Reconfigurable hardware acceleration of CNNs on FPGA-based smart cameras

Kamel Abdelouahab

UNIVERSITÉ CLERMONT AUVERGNE
ÉCOLE DOCTORALE: SCIENCES POUR L'INGÉNIEUR

THÈSE

présentée par

Kamel ABDELOUAHAB

Pour obtenir le grade de:

DOCTEUR DE L'UNIVERSITÉ CLERMONT AUVERGNE

Spécialité: Électronique et Architecture de Systèmes

Titre de la thèse:

Reconfigurable hardware acceleration of CNNs on FPGA-based smart cameras

Thèse soutenue le 11 décembre 2018 devant le jury composé de

Président	M. Jocelyn Sérot
Directeur de thèse	M. François Berry
Co-directeur de thèse	M. Maxime Pelcat
Rapporteurs	M. Daniel Ménard Mme. Francesca Palumbo
Examinateurs	M. Cédric Bourrasset M. Luca Maggiani

DOCTORAL THESIS

Reconfigurable hardware acceleration of CNNs on FPGA-based smart cameras

Author:
Kamel ABDELOUAHAB

Supervisors:
Prof. François BERRY
Dr. Maxime PELCAT

A thesis submitted in fulfillment of the requirements
for the degree of Docteur de l'Université Clermont Auvergne

at the

DREAM Research Group
Institut Pascal

UNIVERSITÉ CLERMONT AUVERGNE
Ecole Doctorale des Sciences Pour l'Ingénieur
Institut Pascal

Abstract

Reconfigurable hardware acceleration of CNNs on FPGA-based smart cameras

by Kamel ABDELOUAHAB

Deep Convolutional Neural Networks (CNNs) have become a *de-facto* standard in computer vision. This success came at the price of a high computational cost, making the implementation of CNNs, under real-time constraints, a challenging task.

To address this challenge, the literature exploits the large amount of parallelism exhibited by these algorithms, motivating the use of dedicated hardware platforms. In power-constrained environments, such as smart camera nodes, FPGA-based processing cores are known to be adequate solutions in accelerating computer vision applications. This is especially true for CNN workloads, which have a streaming nature that suits well to reconfigurable hardware architectures.

In this context, the following thesis addresses the problems of CNN mapping on FPGAs. In Particular, it aims at improving the efficiency of CNN implementations through two main optimization strategies; The first one focuses on the CNN model and parameters while the second one considers the hardware architecture and the fine-grain building blocks.

Keywords: Deep Learning, CNN, FPGA, Dataflow, Direct Hardware Mapping, Smart Camera

UNIVERSITÉ CLERMONT AUVERGNE
Ecole Doctorale des Sciences Pour l'Ingénieur
Institut Pascal

Résumé

Architectures reconfigurables pour l'accélération des CNNs. Applications sur cameras intelligentes à base de FPGAs

par Kamel ABDELOUAHAB

Les Réseaux de Neurones Convolutifs profonds (CNNs) ont connu un large succès au cours de la dernière décennie, devenant un standard de la vision par ordinateur. Ce succès s'est fait au détriment d'un large coût de calcul, où le déploiement des CNNs reste une tâche ardue surtout sous des contraintes de temps réel.

Afin de rendre ce déploiement possible, la littérature exploite le parallélisme important de ces algorithmes, ce qui nécessite l'utilisation de plate-formes matérielles dédiées. Dans les environnements soumis à des contraintes de consommations énergétiques, tels que les nœuds des caméras intelligentes, les cœurs de traitement à base de FPGAs sont reconnus comme des solutions de choix pour accélérer les applications de vision par ordinateur. Ceci est d'autant plus vrai pour les CNNs, où les traitements se font naturellement sur un flot de données, rendant les architectures matérielles à base de FPGA d'autant plus pertinentes.

Dans ce contexte, cette thèse aborde les problématiques liées à l'implémentation des CNNs sur FPGAs. En particulier, ces travaux visent à améliorer l'efficacité des implantations grâce à deux principales stratégies d'optimisation; la première explore le modèle et les paramètres des CNNs, tandis que la seconde se concentre sur les architectures matérielles adaptées au FPGA.

Mot clés : Apprentissage profond, Réseaux de neurones convolutifs, FPGA, Flot de données, Implémentation matérielle, Caméras intelligentes

To my family:
wife, child, and parents

Acknowledgements

This work could not have been carried-out without the guidance of my supervisors: Prof. François Berry and Dr. Maxime Pelcat. Thank you François for your trust and advice[1]. Thank you Maxime for your continuous support and expertise[2].

Moreover, I express my gratitude towards Prof. Jocelyn Sérot[3] and Dr. Cedric Bourrasset[4] for their availability and their prior works related to this thesis. I also thank Dr. Luca Maggiani for his advice, and for our upcoming collaboration ...

Besides the advisors, I would like to thank Prof. Daniel Ménard and Dr. Francesca Palumbo for being members of my thesis committee. Thanks for reading, commenting, and reviewing this manuscript.

I am also grateful to Dr. Christos Bouganis for welcoming me at the Imperial College London last summer. Thanks to Labex IMobS3 for sponsoring this visit, and more generally, thanks to the French ministry of higher education for funding my doctoral studies.

This work benefited from many free and/or open-source tools. To the maintainers of Python, Numpy, Matplotlib, OpenCV, Caffe, GHDL, Caph, Latex ... Thank you.

For the last three year, it has been a pleasure[5] to work within the DREAM research group. In order of appearance, thanks to Sebastien, El Mehdi, Lobna, John[6], Dimia and Abderrahim. It was great to struggle and learn together.

Finally, this work would never be completed without the support of my family. Words cannot describe my gratitude for my parents, my brother, my sisters and of course, my wife. To all of you, thank you for your continuous encouragements and devotion.

[1] Particularly those during lunch-time at «Leclerc». These were the best !
[2] Even over Skype!
[3] Thanks for that VHDL top level generator you've written a Saturday morning !
[4] Thanks for that 365 messy code lines that became the Haddoc tool !
[5] Except «that» period. By the way, I don't thank the person in charge of ZRR at all.
[6] Au secours!

Contents

List of Figures

List of Tables

Glossary

AI Artificial Intelligence. 5, 8

ALM Adaptative Logic Module. 28–31, 44, 77, 87, 90, 93, 97, 99, 106, 112, 114

ASIC Application Specific Integrated Circuits. 23, 27, 32

BNN Binary Neural Network. 12, 62

CLB Configurable Logic Block. 30

CNN Convolutional Neural Network. iii, 1, 4, 5, 7–14, 17–25, 27, 34, 37, 46–51, 53, 55–57, 59, 63, 64, 66, 68–75, 77, 81, 83, 87, 93, 97, 100, 101, 110, 111, 113–115, 119

CPU Central Processing Unit. 21, 22, 27, 31, 34, 48, 49, 59, 67

CTC Computation to Communitcation. 71, 102, 103

DHM Direct Hardware Mapping. 70, 71, 87, 90, 95, 96, 99, 102, 107, 117, 119, 122

DL Deep Learning. 5, 18

dma Direct Memory Access. 70

DP Dot-Product. 90

DPN Data-flow Process Network. 35, 70, 73, 74, 78, 89

DRAM Dynamic Random Access Memory. 21, 53, 55, 57

DSL Domain Specific Language. 35

DSP Digital Signal Processing. 23, 31, 38, 45, 48, 51, 53, 59, 62, 76–81, 93, 106, 111, 119

EWMM Element-Wise Matrix Multiplication. 50, 51

FA Full Adder. 29

FC Fully Connected. 12, 13, 18, 20, 48, 49, 71, 72, 76

FFT Fast Fourier Transform. 48, 51

FIFO First-In First-Out. 44, 71, 77, 88, 89, 121

FIR Finite Impulse Response. 94

FM Feature Map. 9–11, 13, 20, 48–51, 53, 55, 57, 59, 61, 62, 75, 77–80, 84, 92

FPGA Field-Programmable Gate Array. iii, 1–4, 7, 23, 27, 30, 32–34, 36, 45, 47–51, 53, 55, 56, 59, 61, 62, 64, 69, 70, 72, 73, 93, 101, 110, 111, 114, 119

FPS Frames Processed per seconds. 47

FSM Finite State Machine. 116

FxP Fixed Point. 59, 61

GEMM General Matrix Multiplication. 49, 64

GPU Graphics Processing Unit. 7, 21, 22, 24, 27, 31, 34, 48–50, 59, 64, 122

HDL Hardware Description Language. 32–34, 43, 44

HLS High-Level Synthesis. 33–36, 45, 46, 72, 86, 87

HPC High Performance Computing. 47

HPS Hard Processor System. 31

I/O Inputs Outputs. 27, 38, 76, 91

ILSVRC ImageNet Large Scale Visual Recognition Competition. 17

IoU Intersection over Union. 16

IP Intellectual Property. 40, 72, 103

LAB Logic Array Block. 30

LC Logic Cell. 27, 28

LE Logic Element. 28, 29

LFSR Linear Feedback Shift Register. 115, 117, 118

LOA Lower-part-Or Adder. 113, 114

LUT Look-Up Table. 28, 29, 32, 48, 50, 114

MAC Multiply Accumulate. 18, 20, 21, 37, 38, 43–45, 47, 62, 64

mAP Mean Average Precision. 16

MCM Multiple Constant Multiplication. 110

ML Machine Learning. 5, 6

MLP Multi-Layer Perceptron. 6–8, 12

MNIST Modified National Institute of Standards and Technology. 76, 78

MOA Multiple Operand Adder. 96, 99, 106–108, 111–114, 143

MoC Model of Computation. 34, 35, 72, 86

MRED Mean Relative Error Distance. 113, 114

MSE Mean Squared Error. 107

MVCNN Multi-View CNN. 82–85, 122

OCR Optical Character Recognition. 12, 75, 78, 80, 103

PE Processing Element. 21, 51, 53, 54, 66

PSNR Peak signal-to-noise ratio. 145

QNN Quantized Neural Networks. 62, 94

QoS Quality of service. 2

RAM Random Access Memory. 55

ReLU Rectified Linear Unit. 11, 13, 19, 144

RTL Register Transfer Level. 32–34, 38, 67, 87

SC Stochastic Computing. 115–119

SCM Single Constant Multiplication. 87, 94–96, 100, 106–108, 110, 143

SDF Static Data-Flow. 69

SGD Stochastic Gradient Descent. 7

SIMD Single Instruction on Multiple Data. 21

SM Streaming Multiprocessor. 21

SNG Stochastic Number Generator. 115–119

SRAM Synchronous Random Access Memory. 30, 31, 40, 48, 90, 92, 106

SSIM Structural SIMilarity. 145

TPU Tensor Processing Unit. 23

TTQ Trained Ternary Quantization. 62, 63, 94

USPS United States Postal Service. 76, 79

VHDL VHSIC Hardware Description Language. 32, 35, 37, 38, 100

WB Window Buffer. 90

Chapter 1

Introduction

1.1 The Context of Deep Learning and Smart Cameras

Video is arguably the largest data being stored and exchanged, as it accounts for over 70% of today's Internet traffic [Cis17]. On a daily basis, over 800 million hours of video are collected worldwide, and that is only for video surveillance [Woo14]. The acquisition and processing of all this data opens up critical research challenges in both areas of multi-camera networks –during the acquisition phase–, and computer vision –during the processing phase–. In the conventional approach, multi-camera vision systems operate in a centralized fashion in which all data collected by a cameras is sent to a unique processing unit. Then, this central node extracts high level information from the gathered data, inferring for instance the presence of an object in a given scene. Such an approach quickly reaches its limits when the number of cameras increases. In this case, both the computing capabilities of the processing unit and the network bandwidth become limiting factors. This is even more true when the multi-camera vision system is deployed under real-time constraints, which involve usually high resolution video streams, and elevated frame rates. Under these constraints, network technologies fail at providing sufficient bandwidth and become very costly in hardware resources and energy.

