates of different RNN cells including Long Short-Term Memory (LSTM) and Gated Recurrent Unit (GRU). Next, based on the high number of activations that are close to either zero or one, we describe a mechanism named CGPA that exploits the saturation of the activations to reduce computations. Finally, we show the results of CGPA implemented on top of a TPU-like accelerator.

In Chapter 7, we explore state-of-art pruning methods and discuss their main weaknesses. Next, we present a new pruning mechanism based on PCA and relative importance of each neuron's connection. We end this chapter by presenting the experimental results of our pruning mechanism and a comparison against prior works.

Finally, Chapter 8 summarizes the main conclusions of the thesis, and outlines some open research areas for future contributions.

2

Background and Related Work

This chapter presents some background on state-of-the-art machine learning algorithms and cognitive accelerators. Due to the objectives of this thesis, we will focus on Deep Neural Networks (DNNs), which are the mainstream approach for a broad range of cognitive applications. DNNs are computationally and memory intensive, and consume a significant amount of energy. Therefore, custom architectures with optimizations such as pruning or computation reuse mechanisms can provide important benefits. We first describe the basics of different types of DNNs including Multilayer Perceptrons (MLPs), Convolutional Neural Networks (CNNs), Recurrent Neural Networks (RNNs) and Transformers. Next, we illustrate the architecture of two of the most popular and representative DNN accelerators, DaDianNao [14] and TPU [43], that we use as baselines for our experiments. In addition, we elaborate on general optimizations that are usually applied in DNN accelerators, such as linear quantization and pruning, to improve the performance and energy consumption of the baselines. Finally, we outline related works on pruning methods and computation reuse techniques.

2.1 Deep Neural Networks

Deep Neural Networks (DNNs) mimic brain-like functionality based on a simple artificial neuron performing a nonlinear function, a.k.a. activation function, of a weighted sum of the inputs. These artificial neurons are arranged into a number of hidden layers and an output layer, forming feed-forward networks, which means that the outputs of one layer become the inputs of the next layer in the sequence. The "deep" term of DNN comes from including a large number of layers, allowing more accurate models to be built by using additional layers to capture complex patterns and concepts. DNNs are typically used as classifiers, so the last layer of the network often performs a softmax function that generates a probability distribution over a list of potential classes. To

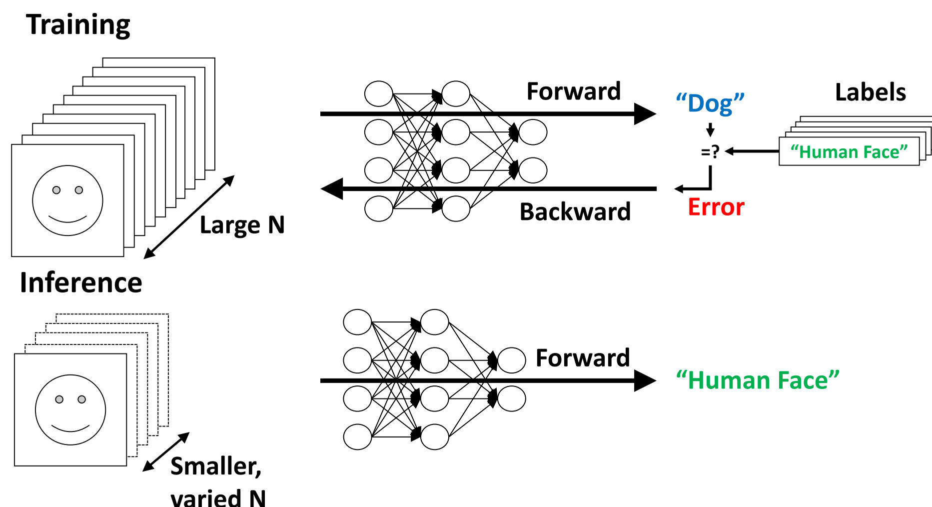
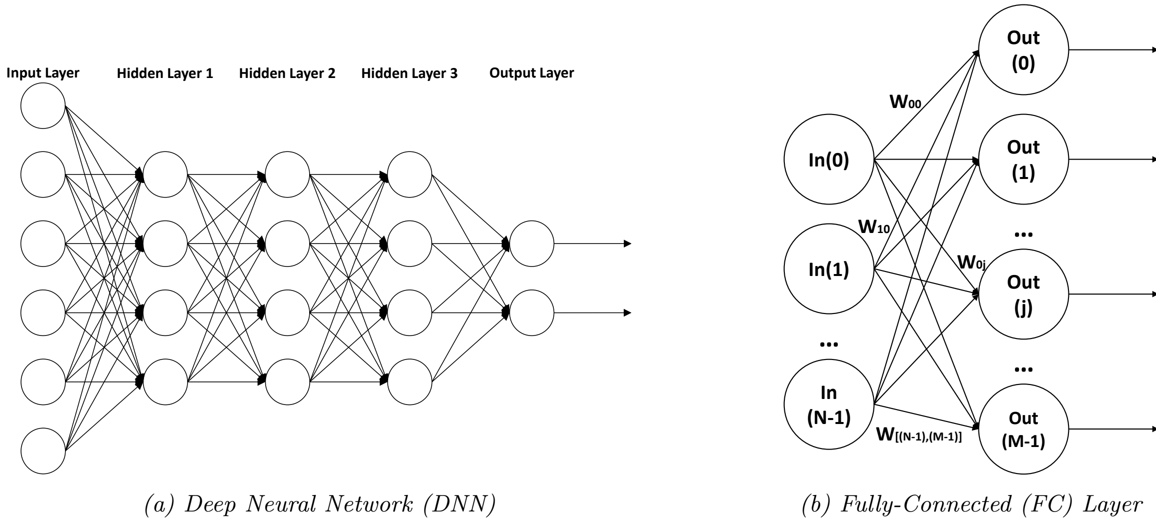


Figure 2.1: Example of DNN training vs DNN inference.

summarize, DNNs are artificial neural networks with multiple hidden layers of artificial neurons between the input and output layers.

DNNs have two phases, training (or learning) and inference (or prediction), which refer to DNN model construction and use respectively. Figure 2.1 shows the difference between both phases. The training procedure determines the weights or parameters of a DNN, adjusting them repeatedly until the DNN achieves the desired accuracy. During training, a large set of sample input data, with its corresponding labels indicating the correct classification, is used to execute the DNN's forward pass and measure the error against the correct labels. Then, the error is used in the DNN's backward pass to update the weights. On the other hand, inference uses the DNN model developed during the training phase to make predictions on unseen data. DNN inference has strict latency and/or energy constraints. Both phases are done using floating point computations, although as described in Section 2.3, there are generalized DNN optimizations, such as quantization, to reduce the inference precision without degrading the quality of the model. Lower precision inference enables better performance and improved energy efficiency for DNN inference. Although the DNN training is complex and computationally expensive, it is usually done only once per model, and then, the learned weights are used in inference as many times as it is required to classify new input data. For this reason, in this thesis we focus on the inference phase.

DNNs can be classified in three main categories. First, Multi-Layer Perceptrons (MLP) [107, 105] consist of multiple Fully-Connected (FC) layers in which every input neuron is connected, via synapses with particular weights, to every output neuron. Figure 2.2a shows an example of an MLP composed of FC layers. Second, Convolutional Neural Networks (CNN) are composed of multiple convolutional layers to extract features, usually followed by one or several FC layers to perform the final classification. CNNs have proved to be particularly efficient for image and video processing [48, 31, 38, 92]. Finally, Recurrent Neural Networks (RNN) consist of multiple layers of cells with feedback connections, stacked on top of each other. RNN cells store information from past executions to improve the accuracy of future predictions. The most popular RNN architectures are the Long-Short Term Memory (LSTM) [35] and the Gated Recurrent Unit (GRU) [15]. In both



(a) Deep Neural Network (DNN)

(b) Fully-Connected (FC) Layer

Figure 2.2: Example of a DNN from the (a) MLP category composed of (b) FC layers.

cases, the cell consists of multiple single-layer FC networks commonly referred as gates.

RNNs and very deep MLPs such as the Transformer model have become the state-of-art solution for sequence processing problems such as machine translation and speech recognition [97, 4, 93]. The Transformer [93] model has recently received special attention from the machine learning community for being extremely efficient in terms of both accuracy and performance. Transformers use attention mechanisms [6, 60] to gather information about the relevant context of a given input (i.e., a word of a sentence), and then encode that context in a vector. The attention mechanism allows to grab context information from distant parts of an input sequence to help understand its meaning, and it is implemented in the form of multiple feed-forward FC layers.

On the other hand, MLPs and CNNs deliver state-of-the-art accuracy for applications such as acoustic scoring in speech recognition [107] or self-driving cars [9] respectively. Most recent works focus on optimizing CNN inference and its convolutional layers. Although CNNs are the most efficient solution for image processing applications, their unique characteristics make it hard to exploit the same techniques on other DNNs. In addition, state-of-art accelerators implementing systolic array architectures, such as TPU [43], have proven to be efficient hardware implementations to perform CNN inference since the weights of a convolutional filter are reused multiple times. However, the FC layers do not have the same reuse properties and may cause the accelerator to be highly underutilized, especially for small batch sizes that are common in inference.

As it can be seen, each type of DNN is especially effective for a specific subset of cognitive applications with varying complexity. Moreover, for each application, each DNN has a different composition of layers with specific operations and unique characteristics. In this thesis we focus on optimizing the performance and energy consumption of hardware accelerators for a variety of DNNs. To illustrate the broad applicability of our techniques, we evaluate the proposals of this thesis on MLPs, including Transformers, CNNs and RNNs for a diverse set of applications. The next subsections provide more details on FC, convolutional and recurrent layers, as these layers take up the bulk of the computations in DNNs. Other types of layers performing pooling, normalization

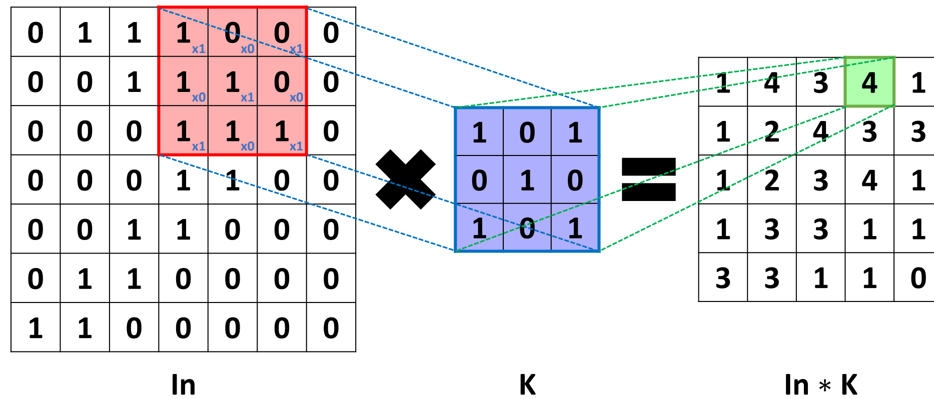


Figure 2.3: Convolutional 2D layer.

or activation functions are also common in modern DNNs. However, these other layers have no synaptic weights and represent a very low percentage of the DNN execution activity.

2.1.1 Fully-Connected Layers

The main performance and energy bottleneck of MLPs, RNNs and Transformers are the FC layers. In an FC layer, each output neuron performs a dot product operation between all the inputs of the layer and their corresponding weights. Figure 2.2b depicts an FC layer. More specifically, the output of neuron j is computed according to the following equation:

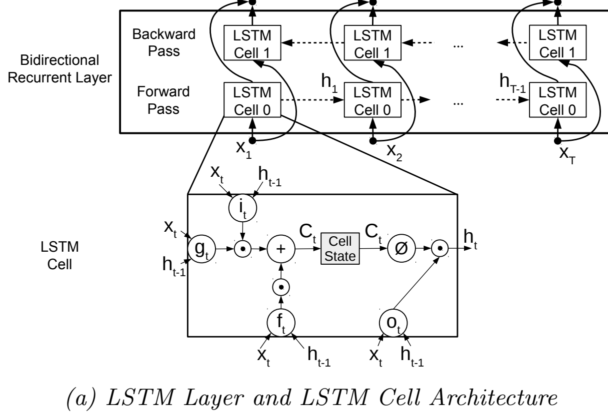
$$out(j) = (\sum_{i=0}^{N-1} w_{ij} * in(i)) + b_j \tag{2.1}$$

where $in(i)$ represents the input vector, w_{ij} is the weight of neuron j for input i, and b_j represents the bias of neuron j. An FC layer with N inputs and M neurons contains $N \times M$ weights, as each neuron has its own set of weights, and requires $2 \times N \times M$ operations. FC layers employed in real applications consist of thousands of neurons and they typically account for most of the computations and memory bandwidth usage of MLPs, RNNs and Transformers. FC layers may also take most of the storage requirements and memory bandwidth usage in CNNs that include a few FC layers at the end of the model to perform the final classification.

2.1.2 Convolutional Layers

Convolutional layers are commonly used for image processing, object recognition and video classification. A convolutional layer implements a set of filters to detect features in the input image. Figure 2.3 depicts an example of a 2D convolutional layer applying a filter of 3×3 over a region of the input data on a single channel.

For video processing, 3D convolution across multiple frames provides higher accuracy than



(a) LSTM Layer and LSTM Cell Architecture

$$i_t = \sigma(W_{ix}x_t + W_{ih}h_{t-1} + b_i) \tag{2.3}$$

$$f_t = \sigma(W_{fx}x_t + W_{fh}h_{t-1} + b_f) \tag{2.4}$$

$$g_t = \phi(W_{gx}x_t + W_{gh}h_{t-1} + b_g) \tag{2.5}$$

$$o_t = \sigma(W_{ox}x_t + W_{oh}h_{t-1} + b_o) \tag{2.6}$$

$$c_t = f_t \odot c_{t-1} + i_t \odot g_t \tag{2.7}$$

$$h_t = o_t \odot \phi(c_t) \tag{2.8}$$

(b) LSTM Cell Computations

Figure 2.4: (a) Bidirectional LSTM layer, architecture of an LSTM cell, and (b) LSTM cell Computations. $\odot$, ϕ and σ are element-wise multiplication, hyperbolic tangent and sigmoid function respectively.

2D convolution [92]. In case of 3D convolution, a filter is defined by $K_x \times K_y \times K_z$ coefficients or weights. Each convolutional layer applies multiple of these filters through the entire input, resulting in multiple output feature maps. Note that, unlike what happens with FC layers, the weights of the different filters are shared by all the neurons in the layer. The concrete formula for calculating an output neuron $out(x, y, z)^{f_o}$ at position (x, y, z) of output feature map f_o is:

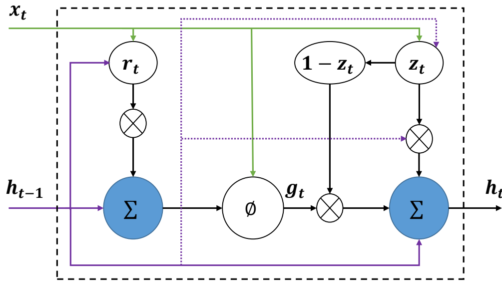
$$out(x,y,z)^{f_o} = \sum_{f_i=1}^{N_{if}} \sum_{k_x=0}^{K_x} \sum_{k_y=0}^{K_y} \sum_{k_z=0}^{K_z} (w_{f_i,f_o}(k_x,k_y,k_z) * in(x+k_x, y+k_y, z+k_z)^{f_i}) \tag{2.2}$$

where $in(x, y, z)^{f_i}$ represents the input at position (x, y, z) in input feature map f_i, and $w_{f_i,f_o}(k_x, k_y, k_z)$ is the synaptic weight at kernel position (k_x, k_y, k_z) in input feature map f_i for filter f_o. Convolutional layers represent most of the computations performed in CNNs.

2.1.3 Recurrent Layers

Recurrent layers include loops or feedback connections, allowing information to persist from one execution of the network to subsequent ones. Long Short Term Memory (LSTM) cells [35] and Gated Recurrent Units (GRU) [15] represent the most successful approaches due to their ability to capture long term dependencies, as they can keep useful information for future predictions over long periods of time. Figure 2.4a shows a bidirectional LSTM layer that includes two different LSTM cells executed in the forward and backward direction respectively. As it can be seen, the same LSTM cell is recurrently executed for each element x_t in the input sequence. Furthermore, the cell also takes as input the output of the previous execution (h_{t-1}).

The bottom of Figure 2.4a shows the architecture of an LSTM cell, whereas Figure 2.4b provides the computations performed inside the cell. The key component is the cell state, c_t, that represents the memory storage for the cell. The cell state is updated by using four gates. The input gate



(a) GRU Cell Architecture

$$z_t = \sigma(W_{zx}x_t + W_{zh}h_{t-1} + b_z) \tag{2.9}$$

$$r_t = \sigma(W_{rx}x_t + W_{rh}h_{t-1} + b_r) \tag{2.10}$$

$$g_t = \phi(W_{gx}x_t + W_{gh}(r_t \odot h_{t-1}) + b_g) \tag{2.11}$$

$$h_t = z_t \odot h_{t-1} + (1 - z_t) \odot g_t \tag{2.12}$$

(b) GRU Cell Computations

Figure 2.5: (a) Architecture of a GRU cell, and (b) Computations performed in a GRU cell. $\odot$, ϕ and σ are element-wise multiplication, hyperbolic tangent and sigmoid function respectively.

(Equation 2.3) decides which information is added to the cell state. The forget gate (Equation 2.4) decides which information is removed from the cell state. The generate gate (Equation 2.5) provides candidate information to be added to the cell state. Finally, the output gate (Equation 2.6) decides which information of the cell state is used to generate the output h_t. Each gate is implemented as an FC layer taking two different inputs: x_t, a.k.a. feed-forward input, and h_{t-1}, a.k.a. recurrent input. FC layers for the different gates take most of the computations in an LSTM network.

RNNs composed of GRUs have a prediction accuracy slightly lower than those made of LSTM cells but with lower computational cost. Figure 2.5b provides the computations performed inside a GRU cell. GRUs are an LSTM variation where the forget and input gates are combined in a single generate gate g_t, and the cell state is merged within the output of the cell h_t. As illustrated in Figure 2.5a, the architecture of a GRU cell only has the generate gate g_t and two more gates, the reset gate r_t and the update gate z_t. The reset gate determines the amount of information from the previous output h_{t-1} that is added by the generate gate to the current output, while the update gate decides to what extent the output h_t should be updated by the generate gate to enable long-term memory. Similar to LSTM networks, the FC layers for the different gates take most of the computations in a GRU network.

2.2 DNN Acccelerators

The main objective of this thesis is to design low-power accelerators for DNNs, which are memory-intensive and require a large amount of storage and computation time, especially for sequence processing applications. A well-known academic work on DNN accelerators is the DianNao [12], and their variations DaDianNao [14] for large scale and ShiDianNao [16] for mobile devices. Although there are several proposals before DianNao [89, 27, 28], they are highly restricted to small neural networks, such as those for recognizing handwritten digits in images [52, 53], and rely too much on accessing main memory, which represents an important overhead in energy consumption when applied to larger DNNs. DianNao was the first accelerator to include its own on-chip SRAM buffers to significantly reduce memory accesses, and DaDianNao further improved this aspect by adding eDRAM to store the weights.

On the other hand, since the advent of Google's Tensor Processing Unit (TPU) [43], which has proven to be an efficient commercial accelerator to perform DNN inference, systolic array architectures have gained popularity. Although TPU supports any kind of neural network, the systolic array architecture is specially efficient for CNNs since the weights of a convolutional filter can be reused multiple times. Conversely, utilization and peak performance of the systolic array for FC layers tend to be significantly lower because the data reuse is much more limited compared to convolutional layers.

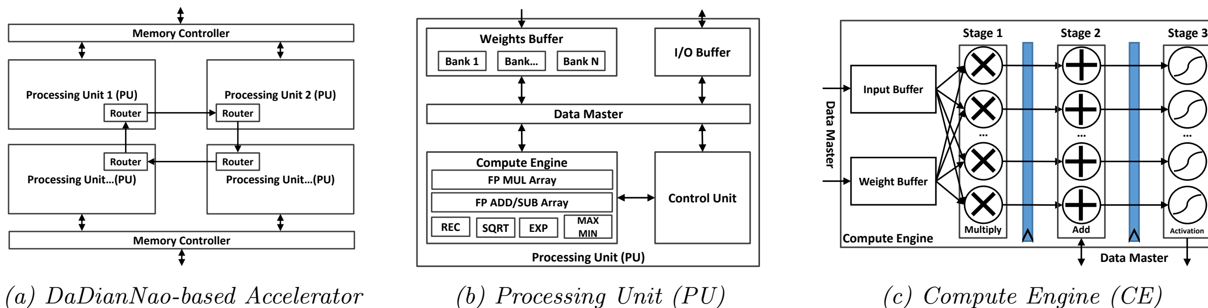
Most of the DNN accelerators presented in recent years are based on DaDianNao or TPU, such as Minerva [76], EIE [25], Eyeriss [13] or FlexFlow [59], among other popular accelerators. Following the same path, we developed two baseline DNN accelerators based on the architecture of DaDianNao and TPU, to later implement our optimization proposals on top of them, adding the required changes in the respective architectures. Initially, we implemented a DaDianNao-based DNN accelerator for sequence processing applications such as speech recognition. Later on, due to the increasing popularity of TPU and the systolic array architectures, we implemented a TPU-like DNN accelerator, allowing us to make fair comparisons with the state-of-the-art solutions. Next subsections describe the architecture of these baseline DNN accelerators in detail.

2.2.1 DaDianNao

Figure 2.6 depicts a high-level block diagram of the architecture of the baseline DaDianNao-based DNN accelerator. The core of the accelerator is the Processing Unit (PU) illustrated in Figure 2.6b. The PU is composed of a Compute Engine (CE) containing the functional units that perform the computations, including an array of multipliers and adders, together with specialized functional units (reciprocal, square root...). On the other hand, the Control Unit (CU) contains the configuration of the DNN and provides the appropriate control signals in every cycle.

Regarding the on-chip storage, the PU includes an eDRAM memory to store the synaptic weights of the different layers. Note that the memory footprint for the weights is typically quite large for common DNNs and, hence, significant on-chip storage is required to avoid excessive off-chip memory traffic that is very costly from an energy point of view [26]. Similar to DaDianNao, eDRAM is used in our design to provide larger on-chip capacity with a small cost in area compared to SRAM. This memory is highly multi-banked to achieve the required bandwidth to feed a large number of functional units in the CE. On the other hand, the I/O Buffer is an SRAM memory with two banks used to store the intermediate inputs/outputs between two DNN layers. Finally, the Data Master is in charge of fetching the corresponding weights and inputs from the on-chip memories, and dispatching them to the functional units in the CE.

The accelerator is configured for different DNNs by loading the neural network model that includes the information of each layer, i.e. input neurons, output neurons, weights and biases. In case all the parameters fit in the on-chip memories of the accelerator, the entire dataset is loaded from main memory. Otherwise, the accelerator loads the amount of elements that can fit on-chip, whereas the rest will be loaded from main memory on demand. By doing so, weights stored on-chip are reused across multiple DNN executions and, hence, no additional off-chip memory access is required for those elements. The accelerator is power gated during idle periods and, hence, weights



(a) DaDianNao-based Accelerator *(b) Processing Unit (PU)* *(c) Compute Engine (CE)*

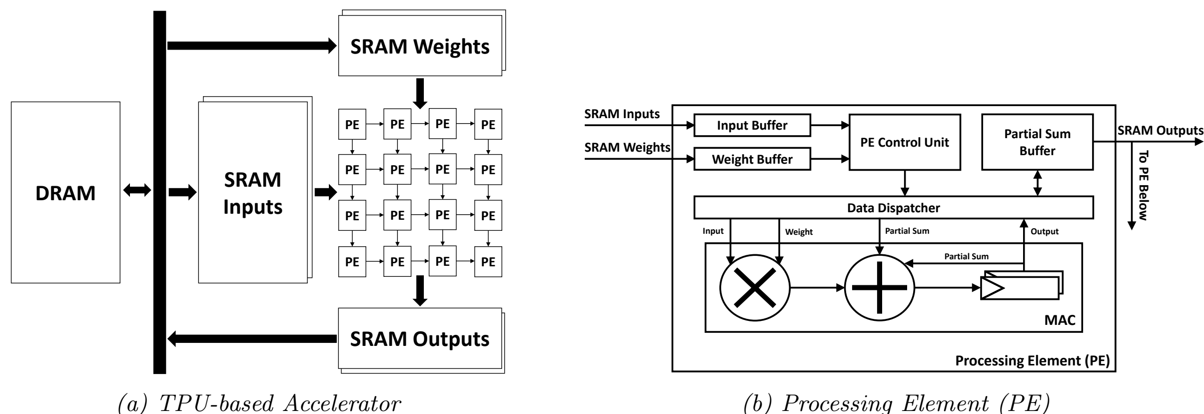
Figure 2.6: Architecture of the baseline DaDianNao-based accelerator showing the (a) tile organization, and the main hardware components of the (b) Processing Unit (PU) and the (c) Compute Engine (CE).

are loaded from main memory at the beginning of processing every sequence or batch of inputs (audio utterance, video...).

State-of-the-art DNN accelerators, such as DaDianNao or TPU, cannot execute a neural network completely unrolled in parallel, since the accelerator would require millions of functional units and would be too expensive, especially when we aim for a low-power design. Instead, because the number of functional units is limited, the execution of a DNN is performed layer by layer, operating on a subset of neurons at a time, depending on a particular dataflow. The dataflow dictates how the inputs and weights are read, processed and reused. Our DaDianNao-based DNN accelerator implements an input stationary dataflow [13], which means that each input of a given layer is read and reused, until all the related computations are done, before reading the next input.

The Compute Engine (CE), illustrated in Figure 2.6c, has an architecture pipelined in three stages. The first stage performs multiplications of an input by the weights of different output neurons. The second stage performs additions to accumulate the result of each output neuron. Finally, the third stage implements special operations to perform activation functions when required. Besides, the CE implements two small local buffers to prefetch inputs and weights, as well as some registers to store the partial results of the functional units. Finally, outputs are sent to the Data Master to be stored in the I/O Buffer, and used as inputs by the next layer. The overall PU is also pipelined so, in the same cycle, the I/O Buffer reads inputs and partial sums, the Data Master reads weight from memory, and the CE performs multiplications and additions.

In addition, multiple instances of processing units can be integrated in the same chip, as shown in Figure 2.6a, to improve performance and accommodate large DNNs. Each PU instance, or tile, includes a router to communicate results with the other tiles. In our design, the different tiles are connected through a ring and the workload is distributed as follows. For FC layers, output neurons are evenly distributed among the tiles. Regarding convolutional layers, the different filters are distributed among the tiles. Finally, for recurrent layers, different tiles process different gates of one LSTM or GRU cell.



(a) TPU-based Accelerator *(b) Processing Element (PE)*

Figure 2.7: Architecture of (a) the baseline TPU-based accelerator and (b) a Processing Element (PE).

2.2.2 TPU

Figure 2.7a shows a high-level schematic of the architecture of the baseline TPU-based DNN accelerator. The main components are the blocks of SRAM used for the inputs, outputs and weights, and the systolic array of processing elements (PEs). All the global SRAM memories are double buffered to load data from main memory while performing computations, and highly multi-banked to achieve the bandwidth required to feed a large number of PEs. Each PE includes a MAC (multiplier-accumulator) with some registers as illustrated in Figure 2.7b. In addition, each PE has three small local buffers to prefetch inputs and weights, as well as for storing the partial sums when required depending on the dataflow.

Traditional DNN accelerators implementing a systolic array architecture follow one of these dataflows [13]: output stationary, weight stationary or input stationary. In output stationary, each PE computes an output neuron at a time. In the weight/input stationary dataflow, each PE preloads a set of weights/inputs from memory to the local buffers, and those are used to perform all the associated computations. TPU implements a weight stationary dataflow which is a good fit for CNNs since there is usually a low number of weights but a large number of inputs. In our case, with the purpose of comparing the different dataflows, we designed two baseline TPU-like accelerators, one with output stationary dataflow and the other with weight stationary dataflow.

2.3 DNN Optimizations

Several state-of-the-art DNN accelerators focus on low power [25, 16, 43]. DNN accelerators have the potential to be inherently fault tolerant [30], or capable of robust computation, in the sense that its accuracy degrades gracefully under adverse conditions. There have been many studies that observed the inherent fault tolerance of DNNs [89, 27, 17]. For example, if a neuron or its connecting links are damaged, due to the distributed nature of information stored in the network, the damage has to be extensive before the overall response of the network is degraded seriously. Thus, in principle, a neural network exhibits a graceful degradation rather than catastrophic failure. This

characteristic can be used to trade accuracy for higher performance and/or energy savings. For example, the accelerator Minerva [76] reduces the voltages of some components to reduce the energy consumption in exchange of reliability, while Stripes [44] changes the precision of the computations in a per layer basis. Popular optimizations for reducing the DNN memory footprint, numerical precision and/or the number of DNN computations include clustering [21], quantization [96] and pruning [26]. This section provides a brief description of each family of optimization techniques.

Clustering uses methods such as K-means to reduce the number of different weights to K centroids. Each weight is then substituted by an index that corresponds to the closest centroid. Since the weights tend to be very similar, the number of centroids per layer can be kept relatively low (in the order of 16-256), which significantly reduces the storage requirements and memory bandwidth for the weights. However, computations still have to be performed in floating point by using the centroids, and the total amount of computations is not reduced at all.

On the other hand, linear quantization maps each value, either weights or inputs, to a discrete set of values uniformly distributed over the whole range of possible values. Values are replaced by indexes, which identify the discrete set of values, and reduces their storage requirements. Unlike clustering, since the quantization is linear, most computations can be done by operating directly on the integer indices rather than the corresponding floating point values. The amount of computations is not reduced, but most computations are simpler since they operate on narrow integer numbers rather than single-precision floating point values. Linear quantization can reduce the DNN computations precision to 8/16 bits with negligible accuracy loss for the vast majority of DNNs. Therefore, linear quantization is particularly efficient in DNN accelerators to not only reduce the model size but also the energy consumption of the computations.

In addition, recent proposals are exploring aggressive non-uniform quantization mechanisms that can further reduce the precision of the DNN parameters. The key idea is to learn the non-uniform quantization parameters during the training of the model, which requires the DNN to be trained from scratch. TTQ [110], DoReFa [108] and LQNets [104] achieve important reductions in the numerical precision (i.e. $< 8b$) with small accuracy loss. Some non-uniform quantization mechanisms, such as LQNets, are compatible with bitwise operations but still require some floating point operations to be performed. Therefore, performance wise, these methods may not be better than uniform quantization but the DNN model can be highly compressed.

Finally, static pruning [26, 102] reduces the model size and the number of computations by removing connections or nodes depending on the weight values. The pruned model may loss accuracy but tends to regain it after retraining while requiring significantly less memory storage and computations. On the other hand, dynamic operation pruning [5, 76], also called activation pruning, avoids computations when the synaptic weight is multiplied by a zero-value input.