In order to address this bandwidth problem, distributed vision systems have been proposed. In this kind of systems, the objective is to decentralize the calculations directly to each camera, or node, of the system. The role of these cameras is not only to acquire the video of a given scene, but also to partially or fully process this video, extracting some *information* from the scene. Such « intelligent » cameras are commonly referred to as *smart cameras*, and embed their own processing capabilities. Thanks to decentralization, smart camera nodes are able to solve the bandwidth bottleneck problem, but introduce new challenges, mainly related to computer vision, and embedded image processing under low-power constraints.

In the last few years, computer vision has been revolutionized by deep learning. Many tasks of computer vision, consisting in extracting structured information from raw data, are now efficiently carried out with *deep learning* based techniques, and particularly Convolutional Neural Networks (CNNs). CNNs have been successfully employed in a large number of vision problems, ranging from image classification and object detection [RF18], to semantic scene labelling and context understanding [KTB15, TKTH18]. In a large number of applications, deep learning outperforms conventional computer vision algorithms, and even human performance in the image classification task [HZRS16]. In the context of camera networks, deep learning and CNNs open new perspectives of system autonomy and reliability.

When considering the embedded processing aspects of smart camera design, the architecture of the hardware, and the nature of the processing core represent critical factors to meet real-time and low power constraints. In this context, Field-Programmable Gate

Arrays (FPGAs) have drawn a lot of attention in the past years because they offer large opportunities for exploiting the fine grain, regular parallelism that most of image processing applications exhibit. This is even more true for deep learning applications, which naturally have a *streaming* nature, and can thus benefit from significant acceleration when running on reconfigurable hardware such as FPGAs.

However, deep learning techniques are very computationally intensive, involving up to billions of operations to properly operate. These high computational workloads currently prevent the real-time implantation of state-of-the-art deep learning in power-constrained environments, and especially smart camera nodes. In this context, the following manuscript addresses the problem of deep learning implementation on embedded reconfigurable hardware to push the boundaries of deep learning embeddability.

1.2 Deep Learning Constraints and Implementation Challenges

The challenges of embedded vision with deep learning are those of conventional computer vision in general, to which are added constraints related to the physical implementation in an embedded platform. Mainly, these constraints can be formalized as:

1. Real-time constraints: Depending on the application, video streams have to be processed at high frame-rates, typically over 30 frames per second when human perception is involved.

2. Quality of service (QoS): Deep learning based methods are known to deliver high reliability. In this manuscript, reliability is quantified using accuracy metrics detailed in the next chapter. However, one may note that deep learning algorithms are usually tolerable to approximate computing with a controlled rate of error, allowing a designer to trade a minimal amount of QoS for efficiency improvements.

3. Embedded environment: These constraints are mainly related to power consumption of the system (typically, under $20W$ in embedded systems), and to the nature of the implementation platfrom (typically under $< 300USD$)

Of course, these constraints are antagonist and a given accelerator has to trade off between computational performance, reliability, and energy efficiency according to the application it implements.

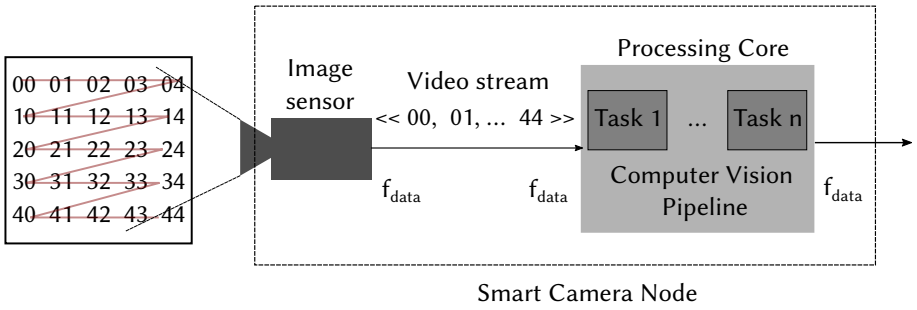


FIGURE 1.1: Diagram of Stream-based dataflow image processing. Data is acquired and processed at the same rate

1.3 Smart Cameras as Dataflow Computer Vision Systems

To meet the real-time requirements of embedded vision, numerous studies advocate the use of the dataflow paradigm in the smart camera nodes [SBB16, Bou16, Mag17].

Diagram 1.1 illustrates the dataflow paradigm. The video of a given scene is acquired by a camera sensor in a raster scan fashion, and is then sent to a processing unit as a continuous *stream* of data. When using dataflow, the objective is to process this stream *at the very rate* it is sent by the sensor, resulting in a real-time execution (i.e. sustainable over long periods without accumulating data or starving processing units). In this case, the number of frames processed per seconds is only a function of the frequency at which the system operates, and of the resolution of the considered video stream. For instance, considering the arrival of one pixel per clock cycle, processing a $720p$ monochrome video stream at 30 FPS imposes a minimum processing frequency of $30 * 1280 * 720 = 27.5MHz$.

$$\text{Frames per seconds} = \frac{\text{Frequency}}{\text{Resolution}}$$

To implement this stream processing, the computation core of a smart camera has to be able to process large processing pipelines. Reconfigurable hardware platforms such as FPGAs shine at this task, and can naturally support streaming workloads thanks to a low granularity of parallelism, and a high hardware flexibility.

As a result, a large number of smart camera architectures rely on FPGA-based processing cores to implement dataflow image processing [YEBM02, SAWM08, LMH04, DBSM07, HPFA07, BB14].

1.4 Contributions

This document mainly addresses the challenge of deep learning implementation on FPGAs. While a variety of studies already address the same problem [APBS18, VKB18], this work is carried out in a context of smart camera networks. As evoked above, the real-time and low-power requirements are critical in this context, and this calls for dedicated methods to make real-time deep learning feasible on low-power devices. In this perspective, the main contributions of this manuscript can be be listed as:

- Exploring dataflow hardware architectures capable of implementing CNN inference on FPGAs.
- Proposing « Tactics » and optimization strategies to reduce the footprint of FPGA-Based CNN accelerators.
- Introducing a design space exploration methodology that searches for the best trade-off between the FPGA resource utilization and the deep learning accuracy.
- Developing tool-flow that automates the mapping of CNN accelerators on FPGA devices.
- Reporting a list of negative methods, which first look promising to optimize CNN mappings, but give negative results when deployed on FPGAs.

1.5 Manuscript Outline

The manuscript is structured in two main parts. The first part includes Chapters 2, 3, 4 and gives a background on CNNs, FPGAs and state-of-the-art methods to accelerate the former on the latter. The second part includes Chapters 5, 6, 7 and presents thesis contributions.

- Chapter 2 motivates the use of deep learning for embedded vision. More particularly, concepts and notions related to CNNs are introduced. The chapter also discusses the computational workload of CNNs, and the conventional hardware accelerators supporting these workloads.
- Chapter 3 briefly recalls the FPGA features and design flow. This chapter particularly highlights the relevance of FPGAs in real-time image processing through end-to-end implementation examples.
- Chapter 4 details how deep learning acceleration can benefit from the evolution of FPGAs technology. A survey of methods and optimization employed in FPGA-Based deep learning acceleration is provided, leading to a classification of the existing approaches into three main categories: algorithmic-based, data-path-based and approximate- computing-based.
- Chapter 5 introduces the first contribution of this work: Model-based optimization of CNN mappings on FPGAs. This chapter can be viewed as a direct follow up to the work of Bourrasset *et al.* [Bou16] in which the use of a dataflow model of computation is advocated. In particular, it proposes a methodology the chapter details a design space exploration methodology that minimizes the resource allocated to dataflow CNN mappings. Finally, the chapter introduces *multi-view* CNNs, and shows how they improve the efficiency of a vision system.
- Chapter 6 discusses the architectural optimization of CNN implementation. In particular, it focuses on improving the previously derived mappings by customizing the hardware architecture of the atomic components; mainly by specializing the multipliers and pipeline the adders. As a proof of concept, the discussed methods are leveraged on to map a CNN application on an FPGA-powered smart camera.
- Finally, chapter 7 lists the methods that fail to optimize the mapping of CNNs on FPGAs. These methods make sense on paper, but, when implemented, give unexpected bad results on current FPGA architectures, calling for new hardware architectures.

Chapter 2

Embedded Deep Learning

With the emergence of low-cost and energy-efficient processing platforms, it is now possible to implement mainstream computer vision tasks on low-power embedded systems. This discipline, known as *embedded vision*, makes numerous applications possible. Among these applications, *smart camera networks* recently gained a lot of research interest.

In a smart camera network, each node is able to pre-process the raw video stream it captures and transform information into mid-level semantic descriptors that are then exploited to extract meaningful information from a given scene. In classical computer vision approaches, these descriptors are hand-crafted and require a domain-specific knowledge to be engineered. However, with the exponential growth of operating image sensors and video sources, a new challenge of embedded vision is to extract high-level semantic information, in a fully *autonomous* fashion, without the need of any domain-specific, human-crafted visual descriptor. In order to address this challenge, Deep Learning (DL) based methods can be used and are currently the *de-facto* standards to solve computer vision tasks where a sufficient amount of data is available to train the system.

In the following material, elements of deep learning for embedded visions are discussed. A special interest is given to CNNs, which is motivated in the first sections of this chapter. Then, the classical CNN layout is described and currently popular models of computer vision are studied. Special interest if given to the computational workload of CNN inference and to the available hardware architectures supporting this workload. This chapter finally discusses lightweight deep learning models, and low-power hardware platforms supporting embedded deep learning for computer vision.

2.1 From Machine learning to Deep Learning

A majority of concepts involved in deep learning are inherited from the Machine Learning (ML) theory and more specifically from feed-forward neural networks. This relationship of deep learning to the whole of Artificial Intelligence (AI) discipline is illustrated in figure 2.1.

Machine Learning is a branch of AI that explores the conception of algorithms that can automatically extract structured information from raw data without being explicitly programmed. In other words, a machine learning algorithm is able to learn how to do some "intelligent" activities outside the notion of programming, which is in contrast with purpose-built algorithms whose behaviours are defined by hand-crafted heuristics that are explicitly defined.

The advantage of ML is clear: instead of creating a distinct, custom set of programs to solve individual tasks of a given a domain, a *single* machine learning algorithm can *learn*, via a process known as training, how to solve multiple tasks (e.g image classification, detection, segmentation ...) of a given domain (e.g. computer vision).

The next subsections discuss the main features constituting deep learning systems, as an introduction to the notion of deep CNN discussed in Section 2.2.

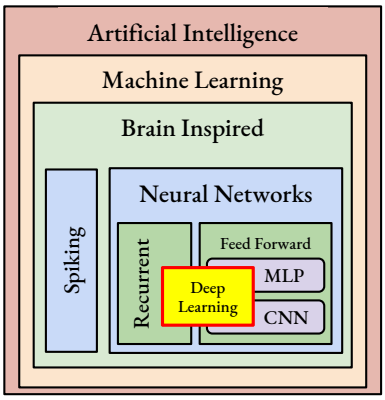


FIGURE 2.1: Deep learning as a sub-field of Artificial Intelligence

2.1.1 Neural Networks and MLPs

The majority of the proposed ML algorithms, deep or not, take their inspiration from the brain itself. As an organ, the brain involves millions of elementary computational elements called *neurons* that are densely interconnected using *synapses*.

Artificial neural networks constitute a class of brain-inspired ML that makes the assumption that the neuron computational model is a weighted sum of input information, followed by a non linear function that produces an *activation*, i.e. that «decides» whether the input signals are sufficient to generate an output signal for the neuron. A common form of Neural Networks are *feed-forward Neural Networks* wherein the neurons are hierarchically arranged onto layers and connections between these neurons do not form a cycle. From the «input» to the «output» of a feed-forward neural network, data is progressively transformed from one representation to another.

One of the most popular classes of feed-forward neural networks are Multi-Layer Perceptrons (MLPs) where each layers' neuron $(\ell - 1)$ is *fully connected* to the neurons of layer ℓ. This means that if M_ℓ is the number of inputs of a given layer ℓ and $N_{\ell-1}$ is the number of outputs (activations) of previous layer $(\ell - 1)$, then $M_\ell = N_{\ell-1}$. Feed-forward neural networks serve as as a substrate for two operations: inference and training. These operations are explained in the next section.

2.1.2 Training and Inference of Neural Networks

As a typical ML setup, neural networks are deployed in two phases. First, the *training* phase works on a known (and often large) training set of data samples to create a model with a modeling power which semantics should be sufficient to interpolate and extrapolate to natural data outside the training set. The output of this phase is the value of the *weights* of the neural network. These weights are then used during the second phase of the deployment named *inference*. The inference works on new data samples (i.e images that were not previously seen during training), and implements the *feed-forward* propagation of the inputs across the network to performed the learned task.

There are multiple ways to train a neural network and derive its weights. The most common approach in computer vision tasks is *supervised learning*, where each training sample is labelled (i.e. where we know what the neural network should ideally produce for this data). This is typically the case in image classification applications, wherein the networks are trained on annotated images where the *class* of the object present in each image of the training set is specified by a human operator.

Unsupervised learning is another approach where none of the training samples is labelled. In this case, the objective is to find *clusters* of similar features on a given dataset.

In the example of image classification, this corresponds to grouping multiple images of a given class, without explicitly knowing which label to put on each group.

Finally, Semi-supervised learning falls in between the two approaches and typically works on a small subset of labelled training data and a large subset of unlabelled data. A semi-supervised machine learning algorithm can for instance use unlabelled data to define the clusters on a dataset and use a small amount of labelled data to label these clusters.

In general, the training of neural networks implements a back-propagation algorithm [LBBH98] which iteratively updates the network parameters to improve the predictive power of the model. Particularly, neural networks can also be *fine-tuned* by using weights of a previously-trained network to initialize the parameters of a new training. These weights are then adjusted to a new constrain, such as a new dataset or reduced precision.

From a computational point of view, the learning phase requires several orders of magnitude more computation than the inference. In fact, back-propagation relies on iterative Stochastic Gradient Descent (SGD) algorithms [Bot10] such ADAM or RMSProp [KB14, HSS12] that require millions of iterations to derive accurate models. In addition, back-propagation calls for high precision arithmetic (float32/float64) to support the computation of small gradients. As a result, the training of a CNN is an energy-consuming process that is usually performed once or a few times per problem, in an off-line fashion, on large clusters of GPUs. Indeed, GPUs are currently the preferred hardware architecture for training neural networks. Moreover, training a state-of-the-art model for computer vision requires several days to complete [KSGH12]. For these reasons, training CNNs with the current learning methods[1] is not feasible on embedded devices. Consequently the acceleration of the training phase is kept beyond the scope of this manuscript. For more details about FPGA-based acceleration of training, the reader is referred to [EH94, OJU+16, KMNM17]. This document is thus concentrated on the inference of neural networks, based on a feed forward propagation.

2.1.3 Forward Propagation in a Neural Network

The inference –also known as forward propagation– of a feed-forward neural network refers to the phase of deploying a pre-learned network to make a prediction (i.e. extract structured information) from a new data sample. In this case, each neuron n of a given layer ℓ applies a learned weight $\Theta[m,n]$ to each of the activations of layers $\ell-1$. This is followed by the addition of a learned bias b_n and the application of a non-linear function, as depicted in equation 2.1:

$$\begin{aligned} &\forall \{l,n\} \in [1,B] \times [1,L] \\ &\qquad a_\ell[n] = \text{act}\left(b_\ell[n] + \sum_{m=1}^{M_\ell} \Theta[m,n] a_{\ell-1}[m]\right) \end{aligned} \tag{2.1}$$

Note that the operations involved in feed-forward propagation can be expressed as the following vector-matrix multiplication:

$$\begin{aligned} &\forall \ell \in [1:L] \\ &\qquad a_\ell = \text{act}\left(b_\ell + \Theta a_{\ell-1}\right) \end{aligned} \tag{2.2}$$

The efficient porting of feed-forward propagation in embedded vision systems is the subject of this thesis. However, rather than targeting MLPs formulations such as the one in Equation 2.1, focus is put on Convolutional Neural Networks (CNNs) for their capacity to overcome MLPs limitations.

[1] Other training methods, referred as *One-Shot Learning*, are introduced in [FFFP06]

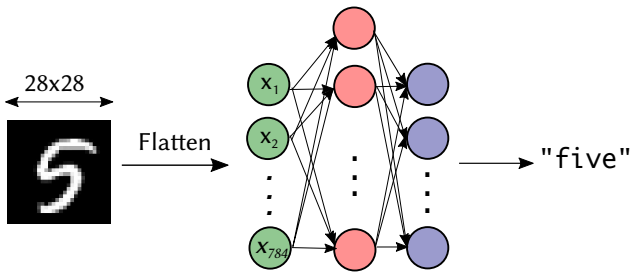


FIGURE 2.2: Feed Forward Propagation

2.1.4 Limitations of MLPs and Motivations of CNNs

MLPs were a popular machine learning solution in the 80. However, these neural networks are often used as classifiers, working on pre-computed representative features, as they are limited in their ability to process data in its natural raw form.

Indeed, MLPs require domain specific features to be hand-crafted before being processed by the AI algorithm. To overcome these limitations, a neural network has to embed more layers of neurons in order to discover its own intermediate representation and automate the feature extraction process. Such networks use a large cascade of layers, each corresponding to a higher level semantics representation [LBH15]. By convention, neural networks with a number of layers exceeding three are referred as *deep* neural network.

However, training deep and fully connected networks such MLPs is an inadequate solution because the derived models are excessively complex and over-parametrized. As shown in equation 2.1, $(M_\ell \times N_\ell)$ weights have to be learned at each layer ℓ. In the context of computer vision, and particularly in the first layer, M_ℓ generally has a large value that corresponds to the *total number of pixels* of a given image. For instance in figure 2.2, each of the N_1 neurons of the first layer has to learn 28×28 weights. As a consequence, a tremendous amount of computations and storage would be needed to train a deep and fully-connected neural net.

Moreover, since the network is receptive to each pixel of the input, it can easily be subject to *over-fitting*. An over-fitted neural network overreacts to all the minor fluctuations in the training data and, consequently, has poor predictive performance for unseen inputs.

Two strategies are advocated to address over-parametrization and over-fitting. The first one is to force a proportion of the weights to have a null value, which is equivalent to partially removing some connections between a layer ℓ and a layer $\ell + 1$ resulting in a *sparsely connected layer*.

The second, known as *weight sharing*, is to reduce the number of weights contributing to an output by replicating them across the inputs using a sliding window over the image space. In this case, each neuron is only receptive to a local neighbourhood instead of being receptive to all of the inputs separately. Combining these two methods resulted to the creation of Convolutional Neural Networks (CNNs), which the invention is associated to LeCun *el al.* [LBD+90]. Later on, it has been empirically demonstrated in [GGS+17] that neural nets need be both deep and convolutional in order to maintain their accuracy performance, especially in computer vision.

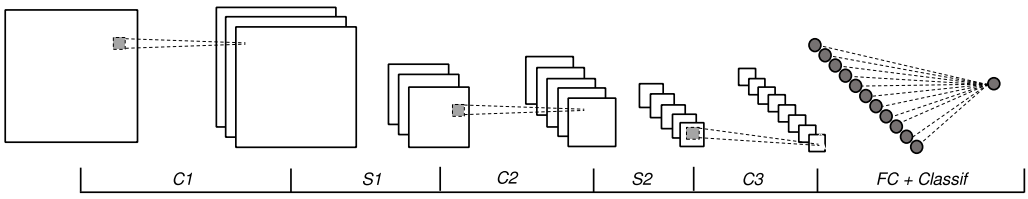


FIGURE 2.3: A typical CNN Structure with 3 convolutional layers, 2 *pool* layers and a single layer MLP

2.2 Deep Convolutional Neural Networks

Convolutional networks are feed-forward neural networks that use convolution instead of general matrix multiplication. Therefore, CNNs have a hierarchical structure that stacks multiple convolution (*conv*) layers. Figure 2.3 illustrates this structure where each *conv* layer includes a set of *three-dimensional* filters that extract features from the input data generating a Feature Map (FM).

TABLE 2.1: Tensors Involved in the inference of a given layer ℓ

	Array	*Dimension*
$\boldsymbol{X}$	Input FMs	$B \times C \times H \times W$
$\boldsymbol{Y}$	Output FMs	$B \times N \times V \times U$
$\boldsymbol{\Theta}$	Learned Filters	$N \times C \times J \times K$
β	Learned biases	N

The depth of a CNN corresponds to to the number of layer it contains, and, the deeper the layer is, the higher is the level of the features it extracts [LBH15]. CNN inference refers to the *feed-forward* propagation of B input images across L layers. A common practice is to manipulate layers, parameters and FMs as multi-dimensional arrays, as listed in table 2.1. Note that when it will be relevant, the type of the layer will be denoted as a super-script, and the position of the layer will be denoted as an under-script.

The dimensions of tensors labelled above by their sizes, are described here-under:

- B: the Batch size (i.e the number of input frames processed by the CNN).
- C: the number of Channels in the considered input image (C_0) or more generally the depth of feature maps that input a CNN layer (C_ℓ).
- H: the Height of the considered input image (H_0) or input feature maps (H_ℓ).
- W: the Width of the considered input image (H_0) or input feature maps (W_ℓ).
- N: the depth of the output feature maps.
- U: the Height of the output feature maps.
- V: the Width of the output feature maps.
- J: the Height of the convolution kernels.
- K: the Width of the convolution kernels.

Next paragraphs detail the best practises and computation of these dimensions.

2.2.1 Common CNN layer types

Recent CNN models may comprise up to hundreds of layers. This section describes the role of each layer type and details the computations it involves.

2.2.1.1 Convolution layers

A convolution layer (*conv*) carries out the feature extraction process by applying a set of 3D-convolution filters $\mathbf{\Theta}^{\text{conv}}$ to a batch B of input volumes $\boldsymbol{X}^{\text{conv}}$. Each input volume has a depth C and can be a color image[2], or an output generated by previous layers in the network.

As illustrated in figure 2.4, applying a 3D-filter to 3D-input results in a 2D *Feature Map (FM)* and, each *conv* layer outputs a set of N two-dimensional features maps.