To summarize, clustering and quantization techniques do not reduce the amount of computations, but they are highly effective at reducing the storage requirements and/or the cost of the computations. On the other hand, static pruning reduces both, the storage requirements and the amount of computations, at the expense of a small accuracy loss. The accelerators proposed in this thesis support all of these optimizations, and efficiently exploit the input/weight repetition originated from them to improve performance and energy efficiency. The following subsections describe in more detail how linear quantization and pruning are applied.

2.3.1 Linear Quantization

Linear quantization [96] is a highly popular technique to map a continuous set of values to a finite set. In the accelerators presented in this research, we apply uniformly distributed linear quantization to the inputs and weights of FC, convolutional and recurrent layers of DNNs. These layers represent close to 100% of the total execution time for typical neural networks. For each input and/or weight v, and each layer l, the quantization is applied according to the following equations:

$$Qval_c = round\left(\frac{value_v}{step_l}\right) * step_l \qquad step_l = \left(\frac{range_l}{C-1}\right) \tag{2.13}$$

$$Qidx_c = round\left(\frac{value_v}{step_l}\right) \qquad C = 2^{n_bits} \tag{2.14}$$

The step is computed as the range divided by the number of clusters (C). The range of the inputs of a layer is obtained via profiling using the training dataset, while the range of the weights is known statically. $Qval_c$ is the cluster centroid that is closest to the original value of the input/weight, while $Qidx_c$ is the integer representation of the centroid. The number of clusters is given by the number of bits that we want to use for the integer representation, which also affects the trade-off between the benefits achieved by applying quantization and the impact in accuracy.

The main benefits of the linear quantization are the reduction in storage, due to a smaller representation of the weights, and the lower computational complexity, due to performing integer operations rather than floating point, with negligible impact in accuracy if enough clusters are provided. Moreover, a side effect of the quantization is that it significantly increases the input/weight repetition which we exploit throughout this thesis.

2.3.2 Pruning and Sparse Accelerators

Pruning reduces the model size and the number of computations with the aim of reducing energy consumption and increasing performance. Pruning can be performed statically or dynamically. As shown in Figure 2.8, static pruning refers to the removal of connections (i.e. synaptic weights) of a DNN model already trained, although it can also be done during the training itself. If all the connections of a node (i.e. neuron) are removed, then the node can also be removed. Usually, the DNN model is trained from scratch, and then, the model is iteratively pruned and retrained until the desired level of pruning and accuracy is achieved.

On the other hand, dynamic pruning refers to the removal of computations due to some neuron inputs being zero or very close to zero. Since the inputs can vary from one execution to another, the effectiveness of this pruning changes dynamically as well. Figure 1.6a shows an example where we assume that x_1, h_0 and h_2 are almost zero. Because the inputs of a neuron are the activation values of the previous layer, this method is usually referred as activation pruning. Note that static pruning reduces both the model size and the number of computations while dynamic pruning only

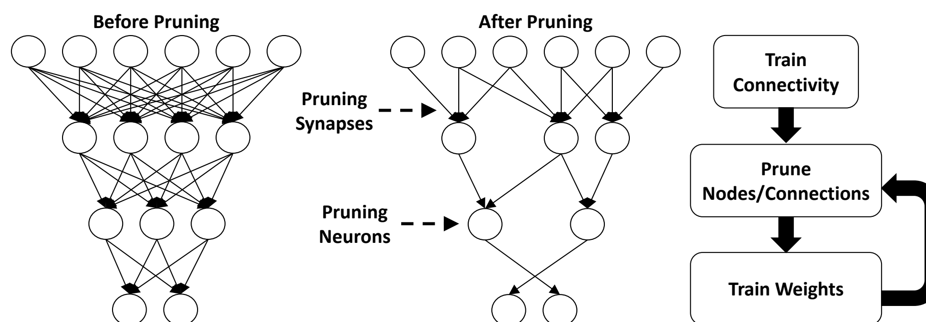


Figure 2.8: Example of static DNN Pruning.

reduces computations. However, dynamic pruning has a negligible impact in accuracy (or even non-existent when no approximations are applied) and does not require retraining. Both ways of pruning are orthogonal so they can be applied at the same time.

The naive way to perform activation pruning is by dynamically skipping individual connections of fully-connected or convolutional layers if their input value is exactly zero [5]. This is common when the activation function is a Rectified Linear Unit (ReLU), which outputs zero for all negative input values. Minerva [76] improved the effectiveness of this method by including values that are close to zero, thus, increasing the number of operations that are avoided with a minor accuracy loss. SnaPEA [1] proposes to early stop the computations of convolutional layers by doing a prediction of the activation values at runtime. After sorting the weights based on their signs (i.e. first positives then negatives), their insight is that if the partial output of a convolution after a certain number of MAC operations is less than a threshold, the final convolution output will likely be negative. As a result, the rest of the computations can simply be ignored, since the output value after the ReLU activation will be zero in any case. However, all these methods can only be applied to DNNs with ReLU activation functions, which are common in MLPs and CNNs but not in RNNs, which typically use sigmoids and hyperbolic tangents. In consequence, Chapter 7 presents a new dynamic pruning method for RNNs that exploits their intrinsic characteristics.

A variety of different static DNN pruning schemes have been recently proposed. Some methods remove connections depending on the weight values [26], others remove nodes taking into account the weights of each node [32, 85]. PCA-based methods for fast pruning have been previously proposed [54, 95, 62, 90, 19, 69, 46]. However, they just project the weights and inputs using the principal components and do not apply retraining. The amount of pruning achieved with these PCA-based methods is extremely limited for modern DNNs. Most of these pruning methods are applied after training a baseline non-pruned network. On the contrary, some methods perform pruning during the training phase by learning the connections or nodes that are redundant [102, 23, 99].

Pruned models become sparse, requiring specialized hardware with a proper dataflow to be efficiently executed. Multiple sparse DNN accelerators [25, 72, 106, 24] have been recently proposed. These accelerators encode the weights in a Compressed Sparse Column/Row (CSC/CSR) format that stores non-zero weights and the number of zeros from a non-zero weight to the next, relative to the column/row. The compressed sparse format is decoded at runtime, which adds to the overheads of having extra metadata. Moreover, sparse accelerators tend to suffer from workload imbalance

problems since each PE (or Tile) may process a different number of non-zero weights. Some sparse accelerators, such as EIE [25], utilize clustering to further reduce weight storage, but do no explore ways to reuse computations as we propose in this thesis.

Recently, structured pruning [102, 109] is being actively researched to reduce the overheads of the sparse format while alleviating the load balancing problem. Structured pruning methods divide the weights into vectors or blocks aiming to prune either the same percentage inside each structure or entire structures depending on the weight values. These structures make the pruning more homogeneous reducing the sparsity and its associated overheads. However, the percentage of pruning achieved is below those of non-structured pruning. In addition, it is not clear which structures are better and how to size them.

Section 2.4 describes in more detail a set of popular pruning schemes. Moreover, we show in Chapter 7 that most of these methods are impractical due to the large number of parameters that have to be tuned for each DNN, while we propose a new method inspired in previously proposed PCA-based techniques and weight pruning schemes. In addition, we show that our method is highly effective and, more importantly, practical for a wide range of contemporary DNNs.

2.4 Main DNN Pruning Schemes

As already discussed, there is a large number of methods to prune DNNs. In general, pruning schemes require retraining of the pruned network to recover the original accuracy, and the resulting model becomes sparse which may incur in some overheads depending on the system, since working with sparse matrices/vectors is more costly than operating on dense arrays for many current systems. Although pruning methods have achieved tremendous success for image classification (e.g. AlexNet [48, 47]), there are very few studies about their effectiveness for other applications such as speech recognition. More importantly, all the schemes rely on heuristics with multiple parameters that require manual tuning, including the percentage of pruning for each layer. The next subsections provide more details on some of the most recent and popular pruning methods.

2.4.1 Near Zero Weights Pruning

Han et al. [26] proposed a pruning method to remove the connections whose weight has an absolute value lower than a given threshold, which is computed using the following equation:

$$Threshold = std(W_l) * qp \tag{2.15}$$

where $std(W_l)$ represents the standard deviation of all weights in layer l and the quality parameter (qp) determines the degree of pruning. The main idea of this heuristic is to remove the weights that are closer to zero. In the paper they report a 90% pruning for AlexNet without accuracy loss. However, the quality parameter is different per layer and the paper does not present any methodology to set it up other than try and error. Note that exploring all possible combinations would be totally unfeasible for networks with many layers since each trial must be followed by a retraining,

which is extremely expensive. Even for AlexNet, which has only 8 layers, the exploration of the design space is huge. For more recent DNNs such as ResNet [31], the winner of the 2015 Imagenet Large Scale Visual Recognition Challenge [39], DenseNet [38] or SENet [37], the winner for 2017, this would be impractical given that they have more than 100 layers.

We implemented this method using a global quality parameter for all the layers, to reduce the search space to just one parameter, and the degree of pruning achieved was more moderate, around 70% in AlexNet. Finally, note that the heuristic they used does not work well if weights are not distributed around zero. In this case, this heuristic does not remove any connection, but there still may be connections that are unimportant compared to the others. For instance, if a neuron has ten input connections, nine of them have weights with a magnitude around 100 and the remaining one has a magnitude of around 10, this latter connection will likely be unimportant. However, this heuristic will not remove it since its weight is much larger than zero.

2.4.2 Node Pruning

He et al. [32] proposed multiple metrics to identify which nodes are redundant for each layer l.

- Entropy:

$$Score(i,l) = \frac{d_i^l(O)}{|O|} * \log_2(\frac{d_i^l(O)}{|O|}) + \frac{a_i^l(O)}{|O|} * \log_2(\frac{a_i^l(O)}{|O|}) \tag{2.16}$$

  where $|O|$ is the total number of frames, and $a_i^l(O)$ and $d_i^l(O)$ are the number of frames which activate or deactivate node i.

- Output Weights Norm (o-norm):

$$Score(i,l,l') = \frac{1}{N^{l'}} \sum_{j=1}^{N^{l'}} \left| W_{ij}^{l'} \right| \tag{2.17}$$

  where l' is the next layer for o-norm.

- Input Weights Norm (i-norm):

$$Score(i,l,l') = \frac{1}{N^{l'}} \sum_{j=1}^{N^{l'}} \left| W_{ji}^{l} \right| \tag{2.18}$$

  where l' is the previous layer for i-norm.

The first metric, called Entropy, examines the activation distribution of each node. A node is considered activated if the output value is greater than a threshold of $0,5$. The idea of this metric is that if one node's outputs are almost identical on all training data, these outputs do not generate variations to later layers and consequently they are not useful. The second and third metrics, called i/o-norm, determines the importance of a neuron based on the average of the weights of its

incoming or outgoing connections. Nodes are sorted by their scores and those with lower scores are removed. The network is then retrained.

All the metrics achieve similar results, around 60% of pruning on TIMIT, a DNN for speech recognition. Note that one still has to decide how much to prune each layer, and they do not provide any heuristic to determine this parameter other than trial and error. Furthermore, we show in Chapter 7 that a blind pruning that randomly selects the nodes to be removed achieves the same results as the aforementioned heuristics.

2.4.3 Similarity Pruning

Another way to detect redundancy, proposed by Srinivas et al. [85], is to measure how similar the nodes are by computing the squared difference of the weights for each pair of nodes using the following equation:

$$Saliency(i,j,l) = \sum_{k=1}^{N} (\|W_{ik} - W_{jk}\|)^2 \tag{2.19}$$

The neurons with the lowest saliency are pruned. In this scheme, retraining is not applied but they achieve a rather moderate 35% of pruning on AlexNet, which is low compared to other methods. This method is only applied to the fully-connected layers. In short, this method avoids retraining but it achieves a low percentage of pruning and requires a huge design space exploration to determine the particular threshold that should be used for each network layer.

2.4.4 Scalpel

Scalpel [102] is an iterative pruning method that determines which nodes to prune during training. To this end, a mask node is added after each original neuron to multiply its output by a parameter alpha that can be either 1 or 0. The method is divided into two steps, in the first step the mask layers are trained depending on a weight decay parameter that determines how much aggressive the pruning is. Then, the nodes for which the mask becomes zero after the training are removed and the network is retrained without the mask layers. This process is repeated multiple times, each time with an increased value of the weight decay, until a loss in accuracy is observed.

For the training phase when the masks are added, Scalpel uses two set of variables, called alphas and betas. Alphas represent the pruning mask that is applied to the output of the nodes y_i during the forward evaluation of the network. That is:

$$Y_i^{'} = \alpha_i * y_i \tag{2.20}$$

where $Y_i^{'}$ are the final outputs passed to the next layer. Alphas cannot be learned by the conventional Backpropagation method [79], since the cost function is not a continuous function of

alpha. To overcome this, Scalpel associates another parameter called beta to each alpha. These betas can take any Real value and are learned during training, as if they were the multiplicative coefficients applied to the outputs. Alphas are updated to 0 or 1 depending on the betas, a threshold and an epsilon offset as shown in Equation 2.21.

$$\alpha_{i|k} = \begin{cases} 1 & T + \epsilon \leq \beta_{i|k} \\ \alpha_{i|k-1} & T \leq \beta_{i|k} < T + \epsilon \\ 0 & \beta_{i|k} < T \end{cases} \tag{2.21}$$

Regularization is applied to the betas during training to penalize high values by using a weight decay parameter. The weight decay parameter also determines the amount of pruning since a high value will increase the penalization to the betas and the backpropagation algorithm will try to reduce them, and lower betas imply that more alphas are zero, which results in more pruning. Finally, multiple iterations of the algorithm are done by increasing the weight decay parameter on each iteration to increase the pruning, until a loss in accuracy is observed. The weight decay and the step that is used to increase it on each iteration have to be manually set and are different for each DNN.

The main drawback of Scalpel is that it requires manual tuning of multiple parameters (threshold, epsilon, learning rate, etc.) for each particular DNN. Besides, its pruning effectiveness is not better than using previous methods, since its main target is pruning the DNN while avoiding sparsity. For instance, Scalpel achieves only 20% of node pruning for AlexNet.

2.4.5 PCA Pruning

Levin et al. [54] proposed a pruning method to remove the nodes by using Principal Components Analysis (PCA). PCA can be used to reduce the number of nodes by computing the correlation matrix of the nodes activity of each layer. We can perform an eigendecomposition of the correlation matrix of the nodes activity to obtain the eigenvectors and eigenvalues. The eigenvalues can be used to rank the importance of a node of the new system.

The algorithm they propose to prune is divided into multiple steps. Starting from the first layer, the correlation matrix of the nodes activity is computed. The nodes activity is measured from multiple inputs of the training set and a pretrained network. The eigenvectors of the correlation matrix (i.e. Principal Components) are ranked by their corresponding eigenvalue and the effect of removing each node (i.e. eigenvector) is measured using the validation set. The nodes that do not increase the error are chosen to be removed. The weights of the layer are projected into the new subspace by multiplying the original weights (W) by the significant eigenvectors (C_l) as shown in Equation 2.22. The procedure continues until all the layers are pruned. Note that this algorithm follows an iterative pruning since the validation and projection is performed after removing each node and stops when the accuracy of the network decreases. This method is only applied to the fully-connected layers and does not require any additional retraining.

$$W \rightarrow W * C_l * C_l^T \tag{2.22}$$

$$W * I \rightarrow W * C_l * C_l^T * I \tag{2.23}$$

Note that this scheme does not actually prune physical nodes or connections but reduces the number of parameters and computations depending on the amount of principal components that are removed. However, since the inputs also have to be projected using the significant eigenvectors as shown in Equation 2.23, both the non-pruned eigenvectors and the projected weights have to be stored. Therefore, the pruning of eigenvectors has to be highly effective in order to actually reduce parameters and computations of the neural network.

The method was originally evaluated using a small feed-forward network of two layers with a time series dataset. We implemented this method for LeNet5 using the MNIST dataset and it achived only around 10% of pruning with negligible accuracy loss. We have also tested this method on a modern Kaldi DNN [107] and the pruning achieved was less than 1%. These low pruning percentages compared to previous methods suggest that pruning effectiveness is quite limited if the pruning method does not include retraining.

2.5 Computation Reuse

DNN optimizations, such as clustering and quantization, favor the appearance of repeated inputs and weights. Several recent studies observed the large number of repeated inputs/weights, and exploit it, together with the inherent fault tolerance of DNNs, to improve performance and energy efficiency of DNN inference by proposing algorithms to reuse computations, with low impact in accuracy. This section describes some of the most popular techniques to reuse computations.

In Chapter 5, we propose CREW, an accelerator implementing a reuse-based mechanism of partial products exploiting weight repetition, coupled with an enhanced weight stationary dataflow to efficiently execute FC layers on a systolic array architecture. In addition, CREW is compared against UCNN, a state-of-art computation reuse technique. UCNN [34] and the mechanism in [20] perform factorization of repeated weights in CNNs to reduce computations. However, as discussed along this thesis, FC layers have different requirements, and the proposed mechanisms do not fully exploit the benefits of computation reuse for those layers. Reuse of partial products with repeated weights has been previously proposed in [100], where the output neurons of a given FC layer are clustered depending on their weights similarity, and the partial products may be reused inside a given cluster of neurons. Note that CREW does not limit the reuse to a given cluster of neurons but can reuse partial products of any output neuron of an FC layer.

The work in [40] proposes RAPIDNN, an In-Memory Processing (PIM) accelerator inspired by memoization and computation reuse techniques. RAPIDNN uses clustering to statically generate a subset of the most representative inputs and weights, and memoizes all the possible partial product combinations. During inference, the inputs and weights are encoded by accessing lookup tables to select which partial products are accumulated for each output. Note that the partial products selected are always an approximation, which may impact the accuracy of the network. Compared to CREW, we do not lose any accuracy by performing linear quantization, and we require fewer accesses to lookup tables since the integer accumulation with our dataflow is efficiently performed

in a systolic array architecture.

On the other hand, a different approach is to exploit the repetition or similarity found in the input activations of each layer. In Chapter 4, we propose DISC, an accelerator exploiting the similarity between consecutive frames of audio/video to reuse computations of consecutive executions of a given layer. In the same line of research, Deep Reuse [67] and the proposal in [41] detect similarities between vectors of inputs in a given CNN layer to approximate computations. Similarly, Diffy [61] exploits the spatial correlation of the neighboring pixels of computational imaging tasks (CI-DNNs) to perform differential computations. However, these works are focused on exploiting computation reuse in a short time span, i.e. during the same execution of a layer of the DNN. In contrast, our accelerator reuses the computations from one execution of the DNN to the next one. Finally, Delta Networks [66] skip computations of RNNs made of GRUs by considering a neuron to be activated only when the difference (or delta) of the current inputs respect the ones from the previous execution is larger than a threshold. Note that this method introduces more error than DISC and requires retraining of the RNN. In addition, our computation reuse scheme supports any sequence processing application regardless of the DNN type.

To summarize, computation reuse techniques can be applied to a single DNN execution or among different DNN executions. Moreover, computation reuse mechanisms may exploit repeated weights, inputs or both, together with the unique properties of each DNN type.

3 Experimental Methodology

In this chapter, we review the methodology employed throughout this thesis in order to evaluate the different DNN accelerators, with special emphasis in the methods and tools employed to assess the performance, energy consumption and area of the hardware architectures. In addition, we describe the various DNN frameworks, models and datasets that are used for the evaluation of the different techniques proposed in this thesis.

The system platform assumed for all the experiments is an heterogeneous architecture, as shown in Figure 3.1, containing a CPU, a GPU and a DNN Accelerator. These three modules have access to a shared main memory. Regarding the execution flow for all the benchmarks, the CPU is used for pre-processing the DNN inputs and post-processing the final outputs. In the meantime, the DNN accelerator is responsible for performing the DNN inference. In addition, the GPU is used for training the DNNs as required.

In the following sections, we first describe the methodology for modeling and evaluating the DNN accelerator's architecture, starting from the hardware simulation to the gate-level synthesis. Second, we outline the tools used in order to measure the performance and power consumption of CPU and GPU systems. Finally, we introduce the DNN software tools used for the experiments.

3.1 Hardware Acceleration Modeling and Evaluation

Figure 3.2 illustrates the different steps that have been followed in order to develop, model and evaluate the various hardware accelerator designs. From left to right, the flowchart first shows, marked in yellow, the architecture parameters, DNN topology information, memory configuration files, logic circuits, and software DNN implementations, that are required for the simulation tools and system platforms employed in the methodology of this thesis. Next, in red, Figure 3.2 shows

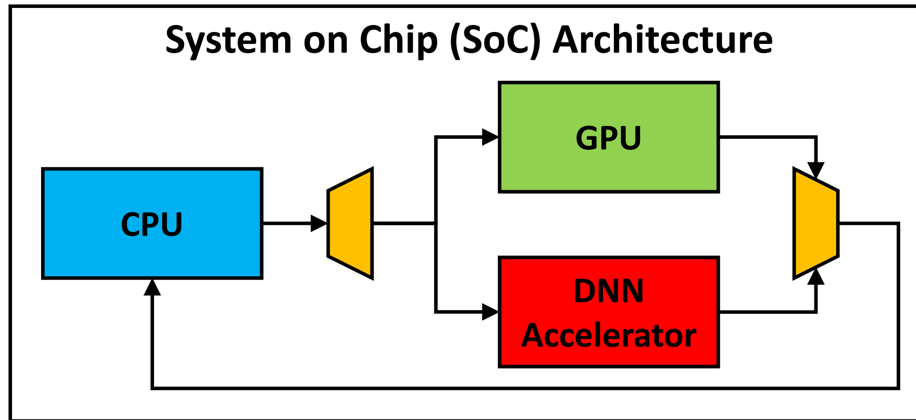


Figure 3.1: The SoC architecture includes a CPU, GPU and a DNN accelerator. The CPU pre-processes the inputs of the DNN. Then, the DNN accelerator performs the inference of the DNN model. Finally, the CPU receives the results of the inference and executes a post-processing step to decode the final outputs. In addition, the GPU is used for DNN training.

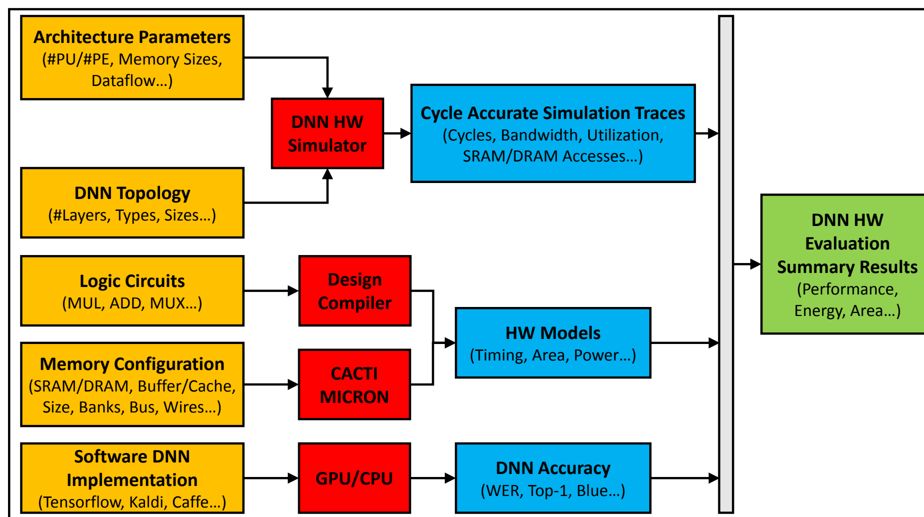


Figure 3.2: Methodology flowchart to model and evaluate the DNN accelerators. The inputs to the simulation tools are marked in yellow, the tools and platforms in red, the partial results from the tools are marked in blue, and the final evaluation summaries in green.

the evaluation tools used. The DNN software implementations are directly executed in a real CPU/GPU to obtain the DNN model accuracy, i.e they are employed as the functional emulator, whereas the other tools perform the timing simulation of the accelerators. Next, the blue rectangles show the partial results collected from each tool. Finally, the partial results are gathered and combined to provide a detailed summary of the performance, energy consumption and area of the DNN accelerators.

In order to evaluate the behavior of the hardware architectures, we have developed and extended two different simulators that accurately model the DNN accelerators presented in Section 2.2. First, we implemented a simulator to model the architecture of DaDianNao described

in Section 2.2.1. Later on, we extended ScaleSim [81], a simulator for neural network accelerators from ARM. ScaleSim models a systolic array architecture like TPU, presented in Section 2.2.2, and supports any kind of neural network, including MLPs, CNNs and RNNs. Both simulators are cycle-accurate, meaning that they count the total execution cycles considering the delay of each component's operations. The simulators are built using Python functions that account for the execution cycles, utilization of the hardware, memory accesses and memory bandwidth. Most parts of the hardware have a deterministic response time given some applications parameters, which greatly simplifies the simulation model.

On the other hand, the combinational logic hardware is implemented in Verilog and synthesized to obtain the area, delay and power using the Synopsys Design Compiler. The efficient IP modules of the DesignWare library are used alongside the technology library of 28/32nm from Synopsys [86]. More specifically, we use the standard low power configuration with 0.78V from the technology library. Design Compiler reports the critical-path delay and the static and dynamic power dissipation of the logic modules. Regarding the dynamic power, it considers a default 50% activity factor for all the signals in the netlist.

Furthermore, we characterize the memory components of the accelerator by obtaining the delay, energy per access and area using CACTI-P [55]. We use the configuration optimized for low power and a supply voltage of 0.78V. To determine the proper frequency of the DNN accelerators, we consider the critical-path-delay and access-time reported by the Design Compiler and CACTI tools, respectively. Then, we take the maximum delay, among the different components, in order to set the cycle time. Finally, the energy consumption of main memory is estimated by using the MICRON power model for LPDDR4 [64].

Regarding the total area, energy and performance estimation of the DNN accelerators, the measurements obtained with the aforementioned tools are combined together with the activity factors, provided by the cycle-accurate simulators, to obtain the dynamic and static energy of each hardware component as well as the execution time. In summary, this methodology allows to quickly run the DNN functionality in a GPU/CPU to asses the accuracy of the model, while performing a fast and accurate prototyping of the hardware accelerator architecture.

3.2 DNN Software Implementations

The different deep learning frameworks used in this thesis (Kaldi, EESEN, Caffe, Tensorflow, PyTorch and OpenSeq2Seq) represent state-of-the-art software implementations for DNNs on CPUs and GPUs. The GPU versions are implemented in CUDA using highly-optimized libraries such as cuBLAS or cuDNN. We have compared the DNN accelerators with the software implementations running on a high performance CPU, an Intel i7 7700K (Kaby Lake), and a modern high-end GPU, an NVIDIA GeForce GTX 1080 (Pascal). The parameters of both CPU and GPU are shown in Table 3.1. We use the RAPL library [94] to collect energy consumption of the CPU, and nvidia-smi (`NVIDIA System Management Interface`) [68] to measure the GPU power dissipation.

Table 3.1: Parameters of the CPU and GPU employed to evaluate the performance and energy consumption of the DNN software implementations.

CPU Parameters		**GPU Parameters**	
CPU	Intel Core i7 7700K	GPU	NVIDIA GeForce GTX 1080
Microarchitecture	Kaby Lake	Microarchitecture	Pascal
Technology	14 nm	Technology	16 nm
Number of Cores/Threads	4/8	Streaming Multiprocessors	20 (2560 threads/SM)
Frequency Base/Boost	4.2/4.5 GHz	Frequency Base/Boost	1607/1733 MHz
L1, L2, L3 Caches	64 KB, 256 KB per core, 8 MB	L1, L2 Caches	48 KB per SM, 2 MB
TDP	91 W	TDP	180 W

Table 3.2: DNNs used in the experimental evaluation of the different proposals of this thesis.

DNN Model	**DNN Type**	**Application**
Kaldi	MLP	Acoustic Scoring
DeepSpeech2 (DS2)	GRU RNN	Speech Recognition
EESEN	LSTM RNN	Speech Recognition
PTBLM	LSTM RNN	Language Modeling
C3D	CNN	Video Classification
AutoPilot	CNN	Self-driving Cars
LeNet5	CNN	Image Recognition
AlexNet	CNN	Image Recognition
GNMT	LSTM RNN	Machine Translation
Transformer	MLP	Machine Translation

3.3 DNN Models and Datasets

An objective of this thesis is to prove that the techniques proposed provide important savings for multiple applications and different DNN architectures. To this end, we evaluate the mechanisms proposed in this thesis on up to ten state-of-the-art DNNs from different application domains, including acoustic scoring, speech recognition, language modeling, image/video classification, self-driving cars and machine translation, as shown in Table 3.2. This section provides a brief description of each DNN. A subset of these DNNs is used to evaluate each technique, and since we have implemented and used different versions of these neural networks for each study, the information corresponding to the composition, size, and accuracy of the DNNs will be provided throughout the remaining chapters.

The suite includes DeepSpeech2 ($DS2$) [4], an RNN implemented in PyTorch [73] that consists of GRU cells, and EESEN [63], an RNN made of LSTM cells, both for end-to-end speech recognition. DNNs for end-to-end speech recognition take as input a sequence of audio frames, where each frame consists of 10 ms of speech and is represented as a set of features (e.g. 120 or 160-element array), and generate the likelihoods of different characters from the target language (e.g. 29, 50 in DS2 and ESSEN respectively) for each frame in the input sequence. In short, DS2 and EESEN are

able to spell the words directly from audio frames. In addition, we include Kaldi [75, 74], an MLP for acoustic scoring, a key task of a speech recognition system, from the popular framework of the same name. Kaldi takes as input a window of 9 frames of speech (current frame, four previous and four next frames), where each frame is represented as an array of 40 features. Then, it generates as output the likelihoods of the 3482 senones, where a senone represents part of a phoneme. We use two different models of DeepSpeech2, $DS2-L$ and $DS2-T$, each having the same composition but trained with a different dataset. DS2-L and Kaldi are trained and evaluated with the Librispeech [70, 71] dataset, while DS2-T and EESEN are trained with the TED-LIUM dataset [80, 18] for spontaneous and noisy speech. Additionally, PTBLM [103] is an LSTM network for language modeling using the Penn Treebank dataset, which is also essential in a speech recognition system.

Furthermore, we employ C3D, a CNN from Facebook [92] that is implemented in Caffe [7]. C3D is a CNN for classifying actions in videos. The input consists of non-overlapping windows of 16 consecutive frames, i.e. the 16 frames for each CNN execution are disjoint. The CNN identifies the action that corresponds to that segment of the video, generating as output the likelihoods for all the possible 101 actions. To evaluate C3D, we use videos from the UCF101 benchmark suite [84, 10]. Moreover, we also use AutoPilot [9], a CNN from NVIDIA implemented in Tensorflow [22, 11] for self-driving cars, that maps raw pixels from a single front-facing camera to steering commands. Autopilot employs as input videos from a single front-facing camera mounted behind the windshield of a car.

In addition, LeNet5 [53] and AlexNet [48] are popular Convolutional Neural Networks (CNNs) for image recognition. LeNet5 is a small CNN to classify written digits. On the other hand, AlexNet is a CNN for classifying color images into 1000 possible classes that range from different animals to various types of objects. To evaluate LeNet5, we use images of digits from the MNIST [52] dataset. Regarding AlexNet [48], we use its second version [47] trained with the ImageNet [39] dataset. These CNNs have been implemented in Tensorflow, and we employ the whole test or validation dataset to measure its accuracy.