In some CNN models, a learned offset β^{conv} –called a *bias*– is added to the 3D-conv results, but this practice is discarded in recent models [SZ14]. The computations involved in feed-forward propagation of *conv* layers are detailed in equation 2.3.

$$\forall \{b, n, u, v\} \in [1, B] \times [1, N] \times [1, V] \times [1, U]$$
$$\boldsymbol{Y}^{\text{conv}}[b, n, v, u] = \beta^{\text{conv}}[n] + \sum_{c=1}^{C} \sum_{j=1}^{J} \sum_{k=1}^{K} \boldsymbol{X}^{\text{conv}}[b, c, v+j, u+k].\mathbf{\Theta}^{\text{conv}}[n, c, j, k] \tag{2.3}$$

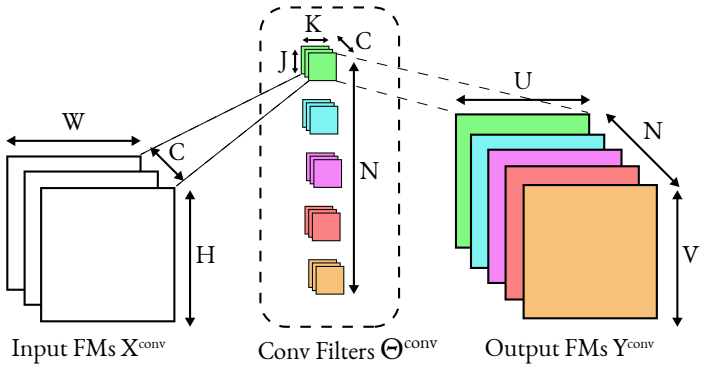


FIGURE 2.4: Example of a Convolutional layer, $C = 3, N = 5$

Applying a 3D-convolution to a 3D-input boils down to applying a mainstream 2D-convolution to each of the 2D-channels of the input, then, at each point, sum the results across all the channels, as shown in equation 2.4

$$\forall n = 1 : N$$
$$\boldsymbol{Y}[n]^{\text{conv}} = \beta^{\text{conv}}[n] + \sum_{c=1}^{C} \textbf{conv2D}\left(\boldsymbol{X}[c]^{\text{conv}}, \mathbf{\Theta}[c]^{\text{conv}}\right) \tag{2.4}$$

The depth of an output FMs, N_ℓ, referring to the number of features extracted by a layer ℓ, is set when designing the CNN topology. The other dimensions of the output FMs can be computed as followed:

$$\begin{cases} V = \dfrac{W - K + 2p_x}{s_x} + 1 \\ U = \dfrac{H - J + 2p_y}{s_y} + 1 \end{cases} \tag{2.5}$$

[2]This is the case of the first *conv* layer, where $C = 3$ for the 3 color components of each pixel.

Where $p_x^{(\ell)}, p_y^{(\ell)}$ refers to horizontal (*resp.* vertical) padding[3] and $s_x^{(\ell)}, s_y^{(\ell)}$ refers to horizontal (*resp.* vertical) stride[4]. In general, popular CNN models use rectangular inputs and rectangular filters in a way that FMs have the same horizontal and vertical dimensions ($W = H, U = V, J = K, p_x = p_y, s_x = s_y$)

2.2.1.2 Activation Layers

Each *conv* layer of a CNN is usually followed by an activation layer that applies a *non-linear* function in an element-wise fashion across the FMs. Early CNNs were trained with TanH or Sigmoid functions but recent models employ the ReLU function that grants faster training times and less computational complexity, as highlighted in [KSGH12].

$$\forall \{b, n, u, v\} \in [1, B] \times [1, N] \times [1, V] \times [1, U]$$
$$\boldsymbol{Y}^{\text{act}}[b, n, h, w] = \text{act}\Big(\boldsymbol{X}^{\text{act}}[b, n, h, w]\Big) \quad | \quad \text{act} := \text{TanH, Sigmoid, ReLU ...} \tag{2.6}$$

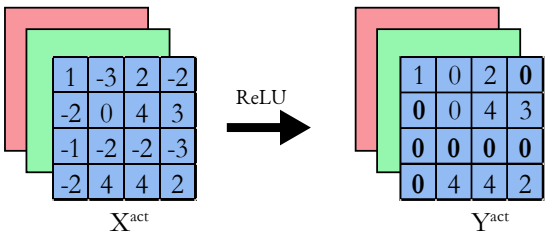


FIGURE 2.5: Example of ReLU activation function

2.2.1.3 Pooling layers

The convolutional and activation parts of a CNN are directly inspired by the cells of visual cortex in neuroscience [HW62]. This is also the case of *pooling* layers, which are periodically inserted in-between successive *conv* layers. As shown in equation 2.7, *pooling* sub-samples each channel of the input FMs by selecting either the *average*, or, more commonly, the *maximum* of a given neighbourhood K. As a results, the dimensionality of a FMs is reduced, as illustrated in figure 2.6

$$\forall \{b, n, u, v\} \in [1, B] \times [1, N] \times [1, V] \times [1, U]$$
$$\boldsymbol{Y}^{\text{pool}}[b, n, v, u] = \max_{p,q \in [1:K]} \Big(\boldsymbol{X}^{\text{pool}}[b, n, v + p, u + q]\Big) \tag{2.7}$$

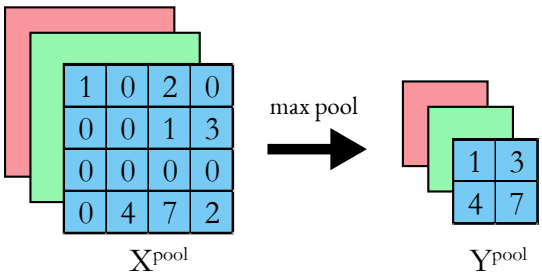


FIGURE 2.6: Illustration of a max pooling layer in a CNNs

[3] Number of zeros put at each edge of the image (North, South, West, East).
[4] Number of pixels (horizontally and vertically) between the application of two successive convolution windows.

2.2.1.4 Fully Connected Layers

When deployed for classification purpose, the CNNs pipeline is often terminated by an MLP referred as Fully Connected (FC) layers. These layers can be seen as *conv* layers with no weight sharing (i.e $W = K$, $H = J$, $U = V = 1$). Moreover, in a same way as *conv* layers, a non-linear function is applied to the outputs of FC Layers.

$$\begin{aligned} &\forall \{b,n\} \in [1,B] \times [1,N] \\ &\boldsymbol{Y}^{\text{fc}}[b,n] = \beta^{\text{fc}}[n] + \sum_{c=1}^{C}\sum_{h=1}^{H}\sum_{w=1}^{W} \boldsymbol{X}^{\text{fc}}[b,c,h,w].\boldsymbol{\Theta}^{\text{fc}}[n,c,h,w] \end{aligned} \tag{2.8}$$

2.2.1.5 Softmax Layer

The Softmax function is a generalization of the Sigmoid function, and "squashes" a N-dimensional vector $\boldsymbol{X}$ to $Sigmoid(\boldsymbol{X})$ where each output is in the range $[0,1]$. The Softmax function is used in various multiclass classification methods, especially in CNNs. In this case, the Softmax layer is placed at the end of the network and the dimension of vector it operates-on (i.e N) represents the number of classes in the considered dataset. Thus, the input of the Softmax is the data generated by the last fully connected layer, and the output is the probability predicted for each class.

$$\begin{aligned} &\forall \{b,n\} \in [1,B] \times [1,N] \\ &Softmax(\boldsymbol{X}[b,n]) = \frac{\exp\left(\boldsymbol{X}[b,n]\right)}{\sum_{c=1}^{N}\exp\left(\boldsymbol{X}[b,c]\right)} \end{aligned} \tag{2.9}$$

2.2.1.6 Batch-Normalization Layers

Batch-Normalization is introduced in [IS15] to speed up training by linearly shifting and scaling the distribution of a given batch of inputs B to have zero mean and unit variance. These layers find also their interest when implementing Binary Neural Networks (BNNs) as they reduce the quantization error compared to an arbitrary input distribution, as highlighted in [HCS+16]. Equation 2.10 details the processing of *batch norm* layers, where the mean μ and the variance σ are statistics collected during the training, α and γ are parameters learned during the training, and ϵ is a hyper-parameter set empirically for numerical stability purposes (i.e avoiding division by zero).

$$\begin{aligned} &\forall \{b,n,u,v\} \in [1,B] \times [1,N] \times [1,V] \times [1,U] \\ &\boldsymbol{Y}^{\text{BN}}[b,n,v,u] = \frac{\boldsymbol{X}^{\text{BN}}[b,n,u,v] - \mu}{\sqrt{\sigma^2 + \epsilon}}\gamma + \alpha \end{aligned} \tag{2.10}$$

2.2.2 Popular CNN Models

CNNs come in multiple shapes with various layers configurations. This section overviews the currently popular CNN models and highlights the improvements they bring. The full topology of the studied CNNs is given in appendix A. Note that more details can be found on a survey of CNN topologies [GWK+17]

LeNet5

Introduced in the late 90s by LeCun *et al.* [LBBH98], LeNet5 was the first commercial success of CNNs, and was deployed for Optical Character Recognition (OCR) of handwritten digits. LeNet5 employs two *conv* layers, respectively with 6 filters of size $(5 \times 5 \times 1)$

and 16 filters of size $(5 \times 5 \times 3)$[5]. These layers are interspersed by two average *pool* layers and terminated by two FC-layers.

AlexNet

The real emergence of CNNs as a prominent technique in the field of computer vision took place in 2012 when Krizhevsky *et al.* submitted AlexNet [KSGH12], the first success of employing CNNs in classifying the ImageNet data-set [DDS+09]. AlexNet was the first network to use a *deep* topology that involved five *conv* layers, three max*pool* and three FC layers. Moreover, AlexNet also introduced the ReLU activation function, which significantly fastened the training process.

The first layer of AlexNet extracts 96 FMs, the second layer *conv2* extracts 256 FMs while layers *conv3*,*conv4* and *conv5* extract 384 FMs. Filter size in these layers ranges from $(11 \times 11 \times 3)$ up to $(3 \times 3 \times 192)$.

VGG

In 2014, Simonyan *et al.* [SZ14] deepen the CNN architecture to 13 *conv* and 3 FC layers. This model, known as VGG, replaces the (11×11) and (5×5) filters by *successive stages* of (3×3) filters. With this method, the size of the receptive field of a given filter remains unchanged while its processing requires less computations, as illustrated in figure 2.7.

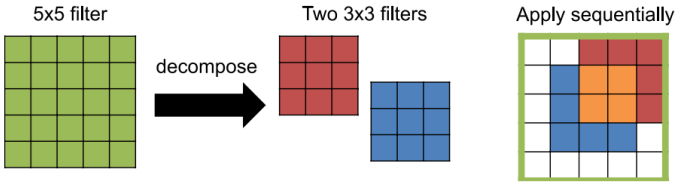


FIGURE 2.7: Decomposing (5×5) filters into two successive stages of (3×3)filters. Used in VGG [SZ14], image from [SCYE17]

By lowering the computational load in the first layers, Simonyan *et al.* can deepen the CNN, which in turn, improves the classification performance by 14,8% on ImageNet. Three versions of this network are proposed with variable depths. Note that the deepest model –known as VGG19– requires 27% more computations then the shallow model –VGG16– to increase the accuracy by only 1%.

GoogLeNet

Szegedy *et al.* [SLJ+15] go deeper and use a CNN of 22 learned layers to outperform VGG accuracy on ImageNet by 3.2%. This network leverages on 9 *micro-networks* [LCY13], each employing variable filter sizes, to capture visual patterns at multiple scales[6]. Each micro-network, known as an *Inception modules*, includes a (3×3) *maxpool*, as well as (1×1), (3×3) and (5×5) convolution layers arranged in a parallel fashion. The results of these operations is concatenated to the output, as illustrated in Fig. 2.8a. GoogleNet also introduces three-dimensional (1×1) convolutions, known as *bottleneck* filters, are placed before (3×3) and (5×5) convolutions as *dimension reduction* filters. Their role is to increase the depth of each layer, and consequently its modeling power, without increasing the computational complexity.

[5] In fact, LeNet5 second conv layer uses 6 filters (5x5x3), 9 filters (5x5x4) and 1 filter (5x5x6)

[6] Each micro-network includes 2 *conv* layers. GoogLeNet is also known as Inception-v1. Inception-v2 and Inception-v3 models have been proposed. Inception-v4 combines the inception architecture with short-cut connections of ResNet

ResNet

Numerous works such in [HS15] report that when the network depth increases too much (> 50 conv layers), accuracy gets saturated and start even to decrease. This degradation is not caused by over-fitting, but by *gradient vanishing*. This phenomenon appears when training a very deep CNN with gradient descent, and causes the gradient to decrease exponentially with L until it « vanishes ».

To address this problem, and deepen the network while avoiding gradient vanishing, Residual Networks are introduced in [HZRS16]. These networks rely on *skip* connections illustrated in Fig. 2.8b. Instead of learning layers that fit a desired mapping from x to $F(x)$, ResNets learn the mapping between x and $F(x) - x$. Authors demonstrate that the so-called *Residual Mapping* is less prone to gradient vanishing and, as a consequence, ResNets can involve hundreds of layers while being more accurate. For instance, the ResNet-152 CNN contains 155 *conv* layers and delivers an error rate of 6.7% on ImageNet, which outperforms the human accuracy by 3.5 % on this dataset!

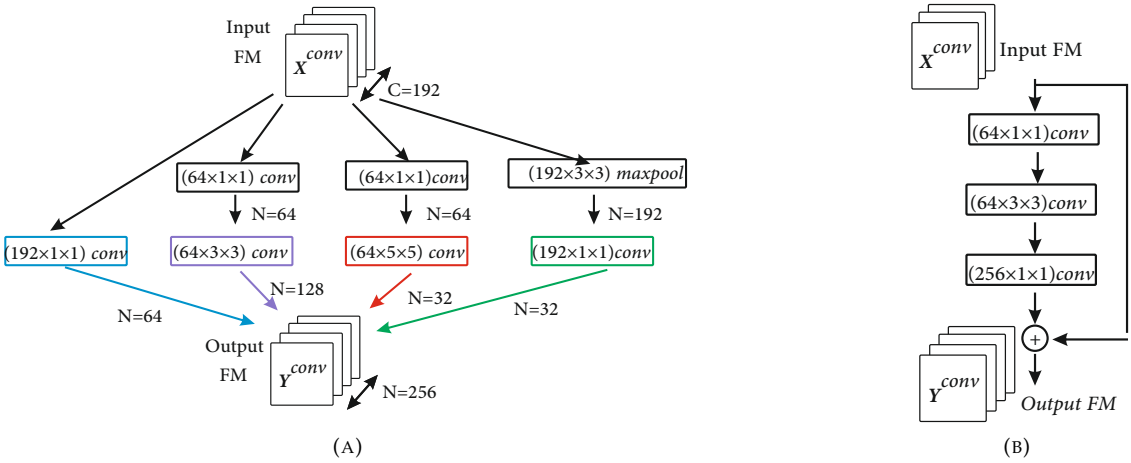


FIGURE 2.8: Advanced CNN Topologies: a-Inception Network, b- Residual Network

2.3 CNN Applications, Datasets and Evaluation Metrics

CNN models, which have been originally proposed to solve image classification problems, are now used in an increasing number of large scale computer vision applications. Indeed, and as stated in section 1, the advantage of machine learning for computer vision is the possibility to re-use the same algorithm for different applications, just by changing the datasets and the labels it processes.

In the sequel, this manuscript focuses on image classification and object detection applications, since they are the most relevant in the context of Smart Camera networks. Nevertheless, CNNs are at the heart of many computer vision algorithms, and have successfully been employed to solve problems related to semantic segmentation [GDDM14, LSD15], saliency maps [ABM13, ZOLW15, KTB15], visual tracking [WOWL15], and many other applications. For more details realted to CNN applications for computer vision, the reader is referred to [GWK+17].

2.3.1 Image Classification with CNNs

The intent of the *classification process* is to categorize an image or a part of an image into one of several classes. In other words, an image classification algorithm outputs the class to which a given object of the image belongs. In this manuscript, the output produced by

a classifier is called a *prediction* while the ground truth value is referred as *annotation* or *label*.

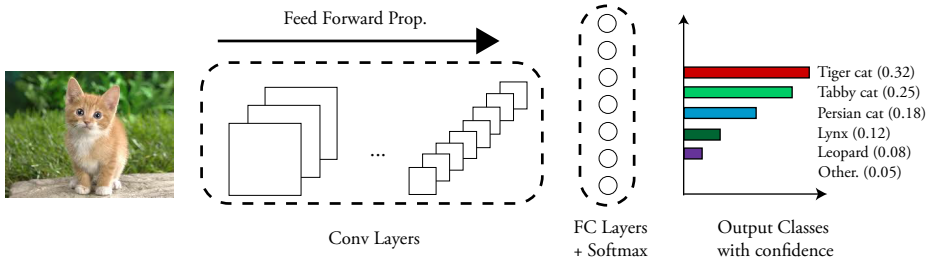


FIGURE 2.9: Example of Image Classification

The performance of image classifiers is usually reported as an *accuracy* rate, defined in equation 2.11. Note that when dealing with datasets involving a very large number of classes, accuracy can be reported as a TOP-N rate (usually Top5). In this case, a sample is considered to be correctly classified if its associated label figures among the top N predictions of the algorithm. This is illustrated in the example of Fig.2.9 which classifies the input image on the left. In this figure, the prediction of the CNN is «Tiger cat» while the ground truth is «Tabby cat». This sample is considered to be correctly classified using the Top5 Accuracy and miss-classified when using the Top1 Accuracy.

$$\text{Accuracy} = \frac{\text{Number of correctly classified samples}}{\text{Total Number of Samples}} \tag{2.11}$$

However, when it comes to comparing different classification algorithms, choosing the classifier with the highest accuracy might not be the best solution. Indeed, the previous definition of accuracy supposes that the classifier outputs a prediction whenever the *confidence* on its prediction exceeds a given *decision threshold*. Varying this decision threshold can heavily impact the outputs of the algorithm, biasing the comparison between two classifiers.

For these reasons, performances of classifiers are evaluated using the precision and recall values. To define these metrics, the following *confusion metric* is built:

TABLE 2.2: Confusion Matrix

		Label	
		Positive	Negative
Prediction	Positive	True Positive (TP)	False Positive (FP)
	Negative	False negative (FN)	True Negative (TN)

With this matrix, precision and recall are defined as:

$$\text{precision} = \frac{TP}{TP + FP} \tag{2.12}$$

$$\text{recall} = \frac{TP}{TP + FN} \tag{2.13}$$

Thanks to these metrics, the *precision-to-recall* curve can be used to compare different algorithms depending on the requirement (high precision at the cost of recall, or high recall with lower precision)[7].

The networks studied in the last section were all originally trained to solve this classification problem, and generally consider the presence of a *unique* class in an image. For these networks, figure 2.11a reports the Top1 and Top5 accuracy rates on the ImageNet dataset.

[7]Examples of the Precision-to-Recall study are nicely explained in
https://www.quora.com/What-is-the-best-way-to-understand-the-terms-precision-and-recall

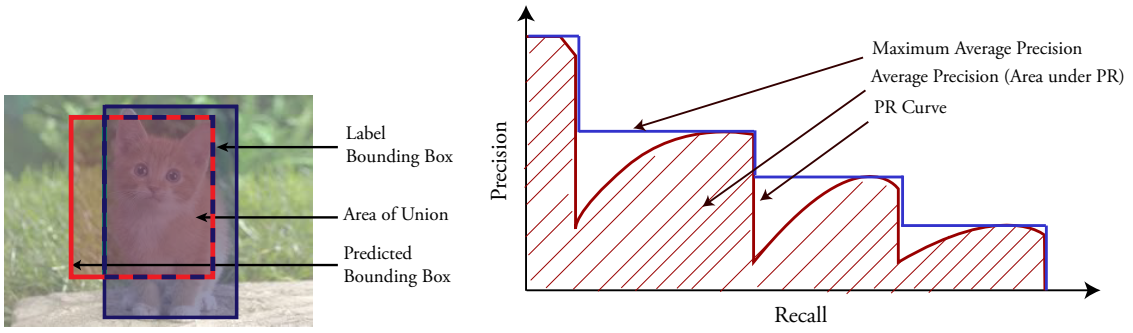
2.3.2 Object Localization and Detection with CNNs

The *object localization* problem is slightly different: the output in no longer the class of a given object, but also its coordinates on the image, given under the form of a *bounding box*. Object localization can also be applied on all the objects in the image, which results in multiple bounding boxes. These objects can belong to different classes, and this problem is known as *object detection*. In these applications, the datasets provide annotations of the classes of the objects present in the image, but also their positions.

To decide if a given detection is positive or not, the Intersection over Union (IoU) ratio, defined in equation 2.14, and depicted in Fig.2.10a, is generally used.

$$\text{IoU} = \frac{\text{Area of predicted bounding box} \cap \text{Area of labeled bounding box}}{\text{Area of predicted bounding box} \cup \text{Area of labeled bounding box}} \tag{2.14}$$

The Performance of CNN detectors is usually reported using the Mean Average Precision (mAP) metric [EVW+10]. This metric is explained in Fig.2.10b, which gives an example of a precision-to-recall curve for single class of the dataset. To compute the average precision (AP) metric, the precision-to-recall curve is divided into a number of segments the where the maximum precision is averaged. This corresponds to finding the total area under the blue curve and dividing it by the number of segments. The mAP metric simply averages the former AP measure over all classes of a given dataset. Note that the mAP50 value considers that a detection is positive if its IoU exceeds 0.5, while The mAP75 considers an IoU that exceeds 0.75.



(A) IoU Measure (B) PR curve and mAP Metric

FIGURE 2.10: Metrics for Object Detectors

The CNN models studied in section 2.2.2 generally constitute a *backbone* for CNN-Based Object detectors. Two approaches can be distinguished:

- *Region Based* approaches rely on a first stage of *region proposal* that outputs bounding box candidates, and a second stage that processes the candidates to output the detections. In the original Region-CNN works (RCNN) [GDDM14], the second stage uses the AlexNet CNN. Later on, improved versions known as Fast-RCNN [Gir15] and Faster-RCNNs [RHGS17] were proposed, and used deeper VGG and ResNet models.

- *Single Shot* approaches, such YOLO [RDGF16, RF18] and SSD [LAE+16], rely only on one network, jointly trained on bounding boxes and labels. Such detectors directly predict class probabilities and bounding boxes in a *single* evaluation, granting them faster inference times. As backbones, SSD is based on the VGG Network while YOLO uses the DarkNet CNN which topology is given in appendix A

2.3.3 Image Datasets Availability as a Boost to CNN

Two key factors made the success of deep CNNs: the availability of annotated datasets and the development of powerful computational platforms. The former factor is discussed in this section and the latter is detailed in sections 2.4 and 2.5.

The availability of massive-sized image databases has provided enough annotated inputs to train robust large-scale feature extractors and accurate classifiers for machine vision. Tab. 2.3 lists these popular and « open » datasets.