Finally, we employ two machine translation networks: $GNMT$ [97] and Transformer [93]. GNMT is an LSTM network for neural machine translation based on the Google Translator, trained using the WMT16 [8] dataset with texts of newspapers from German to English (DE-EN). As described in Section 2.1, the Transformer is a deep MLP with an encoder-decoder architecture that is mainly composed of attention layers. We evaluate the Transformer implementation from the OpenSeq2Seq [49] framework of NVIDIA, using the WMT16 dataset for translating from English to German (EN-DE).

On the other hand, for each DNN and application the standard accuracy metrics are different. Accuracy is reported as Word Error Rate (WER) for speech recognition (lower is better), perplexity for language modeling (lower is better), top-1/top-5 for image/video classification (higher is better), and bilingual evaluation understudy (BLEU) for machine translation (higher is better). Overall, for all the DNNs, we employ the entire evaluation sets from their respective datasets, including several hours of audio/video and a large number of images and texts, to assess the efficiency of the DNN accelerators proposed in this thesis in terms of performance and energy consumption. The workloads employed for the experimental evaluation represent important machine learning applications, and the selected DNNs cover the three main categories: MLPs, CNNs and RNNs.

4

DISC: Differential Input Similarity Computation

In this chapter, we present a technique to improve the inference of DNNs for sequence processing applications. We first provide the analysis of similarity in the inputs of a DNN, and a mechanism to exploit the high degree of input similarity to save computations and memory accesses. Then, we describe DISC, a hardware accelerator implementing the Differential Input Similarity Computation reuse-based DNN inference scheme. Finally, we discuss the experimental results of DISC, and compare them to a CPU, GPU, and a baseline DNN accelerator.

4.1 Input Similarity and Temporal Reuse Analysis

This section analyzes the similarity between inputs of consecutive DNN executions in different layers, and introduces a technique that exploits this similarity to save computations and memory accesses. Table 4.1 shows the DNNs for sequence to sequence applications employed for the analysis, including the model size, baseline accuracy and layer composition. As described in Section 3.3, *Kaldi* is an MLP for acoustic scoring, *EESEN* is an RNN for end-to-end speech recognition, *C3D* is a CNN for classifying actions in videos and *AutoPilot* is a CNN for self-driving cars.

We have evaluated the relative difference of the inputs in consecutive DNN executions for multiple layers. We found that, on average, the relative difference is lower than 14% for the DNNs shown in Table 4.1. Figure 4.1 shows the relative difference in the inputs of the last two FC layers of *Kaldi* for a test audio file. Consecutive inputs tend to be similar, with a relative difference that ranges between 5% and 25%.

In this thesis, we define **input similarity** as the percentage of inputs of a DNN layer that have not changed with respect to the previous execution of the DNN. Despite the small relative differences in the inputs, most of the 32-bit floating-point values are not exactly the same as in

Table 4.1: Deep Neural Networks employed for the analysis of computation reuse. The table only includes Fully-Connected (FC), Convolutional (CONV) and Bidirectional LSTM (BiLSTM) layers, as these layers take up the bulk of computations in DNNs. Other layers, such as ReLU or Pooling, are not shown in the table for the sake of simplicity.

Kaldi [75]: MLP for Acoustic Scoring (18MB)			
Baseline Accuracy: 89.51%, Quantization Accuracy: 89.04%			
Layer	Input Dim	Output Dim	Computation Reuse
FC1	360	360	-
FC2	360	2000	-
FC3	400	2000	75%
FC4	400	2000	66%
FC5	400	2000	56%
FC6	400	3482	66%

EESEN [63]: RNN for Speech Recognition (42MB)				
Baseline Accuracy: 69.03%, Quantization Accuracy: 68.85%				
Layer	In Dim	Out Dim	Cell Dim	Comp. Reuse
BiLSTM1	120	640	320	38%
BiLSTM2	640	640	320	53%
BiLSTM3	640	640	320	56%
BiLSTM4	640	640	320	59%
BiLSTM5	640	640	320	60%
FC1	640	50	-	-

C3D [92]: CNN for Video classification (300MB)				
Baseline Accuracy: 94.86%, Quantization Accuracy: 93.48%				
Layer	Input Dim	Output Dim	Kernel	Comp. Reuse
CONV1	3x16x112x112	64x16x112x112	3x3x3	-
CONV2	64x16x56x56	128x16x56x56	3x3x3	76%
CONV3	128x8x28x28	256x8x28x28	3x3x3	75%
CONV4	256x8x28x28	256x8x28x28	3x3x3	75%
CONV5	256x4x14x14	512x4x14x14	3x3x3	73%
CONV6	512x4x14x14	512x4x14x14	3x3x3	80%
CONV7	512x2x7x7	512x2x7x7	3x3x3	80%
CONV8	512x2x7x7	512x2x7x7	3x3x3	87%
FC1	8192	4096	-	88%
FC2	4096	4096	-	61%
FC3	4096	101	-	54%

AutoPilot [9]: CNN for Self-Driving Cars (6MB)				
Baseline Accuracy: 99.69%, Quantization Accuracy: 99.63%				
Layer	In Dim	Out Dim	Kernel	Comp. Reuse
CONV1	3x66x200	24x31x98	5x5	46%
CONV2	24x31x98	36x14x47	5x5	84%
CONV3	36x14x47	48x5x22	5x5	93%
CONV4	48x5x22	64x3x20	3x3	94%
CONV5	64x3x20	64x1x18	3x3	88%
FC1	1152	1164	-	89%
FC2	1164	100	-	97%
FC3	100	50	-	95%
FC4	50	10	-	82%
FC5	10	1	-	-

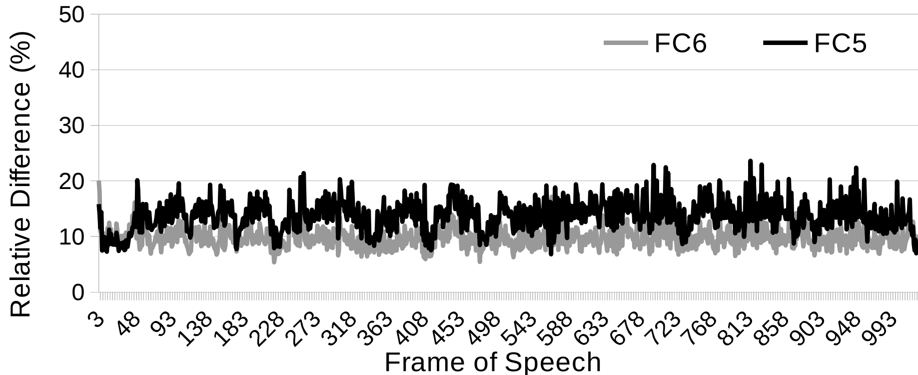


Figure 4.1: Relative difference defined as the Euclidean distance between current and previous input vectors, divided by the magnitude of the input vector in the previous execution.

the previous execution and, therefore, input similarity is small according to this strict definition. However, input quantization can be employed, as discussed in Section 2.3.1, as an effective solution to increase the degree of similarity, exposing more redundant computations with a negligible impact on the accuracy. In this work, we apply uniformly distributed linear quantization to the inputs of the FC, convolutional and recurrent layers, which represent most of the execution time for typical DNNs. Then, we analyze the degree of similarity between the inputs of consecutive DNN executions, and the accuracy loss due to the quantization.

In order to exploit the similarity in the inputs, the key observation is that, if the previous output of a neuron is saved, inputs that have not changed with respect to the previous execution do not require any computation, since their contribution to the output of the neuron has not been modified. In case an input changes, the previous output can be corrected according to the following equation:

$$z_o^{'} = z_o + \sum_{i=1}^{N} \Big((c_i^{'} - c_i) * W_{io} \Big) \tag{4.1}$$

For each layer, we first compute the difference of the previous quantized input and the current one ($d_i = c_i^{'} - c_i$). For each output neuron o, the weight associated to input i is multiplied by its corresponding difference d_i and added to the previous output, only if d_i is different from zero. Otherwise, nothing needs to be done. If the percentage of inputs that remain unmodified is high, i.e. if the input similarity is high, then computing the output of a neuron in this manner requires significantly fewer computations and memory accesses. We refer to this technique as Differential Input Similarity Computation (DISC).

In this thesis, we define the **degree of computation reuse** as the percentage of computations that can be reused from the previous DNN execution, because their input operands have not been modified. Computation reuse and input similarity are highly related, since in case an input is the same as in the previous execution the previous results are reused avoiding all the computations associated with that input.

We analyze the input similarity between frames of the four DNNs, shown in Table 4.1, using multiple configurations: number of clusters, range of the inputs and layers where the quantization is applied. First, we analyze the accuracy loss when applying linear quantization to different layers of the DNN. We start by quantizing the inputs of all the layers using 32 clusters (i.e. 5 bits per input), and observe that the accuracy is highly affected, mainly due to the first layers as their errors are propagated across the entire DNN. Due to this accuracy loss, we selectively apply the quantization layer by layer starting from the last layer, except for *EESEN* and *AutoPilot* where we start from BiLSTM5 and FC4 respectively, since the last FC layers of these networks are fairly small. After testing the accuracy of the DNNs when the quantization is applied to the last layer, we test the accuracy when the technique is applied to the last two layers, and so on until finding the optimum number of layers where the quantization can be applied with negligible loss in accuracy.

After selecting the layers where the quantization is applied, we analyze the impact of the number of clusters, using configurations with 8, 12, 16 and 32 clusters for the linear quantization. We find that linear quantization with 8 and 12 clusters introduces significant accuracy loss in all the DNNs. The configuration with 16 clusters achieves a small accuracy loss in *Kaldi* and *EESEN*, whereas 32 clusters are required in *C3D* and *AutoPilot* to maintain accuracy. When using quantization, the smaller the number of clusters the higher the similarity, since the inputs are constrained to a smaller set of values, but the larger the accuracy loss since input errors are increased. We find that the configuration with 16 clusters provides the best trade-off between similarity and accuracy loss for *Kaldi* and *EESEN*, whereas the configuration with 32 clusters provides the best results for the CNNs.

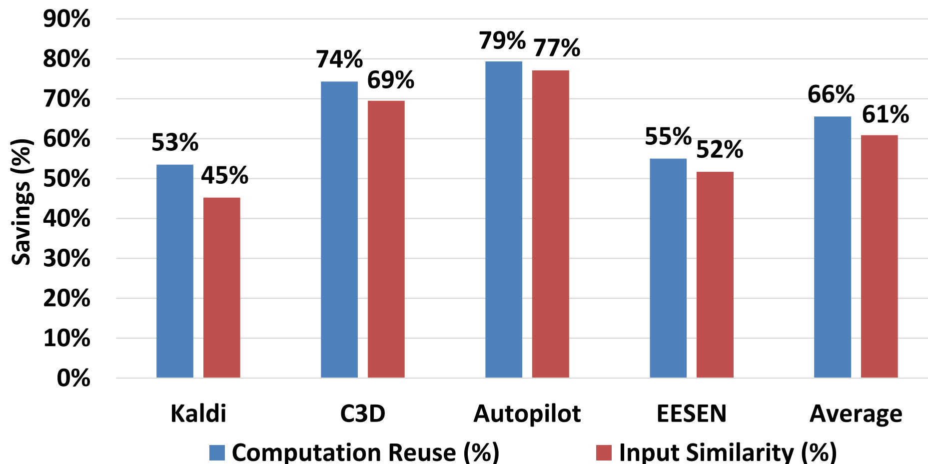


Figure 4.2: Input similarity and computation reuse for various DNNs.

Table 4.1 includes the accuracy for the baseline DNNs and the accuracy when using linear quantization of the inputs. Furthermore, it provides the degree of computation reuse for the layers where linear quantization is applied. In *Kaldi*, quantization is applied to the last four FC layers, obtaining a degree of computation reuse that ranges between 56% (FC5) and 75% (FC3). The accuracy loss is as small as 0.47%. For *C3D*, quantization is used in all the FC and convolutional layers except for CONV1, achieving a computation reuse between 54% (FC3) and 88% (FC1) with an accuracy loss of 1.38%.

On the other hand, quantization is applied in all the layers except the last FC for *EESEN* and *Autopilot*. Note that the number of neurons in the output layers of these DNNs is extremely low and, therefore, the potential for avoiding computations is fairly small compared to the entire DNN. In *AutoPilot*, the accuracy loss is 0.06% and the degree of computation reuse is bigger than 80% in most of the layers. Regarding *EESEN*, the potential for avoiding computations is smaller, but still more than 50% of the computations can be reused across consecutive DNN executions for most of the recurrent layers, with an accuracy loss of just 0.18%.

Figure 4.2 shows the global degree of computation reuse and the input similarity for the various DNNs. When using linear quantization, more than 50% of the computations can be reused across DNN executions in all the DNNs. On average, 61% of the inputs remain unmodified with respect to the previous DNN execution and 66% of the computations can be avoided. Therefore, we can conclude that DISC exhibits a high potential for avoiding computations for different DNN architectures (MLPs, CNNs and RNNs) and different applications (acoustic scoring, speech recognition, video classification and self-driving cars).

4.2 DISC Accelerator

This section describes how the high degree of input similarity and computation reuse, characterized in Section 4.1, can be exploited to improve the performance and energy efficiency of DNN inference. First, we present the main hardware components of the DISC accelerator. Next,

we describe how FC, convolutional and recurrent layers are executed in the accelerator using the computation reuse scheme.

4.2.1 DISC Architecture

In this work, we extend the baseline DaDianNao-based DNN accelerator, presented in Section 2.2.1, to support DISC, and take advantage of the computation reuse found in MLPs, CNNs and RNNs due to the temporal input similarity. The design of the DISC accelerator exploits the high degree of similarity between consecutive inputs to save computations and memory accesses. The main hardware components of DISC are still the same as in the baseline accelerator, illustrated in Figure 2.6, that is, multiple tiles of Processing Units (PU). Each PU includes a Compute Engine (CE) to perform computations, an eDRAM memory to store the weights (i.e. the Weights Buffer), an SRAM memory to store inputs and outputs (i.e. the I/O Buffer), the Data Master to fetch and dispatch the corresponding weights and inputs, and finally the Control Unit (CU) to orchestrate all the execution. In addition, the CU stores the centroids of the clusters employed for the quantization of the inputs of each layer.

The extra hardware required for the computation reuse scheme is modest, as the components already available for DNN computation can also be employed for most of the operations. For instance, the CE is also employed to quantize and compare the inputs in the reuse scheme and, hence, no extra hardware is required for quantization nor for checking the input similarity. In addition to small changes in the control unit, DISC only requires a larger I/O buffer to store the quantized inputs, and the outputs of each layer that will be reused in the next DNN execution.

The vast majority of computations for MLPs, CNNs and RNNs come from FC, convolutional and recurrent layers respectively. Next subsections provide more insights on how these layers are executed in the DISC accelerator with the computation reuse technique.

4.2.2 DISC for FC Layers

The FC layer computes the dot product of the inputs and the weights of each neuron (see section 2.1.1). Figure 4.3 illustrates the execution of an FC layer in the DISC accelerator when using the reuse scheme. Note that the first execution is different, as it computes the DNN from scratch whereas subsequent executions reuse previous results. The top of Figure 4.3 shows how the weights for an FC layer are interleaved in the Weights Buffer. That is, the first weight of every neuron is stored first, then the second weight of every neuron and so on. This layout simplifies the implementation of the computation reuse scheme, as it is simpler to locate all the weights that operate with a given input in case they have to be skipped or accessed to perform corrections.

The I/O Buffer is organized in three different areas. The first area stores the indices, i.e. the quantized inputs, as these values are required to verify whether an input remains unchanged with respect to the previous execution. The second area stores the outputs of all the layers where the computation reuse is exploited, to be later reused by the next DNN execution. Finally, the third area is used as a temporal storage for the inputs/outputs of layers where the reuse scheme is not

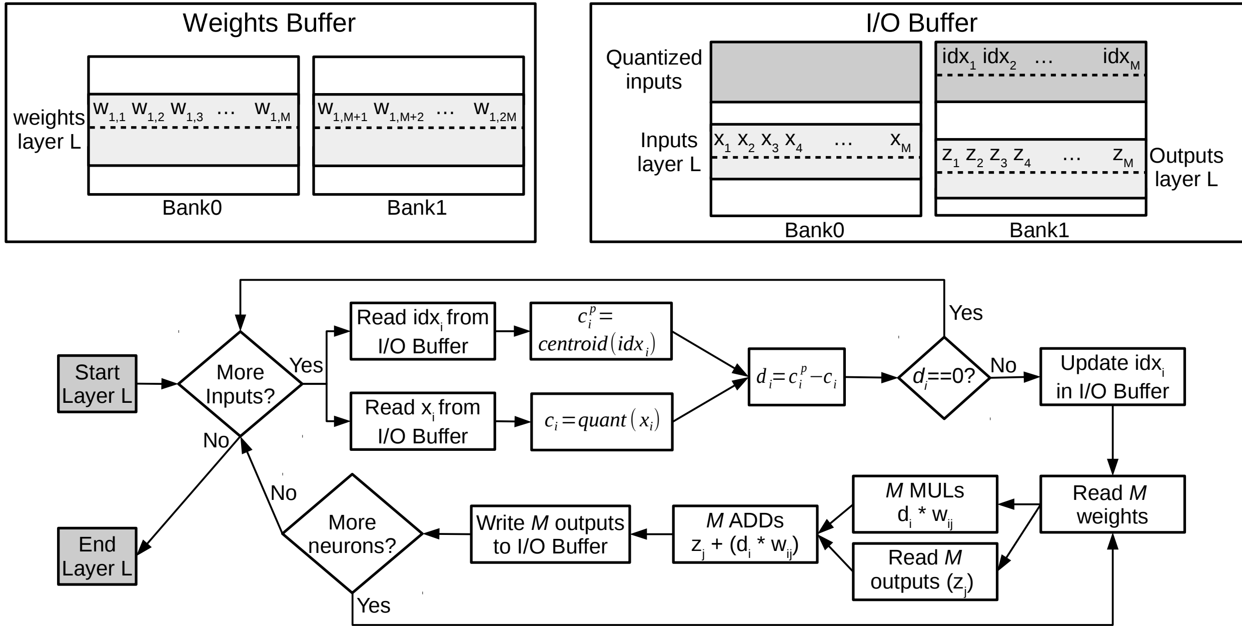


Figure 4.3: FC execution in the DISC Accelerator.

applied. The first two areas are extra storage required to exploit computation reuse, Section 4.4 shows that they represent a very small overhead.

The first execution of an FC layer is as follows: initially, the accelerator reads and quantizes the first input. In parallel, M weights of different neurons are read from the eDRAM. The index of the input quantization is stored in the I/O Buffer to be consumed in the next execution. Then, the accelerator performs M MULs of the input by the weights, followed by M ADDs to accumulate the result of each output neuron. Outputs are then stored in the I/O Buffer, to be used by the next layer and to be reused by the same layer in the next DNN execution. The accelerator is pipelined so, in the same cycle, the I/O Buffer reads one input, the Data Master reads M weights from memory and the CE performs M multiplications and M additions.

Subsequent executions of the FC layer employ the computation reuse scheme. The flowchart in Figure 4.3 illustrates the execution of an FC layer with computation reuse. Initially, the first input and its corresponding index from the previous execution are read from the I/O Buffer. Next, the current input is quantized, while the index is used to fetch the corresponding centroid. The current quantized input is subtracted from the centroid. If the outcome is zero, the input is ignored and all the corresponding computations and memory accesses are skipped. In case the input has changed, the index is updated in the I/O Buffer and all the neurons are corrected using the weights associated to that input. Since the accelerator stores the outputs of the previous execution, corrections can be performed by removing the contribution of the previous input and adding the contribution of the current input. To this end, M weights from different neurons are fetched from memory. Then, M MULs compute the value that has to be added, multiplying the weights by the difference between the current input and the previous one. At the same time, M previous outputs are read from the I/O Buffer. Finally, the adders are employed in order to update the outputs, and the corrected

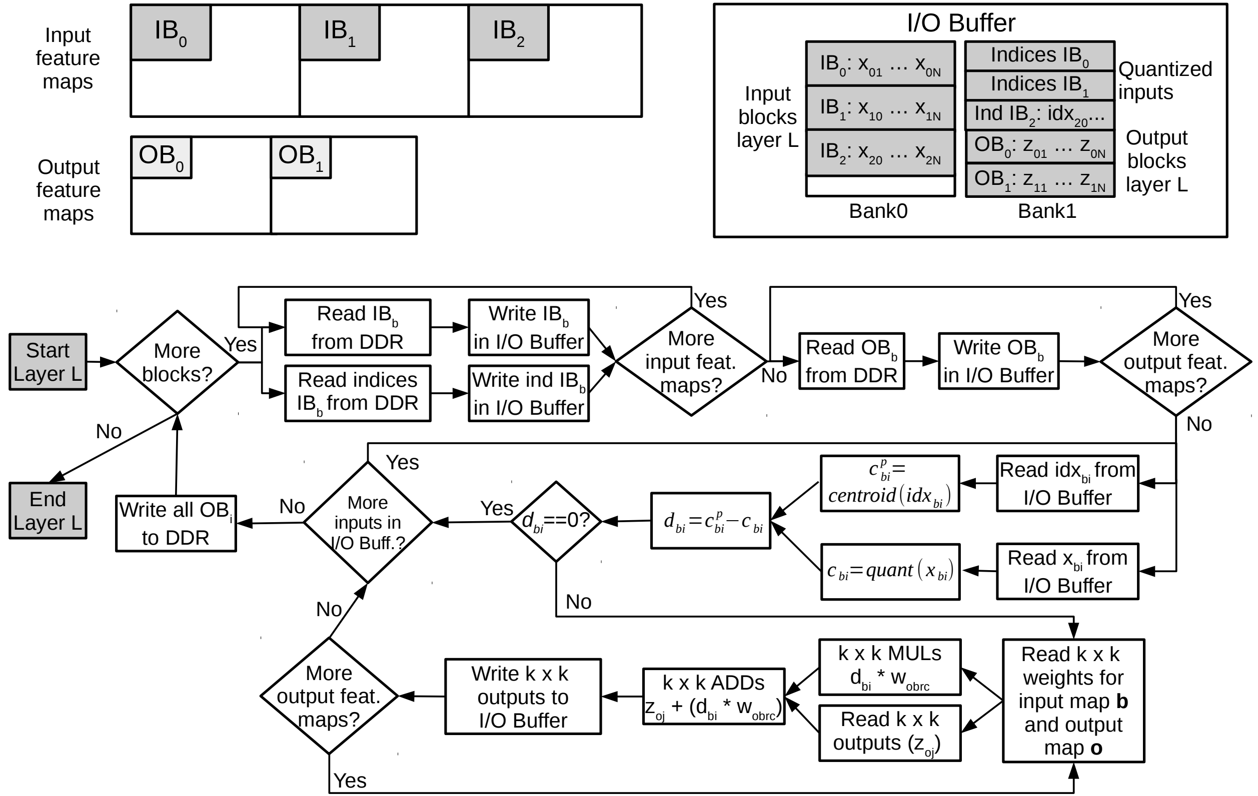


Figure 4.4: Execution of a convolutional layer in the DISC accelerator. IB_b means Input Block from input feature map b, whereas OB_o means Output Block from output feature map o.

results are updated in the I/O Buffer. This process is repeated until all the neurons are corrected for each modified input.

4.2.3 DISC for CONV Layers

In a convolutional layer, the weights of a kernel are reused among different inputs of the same execution. Assuming a kernel of size $k \times k$ and f output filters, $k \times k \times f$ multiplications and additions are required for each input in the conventional scheme. Figure 4.4 illustrates the convolutional layer execution when using the DISC accelerator. The dimensionality of the convolutional layers may be quite large, as shown in Section 4.1. We employ a blocking scheme to reduce the on-chip storage requirements. The I/O Buffer only stores one block for each input feature map and one block for each output feature map, the block size being significantly smaller than the dimensions of the feature maps. In addition, the computation reuse scheme requires storage for the indices of the inputs in the previous DNN execution.

For the first execution, the DISC accelerator loads one block of inputs and processes one input of the block at a time by multiplying it by the weights of the kernels that correspond to that input, and adding the results to the corresponding output neurons. The indices of the quantization and the output results of the convolutional layer are stored in the I/O Buffer and sent to main memory

at the end, to be later used by the next CNN execution.

The flowchart in Figure 4.4 illustrates the subsequent CNN executions exploiting computation reuse. Initially, for each input channel, an input block and its corresponding quantized indices are read from main memory and written to the I/O Buffer. Then, the corresponding output block of each output feature map is read from main memory. Once the current blocks are loaded on-chip, the first input and its corresponding index are read from the I/O Buffer. Then, the input is quantized, while the index is used to access the centroid. The current quantized input is subtracted from the centroid. If the outcome is zero, the input is skipped, avoiding all the corresponding computations. In case the input has changed, the index is updated in the I/O Buffer and the output neurons that operate with that input are corrected. Assuming a kernel size of $k \times k$ with stride one, the accelerator will update $k \times k$ output neurons for each filter. To this end, the accelerator reads $k \times k$ weights, multiplies them by the difference between the current and previous input, while loading $k \times k$ previous outputs from the I/O Buffer. Finally, the adders are employed to update the outputs for the current execution, and the updated results are written in the I/O Buffer. The indices and output blocks are written back to main memory at the end of each input block. This process is repeated until all the neurons have been updated for each modified input.

4.2.4 DISC for Recurrent Layers

A recurrent layer contains one (unidirectional) or two (bidirectional) LSTM/GRU cells that are recurrently executed for each element in the input sequence. An LSTM/GRU cell, described in Section 2.1.3, consists of multiple gates implemented as independent FC layers. Therefore, the execution of a recurrent layer mimics the behavior of FC layers in the DISC accelerator, i.e. each gate is processed as described in Section 4.2.2. However, there are a few differences that make recurrent layers more amenable for the computation reuse technique. First, each recurrent layer is executed back-to-back for every input in the sequence before proceeding to the next layer (see Figure 2.4a). Therefore, RNNs only require extra storage for the inputs/outputs of one layer, whereas MLPs and CNNs require extra storage for all the layers where the computation reuse technique is applied. In other words, temporal locality of the redundant computations is higher in RNNs. Second, all the gates (or FC layers) in one LSTM/GRU cell share the same inputs. Hence, we only compare the inputs once with the previous values and, in case an input remains unmodified, computations and memory accesses are avoided in all the gates.

4.3 Evaluation Methodology

The experimental evaluation of DISC is performed as described in Chapter 3 to asses the performance, energy consumption and area of the hardware accelerator. We have extended the cycle-accurate simulator modeling the baseline architecture of DaDianNao, described in Section 3.1, to accurately model the DISC accelerator presented in Section 4.2, including the computation reuse scheme. Table 4.2 shows the parameters for the experiments. We model an accelerator with four tiles, with a total of 128 FP adders and 128 FP multipliers (32 per tile). The DISC accelerator includes 36 MB of eDRAM for the Weights Buffer (9 MB per tile), which is enough to fit on-chip

Table 4.2: Parameters for the DISC accelerator.

Technology	32 nm
Frequency	500 MHz
# Tiles	4
# 32-bit multipliers	128
# 32-bit adders	128
Weights Buffer	36 MB
I/O Buffer Size	1152KB (Baseline) / 1280KB (DISC)

two of the evaluated DNNs (i.e. *Kaldi* and *Autopilot*), whereas for *C3D* it can store most of the convolutional weights and for *EESEN* it stores the weights of one layer at a time. Finally, the I/O Buffer is sized to fit the blocked input of the *C3D* DNN, which requires 1152 KB for the baseline DaDianNao-based accelerator and 1280 KB when using the reuse scheme of DISC, as extra storage is needed for the input indices (see Figure 4.4). We employ a block size of $16 \times 16 \times 1$ for the CNNs, as we found that it provides a good trade-off between on-chip storage requirements and memory bandwidth usage. Regarding main memory, we model an LPDDR4 of 4 GB with a bandwidth of 16 GB/s (dual channel).

In order to prove that DISC provides important savings for multiple sequence processing applications and different DNN architectures, we use the four state-of-the-art DNNs shown in Table 4.1 from different application domains, including acoustic scoring, speech recognition, video classification and self-driving cars. For all the DNNs, we employ several hours of audio/video to assess the accuracy and performance of the computation reuse scheme. These workloads represent important machine learning applications, and the selected DNNs cover the three existing approaches for sequence processing: MLPs, CNNs and RNNs.

Besides comparing the DISC accelerator with the baseline DaDianNao-based DNN accelerator, we also provide a comparison of DISC with the DNN software implementations running on a high performance CPU and a high-end GPU using the platforms and tools described in Section 3.2.

4.4 Experimental Results

This section evaluates the performance and energy consumption of the DISC accelerator implementing the computation reuse scheme. First, we present the speedups and energy savings achieved by DISC, described in Section 4.2, when implemented on top of the DaDianNao-based DNN accelerator, presented in Section 2.2.1. Next, we characterize the memory overheads of the computation reuse scheme. Finally, we compare the DISC accelerator with the DNN software implementations running on a modern CPU and GPU.

Figure 4.5 shows the speedups achieved by DISC over the baseline DNN accelerator. The computation reuse scheme provides consistent speedups for the four DNNs that range from 1.9x (*Kaldi*) to 5.2x (*Autopilot*), achieving an average performance improvement of 3.5x. The reduction

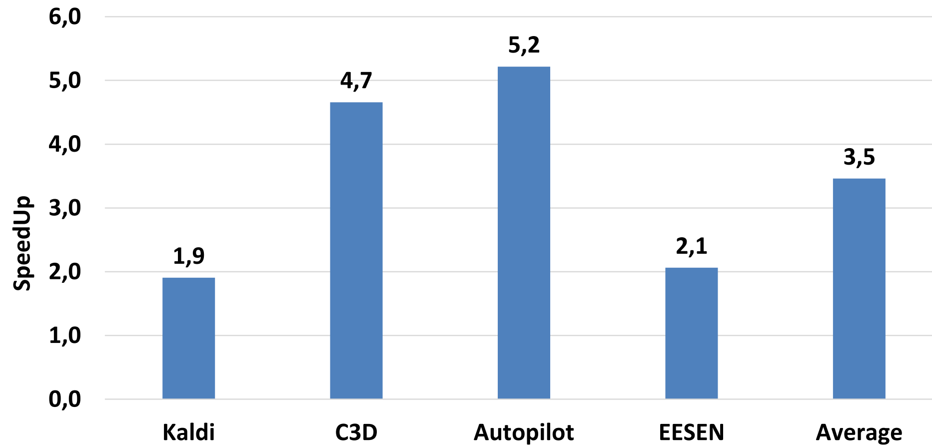


Figure 4.5: Speedups achieved by DISC for each DNN. Baseline configuration is the DaDianNao-based DNN accelerator without any computation reuse technique.