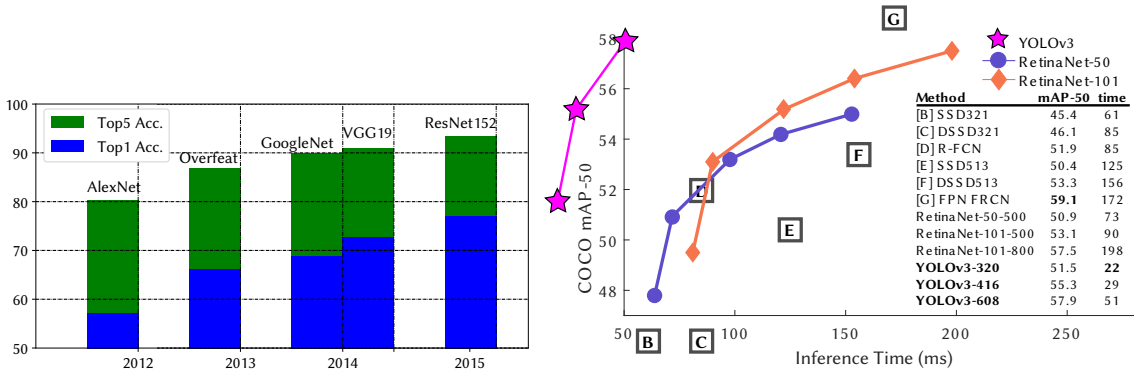
TABLE 2.3: Popular datasets for computer vision applications

Dataset	Resolution	#Classes	#Train Samples	#Test Samples	Application
MNIST [LBD+90]	28 x 28 x 1	10	60 K	10 K	OCR
Cifar10 [Kri09]	32 x 32 x 3	10	50 K	10 K	Classification
Cifar100 [Kri09]	32 x 32 x 3	100	50 K	10 K	
ImageNet [DDS+09]	256 x 256 x 3	1000	1.3 M	100 K	
PascalVOC [EVW+10]	-	20	11.5 K	-	Detection
COCO [LMB+14]	-	91	328 k		
OpenImages [KDA+16]	-	5000	9 M	125.4 K	

Most of the recent CNN improvements took place at the ImageNet Large Scale Visual Recognition Competition (ILSVRC) [RDS+14]. ImageNet [DDS+09] is a visual database that contains –as of 2016– over 10M images where objects present in each picture are hand-annotated. ILSVRC provides 1.3M color images from ImageNet for training and 100k for testing CNN *classifiers*.

Since the first CNN model was proposed to solve ImageNet, and discussed in section 2.2.2, the accuracy of CNN classifiers continues to improve overtime, as summarized in Fig.2.11a.

The same improvements apply to CNN-based object detectors, where two main benchmarks are made available (PASCAL-VOC [EVW+10] and MS-COCO [LMB+14]). Figure 2.11b compares the performance of CNN-based detectors on the COCO dataset. One may note that the Single Shot methods outperform region based methods –R-CNN– in terms inference time, making them more adequate in a context of real-time embedded vision.



(A) Image Classifiers (B) Object Detectors [RF18]

FIGURE 2.11: Performance of CNN-based classifiers and detectors

2.4 Workload and Implementation Challenges on SmartCams

In the context of embedded vision, DL based methods are too computationally intensive for general purpose processors, especially for the ones generally included in smart camera nodes. This section details the computations that deep CNNs involve. Moreover, it describes how these computations exhibit a massive, fine-grain parallelism, which makes the implementation of CNN on embedded devices feasible thanks to dedicated hardware accelerators.

2.4.1 Computations in CNN Inference

As shown in equations 2.3 and 2.8, the processing of CNNs involves an intensive use of the Multiply Accumulate (MAC) operation. All these MAC operations take place at *conv* and FC layers while the remaining parts of network are element-wise transformations that can be generally implemented with low complexity computational requirements.

In this manuscript, the computational workload $\mathcal{C}$ of a given CNN corresponds to the number of MACs it involves during inference[8]. The number of these MACs mainly depends on the topology of the network, and more particularly on the number of *conv* and FC layers and their dimensions. Thus, the computational workload can be expressed as in equation 2.15, where, L_c (*resp.* L_f) is the number of *conv* (*resp.* fully connected) layers, and $\mathcal{C}_\ell^{conv}$ (*resp.* $\mathcal{C}_\ell^{fc}$) is the number of MACs occuring on a given convolution (*resp.* fully connected) layer ℓ.

$$\mathcal{C} = \sum_{\ell=1}^{L_c} \mathcal{C}_\ell^{conv} + \sum_{\ell=1}^{L_f} \mathcal{C}_\ell^{fc} \tag{2.15}$$

$$\mathcal{C}_\ell^{conv} = N_\ell \times C_\ell \times J_\ell \times K_\ell \times U_\ell \times V_\ell \tag{2.16}$$

$$\mathcal{C}_\ell^{fc} = N_\ell \times C_\ell \times W_\ell \times H_\ell \tag{2.17}$$

In a similar way, the number of weights, and consequently the size of a given CNN model, can be expressed as follows:

$$\mathcal{W} = \sum_{\ell=1}^{L_c} \mathcal{W}_\ell^{conv} + \sum_{\ell=1}^{L_f} \mathcal{W}_\ell^{fc} \tag{2.18}$$

$$\mathcal{W}_\ell^{conv} = N_\ell \times C_\ell \times J_\ell \times K_\ell \tag{2.19}$$

$$\mathcal{W}_\ell^{fc} = N_\ell \times C_\ell \times W_\ell \times H_\ell \tag{2.20}$$

For state-of-the-art CNN models, L_c, N_ℓ and C_ℓ can be quite large (see appendix A). This makes CNNs *computationally and memory intensive*, as illustrated in table 2.4, where for instance, the classification of a single frame using the VGG19 Network requires 19.5 Billions MAC operations.

It can be observed in the same table that most of the MACs occur on the convolution parts, and consequently, 90% of the execution time of a typical the inference is spent on *conv* layers [CX14]. By contrast , FC layers marginalize most of the weights, and thus the size of a given CNN model.

[8] Batch size is set to 1 for clarity purposes

TABLE 2.4: Workload of Popular CNN models. Computational workload given as the number of MACs. Accuracy measured on single-crops of ImageNet test-set.

Model	AlexNet [KSGH12]	GoogleNet [SLJ+15]	VGG16 [SZ14]	VGG19 [SZ14]	ResNet101 [HZRS16]	ResNet-152 [HZRS16]
Top1 err (%)	42.9 %	31.3 %	28.1 %	27.3 %	23.6% %	23.0%
Top5 err (%)	19.80 %	10.07 %	9.90 %	9.00 %	7.1 %	6.7 %
L_c	5	57	13	16	104	155
$\sum_{\ell=1}^{L_c} \mathcal{C}_\ell^{conv}$	666 M	1.58 G	15.3 G	19.5 G	7.57 G	11.3 G
$\sum_{\ell=1}^{L_c} \mathcal{W}_\ell^{conv}$	2.33 M	5.97 M	14.7 M	20 M	42.4 M	58 M
Act.	ReLU					
Pool.	3	14	5	5	2	2
L_f	3	1	3	3	1	1
$\sum_{\ell=1}^{L_f} \mathcal{C}_\ell^{fc}$	58.6 M	1.02 M	124 M	124 M	2.05 M	2.05 M
$\sum_{\ell=1}^{L_f} \mathcal{W}_\ell^{fc}$	58.6 M	1.02 M	124 M	124 M	2.05 M	2.05 M
$\mathcal{C}$	724 M	1.58 G	15.5 G	19.6 G	7.57 G	11.3 G
$\mathcal{W}$	61 M	6.99 M	138 M	144 M	44.4 M	60 M

2.4.2 Parallelism in CNNs

The high computational workload of CNNs makes their inference a challenging task, especially on low-energy embedded devices. The key solution to this challenge is to leverage on the extensive concurrency they exhibit. These parallelism opportunities can be formalized as:

- **Batch Parallelism:** CNN implementations can simultaneously classify multiple frames grouped as a *batch B* in order to reuse the filters in each layer, minimizing the number the memory accesses. However, and as shown in Fig.2.12, batch parallelism quickly reaches it limits. This is due to the fact that most memory transactions are made for storing intermediate results and not loading CNN parameters. Consequently, reusing the filters only slightly impacts the overall processing time per image.

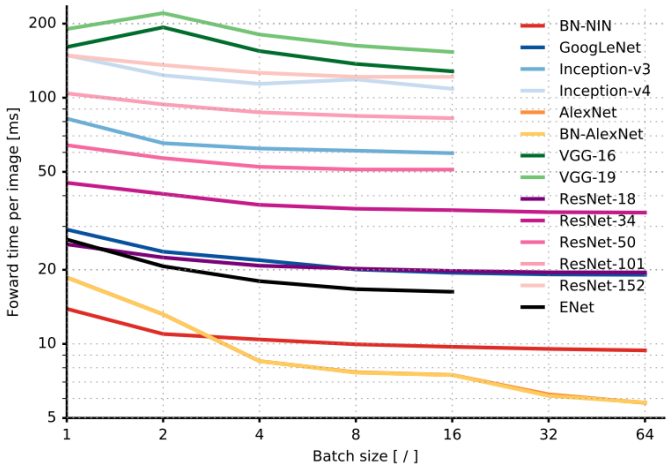


FIGURE 2.12: Performance of Batch Parallelism in popular CNNs: A speedup of 3× in inference time per image is achieved by AlexNet due to better optimization of its FC layers for larger batches [CPC16].

- **Inter-layer Pipeline Parallelism:** CNNs have a feed-forward hierarchical structure consisting of a succession of data-dependent layers. These layers can be executed in a pipelined fashion by launching layer (ℓ) before ending the execution of layer $(\ell - 1)$. This pipelining costs latency but increases throughput.

Moreover, the execution of the most computationally intensive parts (i.e *conv* layers), exhibits the four following types of concurrency:

- **Inter-FM Parallelism:** Each two-dimensional plane of a FM can be processed separately from the others. Meaning that P_N elements of $\boldsymbol{Y}^{\text{conv}}$ can be computed in parallel ($0 < P_N < N$).
- **Intra-FM Parallelism:** In a similar way, pixels of a single output FM plane are data-independent and can thus be processed concurrently by evaluating $P_V \times P_U$ Values of $\boldsymbol{Y}^{\text{conv}}[n]$ ($0 < P_V \times P_U < V \times U$)
- **Inter-convolution Parallelism:** 3D-convolutions occurring in *conv* layers can be expressed as a sum of 2D convolutions as shown in equation 2.4. These 2D convolutions can be evaluated simultaneously by computing concurrently P_C elements ($0 < P_C < C$).
- **Intra-convolution Parallelism:** The 2D-convolutions involved in the processing of *conv* layers can be implemented in a pipelined fashion such as in [Sho94]. In this case $P_J \times P_K$ multiplications are implemented concurrently ($0 < P_J \times P_K < J \times K$).

2.4.3 Memory Accesses

As a consequence of the previous discussion, the inference of a CNN shows large vectorization opportunities that can be exploited by allocating multiple computational resources to concurrently process multiple features. However, this parallelisation can not accelerate the execution of a CNN if no datacaching strategy is implemented. In fact, memory bandwidth is often the bottleneck when processing CNNs.

In *FC* parts, the execution can be memory-bounded because of the high number of weights that these layers contain, and consequently, the high number of memory reads required.

This is expressed in eq.2.21 where $\mathcal{M}_\ell^{fc}$ refers to the number of memory accesses occurring in an FC layer ℓ. This number can be written as the sum of memory accesses reading the inputs $\boldsymbol{X}_\ell^{fc}$, the memory accesses reading the weights $\boldsymbol{\theta}_\ell^{fc}$), and the number of memory accesses writing the results ($\boldsymbol{Y}_\ell^{fc}$).

$$\begin{aligned}
\mathcal{M}_\ell^{fc} &= \text{MemRd}(\boldsymbol{X}_\ell^{fc}) + \text{MemRd}(\boldsymbol{\theta}_\ell^{fc}) + \text{MemWr}(\boldsymbol{Y}_\ell^{fc}) && (2.21)\\
&= C_\ell H_\ell W_\ell + N_\ell C_\ell H_\ell W_\ell + N_\ell && (2.22)\\
&\sim N_\ell C_\ell H_\ell W_\ell && (2.23)
\end{aligned}$$

Note that the fully connected parts of state-of-the-art models involve large values of N_ℓ and C_ℓ, making the memory reading of weights the most impacting factor, as formulated in eq. 2.23. In this context, the batch parallelism discussed above is relevent, and can significantly accelerate the execution of CNNs with a large number of FC layers[9].

In the *conv* parts, the high number of MAC operations results in a high amount of memory accesses, as each MAC requires at least 2 memory reads and 1 memory write[10].

[9] Which is the reason why AlexNet (3 FC Layers) benefits from a considerable acceleration when implementing batch processing, as depicted in Fig.2.12.

[10] This is the best case scenario of a fully pipelined MAC where intermediate results do not need to be loaded.

This number of memory accesses accumulates with the high dimensions of data manipulated by *conv* layers as shown in equation 2.25. If all these accesses are towards external memory (for instance, Dynamic Random Access Memory (DRAM)), throughput and energy consumption will be highly impacted, because a DRAM access engenders high latency and energy consumption, even more than the computation it self [Hor14].

$$\mathcal{M}_\ell^{conv} = \text{MemRd}(\boldsymbol{X}_\ell^{conv}) + \text{MemRd}(\boldsymbol{\theta}_\ell^{conv}) + \text{MemWr}(\boldsymbol{Y}_\ell^{conv}) \tag{2.24}$$

$$= C_\ell H_\ell W_\ell + N_\ell C_\ell J_\ell K_\ell + N_\ell U_\ell V_\ell \tag{2.25}$$

The number of these DRAM accesses, and thus latency and energy consumption, can be reduced by implementing a memory caching hierarchy using on-chip memories. As discussed in the next sections, state-of-the-art CNN accelerators employ register files as well as several levels of caches. The former, being the fastest, is implemented at the nearest of the computational capabilities. The latency and energy consumption resulting from these caches is lower by several orders of magnitude than external memory accesses, as pointed-out in [SCYE17].

2.5 Hardware for mainstream DL

As detailed in the last section, CNN workloads call for important computational resources to exploit the parallelism, as well as the memory caching requirements and to reduce the number of external memory accesses. As a result, efficient execution of CNNs should be achieved on particular hardware architectures. As pointed-out in [SCYE17], we distinguish two main hardware architectures: Temporal and Spatial.

2.5.1 Temporal Architectures

Temporal Architectures appear on multi-cores processors such Central Processing Units (CPUs) and Graphics Processing Units (GPUs). Both rely on the « Single Instruction on Multiple Data (SIMD) » paradigm to capture the parallelism of CNNs. Thanks to SIMD, temporal architectures employ multiple processing elements to simultaneously perform the same operation (i.e MAC) on multiple inputs (i.e feature maps).

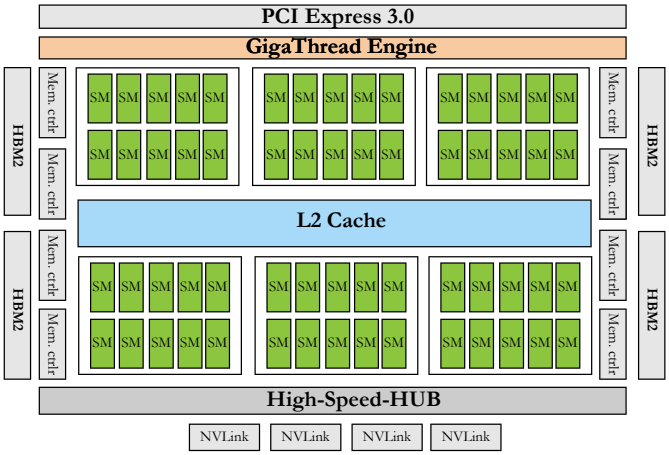
CPUs are the most general purpose and easily programmable computational platforms. Current multi-core CPUs can peak at over 1 TFLOP per second, which is enough to infer, and even train medium and moderately large CNNs [RHR+17]. However, the training and inference of state-of-the-art models were only made possible by a more specialized hardware platform: GPUs.

GPUs are multi-core processors that are specialized at manipulating computer graphics. Current GPUs architectures, such as the Nvidia Pascal [Nvi17] architecture depicted in Fig.2.13a, employ thousands of Processing Elements (PEs) –known as NVidia Cuda cores (Fig.2.13c)–, each having its own computational capabilities by means of floating-point arithmetci units. These cores are grouped into Streaming Multiprocessors (SMs) (Fig.2.13b) that include separate «level 1» Memory caches, and «level 2» caches shared across different SMs.

GPUs use the «single-instruction multiple threads» (SMT) method. They can store in internal registers the context of thousands of concurrent threads (actually, each SM has its own copy of the context of all threads) and fire execution upon availability of thread

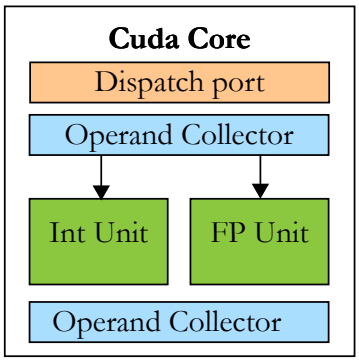
data. GPU boards also include dedicated off-chip memories (GDDRs) with high capacity and bandwidth.

Thanks to this highly parallel structure, GPUs are more efficient than general-purpose CPUs in processing *embarrassingly parallel algorithms*, which is typically the case of classic CNN models. For instance, a GTX 1080Ti GPU is able to peak at 12 TFLOPs when inferring CNNs, which corresponds to 36.87ms of computation time when executing AlexNet inference [11].



(A) Overall GPU Architecture

(B) Streaming Multiprocessor



(C) Cuda Core

FIGURE 2.13: Nvidia Pascal Architecture [Nvi16]

2.5.2 Libraries and development Frameworks

To make the use of CPU and GPU accelerators efficient, specialized libraries for parallel computing, and more particularly deep learning have been developed. This is for instance the case of CuDNN on Nvidia GPUs[12] and MKL-DNN[13] for Intel CPUs. A cross-platform alternative, the DeepCL library[14] provides acceleration for heterogeneous hardware architectures (CPU/GPU/FPGA) through the OpenCL standard [Per17].

Built-upon these libraries, dedicated frameworks for deep learning are proposed. These aim at improving productivity of designing, training and deploying CNNs, such as Caffe [JSD+14] and TensorFlow [ABC+16].

[11] For more details, a CNN Benchmark on GPUs is available in https://github.com/jcjohnson/cnn-benchmarks
[12] https://developer.nvidia.com/cudnn
[13] https://github.com/intel/mkl-dnn
[14] https://github.com/hughperkins/DeepCL

2.5.3 Spatial Architectures

Temporal architectures rely on a centralized control logic for multiple processing elements, following a Von Neumann execution model [Tay06]. These elements can only fetch data from the memory hierarchy and cannot communicate directly with each other, which may limit the performance of a given implementation. In addition, general purpose temporal architectures advocate the ease of programmability at expense of performance and efficiency. By contrast, spatial architectures implement a processing chain where computing elements can directly pass data from one to another. This architecture model naturally fits the *streaming nature* of CNN graphs, and allows the processing elements to include their own control fabric and local memory.

Spatial CNN architectures are often deployed as Application Specific Integrated Circuitss (ASICs) or mapped on FPGA devices. A key advantage of these devices is their ability to support fine-grain parallelism with low energy consumption. This makes spatial architectures particularly efficient when processing irregular parallelism patterns and custom precision computations. As a result, these *dedicated accelerators* deliver superior energy efficiency when compared to temporal architectures, which comes at the price of low programmability and flexibility.

Among a large amount of ASICs for Deep learning [DFC+15,HLM+16,CB16,ACRB16, CES16,RWA+16,MM16,DDL+18], the Tensor Processing Unit (TPU), developed by Google in late 2016, encountered the largest commercial success due its capability to support various Machine Learning algorithms in addition to its tight integration with the TensorFlow framework. However, the major drawback of ASICs remain their lack of reconfigurability and their high production cost.

By contrast, FPGAs benefit from a higher hardware flexibility and reconfigurability at the price of a lower computation per watt ratio. Still, current generation of FPGAs can catch the computational workload of state-of-the-art CNNs thanks to a high density of hard-wired Digital Signal Processing (DSP) blocks that can deliver up to 8 TFLOPs [Int17]. In addition, FPGAs embed a collection of *In-situ* on-chip memories, located next to DSPs, which significantly reduces the needs of external memory accesses.

TABLE 2.5: Comparison of Available Hardware to Accelerate CNN Workload

Hardware	CPU i7-7980HQ	GPU GTX1080Ti	FPGA DE5-Net
Inference Time(ms)[15]	1630	4.31	15
Power (W)[16]	41	206	27

[15] Inference time of AlexNet mesured with Caffe for single batches

[16] Power dissipation of CPU estimated with Intel Power Profiling tools. Power dissipation of GPU estimated with nvidia-smi tools. Power dissipation of FPGA measured with a power-meter in [Wan17]

2.6 Embedded Deep Learning

Because of latency, bandwidth and security concerns, it is more suitable to infer deep learning models locally, next to the sensor, rather than offloading the computations into the cloud [Guo17, RCLC17, EP18]. This is particularly true for smart camera networks where it is critical to extract meaningful information directly from the video streams at the nearest of the image sensors. The challenge of *embedded deep learning* is thus to infer deep and accurate CNN models, on devices with stringent energy consumption (Typically under 20W), and thus limited computational power and low memory resources. These constrains make the state of the art solutions –such the ones listed table 2.5– unfeasible for embedded CNN inference. To address this challenge, two strategies are considered: Top-down and bottom-up.

2.6.1 Lightweight CNN Models for Embedded Vision: The Top-Down Approach to Embed Deep Learning

Top-down approaches design and train networks to *jointly* reach modelling accuracy AND *energy efficiency* to target low-power devices. In other words, the objective is no longer to have networks delivering the best accuracy on a given dataset, but to have networks that can be executed in an embedded system with a tolerable accuracy. In general, the key idea to design such models is to deepen the CNN without increasing its computational workload and/or the number of weights. This is achieved by using *small convolution kernels*, *bottleneck filters* and *inception modules*, as detailed in section 2.2.2. Examples of such *lightweight* CNN models are given in table 2.6

TABLE 2.6: Lightweight CNN Models

Model	AlexNet	ResNet	SqueezeNet	MobileNet v1	MobileNet v2	NasNet Mobile
Top1 err. (%)	42.9	24.7	44.6	29.5	28.1	25.7
Work.(GMACs)	0.72	3.86	0.86	0.64	0.48	0.56
Params. (M)	61	25.5	1.2	4.3	7.8	7.7

Moreover, computations can be further reduced by introducing approximate computing methods that trade a minimal amount of modeling power for efficiency improvements. A review and performance analysis of these methods will be detailed in the next chapter (c.f section 4.4).

2.6.2 Hardware Acceleration Platforms for Embedded Vision: The Bottom-Up Approach to Embed Deep Learning

The bottom-up approach deploys neural networks on the edge by optimizing the hardware architecture towards energy efficiency. This is typically the case of Nvidias' Tegra GPUs, which include a limited number of Processing Cores, or Nvidias' Max-Q GPU, which are under-clocked to satisfy certain power-dissipation requirements.