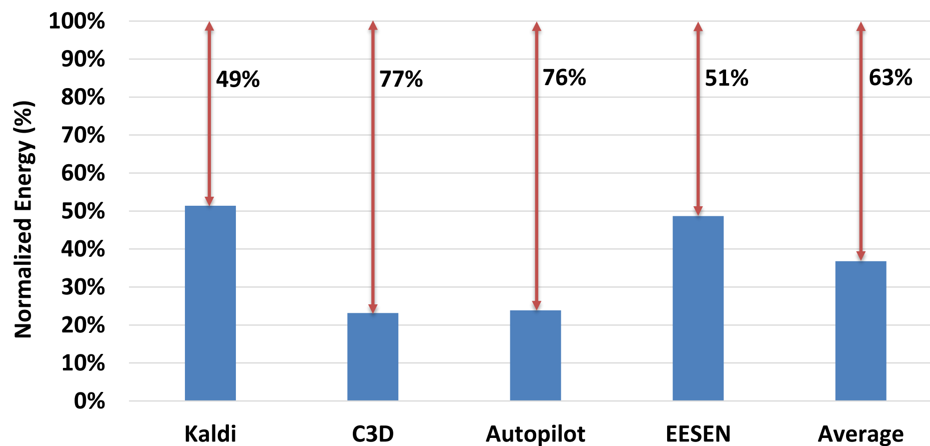


Figure 4.6: Normalized energy of DISC for each DNN. Baseline configuration is the DaDianNao-based DNN accelerator without any computation reuse technique.

in execution time is due to reusing previously computed results, since inputs that remain unmodified with respect to the previous DNN execution do not require any computation or memory access. Furthermore, the overhead of performing the quantization and comparing the current input with the previous one is fairly small, since it is performed per input and not per connection. As one input is normally used in thousands of neurons, performing a comparison to detect that the input has not changed can save thousands of computations and memory accesses. *C3D* and *Autopilot* exhibit the highest degree of computation reuse (see Figure 4.2) and, hence, they obtain the largest performance improvements.

The average power consumption of the DISC accelerator is lower than $3.5W$. Figure 4.6 reports the normalized energy of DISC over the baseline DNN accelerator. On average, the technique reduces the energy consumption of the accelerator by 63%. The energy savings are well correlated with the degree of input similarity and computation reuse reported in Figure 4.2. These energy savings are due to two main reasons. First, dynamic energy is reduced due to the savings in

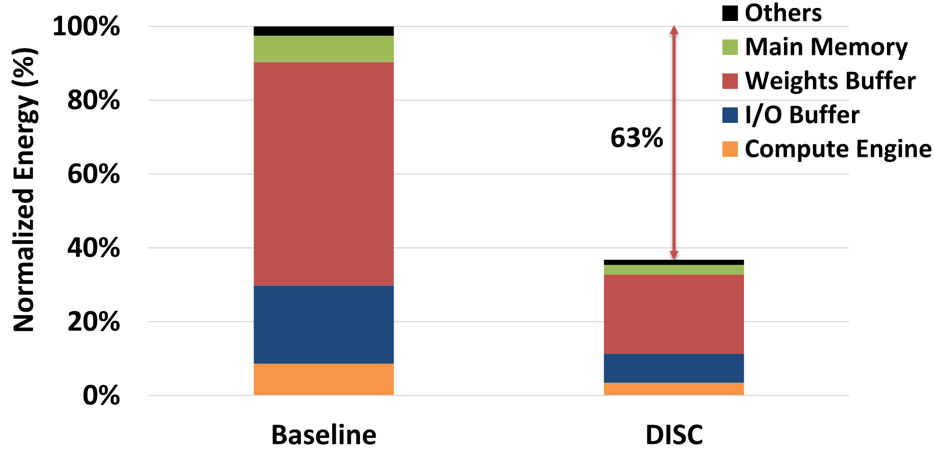


Figure 4.7: Energy breakdown for the baseline DaDianNao-based DNN accelerator and the DISC accelerator using the computation reuse scheme.

computations and memory accesses. Second, the performance improvements shown in Figure 4.5 provide a reduction in static energy. Again, *C3D* and *Autopilot* obtain the largest benefits, achieving a reduction of 77% and 76% in energy respectively.

Figure 4.7 shows the energy breakdown, including the percentage of energy consumed by each hardware component for the baseline DaDianNao-based DNN accelerator (Baseline) and the accelerator implementing the Differential Input Similarity Computation reuse-based scheme (DISC). The figure shows aggregated results for the four neural networks. As it can be seen, the eDRAM for storing the weights consumes most of the energy in both accelerators. The energy savings achieved by DISC are significant in all the components, and are especially large in the on-chip eDRAM memory, since the reuse scheme provides significant savings in memory accesses for fetching the synaptic weights. On the other hand, the reduction in the number of computations also results in smaller energy in the Compute Engine. Note that the energy required for performing the quantization and comparing the current and previous inputs for every DNN execution is also included in the Compute Engine energy consumption. Regarding the I/O Buffer, its energy consumption is significantly reduced since inputs that remain unmodified do not require any access to this on-chip memory.

Table 4.3 reports the memory overheads of the DISC accelerator. For *Kaldi* (MLP), the reuse scheme requires extra storage in the I/O Buffer for the indices (quantized inputs) and outputs of the different layers (see Figure 4.3). For *EESEN* (RNN), the accelerator requires storage for the inputs/outputs of the four gates of one LSTM cell in the I/O Buffer. Note that in the worst case only 66 KB of on-chip storage are required for *Kaldi* and *EESEN*, whereas no additional storage is used in main memory. Regarding the CNNs (*C3D* and *Autopilot*), the I/O Buffer requires additional storage for the indices (see Figure 4.4). The inputs/outputs of each layer are stored in main memory for the CNNs, resulting in a small increase (around 10%) in storage requirements as reported in Table 4.3. We provision the I/O Buffer with enough capacity for the largest DNN (*C3D*): 1152 KB in the baseline accelerator and 1280 KB with DISC. In addition, 1.25 KB are used for the table of centroids. The overall overhead in area of the accelerator is less than 1%, as it increases from 52 mm^2 to 53 mm^2.

Table 4.3: Memory overheads of the DISC accelerator implementing the computation reuse scheme.

	I/O Buffer (KB)		**Main Memory (MB)**	
DNN	**Baseline**	**DISC**	**Baseline**	**DISC**
Kaldi	27	66	18	18
C3D	1152	1280	397	443
Autopilot	160	176	6.6	7.2
EESEN	8	13	42	42

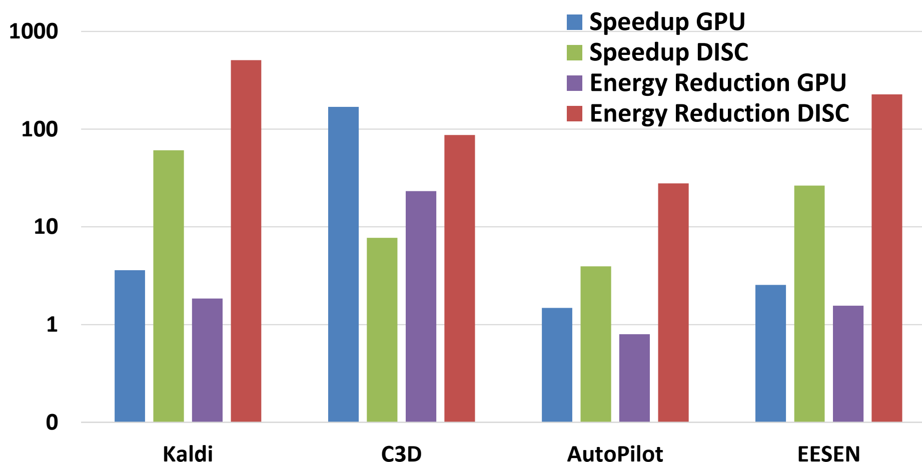


Figure 4.8: Speedup and energy reduction of DISC compared to a high-end GPU. Baseline configuration is the Intel i7 7700K CPU.

The simulator and power model include all the overheads due to the computation reuse technique implemented in DISC: extra accesses to a larger I/O Buffer, extra memory bandwidth usage and memory storage, accesses for fetching centroids and computations for applying linear quantization. As it is shown in Figure 4.5 and Figure 4.6, these overheads are negligible in comparison to the savings in computations and memory accesses, and the net result is an improvement of 9.5x in energy-delay (2.7x in energy and 3.5x in delay).

Finally, Figure 4.8 provides a comparison of the accelerator implementing the Differential Input Similarity Computation reuse-based scheme (DISC) with a modern CPU (i7 7700K) and a high-end GPU (GTX 1080). Regarding the speedups, DISC outperforms both CPU and GPU in all the DNNs except in *C3D*. *C3D* is the largest DNN and it achieves close to peak performance in the GPU. Note that the GTX 1080 exhibits the highest peak performance, as it includes 2560 FPUs at 1.82 GHz versus the 256 FPUs at 500 MHz of the DISC accelerator. However, the GPU also exhibits the highest power dissipation, bigger than 200 W for *C3D*. Regarding energy consumption, the DISC accelerator provides large energy reductions with respect to both CPU and GPU for all the DNNs. On average, the energy reduction over the i7 7700K and the GTX 1080 is 213x and 115x respectively.

4.4.1 Reduced-precision Accelerator

Reduced-precision fixed-point arithmetic has become popular in DNN accelerators [43] as described in Section 2.3. In this work, we evaluate the computation reuse technique on top of an accelerator that employs 32-bit floating-point arithmetic (see Section 4.3). Nevertheless, DISC also provides large improvements in performance and energy consumption when using reduced-precision arithmetic. Note that the scheme is already using quantized inputs and, hence, the input similarity and computation reuse should not be significantly affected when changing from 32-bit floating-point to 8-bit fixed-point. The use of reduced-precision should benefit both the baseline system and the DISC accelerator in the same or similar amount. Therefore, we expect relative numbers, such as speedup or normalized energy, to be very similar to the ones reported in this work when using a reduced-precision accelerator as the baseline. To support this claim, we modified the baseline DNN accelerator to employ 8-bit fixed-point arithmetic, using 8 bits to represent both weights and inputs, and we evaluated its performance for the Kaldi DNN. We found that the input similarity increases from 45% (32-bit floating-point baseline) to 52% (8-bit fixed-point baseline), whereas we obtained a large amount of computation reuse of 58%. DISC provides 1.8x speedup and 45% energy savings for Kaldi DNN when implemented on top of a reduced-precision accelerator. Furthermore, we also verified that the accuracy loss is negligible (significantly lower than 1%).

5

CREW: Computation Reuse & Efficient Weight Storage

In the previous chapter, we presented a reuse-based technique to improve the inference of DNNs for sequence processing applications. We exploited the temporal input similarity of consecutive DNN executions to reduce the number of computations and memory accesses of DNN accelerators. In this chapter, we propose a computation reuse scheme that exploits the spatial locality of repeated weights, to further optimize the inference of DNNs composed of fully-connected (FC) layers. We first present the analysis of weight repetition in FC layers. Then, we describe the mechanism to reuse partial products by leveraging the small number of unique weights of each input neuron on each FC layer. Next, we present CREW, a hardware accelerator implementing this Computation Reuse and Efficient Weight Storage mechanism of partial products, as well as an enhanced dataflow to efficiently execute FC layers on a TPU-like accelerator. Finally, we discuss the evaluation and experimental results of CREW, and provide a comparison against UCNN, a state-of-art computation reuse technique.

5.1 Unique Weights in FC Layers

FC layers have increased their size over time from thousands to millions of parameters to provide better accuracy on complex applications. However, the number of unique weights in each neuron is decreasing. Prior works observed the same effect in the filters of the CNNs [34]. This reduction in unique weights is due to the successful methods to compress the DNN model size such as pruning [26, 102] and quantization [108, 104]. As described in Section 2.3.1, linear quantization [96] is a highly popular technique to map a continuous set of values to a discrete set with a negligible impact in accuracy. One of the side effects of quantization is that it significantly increases the number of repeated weights. In this work, we apply uniformly distributed linear quantization to the weights of the FC layers. Then, we analyze the number of unique weights per input neuron.

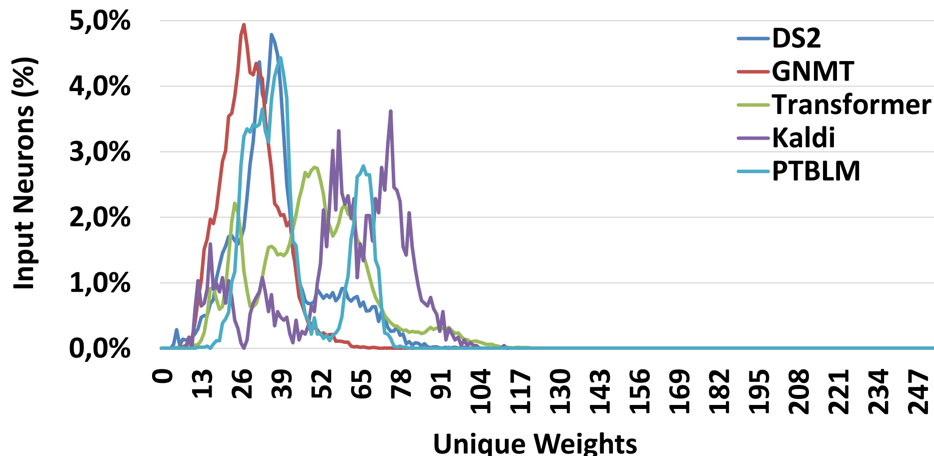


Figure 5.1: Histograms of the unique weights per input neuron from all the FC layers of DS2, GNMT, Transformer, Kaldi and PTBLM.

Typically, DNNs are quantized to 8 bits per weight without any impact in accuracy [43]. We refer to the number of unique weights in a layer as UW, and hence with 8-bit weights $UW \leq 2^8 = 256$. Weight repetition in a fully-connected layer is guaranteed as long as $UW < N * M$, being N and M the number of input and output neurons of the FC layer respectively. This condition is commonly met in modern DNNs. For instance, all the FC layers of the Transformer [93] have more than one million weights.

Weight repetition can be exploited in two different ways: Factorization and Memoization. UCNN [34], a state-of-art accelerator, implements a computation reuse technique by exploiting the factorization of repeated weights. UCNN focuses on optimizing the convolutional layers by grouping and adding the inputs that belong to the same unique weights in a given convolutional window and filter, performing each unique weight multiplication just once per factorization group. UCNN implements FC layers as convolutions, where each output neuron is treated as a $1 \times 1 \times N$ convolutional filter, i.e. the number of channels is the number of inputs (N) in the FC layer. Due to the factorization, inputs are spread out irregularly within each filter and, hence, an indirection table is required to fetch the inputs for each unique weight. Since there are N inputs, the size of each index in the indirection table is $log_2 N$. Note that for FC layers in modern DNNs, $log_2 N$ may be larger than 8 bits. To sum up, UCNN reduces the number of computations in an FC layer. However, it requires $N \times M$ indexes of size $log_2 N$ bits, which may result in a model larger than the original one. Therefore, its indexing overheads to apply factorization on FC layers are not negligible and hinder the benefits of the computation reuse. Not surprisingly, we obtained modest speedups and energy savings when applying UCNN on FC layers as we show in Section 5.5.

On the other hand, we observe that the number of unique weights per input neuron is relatively low. Therefore, we can memoize partial products between inputs and unique weights to largely reduce the number of multiplications. The number of indexes to the partial results required is still $N * M$, but the size of each index will depend on the number of unique weights of its related input neuron. Therefore, if the condition in Equation 5.1 is fulfilled, where q is the number of bits used to quantize and UW_i is the number of unique weights of a given input neuron i, the size of the

*Table 5.1: **UW/I** shows the average number of unique weights per input neuron. **MULs** is the percentage of multiplications of inputs by unique weights with respect to the total number of multiplications in the original model.*

DNN Model	UW/I	MULs (%)
DS2	38	1.67
GNMT	29	0.57
Transformer	49	3.77
Kaldi	59	2.95
PTBLM	43	0.71

model along with the associated memory accesses can be reduced.

$$UW_i \leq 2^{q-1}, \forall i \in \{1, \ldots, N\} \tag{5.1}$$

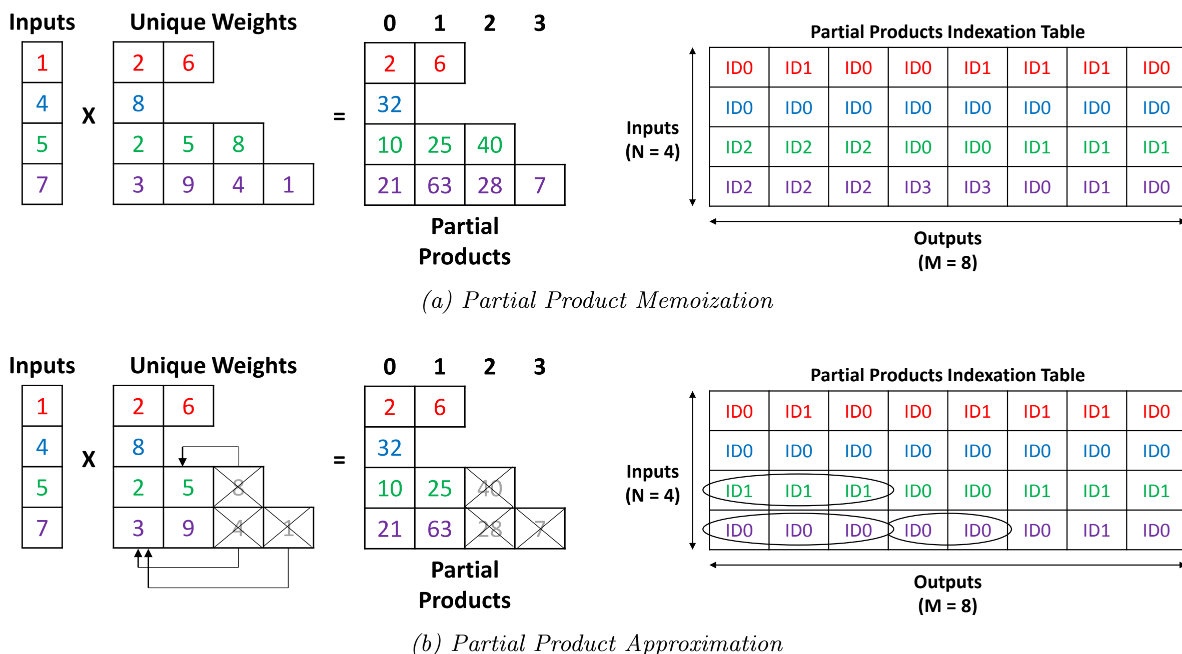
CREW, our proposed computation reuse solution for FC layers, is based on the key observation that, on average, more than 80% of the input neurons among different FC layers of a representative set of DNNs are multiplied by less than 64 unique weights as shown in Figure 1.4. Figure 5.1 provides the histogram of unique weights per input neuron for each DNN listed in Table 5.4. We can see that DS2 and GNMT have a similar distribution of unique weights centered around 34, while the rest have a more dispersed distribution centered around 50. In all the cases, all the input neurons have less than 128 unique weights. Table 5.1 shows the average number of unique weights per input neuron (**UW/I**). On average, each input neuron from the evaluated set of DNNs has 44 unique weights when applying an 8 bit quantization, out of the 256 potential weights per input. Furthermore, Table 5.1 shows the percentage of multiplications of inputs by unique weights (**MULs**). As it can be seen, only between 0.57% and 3.77% of the multiplications are required to compute the FC layers.

5.2 Partial Product Reuse

This section describes how the low number of unique weights per input in FC layers, characterized in Section 5.1, can be exploited for efficient DNN inference. We first present the partial product memoization mechanism (Section 5.2.1), which saves multiplications and storage by leveraging repeated weights in each input, for any FC layer of the network. We then present an optimization, called partial product approximation (Section 5.2.2), to further reduce multiplications and storage at the expense of a minor accuracy loss.

5.2.1 Partial Product Memoization

Our computation reuse method consists of two main steps. First, it detects statically the unique weights of each input for all the FC layers after quantization. Then, it dynamically performs the



(a) Partial Product Memoization

(b) Partial Product Approximation

Figure 5.2: Example of partial product memoization (a) vs partial product approximation (b) using a small FC layer.

partial products between the inputs and their unique weights, and memoizes the partial results. These partial products will be retrieved later to perform the final accumulations of each output neuron. We refer to this scheme as Computation Reuse and Efficient Weight Storage (CREW).

The first step of the proposed approach is to detect the unique weights per input on each FC layer. This step can be done offline since the weights are statically known after training. The unique weights will be stored in a table in the same order as the inputs, that is, the unique weights of the first input will be stored first, next the unique weights of the second input and so on. There is an extra table storing the number of unique weights of each input. Then, we generate a table with the corresponding indexes to the partial products. Note that we are exploiting the partial products of repeated weights in the same spatial position along the M-axis as shown in Figure 1.5. Therefore, the positions of the partial products are statically determined. Each FC layer will have an index table with $N * M$ entries. The indexes of each row N may have a different bit length depending on the number of unique weights of the corresponding input. For example, if the number of unique weights of a given input neuron is 45, then the corresponding row of the table will have M indexes of 6 bits.

The second step of CREW is applied during the dynamic execution of the DNN inference. Each FC will be computed as follows: first the multiplications of each input by their unique weights will be performed and the results memoized into a new table of partial products. Then, the indexes generated offline will be used to access the corresponding partial products of each output neuron, which will be accumulated to generate the final output. Note that each column of the table of indices will contain N indexes to a partial product related with a different input so that by adding them all, the output of each neuron is computed. Figure 5.2a illustrates the tables required by

Table 5.2: Reduction in multiplications and storage.

DNN Model	Saved MULs (%)	Storage Reduction (%)
DS2	98	27
GNMT	99	34
Transformer	96	22
Kaldi	97	16
PTBLM	99	26

CREW for a small FC layer of 4×8 neurons. Note that in the example, the number of unique weights per input is ≤ 4, as well as the number of partial products, so the size of the indices is ≤ 2 bits.

The main benefit of the memoization of partial products is the reduction in multiplications. The number of multiplications per input is reduced to the number of unique weights it has. Moreover, the storage and memory accesses are also reduced since the indexes are smaller than the original weights. Table 5.2 shows the multiplications and storage reduction achieved on a set of state-of-the-art DNNs. On average, we can reduce the storage required for the FC layers of the models by 25% over the quantized networks (taking into account the required metadata for CREW), and save 98% of the multiplications. CREW requires extra metadata such as the number of unique weights per input and the table of partial products. However, since the unique weights represent a small fraction of the total number of weights in the original model ($< 4\%$) and the indices are also smaller than the original weights, the extra storage required for metadata is negligible compared to the savings provided by CREW.

5.2.2 Partial Product Approximation

Partial product memoization reduces multiplications by exploiting weight repetition. The original weights are replaced by the unique weights and the indexes to the partial products. The number of unique weights per input determines the size of the indexes and, hence, the total size of the DNN. We propose to further improve our mechanism by reducing the number of unique weights. From the histograms of the unique weights frequency usage shown in Figure 5.3, we observe that on average more than 50% of the unique weights have a frequency of use lower than 1%. Based on this observation we propose to approximate the less frequent unique weights by its closest value in the remaining unique weights.

The partial product approximation technique reduces the number of unique weights of some (potentially all) input neurons to the nearest and smaller power of two. For this purpose, a subset of unique weights, as well as their respective partial products, are approximated by similar unique weights/products of the same input. By reducing the number of unique weights to a lower power of two, the index bit length can be reduced. For example, if an input neuron has 38 unique weights, we can approximate 6 of them so that the number of unique weights becomes 32, reducing all the indexes related to this input by one bit.

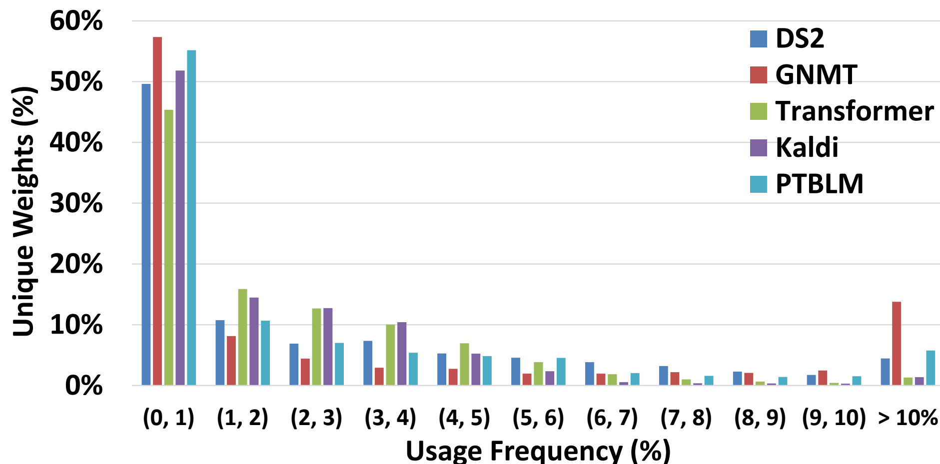


Figure 5.3: Usage frequency histograms of the unique weights per input neuron for all the FC layers of DS2, GNMT, Transformer, Kaldi and PTBLM. The frequency of use is the number of times each unique weight is repeated divided by the total number of weights for each input neuron.

Algorithm 5.1 Partial Product Approximation Heuristic

1: $W = LoadWeights();$
2: $Thr = Constant;$
3: **for** Input Neuron (i) **do**
4: $UW, Frequency = UniqueWeights(W[i], counts = True);$
5: $Current2Power = 2^{(log_2 UW)};$
6: $Low2Power = Current2Power/2;$
7: $dist_W = UW - Low2Power;$
8: $F = Sort(Frequency);$
9: $num_weights = Size(W[i]);$
10: $low_freq_W, del_uw = LowFreqSum(UW, F, dist_W);$
11: $WR = (low_freq_W/num_weights);$
12: **if** $WR < Thr$ **then**
13: $sim_uw = ClosestWeights(UW, del_uw);$
14: $new_W = ReplaceApproximate(W, sim_uw, del_uw);$
15: **end if**
16: **end for**
17: $Accuracy = Inference(new_W);$

Using the example of Figure 5.2, Figure 5.2b shows the approximation of three unique weights from two different input neurons that result in a reduction of one bit in the corresponding indexes of the index table. As a result of this optimization, all the indexes of the example require only one bit.

The reduction in storage and memory accesses obtained from this optimization may affect accuracy if it is not carefully applied. There is a trade-off between the number of weights that can be approximated without accuracy loss (or with a negligible loss) and the benefits obtained in terms of performance and energy consumption during DNN inference. The heuristic used to select which weights are approximated is shown in Algorithm 5.1. First, for each input neuron we compute the number of weights to be approximated by measuring the distance ($dist_W$) between

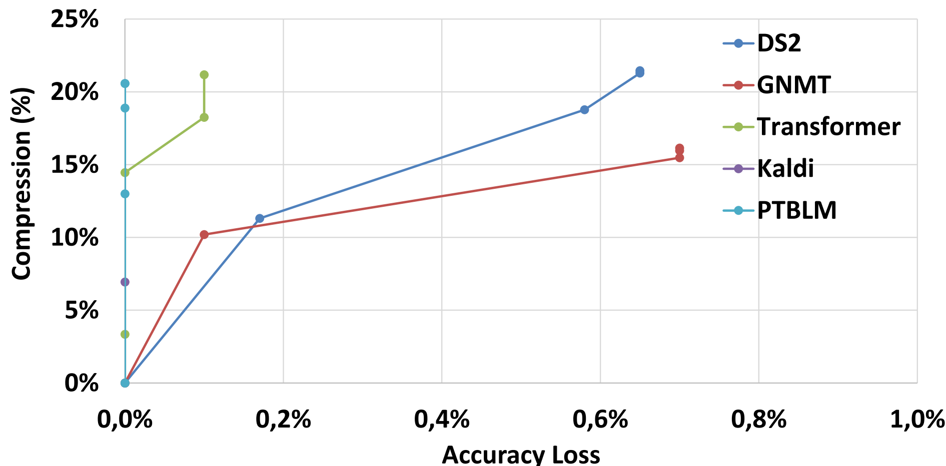
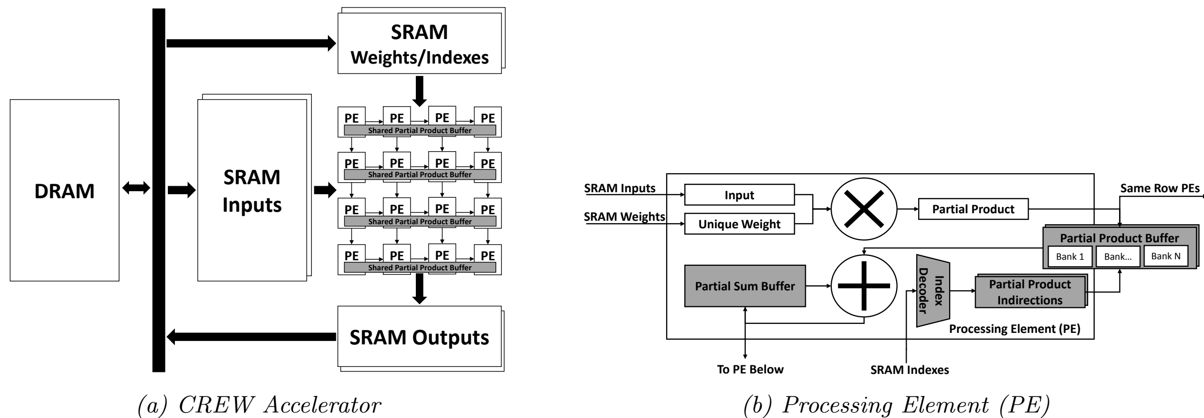


Figure 5.4: Accuracy loss versus compression ratio over partial product memoization (CREW without approximation) for different thresholds of the heuristic for unique weight approximation. Thresholds tested go up to 20% in steps of 5%, where the initial 0% threshold means no weight approximation.

the number of unique weights (UW) and the closest smaller power of two ($Low2Power$). Then, we select the weights to approximate by sorting the frequencies from lowest to highest, that is the number of times each unique weight is repeated in the original model. The rationale is that approximating the weights that are less frequently used should introduce the smallest distortion in the model. The frequencies of the $dist_W$ lowest weights are added (low_freq_W) to determine the relevance of the weights that will be approximated (WR), and the result is compared against a threshold (Thr): if the total frequencies of the less common weights (WR) is smaller than the threshold (Thr) then the weights are considered to have a small importance in the layer and are selected for approximation; otherwise, approximation is not used for this input. Finally, the selected weights (del_uw) are replaced in the original model by their closest unique weight (sim_uw) and the new model accuracy is tested. This heuristic can be generalized to reduce more than one bit by shrinking the number of unique weights to different powers of two while fulfilling the threshold condition.

A small threshold will limit the benefits achieved since less input neurons will be able to reduce their number of unique weights. On the other hand, a high threshold value may negatively impact the DNN accuracy since important weights may be approximated. We performed a sensitivity analysis on a representative set of DNNs to determine a proper threshold. Figure 5.4 shows that we can achieve, on average, an extra 17% model compression over the partial product memoization mechanism (CREW without accuracy loss as described in Section 5.2.1) while losing less than 1% of accuracy in absolute terms. We observed that with a threshold of 10%, the approximation covers more than 90% of the input neurons, which means that 90% of the indices are being reduced by one bit. For the Transformer and PTBLM networks the accuracy loss is almost negligible, so we tested a more aggressive approximation trying to reduce up to 2 bits per index. By doing so, we achieved an extra 35% model compression while losing less than 1% of accuracy. In summary, we can achieve extra improvements in performance and energy consumption during DNN inference in scenarios where the user is willing to accept a very minor accuracy loss.



(a) CREW Accelerator *(b) Processing Element (PE)*

Figure 5.5: Architecture of (a) the CREW accelerator and (b) a Processing Element (PE). The PE components shaded in gray represent the required extra hardware for CREW. The partial product buffer is shared among all or a subset of PEs of the same row of the array.

5.3 CREW Accelerator

This section describes the hardware support required to implement CREW. First, we present the main hardware components of the CREW accelerator for DNN inference. Next, we describe how FC layers are executed in the accelerator using CREW with an enhanced weight stationary dataflow. Finally, we describe how to efficiently support other types of layers and networks such as CNNs.

5.3.1 Architecture

In this work, we extend the baseline TPU-based DNN accelerator, presented in Section 2.2.2, to support CREW, and take advantage of the computation reuse found in the FC layers of different DNNs due to the spatial locality of the repeated weights. CREW exploits the high degree of repeated weights at each input neuron to save computations and memory accesses. Figure 5.5a shows a high-level schematic of the architecture of the CREW accelerator. The main hardware components of CREW are still the same as in the baseline TPU-like accelerator, illustrated in Figure 2.7, that is, the blocks of SRAM used for the inputs, outputs, unique weights and indexes, and the systolic array of processing elements (PEs). Each PE includes functional units to perform computations. All the global SRAM memories are double buffered to load data from main memory while performing computations, and highly multi-banked to achieve the bandwidth required to feed a large number of PEs.

Figure 5.5b shows the architecture of a Processing Element (PE) of the CREW accelerator. Each PE has a multiplier and an adder which can be used independently, that is, the multipliers are used to perform the unique weight multiplications and can be power-gated during the rest of the execution. On the other hand, the adders are used during the entire execution to accumulate the partial products of each output into the Partial Sum Buffer. Each PE requires additional memory buffers to store the partial products and the corresponding indexes to access them. The indexes

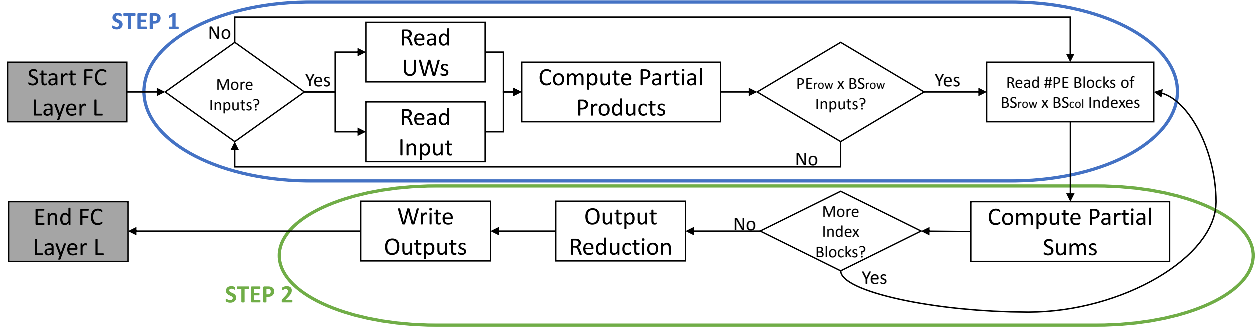


Figure 5.6: CREW Execution Dataflow. PE_{row} is the number of PEs per row in the systolic array, and BS_{row} and BS_{col} determine the block size, i.e. the number of indexes per block. BS_{row} refers to the number of indexes relative to different input neurons, while BS_{col} refers to indexes associated to the same input neuron but used in the computation of different output neurons.

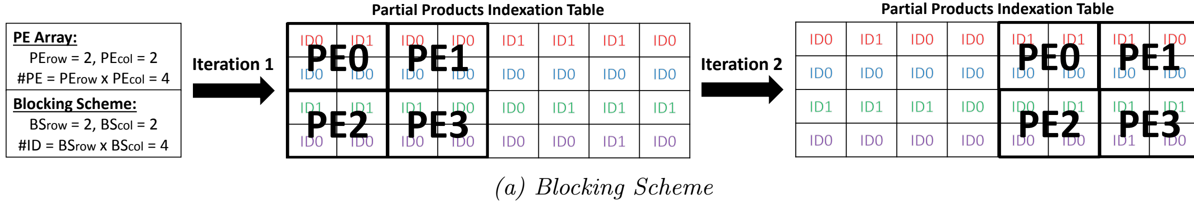
are decoded from the compressed stream and padded to 8 bits to be stored in the partial product indirections buffer. The partial products are stored in a multi-banked buffer shared among all the PEs from the same row of the array.

5.3.2 Dataflow

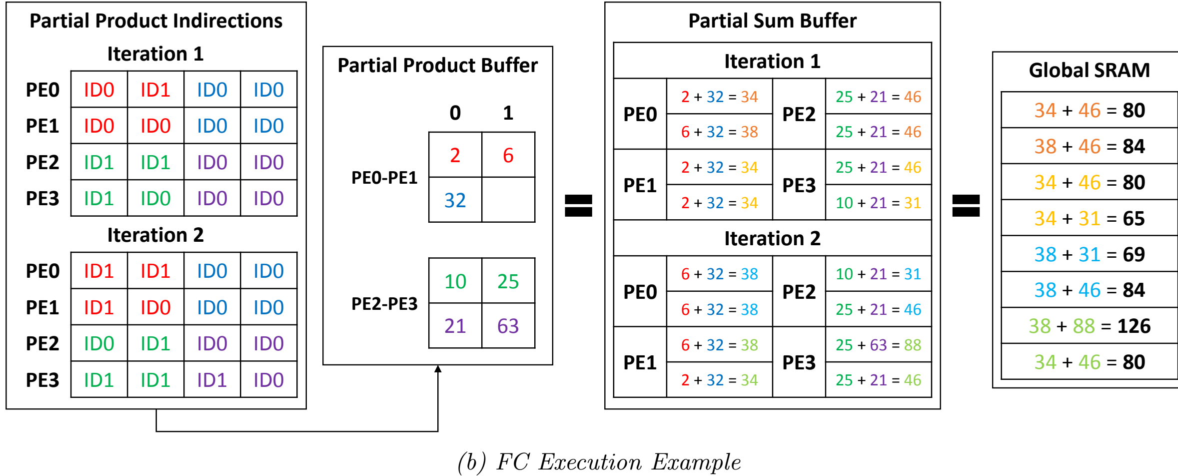
We implement an enhanced weight stationary dataflow to provide better efficiency in the execution of the FC layers with CREW. As described in Section 2.2, traditional DNN accelerators follow one of these dataflows: output stationary, weight stationary or input stationary. In output stationary, each PE computes an output at a time. In the weight/input stationary dataflow, each PE pre-loads a weight/input from memory to its local register, and those are used to perform all the associated computations. FC layers do not tend to be efficiently executed on systolic arrays because the potential reuse is much more limited compared to convolutional layers. Given a batch size of one, only the inputs can be reused multiple times to compute each output, making all the common dataflows inefficient to exploit the resources of the systolic array. We propose to use an enhanced weight stationary dataflow coupled with a blocking scheme.

The CREW accelerator executes the FC layers in two main steps carried out in parallel as is shown in the flowchart of Figure 5.6, where both steps are marked in different colors. In the first step, marked in blue, the accelerator starts by reading an input and its number of unique weights. Then, the unique weights of the input are read from memory. The input is broadcasted to a row of PEs and the unique weights are distributed along the PEs of the same row. Each PE will multiply the input and its unique weights and store the partial results into the Partial Product Buffer shared by all the PEs of a given row. This step is repeated until all the partial products of unique weights for the FC layer are computed. Note that the multipliers can be power gated after doing all the unique multiplications.

As it is shown in the schematic of Figure 5.6, when a given number of partial products is stored in the shared buffer of each row of PEs, the second step starts in parallel with the first one. The number of partial products required is determined by a block size parameter of $BS_{row} \times BS_{col}$ so



(a) Blocking Scheme



(b) FC Execution Example

Figure 5.7: FC Execution in the CREW Accelerator.

that when the unique weights of BS_{row} inputs have been processed for each row of the systolic array (PE_{row}), the second step begins. The block size also determines the number of indexes ($BS_{row} \times BS_{col}$) stored in the index table of each PE and the number of partial outputs (BS_{col}) that each PE is computing. During the second step, the accelerator reads one block of $BS_{row} \times BS_{col}$ indexes for each PE to perform the partial sums, as illustrated by the green marker in Figure 5.6. The blocks of indexes are constructed offline and stored consecutively in main memory. Each PE will receive a block of indexes pointing to partial products stored in its shared buffer. Therefore, each PE in a given row of the array is computing a different set of partial outputs, while the PEs in the same column are computing different partial sums for the same set of outputs.

The indexes of a block arrive in a compressed format to each PE, that is, all the BS_{col} indexes of a given row of the block have the same size, but for each BS_{row} the size of the indexes may be different. The reasoning behind the compressed format is that the blocks are composed of indexes related to different input neurons which may have a different number of unique weights, and so indexes exhibit variable size. All the indexes of a block are stored consecutively in memory starting by the BS_{col} indexes of the first row of the block. Each PE includes a specialized hardware decoder to decompress the indexes and store them uncompressed (8 bits per index) in the Partial Product Indirections buffer. The number of indexes in a block is fixed and known by the control unit of each PE. Similarly, the sizes of the indexes related to each input neuron are computed offline, stored in memory, and sent along the indexes of each block when needed. Note that a single value of three bits per input neuron is enough to determine the sizes of all the indexes. The hardware required to perform the decoding of each index is relatively simple since all the information is statically available. On the other hand, the partial products between inputs and unique weights can be

computed in parallel to the partial sums, along with the loading and decoding of the next blocks of indexes from the global buffer. To this end, the shared Partial Product Buffer and the indirection table of each PE are double buffered. To avoid collisions, the shared buffer of partial products is highly multi-banked and each PE starts to access partial products relative to input neurons with a different offset located into a different bank.

Continuing the example from Section 5.2, Figure 5.7 shows an FC execution in the CREW accelerator following our dataflow and blocking scheme. Considering a systolic array of 2×2 PEs and a block size of 4 indexes, the blocks of indexes are distributed in two iterations as shown in Figure 5.7a. Figure 5.7b illustrates the second step of the dataflow, having all the partial products computed in the first step stored in the shared buffer of each row of PEs, and all the decoded indexes in the indirections buffer. At each cycle, each PE reads an index of its block, the associated partial product and the current partial sum of the corresponding output to perform the next accumulation. The partial sum buffer depicted in the figure shows the additions performed by each PE in each iteration. Once the PE has processed all the indexes of the block, it proceeds to compute the subsequent block. This step will be repeated until all the blocks are processed. Finally, a reduction of all the partial summations is performed from top to bottom of the systolic array, writing the final results into the SRAM output buffer as shown in the example.

In summary, each PE will compute partial sums of multiple neurons by accessing to different partial products associated to the same set of inputs. The two main steps are repeated until all the partial products of each input are computed and all the indexes processed. Note that the partial products are computed only once and reused during the rest of the FC layer execution.

5.3.3 Design Flexibility

A key parameter in the dataflow of CREW and its hardware implementation is the block size. The block size affects the performance and the size of the local buffers in each PE. A large block size will require sizable local buffers in each PE. On the other hand, the block size has to be large enough to be able to perform computations while loading from memory the next blocks of indexes, in order to hide main memory latency, and avoid collisions to the shared buffers. To avoid large interconnection overheads in case of a considerable number of columns in the systolic array, the shared buffers can be replicated and shared among PEs of a given subset for the same row of the array. Therefore, by adjusting the block size and resizing the buffers, CREW can keep the scalability of the accelerator to any systolic array size. Although our accelerator is specialized in efficient inference of FC layers by using CREW, it can support any kind of layer such as convolutionals as executed in a TPU-like accelerator, without obtaining the benefits of the reuse mechanism.

5.4 Evaluation Methodology

The experimental evaluation of CREW is performed as described in Chapter 3 to asses the performance, energy consumption and area of the hardware accelerator. We have extended ScaleSim [81], the cycle-accurate simulator for DNN accelerators described in Section 3.1, to accurately model three different systems: the baseline TPU-based DNN accelerator, UCNN [34] and the CREW

Table 5.3: Parameters for the accelerators.

Common Parameters	
Technology	32 nm
Frequency	500 MHz
# PEs	16 x 16
Block Size	16 x 16
Total Global SRAM Buffers Size	24 MB
UCNN Parameters	
Input/Weight Indirections Buffer Size/PE	0.75 KB
Input/Weight Buffer Size/PE	272 B
Partial Sum Buffer Size/PE	0.75 KB
CREW Parameters	
Partial Product Indirections Buffer Size/PE	0.5 KB
Partial Product Buffer Size/PE	1 KB
Partial Sum Buffer Size/PE	0.75 KB

accelerator presented in Section 5.3. ScaleSim models a systolic array architecture and supports any kind of neural network, including CNNs, MLPs and RNNs.

Table 5.3 shows the parameters for the experiments. The baseline configuration is a TPU-like architecture clocked at 500 MHz with an output stationary dataflow [13]. It includes a systolic array of 16×16 (256 PEs) and 24 MB of global on-chip SRAM. Inputs and weights are quantized to 8-bit integers without any accuracy loss for a modern set of DNNs, whereas activation functions are performed in 32-bit floating point. On top of this architecture, we have implemented UCNN as described in [34] and the CREW scheme as presented in Section 5.3. The sizes of the additional local buffers required by UCNN and CREW are provided in Table 5.3. CREW requires three local SRAM buffers on each PE: the Partial Product Indirections buffer, the Partial Product Buffer shared between multiple PEs of the same row and the Partial Sum Buffer. All the local buffers are sized taking into account the block size parameter of the CREW dataflow. We employ a block size of 16×16 for all the DNNs, as we found that it provides a good trade-off between on-chip storage requirements and performance. As a result, all the local buffers require less than 1KB of capacity per PE. These overheads are taken into account for the area, timing and energy evaluation of the accelerator. Regarding main memory, we model an LPDDR4 of 8 GB with a bandwidth of 16 GB/s (dual channel).

In order to prove that CREW provides important savings for multiple applications and different DNN architectures, we use the five state-of-the-art DNNs shown in Table 5.4 from different application domains, including speech recognition, machine translation and language modeling. For all the DNNs, we employ the entire evaluation sets from their respective datasets, including several hours of audio and a large number of texts, to assess the efficiency of the computation reuse scheme in terms of performance and energy consumption. These workloads represent important machine learning applications, and the selected DNNs cover three of the most common approaches for sequence processing: MLPs, LSTMs and GRUs.

Table 5.4: DNNs employed for the experimental evaluation of CREW. The model size accounts for the original FP parameters of the FC layers where CREW is applied.

DNN Model	Size (MB)	Accuracy
DS2	144	10.32% (WER)
GNMT	518	29.8% (BLEU)
Transformer	336	28.0% (BLEU)
Kaldi	18	10.85% (WER)
PTBLM	137	78.15% (Perplexity)

5.5 Experimental Results

This section evaluates the performance and energy consumption of the CREW accelerator implementing the computation reuse scheme. First, we present the speedups and energy savings achieved by CREW compared to the baseline TPU-based DNN accelerator and UCNN. In order to do a fair comparison, we evaluate the factorization of UCNN using the same blocking dataflow as CREW. Next, we present the additional benefits of CREW when applying the partial product approximation presented in Section 5.2.2. Finally, we discuss the accelerator overheads.

5.5.1 CREW Evaluation

Figure 5.8 shows the speedups achieved by CREW and UCNN over the baseline TPU-based DNN accelerator. CREW provides consistent speedups for the five DNNs that range from 2.26x (*Kaldi*) to 2.96x (*GNMT*), achieving an average performance improvement of 2.61x. The reduction in execution time is due to reusing previously computed partial products. The number of multiplications is dramatically reduced since only the unique weights are multiplied by their corresponding inputs. Furthermore, the overhead to access the previously memoized partial products is small since the indexes are smaller than the original weights. In addition, the partial sums are computed concurrently while loading the next blocks of indexes from memory, efficiently exploiting all the resources of the accelerator. Note that we use the partial product memoization scheme presented in Section 5.2.1, so the speedups reported in Figure 5.8 are achieved without any accuracy loss. Impact on accuracy loss and performance of partial product approximation are presented in Section 5.5.2.

We achieve more than 2x performance speedup compared to the factorization scheme of UCNN, since CREW is much more effective for FC layer inference. Although UCNN supports FC layers, our results show that it achieves an average speedup of 25% compared to the baseline. The multiplications are reduced in a similar percentage compared to CREW. However, the overheads to index the inputs of each factorization group in an FC layer are quite high, hindering the benefits of the computation reuse scheme.

The average power consumption of the CREW accelerator is $8W$. Figure 5.9 reports normalized energy savings. On average, CREW reduces the energy consumption of the accelerator by 2.42x. The energy savings are well correlated with the reduction of the model size due to the low number

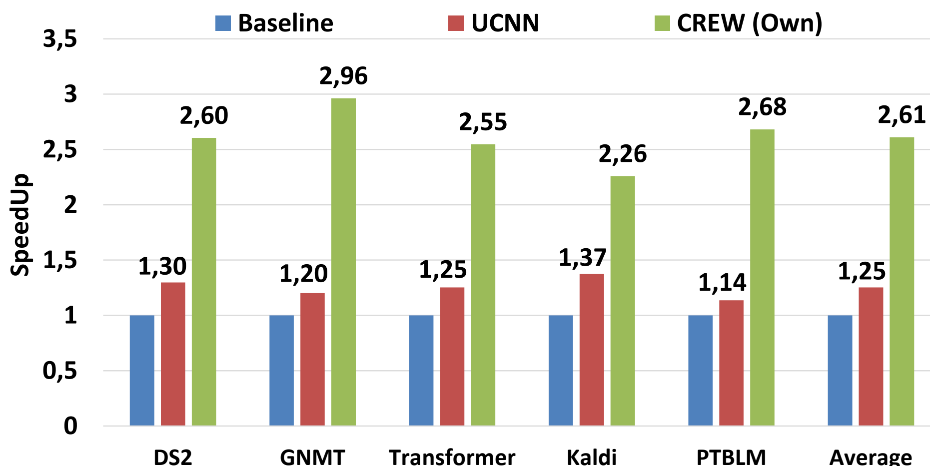


Figure 5.8: Speedups achieved by CREW and UCNN for each DNN. Baseline configuration is the TPU-like DNN accelerator without any computation reuse mechanism.

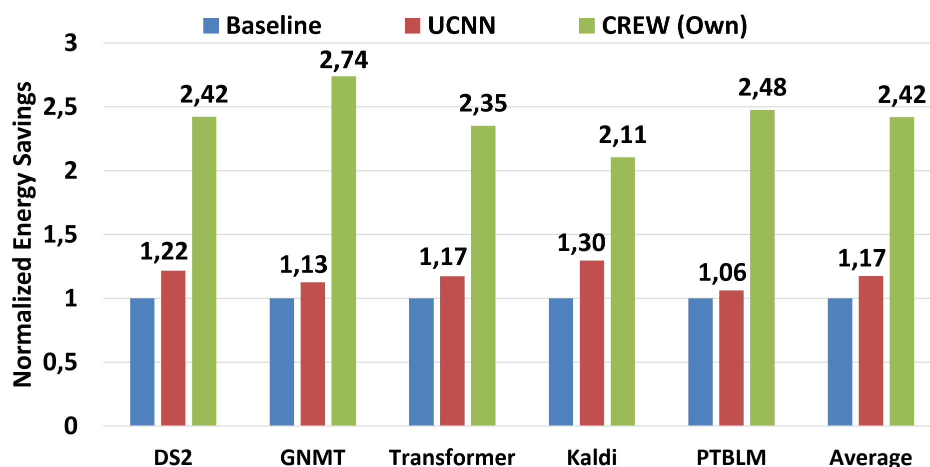


Figure 5.9: Normalized energy savings for each DNN. Baseline configuration is the TPU-like DNN accelerator without the computation reuse technique.

of unique weights and the small size of the indexes compared to the original weights. These energy savings are due to two main reasons. First, dynamic energy is reduced due to the savings in computations and memory accesses. Second, the performance improvements shown in Figure 5.8 provide a reduction in static energy. Again, we reduce energy by more than 2x on average compared to both the baseline and UCNN.

5.5.2 CREW-PPA Evaluation

As described in Section 5.2.2, we propose to extend CREW with partial product approximation to further reduce multiplications and the storage for the indexes. With this technique, some unique weights are discarded and replaced by the most similar remaining unique weights. Note that all the processing for this optimization is performed offline and it does not require any additional hardware

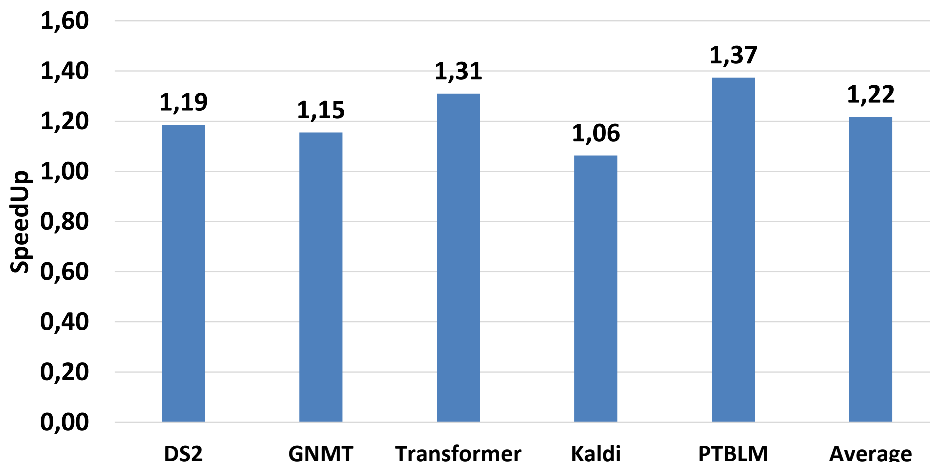


Figure 5.10: Speedup of partial product approximation with less than 1% accuracy loss. Baseline configuration is the CREW accelerator.

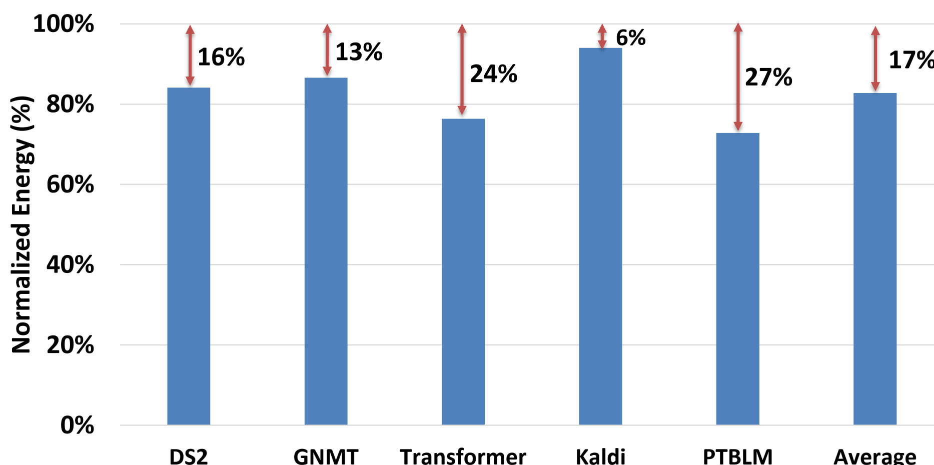


Figure 5.11: Normalized energy of partial product approximation with less than 1% accuracy loss. Baseline configuration is the CREW accelerator.

support, so it can be implemented directly on top of CREW. This optimization effectively reduces the amount of computations and memory bandwidth usage, but it has to be carefully applied to guarantee a negligible impact on DNN accuracy.

Figure 5.10 and Figure 5.11 show the speedup and normalized energy obtained of the partial product approximation optimization when the maximum accuracy loss is set to 1%. On average, it achieves more than 1.2x speedup with an energy reduction of 17%, on top of the CREW accelerator. Note that this optimization can be customized to the user requirements in terms of accuracy versus performance and energy consumption.

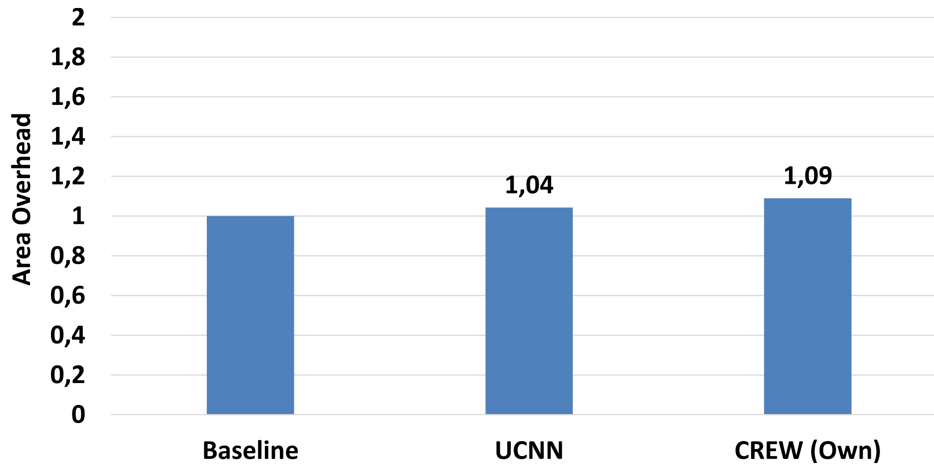


Figure 5.12: Area overhead of UCNN and CREW. Baseline configuration is the TPU-like DNN accelerator without any computation reuse technique.

5.5.3 Overheads

CREW requires extra storage in the PEs of the accelerator as shown in Table 5.3. These buffers together with the index decoders are small compared to the global SRAM, so they represent a small portion of the area and energy of the accelerator. Figure 5.12 shows that CREW represents a 9% increase in area, compared to the baseline TPU-like accelerator, as it increases from $101.14mm^2$ to $110.15mm^2$. On average, the energy consumed by the additional hardware of CREW represents less than 4% of the total energy. In comparison, UCNN's extra storage results in an increase in area of 4%, while representing less than 3% of overall energy consumption. We believe CREW's area and energy overheads are acceptable considering the large performance and energy improvements as reported in section 5.5.1.

6

CGPA: Coarse-Grained Pruning of Activations

Previous chapters have explored different computation reuse techniques that dramatically improved the efficiency of the hardware accelerators for DNN inference in terms of performance and energy consumption. Despite these improvements, the memory hierarchy still represents a major bottleneck of the DNN accelerators, due to the huge number of DNN parameters to process. In the remaining chapters of this thesis, we explore various pruning methods, including static/dynamic and node/link pruning mechanisms, in order to further reduce the DNN model size, the number of memory accesses and the number of computations.

In this chapter, we focus on improving RNN inference, by proposing a dynamic activation pruning mechanism applied at a coarse granularity. We first present an analysis of the values of the activation functions that are most commonly used in the gates of different RNN cells including Long Short-Term Memory (LSTM) and Gated Recurrent Unit (GRU). Next, based on the high number of activations that are close to either zero or one, we propose CGPA, a mechanism that exploits the saturation of the activation functions to reduce the amount of computation. Finally, we provide the results of CGPA implemented on top of a TPU-like accelerator.

6.1 Analysis of RNN Activations

LSTM and GRU networks are the two most successful RNN architectures. Both employ sigmoid and hyperbolic tangent activation functions. This section analyzes the activation values of different RNNs, and introduces a technique that exploits the activation values that are close to zero or one in order to save computations and memory accesses. Table 6.1 shows the RNNs employed for the analysis. As described in Section 3.3, DeepSpeech2 ($DS2$) [4] consists of multiple GRU layers employed to perform end-to-end speech recognition. We use two DS2 models, a model trained with

Table 6.1: RNNs employed for the evaluation. Only LSTM and GRU layers are included since these layers take up the bulk of computations in RNNs.

DS2 (145MB)				GNMT (800MB)				PTBLM (250MB)			
DS2-L WER: 10.2% - DS2-T WER: 29.2%				Bleu: 29.8%				Perplexity: 78.1%			
Layer	Input Dim	Output Dim	Cell Dim	Layer	In Dim	Out Dim	Cell Dim	Layer	In Dim	Out Dim	Cell Dim
BiGRU1	1472	800	800	BiLSTM1	2048	2048	1024	UniLSTM1	3000	1500	1500
BiGRU2	1600	800	800	UniLSTM2	3072	1024	1024	UniLSTM2	3000	1500	1500
BiGRU3	1600	800	800	UniLSTM3	2048	1024	1024				
BiGRU4	1600	800	800	UniLSTM4	2048	1024	1024				
BiGRU5	1600	800	800	UniLSTM5	3072	1024	1024				
				UniLSTM6	3072	1024	1024				
				UniLSTM7	3072	1024	1024				
				UniLSTM8	3072	1024	1024				

$$i_t = \sigma(W_{ix}x_t + W_{ih}h_{t-1} + b_i) \quad (6.1)$$

$$f_t = \sigma(W_{fx}x_t + W_{fh}h_{t-1} + b_f) \quad (6.2)$$

$$g_t = \phi(W_{gx}x_t + W_{gh}h_{t-1} + b_g) \quad (6.3)$$

$$o_t = \sigma(W_{ox}x_t + W_{oh}h_{t-1} + b_o) \quad (6.4)$$

$$c_t = f_t \odot c_{t-1} + i_t \odot g_t \quad (6.5)$$

$$h_t = o_t \odot \phi(c_t) \quad (6.6)$$

(a) LSTM Cell Computations

$$z_t = \sigma(W_{zx}x_t + W_{zh}h_{t-1} + b_z) \quad (6.7)$$

$$r_t = \sigma(W_{rx}x_t + W_{rh}h_{t-1} + b_r) \quad (6.8)$$

$$g_t = \phi(W_{gx}x_t + W_{gh}(r_t \odot h_{t-1}) + b_g) \quad (6.9)$$

$$h_t = z_t \odot h_{t-1} + (1 - z_t) \odot g_t \quad (6.10)$$

(b) GRU Cell Computations

Figure 6.1: Computations performed in (a) LSTM and (b) GRU cells. $\odot$, ϕ and σ are element-wise multiplication, hyperbolic tangent and sigmoid function respectively.

the Librispeech [70] dataset ($DS2 - L$) and a Tedlium trained model for spontaneous and noisy speech ($DS2-T$). GNMT [97] is an LSTM network for neural machine translation trained with the WMT16 dataset with texts of newspapers (DE-EN). Finally, PTBLM [103] is an LSTM network for language modeling using the Penn Treebank dataset. Accuracy is reported as Word Error Rate (WER), bilingual evaluation understudy (BLEU) and perplexity respectively.

Figure 6.1a and Figure 6.1b describe the computations performed in the LSTM and GRU cells respectively. Both are composed of multiple gates that are implemented as single-layer Fully-Connected (FC) networks taking two different inputs: x_t, a.k.a. feed-forward input from previous layer, and h_{t-1}, a.k.a. recurrent input from the same layer. Evaluation of FC layers for the different gates takes most of the computations in an RNN. Careful analysis of the LSTM equations reveals that if a neuron of the input gate (i_t) is saturated to zero, we could save the computations and memory access of its peer neuron in the generate gate (g_t) and vice versa. Similarly, if the hyperbolic tangent of the cell state (c_t) equals to zero, we could save the computations and memory accesses of its peer neuron in the output gate (o_t) needed for computing the cell output (h_t). Note that if the output gate (o_t) is saturated to zero we can only save the computations related to the activations since the cell state has to be computed for the next timestep, and if the forget gate (f_t) is saturated to zero, we cannot save computations since the cell state from the previous timestep has already been computed. A similar analysis can be done to the GRU equations where we could

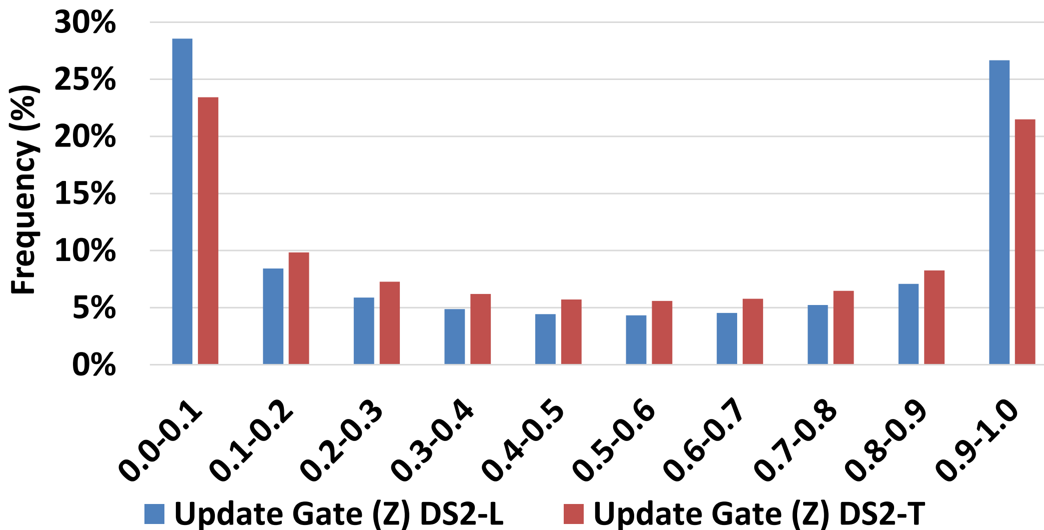


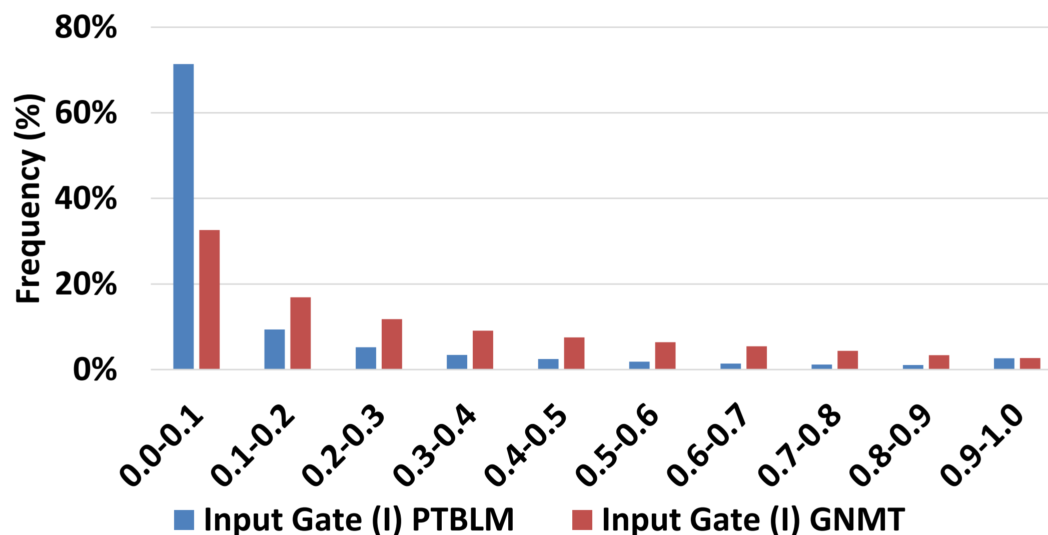
Figure 6.2: Histogram of activations of the Update Gate (Z) for DS2-L and DS2-T.

save computations and memory accesses of neurons from the generate gate (g_t) when the peer neuron in the update gate (z_t) is saturated to one. Note that in case that a neuron of the generate gate (g_t) is zero, the update gate (z_t) still has to be computed since it is multiplied by the output cell of the previous timestep.

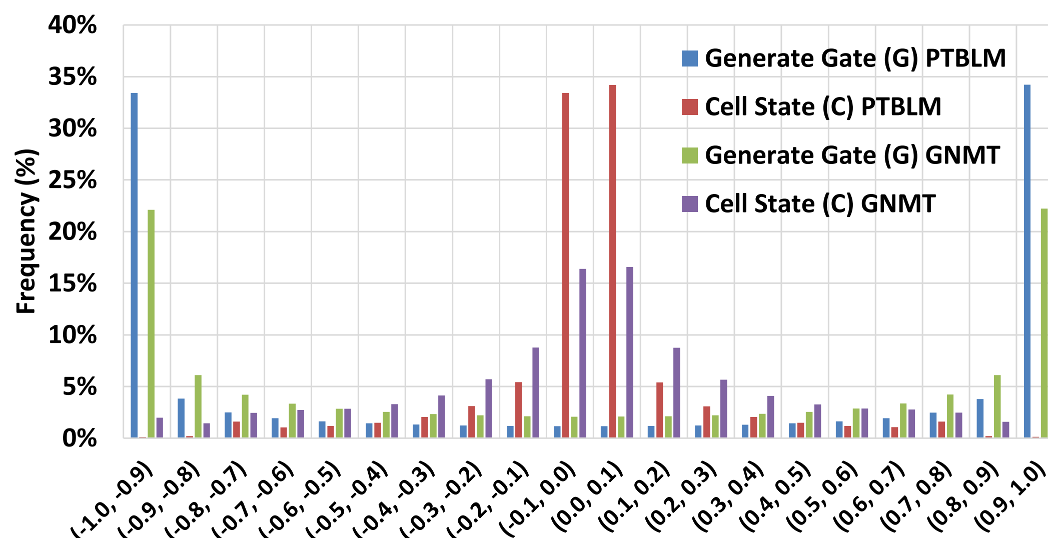
Figure 6.2 shows the histogram of activation values of the GRU's update gate (Z) for DeepSpeech2 trained with the Librispeech ($DS2-L$) and Tedlium ($DS2-T$) datasets in steps of 0.1. The range of the histogram is from 0 to 1 since the sigmoid is used as activation function. We found that between $40-60\%$ of the values are near the saturation points of $0-0.1$ and $0.9-1.0$ (i.e. $20-30\%$ at each end of the histogram), while the rest of the values are about equally distributed. Therefore, a significant percentage of the activations of the update gate (z_t) are close to the saturation points (zero or one), pointing out potential to save computations.

Figure 6.3a shows the histogram of the activation values (sigmoid activation) of the LSTM's input gate (I) for PTBLM and GNMT. We can see a high concentration of values in the range of $0-0.1$: 70% for the PTBLM and 33% for the GNMT network. On the other hand, Figure 6.3b shows the histogram of activations of the generate gate (G) and the cell state (C) for PTBLM and GNMT. The range of the histograms is from -1 to 1 since the hyperbolic tangent is used as activation. We can see that the activation values of the cell state (C) are concentrated around zero: more than 60% of activations in PTBLM and more than 30% in GNMT are in the range $[-0.1, 0.1]$. Furthermore, the activation values of the generate gate (G) are concentrated at both ends of the histogram, with values close to -1 or 1. The large percentage of near-zero activations in both input gate (I) and cell state (C) points out significant potential for saving computations in LSTM cells.

In this work, we propose to avoid computations of entire neurons in LSTM and GRU cells whenever the activation of some gates is close to zero or one. To this end, we define low and high thresholds so that activation values that are lower than a small threshold t_l are set to zero and all the activation values that are higher than t_h in absolute value are set to one. We apply these thresholds in gates that allow saving computations, i.e. update gate (z_t) in GRU cells and



(a) Histogram of activations of the Input Gate (I)



(b) Histogram of activations of the Generate Gate (G) and Cell State (C)

Figure 6.3: Input Gate (I) (a), Generate Gate (G) and Cell State (C) (b) histogram of activations for the GNMT and PTBLM RNNs.

input gate (i_t) and cell state (c_t) in LSTM cells. We call this technique Coarse-Grained Pruning of Activations (CGPA) since it prunes whole neurons from the gates.

Setting appropriate thresholds is key to achieve a good trade-off between savings in computations and accuracy. We evaluated different high and low thresholds for the set of RNNs shown in Table 6.1, ranging from $0.9 - 1.0$ and $0.0 - 0.1$ with steps of 0.01 respectively. In general, a high threshold (t_h) of 0.95 and a low threshold (t_l) of 0.05 provide significant savings with negligible accuracy loss across our set of RNNs. However, we have empirically selected a particular set of thresholds for each RNN in order to maximize the savings while maintaining the accuracy.

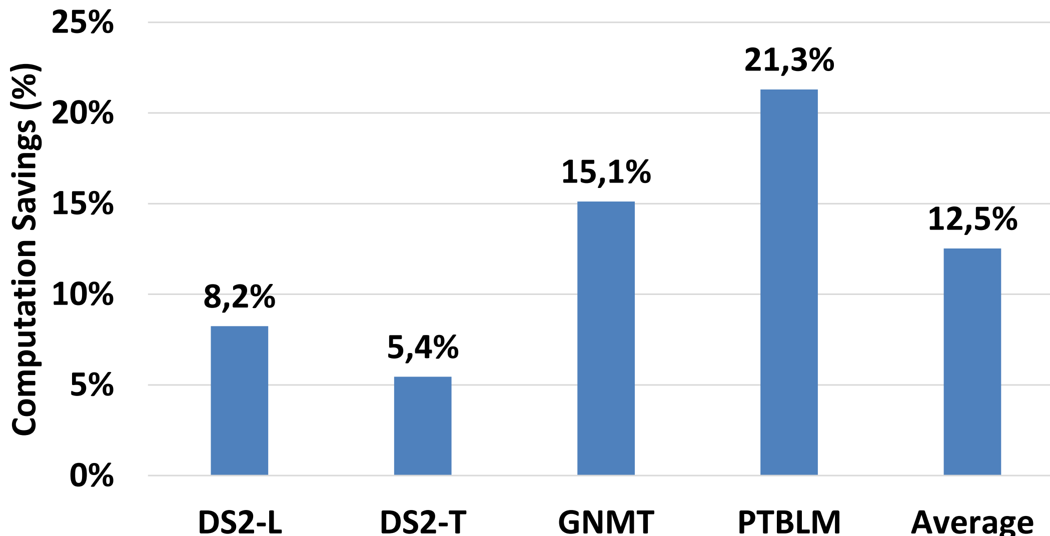


Figure 6.4: Computation savings for each RNN.

Note that we are introducing a small distortion in the RNNs to clamp some activations to zero or one. In order to improve the effectiveness of CGPA, we can include the thresholds during the training of the RNN, so the model learns to compensate these small deviations in the activations. As a proof of concept, we verified that including CGPA during training results in larger savings: in GNMT the savings in computations are increased by a factor of approximately 3x with the same negligible accuracy loss. In other words, training with CGPA allows to use more generous thresholds in order to increase the savings, since more activations are clamped to zero or one, while retaining accuracy by a large extent. Accuracy loss has been computed in absolute terms for each of the RNNs. The GNMT network accuracy loss is 0.8 in absolute terms which means that the BLEU has been reduced from 29.8% to 29.0%. The PTBLM network accuracy loss is 2.43 in perplexity, while the accuracy loss of $DS2-L$ and $DS2-T$ is 0.92 and 1.65 of the WER respectively.

We have used our selection of thresholds to compute the number of activations that are saturated layer-wise. We have observed that the number of saturations in the update gate (z_t) of the RNNs with GRU cells are homogeneous across most of the layers. DeepSpeech2 (DS2) with Librispeech ($DS2-L$) and Tedlium ($DS2-T$) have on average 24% and 15% saturations per layer respectively. On the other hand, the LSTM networks have more variance across layers with a minimum of 40% and a maximum of 60% saturations per layer taking into account both the input gate (i_t) and cell state (c_t) activations. Figure 6.4 shows the computation savings that are achieved for each RNN. We can see that the computation savings are directly related to the histograms of the activations. On average, 12.5% of the computations and memory accesses can be avoided with a small accuracy loss of less than 1.5%.

6.2 CGPA Accelerator

This section describes the hardware support required to implement CGPA. First, we present the main hardware components of the CGPA accelerator for RNN inference. Then, we describe how LSTM and GRU layers are executed in the accelerator using CGPA.

6.2.1 CGPA Architecture

In this work, we extend the baseline TPU-based DNN accelerator, presented in Section 2.2.2, to support CGPA, and take advantage of the saturated activations of the RNN gates to avoid computations and memory accesses. The main hardware components of the CGPA accelerator are still the same as in the baseline TPU-like accelerator, illustrated in Figure 2.7, that is, the blocks of SRAM used for the inputs, outputs and weights, and the systolic array of processing elements (PEs). Each PE includes a MAC (multiplier-accumulator) and some registers. All the global SRAM memories are double buffered to load data from main memory while performing computations, and highly multi-banked to achieve the bandwidth required to feed a large number of PEs.

CGPA requires a comparator to determine which neurons must be executed depending on the activation values. The comparator is used for each neuron to determine whether its activation is smaller/larger than the given threshold. In addition, a small SRAM buffer is required to store the resulting bitmask from the comparisons. The bitmask contains one bit per neuron indicating whether the peer neuron in another gate has to be computed or skipped.

The dataflow of the baseline TPU-like accelerator has been extended to implement CGPA for GRU and LSTM cells. The following subsections describe the execution of GRU and LSTM layers with CGPA.

6.2.2 CGPA for GRU Layers

The execution of GRU cells under CGPA is described in the pseudo-code Algorithm 6.1. The accelerator starts by computing all the neurons of the update gate (z_t) and reset gate (r_t). Each neuron activation of the update gate is compared with the high threshold to generate a bitmask that determines which neurons from the generate gate are executed. The bitmask is stored in a small SRAM buffer. Finally, the generate gate (g_t) and cell output (h_t) are computed by checking the bitmask from the buffer. If the activation of the update gate is greater or equal than the high threshold for a given neuron, its corresponding bit in the bitmask will be zero indicating that the computations of the peer neuron from the generate gate can be avoided. In this case, the output of the cell will be the same as the output in the previous timestep. Otherwise, the generate gate and the cell output is computed as usual. The control unit is modified to schedule multiple neurons of the generate gate in the array of PEs after checking the bitmask.

Algorithm 6.1 Accelerator GRU Execution

```
1: High Threshold (t_h) = Constant
2: for Timestep (t) do
3:    Compute Update Gate (z_t);
4:    Compute Reset Gate (r_t);
5:    {Accelerator Executes Multiple Neurons in Parallel}
6:    for Neuron (n) do
7:       if z_tn ≥ t_h then
8:          {Generate Gate (g_tn) Computations Avoided}
9:          Cell Output (h_tn) = h_tn−1;
10:      else
11:         Compute Generate Gate (g_tn);
12:         Compute Cell Output (h_tn);
13:      end if
14:   end for
15: end for
```

6.2.3 CGPA for LSTM Layers

The execution flow for a layer of LSTMs is depicted in the pseudocode of Algorithm 6.2. The accelerator computes all the neurons of the input (i_t) and forget (f_t) gates. After each activation of the input gate, a comparison with the low threshold is performed to generate a bitmask that determines which neurons from the generate gate are executed. The generate gate (g_t) and the cell state (c_t) are computed by checking the bitmask. If an activation of the input gate is lower or equal than the low threshold for a given neuron, the computations of its peer neuron from the generate gate are avoided and the cell state is computed as the product of the forget gate by the previous cell state. The activations of the cell state are compared against the low threshold to generate a new bitmask that determines which neurons from the output gate (o_t) are executed. If an activation of the cell state in absolute value is lower or equal than the low threshold for a given neuron, the computations of its peer neuron from the output gate are avoided and the cell output is set to zero. Otherwise, the output gate and the cell output is computed as usual. As in the GRU's execution, the control unit is modified to schedule multiple neurons in the array of PEs after checking the bitmasks.

6.2.4 Design Flexibility

Note that CGPA does not increase the complexity of the control unit of the accelerator significantly, since we are skipping entire neurons. We only need to schedule the neurons that require computations in the array of PEs while the rest are skipped by checking the bitmask generated from the comparators. Unlike a sparse accelerator that requires multiple indices to access the data and a complex control unit, we just need a constant offset with the size to skip the weights of an entire neuron, so the control unit computes the required addresses as indicated by the bitmask.

Algorithm 6.2 Accelerator LSTM Execution

```
1: Low Threshold (t_l) = Constant
2: for Timestep (t) do
3:    Compute Input Gate (i_t);
4:    Compute Forget Gate (f_t);
5:    {Accelerator Executes Multiple Neurons in Parallel}
6:    for Neuron (n) do
7:       if i_tn ≤ t_l then
8:          {Generate Gate (g_tn) Computations Avoided}
9:          Cell State (c_tn) = f_tn c_tn−1;
10:      else
11:         Compute Generate Gate (g_tn);
12:         Compute Cell State (c_tn);
13:      end if
14:   end for
15:   for Neuron (n) do
16:      c_tn = abs(tanh(c_tn));
17:      if c_tn ≤ t_l then
18:         {Output Gate (o_tn) Computations Avoided}
19:         Cell Output (h_tn) = 0;
20:      else
21:         Compute Output Gate (o_tn);
22:         Compute Cell Output (h_tn);
23:      end if
24:   end for
25: end for
```

6.3 Evaluation Methodology

The experimental evaluation of CGPA is performed as described in Chapter 3 to asses the performance, energy consumption and area of the hardware accelerator. We have extended ScaleSim [81], the cycle-accurate simulator for DNN accelerators described in Section 3.1, to accurately model two different systems: the baseline TPU-based DNN accelerator and the CPGA accelerator presented in Section 6.2. ScaleSim models a systolic array architecture which is specialized on CNNs but supports any kind of neural network, including RNNs, like TPU does.

For the evaluation of CGPA, we set the configuration parameters of the accelerator architecture to match the TPU: a systolic array of $256x256$ PEs, 24 MB of total on-chip SRAM and a frequency of 700 MHz. However, we use an output stationary dataflow [13], as we empirically found that it is more efficient than weight stationary (originally used in TPU) for our set of RNNs. In output stationary, each PE computes the output of a different neuron. Therefore, the weights are stored and forwarded from top to bottom and the inputs are injected in the systolic array from left to right. As in TPU, the inputs and weights are quantized to integers of 8-bits without any accuracy loss so that the operations of the convolutional and fully-connected layers are performed with integers. However, the intermediate layers and the activation functions are performed in 32-bit floating point to maintain the accuracy. Regarding main memory, we model an LPDDR4 of 8 GB with a bandwidth of 16 GB/s (dual channel). As previously described, CGPA also requires a

floating point comparator to determine which neurons must be executed depending on the activation values. In addition, a small SRAM buffer of 512B is included to store the resulting bitmask from the comparisons. All these overheads are taken into account for the area, timing and energy evaluation of the accelerator.

In order to prove that CGPA provides important savings for multiple applications, we use the four state-of-the-art RNNs shown in Table 6.1 from different application domains, including speech recognition, machine translation and language modeling. For all the RNNs, we employ the entire evaluation sets from their respective datasets, including several hours of audio and a large number of texts, to assess the efficiency of the pruning of activations in terms of performance and energy consumption. These workloads represent important machine learning applications, and the selected RNNs cover the two most common approaches for implementing recurrent layers: LSTMs and GRUs.

6.4 Experimental Results

This section evaluates the performance and energy consumption of CGPA when implemented on top of the TPU-like accelerator as described in Section 6.2.

Figure 6.5 shows the speedups achieved by CGPA. The technique provides consistent speedups for the four RNNs that range from 1.07x ($DS2 - T$) to 1.21x (PTBLM), achieving an average performance improvement of 1.12x. The reduction in execution time is due to avoiding the computation of neurons from gates of the recurrent cells that would be multiplied by an activation value that is close to saturation. Furthermore, the overhead of performing the comparison of the activation values with the thresholds is fairly small, since it is performed per output and not per connection, and not for all the cell gates. Furthermore, comparisons can be done in parallel with useful computations in the systolic array, hiding their latency by a large extent. As one output neuron normally requires hundreds of inputs, performing a comparison to detect whether the output of the neuron will be multiplied by an almost zero value can save hundreds of computations and memory accesses. GNMT and PTBLM exhibit the highest degree of activation values close to saturation (see Section 6.1) and, hence, they obtain the largest computation reduction and performance improvements.

The average power consumption of the CGPA accelerator is lower than $22W$. Figure 6.6 reports energy consumption normalized to the baseline. On average, CGPA reduces energy consumption of the accelerator by 12%. The energy overhead of the accelerator is less than 0.001% for all the networks. The most energy consuming component is the on-chip SRAM memory which represents 60% of the overall energy of the accelerator while the array of PEs and the main memory consume 10% and 30% respectively. The energy savings are strongly correlated with the percentage of activation values close to saturation and the computation reduction reported in Figure 6.4. These energy savings are due to two main reasons. First, dynamic energy is reduced due to the savings in computations and memory accesses. Second, the performance improvements shown in Figure 6.5 provide a reduction in static energy. Again, GNMT and PTBLM obtain the largest benefits, achieving a reduction of 14% and 20% in energy respectively.

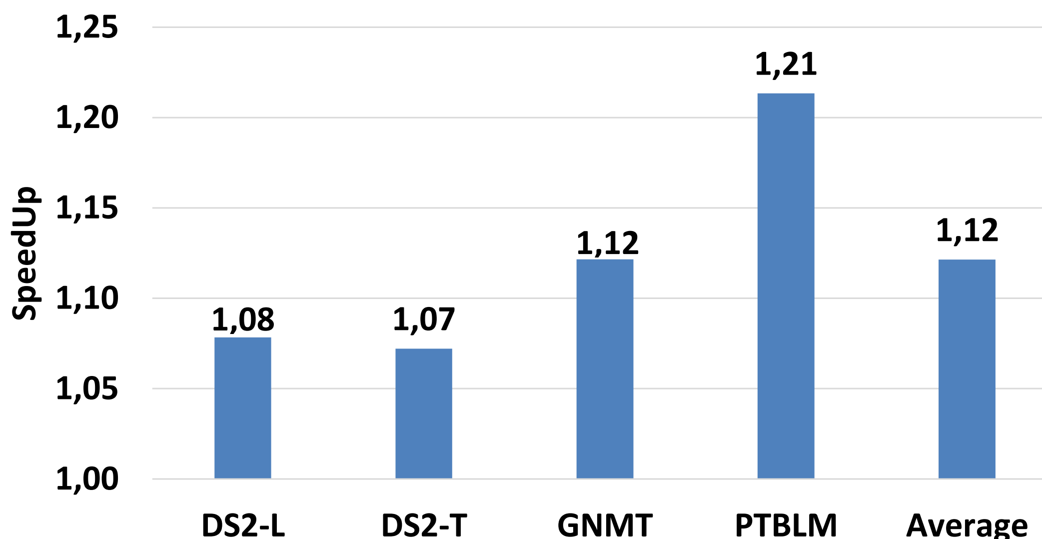


Figure 6.5: Speedups achieved for each RNN. Baseline configuration is the TPU-like accelerator without CGPA.

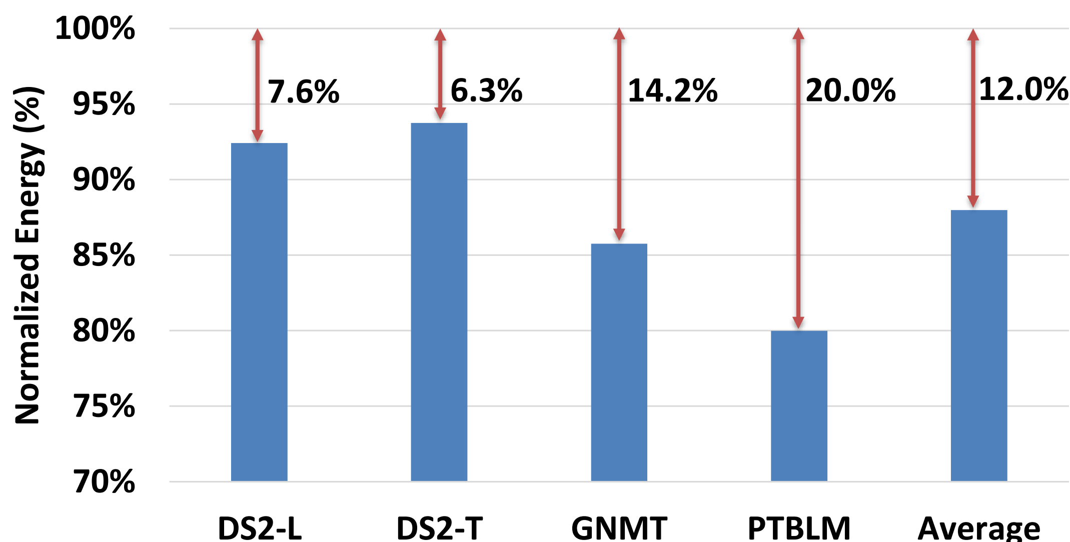


Figure 6.6: Normalized energy for each RNN. Baseline configuration is the TPU-like accelerator without CGPA

The above evaluation includes all the overheads due to CGPA: accesses to an small SRAM buffer of 512B and floating point comparisons of the activations. As it is shown in Figure 6.5 and Figure 6.6, these overheads are negligible in comparison to the savings in computations and memory accesses, and the net result is an improvement of 1.25x in energy-delay (1.12x in energy and 1.12x in delay). The overall area overhead of the accelerator is less than 0.003%, as it increases from $353.93mm^2$ to $353.94mm^2$.

7

PCA+UC: The (Pen-) Ultimate DNN Pruning

In the previous chapter, we presented a mechanism to dynamically prune activations of RNNs at a coarse granularity. We exploited the sigmoid and hyperbolic tangent activation functions resulting in close to saturation values, and the intrinsic element-wise multiplications of the LSTM and GRU equations, to reduce the number of computations and memory accesses of RNN accelerators. Continuing the research of pruning methodologies, in this chapter we first explore state-of-art methods to statically prune nodes and connections, and discuss their main weaknesses. Next, we present a new static pruning mechanism based on PCA and relative importance of each neuron's connection. We end this chapter by presenting the experimental results of our pruning mechanism and a comparison against prior works.

7.1 Weaknesses of Previous Static Pruning Schemes

In this section, we highlight the main weaknesses of popular DNN pruning schemes. For quantitative evaluations, we use three different DNNs, described in Section 3.3, whose parameters are shown in Table 7.1. *Kaldi* is an MLP for acoustic scoring, while *LeNet5* and *AlexNet* are popular CNNs for image recognition. LeNet5 classifies written digits and AlexNet classifies color images into 1000 possible classes that range from different animals to various types of objects.

We first evaluate the effectiveness of the schemes previously proposed to select the connections and/or nodes that are removed from the model, and compare them with a blind pruning scheme that randomly selects the connections/nodes. We report the accuracy loss and the amount of pruning achieved for the networks shown in Table 7.1. More specifically, we implemented the near-zero pruning (which applies to connections), and two of the node pruning methods: the similarity and the i-norm pruning (see Section 2.4). Although similarity pruning does not use retraining, we

Table 7.1: DNNs employed for the pruning study. Kaldi is an MLP for acoustic scoring, AlexNet is a CNN for image classification and LeNet5 is a CNN for digit classification. The table only includes Fully-Connected (FC) and Convolutional (CONV) layers, as these layers take up the bulk of computations in DNNs. Other layers, such as ReLU or Pooling, are not shown for the sake of simplicity.

Kaldi (18MB)			**AlexNet (200MB)**				**LeNet5 (12MB)**			
Accuracy: 89.51%			Accuracy: 57.48%				Accuracy: 99.34%			
Layer	Input Dim	Output Dim	Layer	In Dim	Out Dim	Kernel	Layer	In Dim	Out Dim	Kernel
FC1	360	360	CONV1	3*224*224	64*55*55	11*11	CONV1	1*28*28	32*28*28	5*5
FC2	360	2000	CONV2	64*27*27	192*27*27	5*5	CONV2	32*14*14	64*14*14	5*5
FC3	400	2000	CONV3	192*13*13	384*13*13	3*3	FC1	7*7*64	1024	-
FC4	400	2000	CONV4	384*13*13	384*13*13	3*3	FC2	1024	10	-
FC5	400	2000	CONV5	384*13*13	256*13*13	3*3				
FC6	400	3482	FC1	5*5*256	4096	-				
			FC2	4096	4096	-				
			FC3	4096	1000	-				

include it in all the methods for a fair comparison. We report results for different overall pruning percentages starting from 10% and increasing it by steps of 10%. The parameters of each pruning scheme are manually adjusted to attain the target percentage of global pruning. For instance, for near-zero pruning, we tried different values of the *quality parameter (qp)* until the target percentage of pruning was attained. Note that the percentage of pruning applied to each individual layer is determined by the particular heuristics used by each method and is not uniform across layers, i.e. some layers are pruned more aggressively than others.

Figure 7.1 shows the comparison between near-zero and random pruning of connections in terms of Word Error Rate (WER is the main metric used in speech recognition; lower is better) for the Kaldi DNN. We can observe that for 10-20% pruning both methods achieve very similar accuracy. For 30-80% pruning, near-zero is slightly better, and for 90% pruning random is slightly more accurate. Random pruning achieves up to 50% of pruning with negligible accuracy loss, whereas near-zero pruning achieves up to 70%. Therefore, the near-zero scheme can achieve up to 20% more pruning, but the random scheme still performs quite well. To further reinforce this conclusion, we performed multiple tests with the random scheme using different seeds to obtain different pruning patterns. For all the random experiments the accuracy obtained after retraining was almost the same, with smaller differences of less than 0.2%.

Figure 7.2 shows the comparison between the different methods to prune nodes for the Kaldi DNN. We observe very minor differences in terms of accuracy among all the methods, so there is no clear winner. For node pruning, the last layer cannot be pruned since these neurons generate the output values that are used by the application. For instance, in Kaldi these are the probabilities used by the Viterbi beam search. Therefore, the maximum degree of pruning that can be achieved is around 60% of the nodes. We can see that the differences in WER are less than 1% in all the cases, random being slightly better when the global pruning is high (50% and 60%).

Results for LeNet5 are shown in Figure 7.3 for link pruning and Figure 7.4 for node pruning. In this case, LeNet5 has a high tolerance to errors and the accuracy is well maintained until pruning 90% of the network for both types of pruning. At that point, the accuracy starts to decrease, being the random scheme slightly worse in the case of link pruning, but only by around 1%, whereas for node pruning there are no significant differences between random, i-norm and similarity schemes

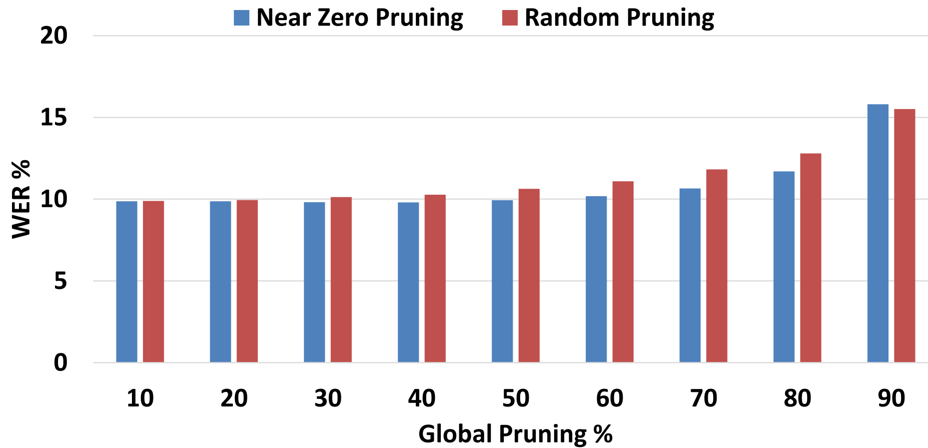


Figure 7.1: Comparison between near-zero and random pruning of the Kaldi DNN for different percentages of global pruning.

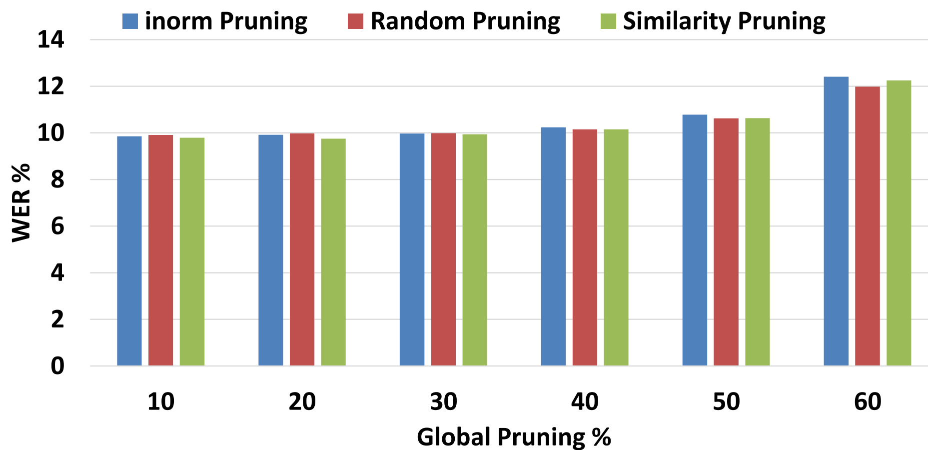


Figure 7.2: Comparison between i-norm, similarity and random pruning of the Kaldi DNN for different percentages of global pruning.

for all percentages of pruning.

Figure 7.5 shows the comparison between near-zero and random pruning of connections in terms of Top-1 accuracy for AlexNet. We can observe that up to 50% of pruning the accuracy is largely recovered for both methods. Then, the random pruning starts to decrease the accuracy while the near-zero is able to maintain it until 80%, and beyond that point it drops significantly.

Note that some studies in the literature report results for pruning that require the exploration of a huge design space, which may be impractical for some DNNs. For instance, the near-zero pruning scheme could obtain better results by applying a different quality parameter (qp) for each network layer, as reported in the original paper [26]. However, trying different values of qp for each layer requires to explore an exponential number of configurations (exponential with the number of layers). Taking into account that for each configuration a retraining is needed, and retraining is extremely costly (can take several days in large networks), we claim that such strategies based on

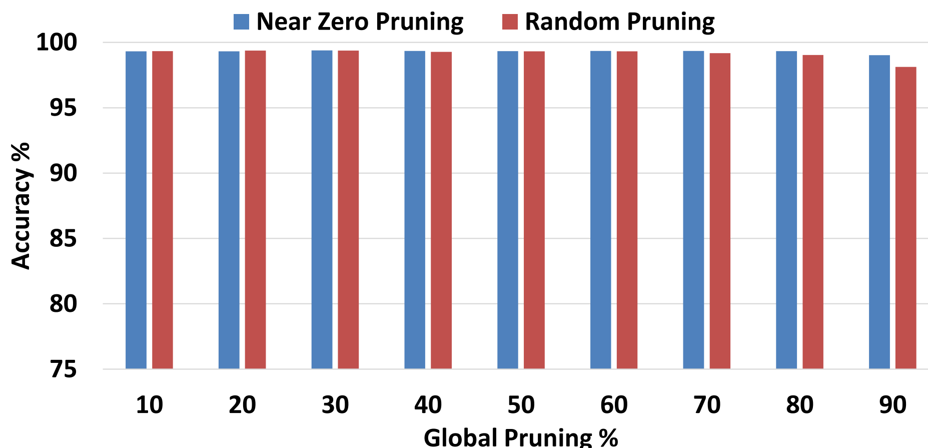


Figure 7.3: Comparison between near-zero and random pruning of LeNet5 for different percentages of global pruning.

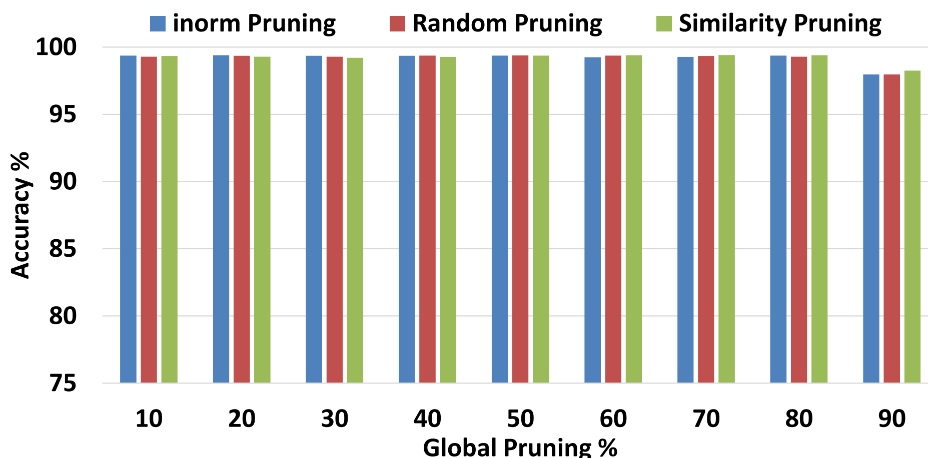


Figure 7.4: Comparison between i-norm, similarity and random pruning of LeNet5 for different percentages of global pruning.

exploring an exponential number of configurations are impractical for many contemporary DNNs, which have hundreds of layers.

In summary, we observed that in terms of accuracy the random node pruning behaves almost equal to the rest of the methods. On the other hand, the random pruning of connections is somewhat less effective than the analyzed schemes. In other words, the heuristics to choose which neurons to prune are irrelevant while the heuristics to choose the connections may impact the final result. Previous works [57] have done a similar analysis with a different set of applications and neural networks (i.e. VGG [83], ResNet [31] and DenseNet [38]), which reinforce the observations made in this work. Besides, a main weakness of current pruning methods is their need of tuning one or multiple parameters through a trial and error process. This is extremely costly since each experiment requires retraining the network, and is impractical if the number of configurations to explore is too high (e.g. exponential with number of network layers).

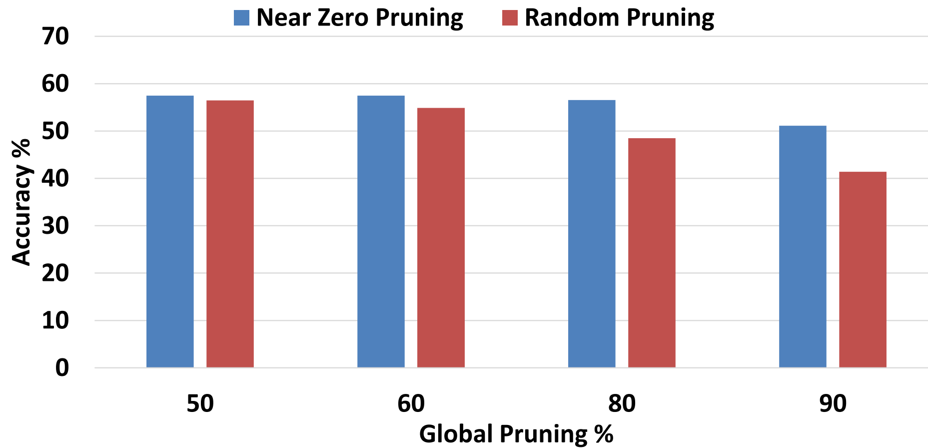


Figure 7.5: Comparison between near-zero and random pruning of AlexNet for different percentages of global pruning.

7.2 PCA+UC Pruning Method

The proposed pruning method of this work consists of two main steps. First, it performs a node pruning based on a PCA analysis of the data produced by each fully-connected layer. Next, some of the remaining connections are pruned based on their importance relative to the rest of incoming connections of the same neuron. The proposed scheme is not iterative and requires only two retraining operations, one after each of the two steps. We refer to it as PCA plus Unimportant Connections (PCA+UC) pruning.

7.2.1 Node Pruning Through PCA

The first step of the proposed approach is to prune redundant neurons in each layer through a Principal Components Analysis (PCA) [91]. PCA is a well-known statistical method to summarize data, and is typically used to reduce the dimensionality of a dataset. PCA transforms a set of observations from different variables with high correlation into a set of principal components without linear correlation. Therefore, one of its main usages is to determine redundancy.

In the context of DNN pruning, PCA is used to reduce the number of nodes of a layer. Each layer can be regarded as a system that for each evaluation generates an output value represented as a n-dimensional vector, where n is the number of neurons of this layer. PCA allows us to represent a set of n-dimensional values in a different coordinate system, without loosing any information, by applying a linear transformation:

$$
\begin{aligned}
New_1 &= Old_1 * a_1 + Old_2 * b_1 + \cdots + Old_n * z_1 \\
New_2 &= Old_1 * a_2 + Old_2 * b_2 + \cdots + Old_n * z_2 \\
&\cdots \\
New_n &= Old_1 * a_n + Old_2 * b_n + \cdots + Old_n * z_n
\end{aligned}
\tag{7.1}
$$

Besides, in the new system the components are orthogonal (i.e., they do not contain any redundant information) and are ordered from higher to lower variance (i.e., from more to less information). If the original data presents high correlation among some of the n dimensions, in the transformed coordinate system the last components will have very low variance. If we remove these low-variance components, we can represent the data in a lower-dimensional system with practically no loss of information.

In our case, we use PCA only to determine how many neurons we need to preserve in each layer. The retraining process applied after pruning will adjust the weights so that the output of the pruned layer is equivalent to computing the original neurons and then applying the linear transformation dictated by the PCA. In other words, the original n-dimensional output is not needed, and the pruned network is expected to produce the same results as if the original n-dimensional output was computed and the PCA transformation was applied afterwards.

The steps to apply the PCA-based pruning are as follow. First, we generate a trace of the outputs of the nodes of each fully-connected layer (outputs are taken after the activation function). A subset of the training dataset can be used to generate the trace. In our experiments, we use around 1% of the training set of each DNN to generate these traces. Then, we apply the PCA to the trace, which gives us the variance coefficients in the transformed coordinate system. Next, we compute how many of the lower-variance components can be removed while still keeping 95% of the original variance. The number of remaining components is the number of neurons that we keep for this layer. Which nodes to keep and which are removed is irrelevant as demonstrated in the previous section, so they are randomly chosen. Once we determine the number of neurons in all layers, a single retraining of the pruned network is performed.

PCA cannot be directly applied to convolutional layers because a full feature map is considered as a single node which generates a volume of data. Since removing entire feature maps also has a high impact in accuracy we decided not to apply this step to convolutional layers. Note that previously proposed schemes for node pruning are also very inefficient for convolutional layers for the same reason.

Figure 7.6 shows the cumulative variance of a sample fully-connected layer of AlexNet, Kaldi and LeNet5. For the sample layer of AlexNet, we can see that 50% of the nodes keep 95% of the original information, so 50% can be pruned. For the sample layer of Kaldi, we can see that 60% of the nodes keep 95% of the original information, so 40% can be pruned. Finally, for the sample layer of LeNet5, the benefits are much higher since we can remove 70% of the nodes while still keeping 95% of the information.

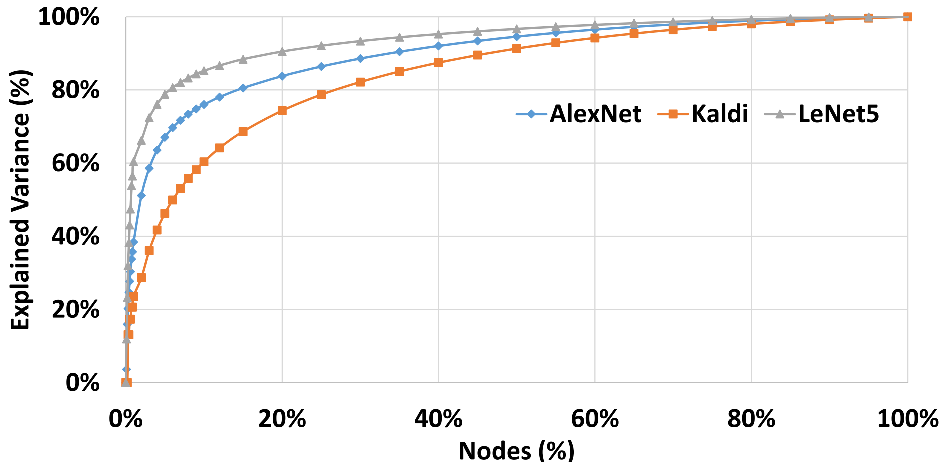


Figure 7.6: Cumulative variance of a sample layer of AlexNet, Kaldi and LeNet5.

7.2.2 Pruning Unimportant Connections

The second step of the method consists of pruning unimportant connections after removing the redundant neurons. The rationale behind this approach is the following. The node pruning performed in the first step gives us a network with the minimal number of neurons in each fully-connected layer, while keeping the original accuracy. In the resulting topology there is no opportunity to remove further neurons; however, some of the connections may still have minor impact and can be removed to further reduce the size of the network. The obvious case are connections whose weight is zero. They clearly can be ignored without affecting the output. We could also remove all connections whose weight is close to zero, as the near-zero pruning [26] does. However, the importance of a connection is not necessarily related to how close to zero its weight is. For instance, a weight of 0.1 would be unimportant if the rest of the weights for the same neuron are in the order of 1 or greater, but will be important if the rest of the weights are similar or smaller. In a similar manner, a connection with a weight very different to zero, say for instance 10, will be unimportant if the rest of the connections of the same neuron are in the order of 100.

We propose to measure the importance of a connection as a function of the absolute value of its weight and the average absolute value of all the incoming weights of the same neuron. In other words, a connection is considered unimportant if the magnitude of its weight is small compared with the other weights of the same node. This step is applied to all layers including convolutionals where a full feature map is considered as a single node.

A first idea to measure the importance of a connection would be to compute the ratio of the absolute value of its weight to the average absolute value of all weights of the same neuron. This works relatively well if the distributions of the weights are centered around zero for all neurons. Besides, this metric is insensitive to scaling. That is, if all weights are multiplied by a given constant, the pruning scheme would still affect the very same weights. However, this metric is not insensitive to displacements (translations). That is, if we have another neuron whose weights are about the same but with an added offset (i.e, each weight of the new neuron results from adding a constant to a different weight of the other neuron), this metric would result in a different pruning,

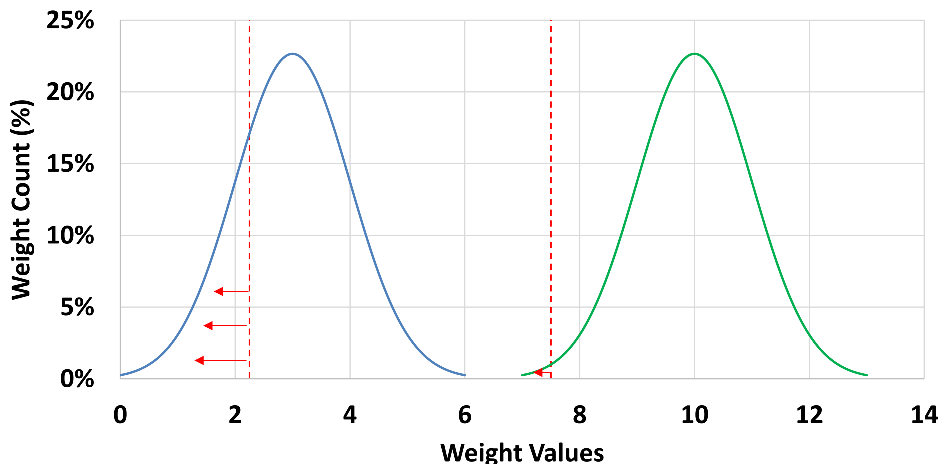


Figure 7.7: Example of the weights distribution translation pruning problem.

in spite of the fact that the weight distribution of the two neurons have exactly the same shape, one being displaced by a constant with respect to the other. Figure 7.7 illustrates the translation problem with an example of two distributions of weights. Assuming a threshold of 75% of the mean, the weights on the left side of the red line would be pruned in each case. We can see that the number of weights pruned for the distribution centered on three would be much higher than for the distribution centered on ten, although the only difference between them is a translation of seven.

We want a metric to measure the importance of a connection that is insensitive to translation and scaling of the weights. To this end, we first take the absolute value of all weights. Then, for each neuron, we substract the minimum absolute value of the weights of this node to the rest of the weights of that node. Finally, for each node we compute the mean of the resulting values and remove the connections whose value is smaller than 75% of the mean.

An observation to make is that after applying the pruning of connections the resulting network model will be sparse. Executing a sparse model is normally less efficient than a dense model so the pruning ratios achieved by this step must be significant to compensate for this penalty.

To summarize, the proposed scheme consists of two steps. First, we perform a node pruning based on a PCA analysis, to keep the minimum number of nodes that generate practically the same information as the original ones. Then, we remove the remaining unimportant connections, as identified by those connections whose weight has an absolute value that is small compared with the rest of the connections of the same node. Retraining is applied only once after each of the two steps.

7.3 Evaluation Methodology

The goal of the experimental evaluation of this study is to prove that the PCA+UC pruning scheme provides pruning ratios that are similar or better than previous schemes in spite of not being

iterative, unlike previous schemes, which makes them very costly or impractical. The schemes used for comparison are the **Baseline** where no pruning is applied, the **Near Zero Weights** using the connection pruning method described in Section 2.4.1, the **Input Weights Norm** which applies the node pruning method described in Section 2.4.2, the **Similarity** node pruning method described in Section 2.4.3, and the **Random Weights** and **Random Nodes** which are simplistic methods that randomly choose connections or nodes to prune given a target percentage. We do not include results for Scalpel (Section 2.4.4) and PCA Pruning (Section 2.4.5) since they perform worse than the above schemes.

To evaluate the pruning ratios achieved by PCA+UC, we evaluate it on three state-of-the-art DNNs for two different application domains, one for acoustic scoring in speech recognition and two for image classification. We use the DNNs shown in Table 7.1. All the networks and pruned models have been implemented in Tensorflow [22]. For all the DNNs, we employ the whole test or validation sets from their respective datasets, including several hours of audio and a large number of images, to assess the efficiency of the pruning methods in terms of accuracy and pruning ratios.

7.4 Experimental Results

This section evaluates the proposed PCA+UC pruning scheme, described in Section 7.2, on different DNNs in terms of accuracy and amount of pruning. First, we present a sensitivity analysis to setup appropriate threshold parameters for our pruning scheme. Then, we compare our method with some previously proposed pruning strategies.

7.4.1 Sensitivity Analysis

The proposed pruning scheme uses two thresholds to control the amount of pruning and the loss of accuracy. Our key goal is to find values for these parameters that achieve high efficiency, i.e. large amount of pruning with negligible accuracy loss, for a wide range of DNNs. Therefore, the user does not have to manually tune these thresholds for each specific DNN, as it happens with other pruning schemes.

In the PCA+UC scheme, the first step (node pruning) is applied by keeping only as many neurons as PCA components are needed to represent 95% of the original information (coefficient of variance). In the second step, connections are pruned based on its relative importance compared to the other connections of the same neuron.

We performed a sensitivity analysis using the Kaldi DNN since the accuracy of Kaldi is highly sensitive to model changes and the model size is reasonable. The parameters obtained from this analysis are used for all the networks, achieving the results shown in Section 7.4.2, which confirms that these parameters work well for different networks.

Table 7.2 shows the results for the Kaldi DNN after applying the first step of the PCA+UC pruning method using different thresholds for the coefficient of variance (CV). The accuracy of Kaldi is measured as Word Error Rate (WER), so lower is better. As it can be seen, with 99% of

Table 7.2: Kaldi results after the PCA step (first step) of the proposed pruning scheme using different coefficients of variance. (Baseline WER=10.04%)

CV (%)	WER (%)	Weights Pruned(%)	FLOPS Removed(%)
99	10.00	15	15
95	10.24	45	45
90	10.62	64	64

Table 7.3: Kaldi results after the unimportant connections pruning step (second step) of the proposed pruning scheme for different thresholds of the mean. (Baseline WER=10.04%)

CV(%)	Mean(%)	WER(%)	Weights Pruned(%)	FLOPS Removed(%)
95	100	10.62	77	77
95	75	10.28	70	70
95	50	10.26	63	63

variance we can maintain the same accuracy or even slightly better, but the percentage of pruning is dramatically reduced to only 15%. In contrast, if we use 90% of variance, the pruning is quite high (64%) but the accuracy is slightly affected (0.58% loss). Finally, if we use 95% of variance the accuracy is almost the same (only 0.2% of loss) and the percentage of pruning is 45%. Therefore, we use 95% of coefficient of variance as the by default value for our scheme. In case that a small loss of accuracy could be assumed, 90% of CV could be a good choice in order to achieve a higher degree of pruning.

Table 7.3 shows the results after the second step of the PCA+UC pruning method using different thresholds for the percentage of the mean (computed as described in Section 7.2.2), which determines when a connection is pruned for the Kaldi DNN. We can conclude that 75% is the most adequate threshold. A higher threshold can prune more links but looses some accuracy, whereas a lower threshold has about the same accuracy but is less effective at pruning.

In summary, we set the by default pruning configuration to work with 95% for the coefficient of variance to remove nodes and 75% of the mean of the weights of each node to remove connections.

7.4.2 PCA+UC Evaluation

The main benefit of PCA+UC is the time required to complete the pruning. For instance, AlexNet takes around 3 days to finish a retraining step on a GTX 1080 GPU. PCA+UC requires two retraining steps so it will take 2x the retraining time, i.e. less than a week for AlexNet. On the other hand, for the methods that require exploring multiple configurations per layer such as the Near-Zero pruning, finding the appropriate pruning percentages for all layers requires n^l retraining steps, being n the number of configurations analyzed per layer and l the number of layers. Since AlexNet has eight layers, even for a very low value of n such as 3, the pruning process would take

Table 7.4: Accuracy (WER) and percentage of weights and computations removed by different pruning schemes for the Kaldi DNN.

Pruning Method	WER(%)	Weights Pruned(%)	FLOPS Removed(%)
Baseline	10.04	0	0
Near Zero Weights	10.18	60	60
Random Weights	10.27	40	40
Input Weights Norm	10.24	40	40
Similarity	10.15	40	40
Random Nodes	10.15	40	40
PCA+UC	10.28	70	70

around 54 years in a high-end GPU, or half a year in a farm of 100 GPUs. Considering that current DNNs, such as Resnet152 or Densenet201, include hundreds of layers the applicability of previous methods may not be feasible.

Table 7.4 shows the pruning effectiveness for the Kaldi DNN. For each pruning scheme we report the maximum pruning we could achieve with negligible accuracy loss (less than 0.25% in all cases). We can see that PCA+UC achieves the highest degree of pruning, resulting in a 70% reduction of the weights and 70% reduction of the number of computations. The next best scheme is the near-zero pruning, which achieves a 60% reduction in both weights and computations. Since in Kaldi all the layers are fully-connected, the reduction in weights and the reduction in computations is the same. Note that the PCA+UC scheme only requires to retrain the DNN twice whereas for the other methods we have to carry out an iterative search to find the maximum percentage of pruning with negligible accuracy loss, and the DNN has to be retrained for each pruning percentage.

Table 7.5 shows the results for the LeNet5 DNN. In this case, accuracy is measured as the top-1 so higher is better. Unlike Kaldi, in LeNet5 there are both convolutional and fully-connected layers. Most computations come from the convolutional layers while most of the weights are due to the fully-connected layers. Since some node pruning methods such as the similarity pruning can only be applied to fully-connected layers, they achieve a significant reduction in weights, but quite moderate in computations. We can observe that PCA+UC prunes 79% of the weights and 52% of the computations, which is the second best in terms of weight and computation reduction, only slightly below the near-zero.

Table 7.6 shows the results for AlexNet. We can observe that PCA+UC prunes 67% of the weights and 51% of the computations. It does not prune the convolutionals as much as the previous heuristics so the reduction in computations is lower, but it achieves higher accuracy.

We have also evaluated the Unimportant Connections (UC) pruning alone to demonstrate that it is more effective than Near-Zero pruning. Table 7.7 shows the results for 70-90% of pruning of the Kaldi DNN. We can see that UC achieves better accuracy with the same amount of pruning and, hence, taking into account the importance of the connections relative to each node is more effective than just considering the magnitude of all the weights on each layer.

Table 7.5: Accuracy (Top-1) and percentage of weights and computations removed by different pruning schemes for LeNet5.

Pruning Method	Top-1(%)	Weights Pruned(%)	FLOPS Removed(%)
Baseline	99.34	0	0
Near Zero Weights	99.33	80	80
Random Weights	99.31	60	60
Input Weights Norm	99.37	87	20
Similarity	99.4	87	20
Random Nodes	99.29	87	20
PCA+UC	99.41	79	52

Table 7.6: Accuracy (Top-1) and percentage of weights and computations removed by different pruning schemes for AlexNet.

Pruning Method	Top-1 (%)	Weights Pruned(%)	FLOPS Removed(%)
Baseline	57.48	0	0
Near Zero Weights	56.55	80	87
Random Weights	56.46	50	63
PCA+UC	59.7	67	51

Table 7.7: Link Pruning Comparison for the Kaldi DNN. (Baseline WER=10.04%)

Weights Pruned (%)	Near-Zero WER (%)	UC WER (%)
70	10.67	10.53
80	11.39	11.10
90	14.62	13.56

To summarize, using the pruning configuration parameters described in Section 7.4.1, PCA+UC achieves pruning percentages of 70% in Kaldi, 79% in LeNet5 and 67% in AlexNet with negligible accuracy loss.

8

Conclusions and Future Work

In this chapter, the main conclusions and contributions of this thesis are summarized, and some open-research areas for future work are presented.

8.1 Conclusions

The purpose of this thesis is to design low-power accelerators for cognitive computing applications, by providing efficient hardware architectures and techniques to improve performance and energy-efficiency of DNN-based systems.

In this thesis, we first show that many modern DNNs usually process a sequence of data (e.g., frames of audio or video) and consecutive neuron's inputs exhibit a high degree of similarity for all neurons, including the hidden layers. Then, we propose a new mechanism that exploits this similarity, in order to avoid a large percentage of the activity by reusing the results of the previous execution. We analyze the degree of input similarity for several DNNs and observe that, if linear quantization is used, on average more than 60% of the inputs of fully-connected, convolutional and recurrent layers remain unmodified with respect to the previous execution. Based on this observation, we propose DISC, a hardware extension for DNN accelerators to exploit this similarity.

The proposed reuse scheme checks which inputs have changed with respect to the previous execution. Inputs that remain unmodified are ignored, avoiding the associated computations and memory accesses, whereas modified inputs are used to correct the previous output result of each neuron in that layer. We implement the reuse scheme on top of a state-of-the-art DNN accelerator. We show that DISC requires minor changes, mainly additional memory storage for saving the outputs of the DNN layers. Our experimental results show that, on average, DISC provides 63% energy savings and 3.5x speedup, while it only requires a minor increase in the area of the accelerator

(less than 1%). The scheme works for MLPs, CNNs and RNNs from different applications, including speech recognition, video classification and self-driving cars.

In order to further improve the inference of FC-based DNNs, we propose CREW, a Computation Reuse and Efficient Weight Storage scheme that exploits the spatial locality of repeated weights. In the second contribution of this thesis, we show that modern DNNs exhibit a high degree of weight repetition in FC layers, resulting in a low number of unique weights per input neuron. Then, we implement CREW on top a TPU-like accelerator, exploiting weight repetition to reduce multiplications and memory accesses. The proposed reuse scheme only performs the multiplications of the inputs by their respective unique weights, avoiding many redundant computations and their associated memory accesses. The final product accumulations are performed by indexing a buffer that stores the results of the partial products. Besides, the indices required to access the buffer of partial products are small, since their size depend on the number of unique weights, thus the total network model size is significantly reduced. CREW also includes an enhanced weight stationary dataflow that processes blocks of indices and partial products.

We show that CREW requires minor hardware changes over a state-of-the-art accelerator, mainly additional memory storage for saving the partial products and the indices. Our experimental results show that, on average, CREW provides 2.42x energy savings and 2.61x speedup, while it only requires a minor increase in the area of the accelerator (less than 9%). We show that the scheme works for any DNN composed of FC layers such as MLPs and RNNs from different applications, including speech recognition and machine translation.

After proposing different computation reuse techniques to optimize the inference of DNN accelerators, we observed that the memory hierarchy still remains the main bottleneck of these systems, due to the huge number of DNN parameters to process. In consequence, we explore various pruning mechanisms to reduce the model size and the number of computations, including static/dynamic and node/link pruning mechanisms.

In the third proposal of this thesis, we show that modern RNNs exhibit a significant percentage of activation values saturated towards one or zero in different gates of LSTM and GRU cells. We propose CGPA, a Coarse-Grained Dynamic Pruning of Activations technique that exploits these saturated activation values to save computations and memory accesses. As the output of neurons from different gates are multiplied element-wise, CGPA skips the evaluation of a neuron whenever the output of its peer neuron from a different gate is saturated. We implement CGPA on top of a TPU-like accelerator. Our experimental results show that, on average, CGPA provides 12% energy savings and 1.12x speedup, while it only requires a minor increase in the area of the accelerator (less than 0.003%). CGPA provides benefits in performance and energy consumption for RNNs from different applications, including speech recognition, machine translation and language modeling.

Finally, we explore various methods of static weight pruning to further reduce the DNN model size and the number of computations. In the last work of this thesis, we focus on static pruning methodologies. We show that current DNN pruning schemes require manual tuning of multiple parameters by means of a trial and error process, often resulting in an exponential number of configurations to be evaluated, which may be impractical since each experiment requires a retraining of the network, which is an extremely costly operation. We propose the PCA+UC pruning scheme that overcomes this weakness. PCA+UC consists of two steps, one to remove redundancy at

the neuron level, and another to remove redundancy at the connection level. The first step is based on a Principal Component Analysis (PCA) to remove nodes while the second step removes unimportant connections (UC) based on a novel metric that measures the importance of each link. The proposed scheme requires only two retraining operations and achieves results similar or even better than state-of-the-art iterative pruning methods.

8.2 Contributions

In this thesis, different techniques and schemes, from software to hardware level, have been proposed to improve the performance and energy efficiency of cognitive accelerators for DNN inference. As a result of this thesis, we provide several accelerator designs considering the different trade-offs of the hardware architectures and the DNN models in terms of performance, energy consumption and DNN accuracy. The main contributions obtained through this dissertation are summarized as follows.

In first place, we perform an analysis of the input similarity between consecutive DNN executions of sequence processing applications. Then, based on the high degree of input similarity, we propose DISC, a hardware accelerator implementing a Differential Input Similarity Computation technique to reuse the computations of the previous execution, instead of computing the entire DNN. We observe that, on average, more than 60% of the inputs of any neural network layer tested exhibit negligible changes with respect to the previous execution. Avoiding the memory accesses and computations for these inputs results in 63% energy savings on average. This work has been published in the 45th IEEE/ACM International Symposium on Computer Architecture (ISCA):

- Marc Riera, Jose-Maria Arnau and Antonio González, "Computation Reuse in DNNs by Exploiting Input Similarity," ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA), Los Angeles, CA, 2018, pp. 57-68.

In second place, we propose to further optimize the inference of FC-based DNNs. We first analyze the number of unique weights per input neuron of several DNNs. Exploiting common optimizations, such as linear quantization, we observe a very small number of unique weights per input for several FC layers of modern DNNs. Then, to improve the energy-efficiency of FC computation, we present CREW, a hardware accelerator that implements a Computation Reuse and an Efficient Weight Storage mechanism to exploit the large number of repeated weights in FC layers. CREW greatly reduces the number of multiplications and provides significant savings in model memory footprint and memory bandwidth usage. We evaluate CREW on a diverse set of modern DNNs. On average, CREW provides 2.61x speedup and 2.42x energy savings over a TPU-like accelerator. Compared to UCNN, a state-of-art computation reuse technique, CREW achieves 2.10x speedup and 2.08x energy savings on average. This work has been submitted for publication and is currently under review:

- Marc Riera, Jose-Maria Arnau and Antonio González, "CREW: Computation Reuse and Efficient Weight Storage for Hardware-accelerated MLPs and RNNs," Under Review, 2020.

In third place, we propose a mechanism to optimize the inference of RNNs, including both LSTM and GRU architectures. RNN cells perform element-wise multiplications across the activations of different gates, sigmoid and tanh being the common activation functions. We perform an analysis of the activation function values, and show that a significant fraction are saturated towards zero or one in popular RNNs. Then, we propose CGPA to dynamically prune activations from RNNs at a coarse granularity. CGPA avoids the evaluation of entire neurons whenever the outputs of peer neurons are saturated. CGPA significantly reduces the amount of computations and memory accesses while avoiding sparsity by a large extent, and can be easily implemented on top of conventional accelerators such as TPU with negligible area overhead, resulting in 12% speedup and 12% energy savings on average for a set of widely used RNNs. This work has been published in the IEEE Micro Journal:

- Marc Riera, Jose-Maria Arnau and Antonio González, "CGPA: Coarse-Grained Pruning of Activations for Energy-Efficient RNN Inference," in IEEE Micro, vol. 39, no. 5, pp. 36-45, 1 Sept.-Oct. 2019.

Finally, in the last contribution of this thesis we focus on static DNN pruning methodologies. We highlight the weaknesses of state-of-the-art pruning methods and solve them by proposing a more effective and practical scheme. DNN pruning reduces memory footprint and computational work of DNN-based solutions by removing connections and/or neurons that are ineffectual. Sparse DNN accelerators efficiently execute the pruned models and improve the performance and energy consumption over typical dense accelerators such as TPU. However, we show that prior pruning schemes require an extremely time-consuming iterative process that requires retraining the DNN many times to tune the pruning hyperparameters. Then, we propose a DNN pruning scheme based on Principal Component Analysis and relative importance of each neuron's connection that automatically finds the optimized DNN in one shot without requiring hand-tuning of multiple parameters. This work has been submitted for publication and is currently under review:

- Marc Riera, Jose-Maria Arnau and Antonio González, "(Pen-) Ultimate DNN Pruning," Under Review, 2020.

8.3 Open-Research Areas

The work proposed in this thesis can be extended by improving the hardware accelerated DNN systems to make them more adaptive for new, more complex DNN models that are to come in the near future. As described in Chapter 2, the state-of-the-art DNNs are fastly growing and evolving in order to reach accuracy levels similar or even higher than those of humans. DNNs are not only used in simple classification tasks but in a wide range of areas such as autonomous driving or health care systems. Even the most mature areas of computer systems are beginning to use machine learning techniques. The growing interest in expanding the areas of use of DNNs together with their constant evolution to achieve better accuracy levels in multiple applications, may lead to new types of deep learning algorithms and models. A clear example is the recent residual neural networks [31, 38] and the attention mechanisms [60, 6], used in the Transformer [93] model, which have considerably

improved the efficiency of sequence processing applications. These new DNN models and algorithms introduce new difficulties and challenges to be efficiently executed, especially for low-power devices such as mobile and embedded systems. Therefore, our DNN accelerators and techniques should be extended to accommodate these new DNN architectures that impose even higher computation and memory requirements.

On the other hand, in this thesis we have focused on proposing accelerators and techniques to perform efficient DNN inference. However, DNN training is also a costly procedure that may take several days/weeks for each DNN to complete given a known model with a new dataset, or even months when building a new model from scratch. Furthermore, some cognitive applications, such as the ones used in health care systems, require the DNN models to be constantly updated with new information, which makes on-device training for mobile and embedded systems an interesting approach to explore in the future. The training procedure has two main steps, the forward pass and the backward pass. Although the computations performed for DNN training and DNN inference are similar because the forward pass is the same in both, the training phase requires more computations and memory storage due to the backward pass, making its execution much more complex and challenging for low-power devices. Therefore, another interesting path of research for future work is to develop DNN accelerators for both efficient training and inference.

As described in Section 2.3.2, pruning methods and sparse accelerators are gaining popularity to reduce the model size and the number of computations to perform DNN inference. In this thesis, we have explored a variety of static and dynamic pruning schemes. However, there is still potential to improve these methods by using different metrics to detect the significance of the nodes and connections, new mechanisms such as the ones targeting specific structures to prune, or techniques to prune during the training of the DNN from scratch while testing multiple hyperparameters. In addition, in this thesis we have not explored the design of sparse accelerators, which are required to efficiently execute the pruned models. Therefore, accelerating sparse DNN models through customized architectures is also a challenging and interesting topic. Moreover, the need to prune the DNN models is given because they are created oversized from the start, causing many connections and neurons to be redundant. Another interesting research path for future work would be the design of new methodologies to build the DNN models from scratch, while trying to avoid as much as possible the common oversizing and redundancy of the neural networks.

Modern cognitive applications usually require the integration of several components in order to achieve high accuracy. As an example, Automatic Speech Recognition (ASR) systems include not only the DNN model for acoustic scoring but also a language rescoring model plus a decoder or graph search algorithm to generate the final outputs. Other popular applications such as real-time speech translation may require multiple DNNs to be execute in parallel. Each of these components or stages may have a unique computational behavior with specific properties, requiring their own customized hardware architecture in order to achieve a highly efficient system. Thus, the design of an heterogeneous system targeted for mobile and embedded devices, which are operated with a small form-factor battery, is required to obtain the best trade-off in the sense of real-time performance, energy consumption and area of the chip.

Finally, looking at a more distant future our goal would be to design a re-configurable cognitive computing accelerator that supports all kind of DNN models, and in general any machine learning algorithm, including multiple mechanisms and techniques to efficiently execute different applica-

tions. This would not only require a more complex and flexible hardware architecture than the current ones, but also better software support including user interfaces and compilers to efficiently integrate the execution of the different applications, dataflows and optimizations. Due to the fastly evolving machine learning algorithms and applications, we are still far away from achieving this goal, but even so, it is a line of research of great interest and with much potential ahead.

Bibliography

[1] V. Akhlaghi, A. Yazdanbakhsh, K. Samadi, R. K. Gupta, and H. Esmaeilzadeh. Snapea: Predictive early activation for reducing computation in deep convolutional neural networks. In *ISCA*, 2018.

[2] Jorge Albericio, Patrick Judd, Tayler Hetherington, Tor Aamodt, Natalie Enright Jerger, and Andreas Moshovos. Cnvlutin: Ineffectual-neuron-free deep neural network computing. In *ISCA*, 2016.

[3] Manoj Alwani, Han Chen, Michael Ferdman, and Peter Milder. Fused-layer cnn accelerators. In *MICRO*, 2016.

[4] Dario Amodei, Sundaram Ananthanarayanan, Rishita Anubhai, Jingliang Bai, Eric Battenberg, Carl Case, Jared Casper, Bryan Catanzaro, Qiang Cheng, Guoliang Chen, Jie Chen, Jingdong Chen, Zhijie Chen, Mike Chrzanowski, Adam Coates, Greg Diamos, Ke Ding, Niandong Du, Erich Elsen, Jesse Engel, Weiwei Fang, Linxi Fan, Christopher Fougner, Liang Gao, Caixia Gong, Awni Hannun, Tony Han, Lappi Johannes, Bing Jiang, Cai Ju, Billy Jun, Patrick LeGresley, Libby Lin, Junjie Liu, Yang Liu, Weigao Li, Xiangang Li, Dongpeng Ma, Sharan Narang, Andrew Ng, Sherjil Ozair, Yiping Peng, Ryan Prenger, Sheng Qian, Zongfeng Quan, Jonathan Raiman, Vinay Rao, Sanjeev Satheesh, David Seetapun, Shubho Sengupta, Kavya Srinet, Anuroop Sriram, Haiyuan Tang, Liliang Tang, Chong Wang, Jidong Wang, Kaifu Wang, Yi Wang, Zhijian Wang, Zhiqian Wang, Shuang Wu, Likai Wei, Bo Xiao, Wen Xie, Yan Xie, Dani Yogatama, Bin Yuan, Jun Zhan, and Zhenyao Zhu. Deep speech 2 : End-to-end speech recognition in english and mandarin. In *ICML*, 2016.

[5] Arash Ardakani, Carlo Condo, and Warren J Gross. Activation pruning of deep convolutional neural networks. In *GlobalSIP*, 2017.

[6] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. Neural machine translation by jointly learning to align and translate. In *ICLR*, 2015.

[7] Berkeley AI Research (BAIR). Caffe software. http://caffe.berkeleyvision.org/. Accessed: 2020-04-23.

[8] Ondřej Bojar, Rajen Chatterjee, Christian Federmann, Yvette Graham, Barry Haddow, Matthias Huck, Antonio Jimeno Yepes, Philipp Koehn, Varvara Logacheva, Christof Monz, et al. Findings of the 2016 conference on machine translation. In *Proceedings of the First Conference on Machine Translation: Volume 2, Shared Task Papers*, pages 131–198, 2016.

[9] Mariusz Bojarski, Davide Del Testa, Daniel Dworakowski, Bernhard Firner, Beat Flepp, Prasoon Goyal, Lawrence D Jackel, Mathew Monfort, Urs Muller, Jiakai Zhang, et al. End to end learning for self-driving cars. *arXiv*, 2016.

[10] UCF Center for Research in Computer Vision. Ucf101 action recognition benchmark. http://crcv.ucf.edu/data/UCF101.php. Accessed: 2020-04-23.

[11] Sully Chen. Autopilot tensorflow. https://github.com/SullyChen/Autopilot-TensorFlow. Accessed: 2020-04-23.

[12] Tianshi Chen, Zidong Du, Ninghui Sun, Jia Wang, Chengyong Wu, Yunji Chen, and Olivier Temam. Diannao: A small-footprint high-throughput accelerator for ubiquitous machine-learning. In *ASPLOS*, 2014.

[13] Y. Chen, J. Emer, and V. Sze. Eyeriss: A spatial architecture for energy-efficient dataflow for convolutional neural networks. In *ISCA*, 2016.

[14] Yunji Chen, Tao Luo, Shaoli Liu, Shijin Zhang, Liqiang He, Jia Wang, Ling Li, Tianshi Chen, Zhiwei Xu, Ninghui Sun, et al. Dadiannao: A machine-learning supercomputer. In *MICRO*, 2014.

[15] Kyunghyun Cho, Bart van Merrienboer, Çaglar Gülçehre, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. Learning phrase representations using RNN encoder-decoder for statistical machine translation. *CoRR*, 2014.

[16] Z. Du, R. Fasthuber, T. Chen, P. Ienne, L. Li, T. Luo, X. Feng, Y. Chen, and O. Temam. Shidiannao: Shifting vision processing closer to the sensor. In *ISCA*, 2015.

[17] Zidong Du, Krishna Palem, Avinash Lingamneni, Olivier Temam, Yunji Chen, and Chengyong Wu. Leveraging the error resilience of machine-learning applications for designing highly energy efficient accelerators. In *ASP-DAC*, 2014.

[18] Laboratoire d'Informatique de l'Université du Maine (LIUM). Ted-lium benchmark. https://www.openslr.org/7/. Accessed: 2020-04-23.

[19] A. P. Engelbrecht. A new pruning heuristic based on variance analysis of sensitivity information. *IEEE Transactions on Neural Networks*, 2001.

[20] James Garland and David Gregg. Low complexity multiply-accumulate units for convolutional neural networks with weight-sharing. *TACO*, 2018.

[21] Yunchao Gong, Liu Liu, Ming Yang, and Lubomir Bourdev. Compressing deep convolutional networks using vector quantization. *CoRR*, 2014.

[22] Google. Tensorflow framework. https://www.tensorflow.org/. Accessed: 2020-04-23.

[23] Yiwen Guo, Anbang Yao, and Yurong Chen. Dynamic network surgery for efficient dnns. In *NIPS*, 2016.

[24] Udit Gupta, Brandon Reagen, Lillian Pentecost, Marco Donato, Thierry Tambe, Alexander M Rush, Gu-Yeon Wei, and David Brooks. Masr: A modular accelerator for sparse rnns. In *PACT*, 2019.

[25] Song Han, Xingyu Liu, Huizi Mao, Jing Pu, Ardavan Pedram, Mark A Horowitz, and William J Dally. Eie: efficient inference engine on compressed deep neural network. In *ISCA*, 2016.

[26] Song Han, Jeff Pool, John Tran, and William J. Dally. Learning both weights and connections for efficient neural networks. In *NIPS*, 2015.

[27] A. Hashmi, H. Berry, O. Temam, and M. Lipasti. Automatic abstraction and fault tolerance in cortical microachitectures. In *ISCA*, 2011.

[28] Atif Hashmi, Andrew Nere, James Jamal Thomas, and Mikko Lipasti. A case for neuromorphic isas. In *ASPLOS*, 2011.

[29] Johann Hauswald, Michael A Laurenzano, Yunqi Zhang, Cheng Li, Austin Rovinski, Arjun Khurana, Ronald G Dreslinski, Trevor Mudge, Vinicius Petrucci, Lingjia Tang, et al. Sirius: An open end-to-end voice and vision personal assistant and its implications for future warehouse scale computers. In *ASPLOS*, 2015.

[30] Simon S Haykin et al. Neural networks and learning machines, 2009.

[31] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. *CoRR*, 2015.

[32] T. He, Y. Fan, Y. Qian, T. Tan, and K. Yu. Reshaping deep neural network for fast decoding by node-pruning. In *ICASSP*, 2014.

[33] K. Hegde, R. Agrawal, Y. Yao, and C. W. Fletcher. Morph: Flexible acceleration for 3d cnn-based video understanding. In *MICRO*, 2018.

[34] K. Hegde, J. Yu, R. Agrawal, M. Yan, M. Pellauer, and C. Fletcher. Ucnn: Exploiting computational reuse in deep neural networks via weight repetition. In *ISCA*, 2018.

[35] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural Computation*, 9(8):1735–1780, 1997.

[36] Matthew B Hoy. Alexa, siri, cortana, and more: an introduction to voice assistants. *Medical reference services quarterly*, 37(1):81–88, 2018.

[37] Jie Hu, Li Shen, and Gang Sun. Squeeze-and-excitation networks. *CoRR*, 2017.

[38] Gao Huang, Zhuang Liu, and Kilian Q. Weinberger. Densely connected convolutional networks. *CoRR*, 2016.

[39] Imagenet. Imagenet datset. www.image-net.org/. Accessed: 2020-04-12.

[40] Mohsen Imani, Mohammad Samragh, Yeseong Kim, Saransh Gupta, Farinaz Koushanfar, and Tajana Rosing. Deep learning acceleration with neuron-to-memory transformation. In *HPCA*, 2020.

[41] X. Jiao, V. Akhlaghi, Y. Jiang, and R. K. Gupta. Energy-efficient neural networks using approximate computation reuse. In *DATE*, 2018.

[42] Daniel A Jiménez and Calvin Lin. Dynamic branch prediction with perceptrons. In *HPCA*, pages 197–206, 2001.

[43] Norman P. Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, Rick Boyle, Pierre-luc Cantin, Clifford Chao, Chris Clark, Jeremy Coriell, Mike Daley, Matt Dau, Jeffrey Dean, Ben Gelb, Tara Vazir Ghaemmaghami, Rajendra Gottipati, William Gulland, Robert Hagmann, C. Richard Ho, Doug Hogberg, John Hu, Robert Hundt, Dan Hurt, Julian Ibarz, Aaron Jaffey, Alek Jaworski, Alexander Kaplan, Harshit Khaitan, Daniel Killebrew, Andy Koch, Naveen Kumar, Steve Lacy, James Laudon, James Law, Diemthu Le, Chris Leary, Zhuyuan Liu, Kyle Lucke, Alan Lundin, Gordon MacKean, Adriana Maggiore, Maire Mahony, Kieran Miller, Rahul Nagarajan, Ravi Narayanaswami, Ray Ni, Kathy Nix, Thomas Norrie, Mark Omernick, Narayana Penukonda, Andy Phelps, Jonathan Ross, Matt Ross, Amir Salek, Emad Samadiani, Chris Severn, Gregory Sizikov, Matthew Snelham, Jed Souter, Dan Steinberg, Andy Swing, Mercedes Tan, Gregory Thorson, Bo Tian, Horia Toma, Erick Tuttle, Vijay Vasudevan, Richard Walter, Walter Wang, Eric Wilcox, and Doe Hyun Yoon. In-datacenter performance analysis of a tensor processing unit. In *ISCA*, 2017.

[44] P. Judd, J. Albericio, and A. Moshovos. Stripes: Bit-serial deep neural network computing. *IEEE Computer Architecture Letters*, 2017.

[45] Veton Kepuska and Gamal Bohouta. Next-generation of virtual personal assistants (microsoft cortana, apple siri, amazon alexa and google home). In *CCWC*, 2018.

[46] Randal A. Koene and Yoshio Takane. Discriminant component pruning: Regularization and interpretation of multilayered backpropagation networks. *Neural Comput.*, 1999.

[47] Alex Krizhevsky. One weird trick for parallelizing convolutional neural networks. *CoRR*, 2014.

[48] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In *NIPS*, 2012.

[49] Oleksii Kuchaiev, Boris Ginsburg, Igor Gitman, Vitaly Lavrukhin, Carl Case, and Paulius Micikevicius. Openseq2seq: extensible toolkit for distributed and mixed precision training of sequence-to-sequence models. *CoRR*, 2018.

[50] Hsiang-Tsung Kung. Why systolic architectures? *Computer*, 0(1):37–46, 1982.

[51] Sun-Yuan Kung and Jenq-Neng Hwang. A unified systolic architecture for artificial neural networks. *Journal of Parallel and Distributed Computing*, 6(2):358–387, 1989.

[52] Yann LeCun. Mnist datset. http://yann.lecun.com/exdb/mnist/. Accessed: 2020-04-20.

[53] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.

[54] Asriel U. Levin, Todd K. Leen, and John E. Moody. Fast pruning using principal components. In *NIPS*, 1994.

[55] S. Li, K. Chen, J. H. Ahn, J. B. Brockman, and N. P. Jouppi. Cacti-p: Architecture-level modeling for sram-based structures with advanced leakage reduction techniques. In *IEEE/ACM ICCAD*, 2011.

[56] Richard Lippmann. An introduction to computing with neural nets. *ASSP*, 4(2):4–22, 1987.

[57] Zhuang Liu, Mingjie Sun, Tinghui Zhou, and Gao Huang andTrevor Darrell. Rethinking the value of network pruning. *CoRR*, 2018.

[58] Gustavo López, Luis Quesada, and Luis A Guerrero. Alexa vs. siri vs. cortana vs. google assistant: a comparison of speech-based natural user interfaces. In *AHFE*, 2017.

[59] W. Lu, G. Yan, J. Li, S. Gong, Y. Han, and X. Li. Flexflow: A flexible dataflow accelerator architecture for convolutional neural networks. In *HPCA*, 2017.

[60] Minh-Thang Luong, Hieu Pham, and Christopher D Manning. Effective approaches to attention-based neural machine translation. *CoRR*, 2015.

[61] Mostafa Mahmoud, Kevin Siu, and Andreas Moshovos. Diffy: A déjà vu-free differential deep neural network accelerator. In *MICRO*, 2018.

[62] L. Mauch and B. Yang. Selecting optimal layer reduction factors for model reduction of deep neural networks. In *ICASSP*, 2017.

[63] Yajie Miao, Mohammad Gowayyed, and Florian Metze. Eesen: End-to-end speech recognition using deep rnn models and wfst-based decoding. In *ASRU*, 2015.

[64] Micron. System power calculators. https://www.micron.com/support/tools-and-utilities/power-calc. Accessed: 2020-04-23.

[65] Riccardo Miotto, Fei Wang, Shuang Wang, Xiaoqian Jiang, and Joel T Dudley. Deep learning for healthcare: review, opportunities and challenges. *Briefings in bioinformatics*, 19(6):1236–1246, 2018.

[66] Daniel Neil, Jun Haeng Lee, Tobi Delbruck, and Shih-Chii Liu. Delta networks for optimized recurrent network computation. In *ICML*, 2017.

[67] Lin Ning and Xipeng Shen. Deep reuse: Streamline cnn inference on the fly via coarse-grained computation reuse. In *ICS*, 2019.

[68] NVIDIA. NVIDIA System Management Interface. https://developer.nvidia.com/nvidia-system-management-interface. Accessed: 2020-04-23.

[69] D. W. Opitz. Analyzing the structure of a neural network using principal component analysis. In *ICNN*, 1997.

[70] Vassil Panayotov, Guoguo Chen, Daniel Povey, and Sanjeev Khudanpur. Librispeech: an asr corpus based on public domain audio books. In *ICASSP*, 2015.

[71] Vassil Panayotov and Daniel Povey. Librispeech asr corpus benchmark. http://www.openslr.org/12/. Accessed: 2020-04-23.

[72] A. Parashar, M. Rhu, A. Mukkara, A. Puglielli, R. Venkatesan, B. Khailany, J. Emer, S. W. Keckler, and W. J. Dally. Scnn: An accelerator for compressed-sparse convolutional neural networks. In *ISCA*, 2017.

[73] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. Pytorch: An imperative style, high-performance deep learning library. In *NIPS*, pages 8024–8035, 2019.

[74] Daniel Povey. Kaldi software. http://kaldi-asr.org/. Accessed: 2020-04-23.

[75] Daniel Povey, Arnab Ghoshal, Gilles Boulianne, Lukas Burget, Ondrej Glembek, Nagendra Goel, Mirko Hannemann, Petr Motlicek, Yanmin Qian, Petr Schwarz, et al. The kaldi speech recognition toolkit. In *ASRU*, 2011.

[76] Brandon Reagen, Paul Whatmough, Robert Adolf, Saketh Rama, Hyunkwang Lee, Sae Kyu Lee, José Miguel Hernández-Lobato, Gu-Yeon Wei, and David Brooks. Minerva: Enabling low-power, highly-accurate deep neural network accelerators. In *ISCA*, 2016.

[77] Marc Riera, Jose-Maria Arnau, and Antonio González. Computation reuse in dnns by exploiting input similarity. In *ISCA*, 2018.

[78] Marc Riera, Jose-Maria Arnau, and Antonio González. Cgpa: Coarse-grained pruning of activations for energy-efficient rnn inference. *IEEE micro*, 39(5):36–45, 2019.

[79] Raúl Rojas. *Neural networks: a systematic introduction*. Springer Science & Business Media, 2013.

[80] Anthony Rousseau, Paul Deléglise, and Yannick Esteve. Ted-lium: an automatic speech recognition dedicated corpus. In *LREC*, 2012.

[81] Ananda Samajdar, Yuhao Zhu, Paul Whatmough, Matthew Mattina, and Tushar Krishna. Scale-sim: Systolic cnn accelerator simulator. *arXiv*, 2018.

[82] Jürgen Schmidhuber. Deep learning in neural networks: An overview. *Neural networks*, 61:85–117, 2015.

[83] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv*, 2014.

[84] Khurram Soomro, Amir Roshan Zamir, and Mubarak Shah. Ucf101: A dataset of 101 human actions classes from videos in the wild. *arXiv*, 2012.

[85] Suraj Srinivas and R. Venkatesh Babu. Data-free parameter pruning for deep neural networks. *CoRR*, 2015.

[86] Synopsys. Synopsys eda tools. https://www.synopsys.com/. Accessed: 2020-04-23.

[87] Hamid Tabani, Jose-Maria Arnau, Jordi Tubella, and Antonio Gonzalez. An ultra low-power hardware accelerator for acoustic scoring in speech recognition. In *PACT*, 2017.

[88] Siri Team. Hey siri: An on-device dnn-powered voice trigger for apple's personal assistant. https://machinelearning.apple.com/2017/10/01/hey-siri.html. Accessed: 2020-03-25.

[89] O. Temam. A defect-tolerant accelerator for emerging high-performance applications. In *ISCA*, 2012.

[90] Philippe Thomas and Marie-Christine Suhner. A new multilayer perceptron pruning algorithm for classification and regression applications. *Neural Process. Lett.*, 2015.

[91] M. E. Tipping and Christopher Bishop. Probabilistic principal component analysis. *Journal of the Royal Statistical Society, Series B*, 1999.

[92] Du Tran, Lubomir Bourdev, Rob Fergus, Lorenzo Torresani, and Manohar Paluri. Learning spatiotemporal features with 3d convolutional networks. In *ICCV*, 2015.

[93] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *NIPS*, 2017.

[94] V. M. Weaver, M. Johnson, K. Kasichayanula, J. Ralph, P. Luszczek, D. Terpstra, and S. Moore. Measuring energy and power with papi. In *ICPP*, 2012.

[95] Wei Wen, Chunpeng Wu, Yandan Wang, Yiran Chen, and Hai Li. Learning structured sparsity in deep neural networks. In *NIPS*, 2016.

[96] Bernard Widrow, Istvan Kollar, and Ming-Chang Liu. Statistical theory of quantization. *IEEE Transactions on instrumentation and measurement*, 1996.

[97] Yonghui Wu, Mike Schuster, Zhifeng Chen, Quoc V Le, Mohammad Norouzi, Wolfgang Macherey, Maxim Krikun, Yuan Cao, Qin Gao, Klaus Macherey, et al. Google's neural machine translation system: Bridging the gap between human and machine translation. *arXiv*, 2016.

[98] Wayne Xiong, Jasha Droppo, Xuedong Huang, Frank Seide, Mike Seltzer, Andreas Stolcke, Dong Yu, and Geoffrey Zweig. Achieving human parity in conversational speech recognition. *arXiv*, 2016.

[99] Tien-Ju Yang, Yu-Hsin Chen, and Vivienne Sze. Designing energy-efficient convolutional neural networks using energy-aware pruning. *CoRR*, 2016.

[100] A. Yasoubi, R. Hojabr, and M. Modarressi. Power-efficient accelerator design for neural networks using computation reuse. *IEEE Computer Architecture Letters*, 2017.

[101] Reza Yazdani, Jose-Maria Arnau, and Antonio González. Unfold: A memory-efficient speech recognizer using on-the-fly wfst composition. In *MICRO*, 2017.

[102] Jiecao Yu, Andrew Lukefahr, David Palframan, Ganesh Dasika, Reetuparna Das, and Scott Mahlke. Scalpel: Customizing dnn pruning to the underlying hardware parallelism. In *ISCA*, 2017.

[103] Wojciech Zaremba, Ilya Sutskever, and Oriol Vinyals. Recurrent neural network regularization. *arXiv*, 2014.

[104] Dongqing Zhang, Jiaolong Yang, Dongqiangzi Ye, and Gang Hua. Lq-nets: Learned quantization for highly accurate and compact deep neural networks. In *ECCV*, pages 365–382, 2018.

[105] Jiajun Zhang and Chengqing Zong. Deep neural networks in machine translation: An overview. *IEEE Intelligent Systems*, 2015.

[106] S. Zhang, Z. Du, L. Zhang, H. Lan, S. Liu, L. Li, Q. Guo, T. Chen, and Y. Chen. Cambricon-x: An accelerator for sparse neural networks. In *MICRO*, 2016.

[107] Xiaohui Zhang, Jan Trmal, Daniel Povey, and Sanjeev Khudanpur. Improving deep neural network acoustic models using generalized maxout networks. In *ICASSP*, 2014.

[108] Shuchang Zhou, Yuxin Wu, Zekun Ni, Xinyu Zhou, He Wen, and Yuheng Zou. Dorefa-net: Training low bitwidth convolutional neural networks with low bitwidth gradients. *CoRR*, 2016.

[109] X. Zhou, Z. Du, Q. Guo, S. Liu, C. Liu, C. Wang, X. Zhou, L. Li, T. Chen, and Y. Chen. Cambricon-s: Addressing irregularity in sparse neural networks through a cooperative software/hardware approach. In *MICRO*, 2018.

[110] Chenzhuo Zhu, Song Han, Huizi Mao, and William J Dally. Trained ternary quantization. *arXiv*, 2016.