Spatial Architectures for Embedded Vision have also been proposed. For instance, Intels' Movidius Neural Computing Stick (NCS) is a platform built around an application specific chip known as the Myriad Vision Processing Unit. The NCS can accelerate the execution of AlexNet by x6.13 when plugged on a RaspberryPi3 board[17] at the price of 1.3W more power dissipation. Popular platforms to accelerate embedded deep learning applications are compared in table 2.7.

[17] Simply through a USB interface
[18] As of June 2018 on mouser.eu

TABLE 2.7: Popular Embedded CNN Accelerators

Platform	RasPi3	Movidius NCS	Jetson TX1	DE1-SoC
CPU	Cortex A53	Cortex A53	Cortex A57	Cortex A9
Accelerator	- (CPU only)	Myriad2 VPU (ASIC)	Tegra X1 (GPU)	Cyclone V (FPGA)
Cost (€)[18]	43	66+43	366.0	210.0
Inference time (ms)	1803	294.0	30.3	205.5
Power(W)	1.3	0.75 + 1.3	5.0	2.1

2.7 Conclusions

This chapter has covered the recent developments of embedded deep learning. For computer vision, deep learning methods based on CNNs currently offer tremendous opportunities of deploying new services and products, especially in a smart camera context. Moving deep learning to embedded systems, as shown in this chapter, is a challenging task but greatly increases the potential and reliability of embedded vision. With this objective in mind, next chapters discuss novel solutions, based on reconfigurable hardware, for embedding deep learning into energy constrained embedded devices.

Chapter 3

Reconfigurable Hardware for Embedded Vision

When evaluating hardware platforms to accelerate a domain specific applications, the trade-off between flexibility, performance and power consumption is always considered. On one end of the spectrum, general purpose processors such CPUs and GPUs provide a high degree of programmability while offering a relatively low performance/watt ratio. On the other end of the spectrum, ASICs deliver better performance per watt at the price of low flexibility and high production coasts.

FPGAs somehow stay between the two and deliver a good compromise between the three metrics. This is especially true in the case of CNN acceleration; While FPGAs have not been known for offering top performance when compared to ASICs and GPUs, they are known to provide superior energy efficiency (vs GPUs) and better flexibility (vs ASICs).

This chapter gives an introduction to FPGAs and details their reconfigurable resources and design flow. In a context of computer vision, this chapter also demonstrates how FPGAs can exploit the streaming-processing model of computation of many vision applications to accelerate their execution. To support this claim, the third section describes an implementation example which studies the dataflow implementation of image convolution. Finally, the relevance of FPGAs for CNN acceleration is discussed.

3.1 FPGA Architecture

Since their introduction in the late 80s, Field-Programmable Gate Arrays (FPGAs) have been providing a growing amount of computational resources operating at higher frequencies. These technological improvements have allowed the implementation of increasingly complex applications on FPGA-powed platforms.

In it simplest form, the architecture of an FPGA is at least composed of three elements, as illustrated on Fig 3.1,

- **Logic Cells (LCs)** are the building blocks of an FPGA and are arranged in matrix form. As shown in the Fig 3.1, each component is *programmable*, and identical to the others.
- **The Interconnect Network** links logical resources together to implement complex functions. Within an FPGA, interconnections are often hierarchical, where each level of the hierarchy operates at a different transmission frequency.
- **Configurable Inputs/Outputs (I/Os)** are circuits that interface the FPGA with the external environment. Each input/output circuit controls a *pin* of the FPGA device and can be set as an input, an output, a bidirectional signal, or can be unused staying a high impedance state.

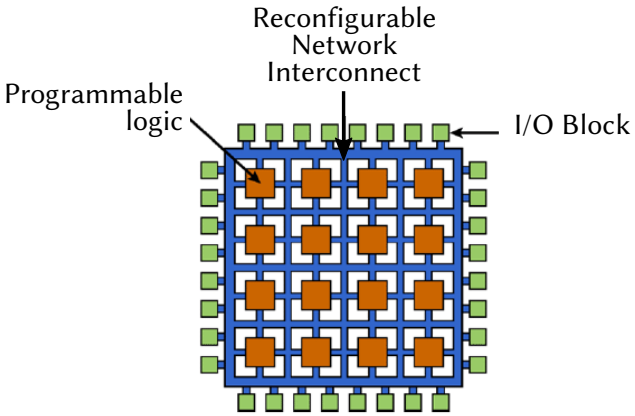


FIGURE 3.1: Simplified Block diagram of an FPGA device

3.1.1 Logic Cells

The basic function of logic cells is to provide calculation and storage capabilities to the FPGA. A logic cell is typically composed of a Look-Up Table (LUT) for implementing the combinatorial part and a storage element (register) for implementing the sequential functions. A LUT of N inputs behaves like a memory with 2^N entries; in order to perform a combinatorial function, the truth table corresponding to the desired boolean equation is loaded into this memory.

According to the FPGA manufacturer, LCs come in relatively similar shapes. At Intel, FPGA building blocks are referred as Logic Elements (LEs) and have six inputs: four of them come from the interconnection network, one from the carry chain and one from the register. The control signals of the registers also come from the interconnection network, as shown in figure. 3.2. The logic element illustrated in the last figure can operate in two modes: the *normal* mode implements combinatorial and sequential functionalities while the *arithmetic* mode implements operations like additions, accumulations or comparisons.

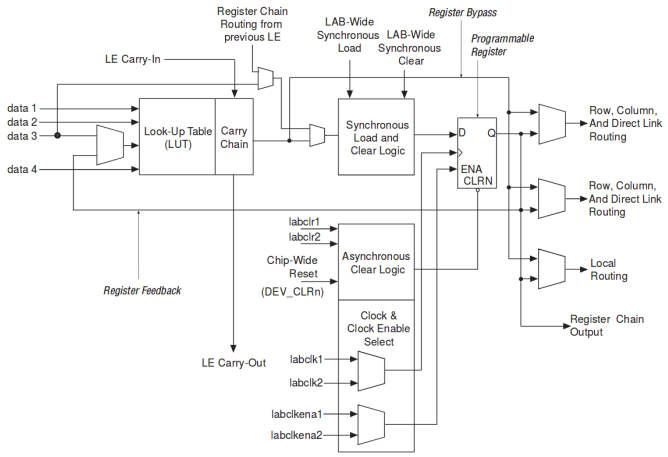


FIGURE 3.2: Logic Element of an Intel Cyclone III FPGA [Int14a]

In modern FPGA devices, the general structure detailed above has evolved in favour of complex blocks that offer greater flexibility and better optimization in resource placement. In this context, Intel introduced the Adaptative Logic Modules (ALMs) (see Fig.3.3), which are the new elementary blocks replacing the logic elements. Similarly, Xilinx (resp.

Actel) introduced *Slices* (resp. *VersaTile*) as elementary logic blocks. These blocks are relatively similar in their structures[1] and have several operating modes. The next paragraph lists these modes using Intels' ALMs as examples.

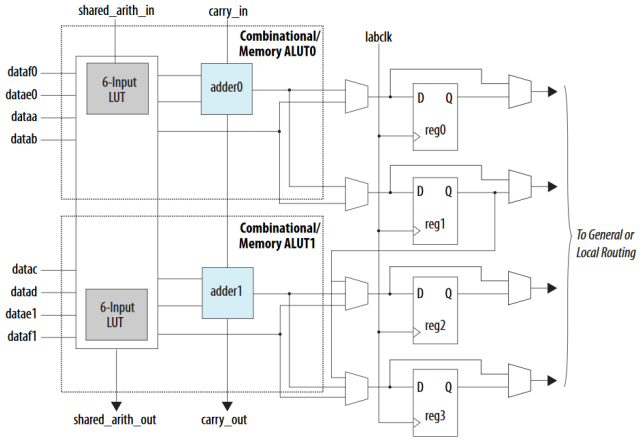


FIGURE 3.3: Structure of an Adaptive Logic Module [Int18a]

Each ALM has four registers and higher number of inputs. Moreover, the operating modes have been extended when compared to the LE structure:

- In the *normal* mode, a single ALM can implement two combinatorial functions of four inputs, or a single function of six inputs.
- In the *extended LUT* mode, the ALM can be connected to seven inputs but the output can not be synchronized by the register. This mode is commonly used for the if-else statements of hardware description languages. The eighth entry of the unused ALM can be potentially connected to an output register, allowing synthesis tools to optimize the surface utilization. This optimization, known as the register packing technique, groups in the same ALM the purely combinatorial functions of the LUTs and the storage of a signal from outside the block.
- In the in *arithmetic* mode, an ALM infers two four-input LUTs and two FAs. Each adder is hard-wired and can perform a binary addition. These two FAs can also be chained to perform an addition a two-bits with and input carry.
- Finally, the ALM in the *shared arithmetic* mode can implement a three-input add. In this mode, the ALM is configured with 4 four-input LUTs. Each LUT either computes the sum of three inputs or the carry of three inputs. The output of the carry computation is fed to the next FA using a dedicated connection called the shared arithmetic chain. This shared arithmetic chain can significantly improve the performance of an adder tree by reducing the number of summation stages required to implement an adder tree.

3.1.2 Interconnect Networks

After highlighting the reconfigurability of logic cells, this section focuses on the Interconnect Networks of FPGAs. As explained above, logic blocks of an FPGA are interconnected to each other through a programmable network. This network provides connections between the logic resources to implement a user-defined circuit.

[1] The main nuances between Intel ALMs, Xilinx Slices, and Actel VersaTiles can be found in http://ee.sharif.edu/~asic/Docs/fpga-logic-cells_V4_V5.pdf

The routing interconnect consists of wires and programmable switches –configured using a programmable technology – that form the desired connection. The interconnect network differs from one FPGA to another, however, most FPGAs share a similar routing structure that involves horizontal and vertical routing tracks which are interconnected through switch boxes. Programming these switches interconnects the desired blocks, as depicted in Fig.3.4[2].

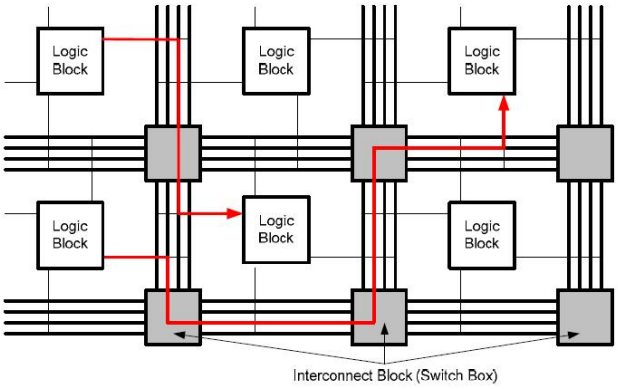


FIGURE 3.4: Scheme of an Interconnect Network of an FPGA

The connections between the blocks is a key factor in the performance of a given FPGA implantation. The longer the connection between two blocks is, the higher the transmission delay will be, lowering the maximum operating. The placement of the elements on the FPGA is therefore a crucial element for performance.

With the evolution of FPGAs, interconnection systems have been redesigned to improve the frequency performance. In recent FPGAs, the interconnections are hierarchical and the logic resources are grouped into larger entities (Logic Array Blocks (LABs) at Intel and Configurable Logic Blocks (CLBs) at Xilinx). Each level of this hierarchy operates at a different transmission frequency. For instance, in Intel FPGAs, elements within the same LAB [Int14a] have privileged communications. Thus, complex functions can be implemented without passing through the interconnection network, which in turn results in better operating frequencies.

This concept is illustrated in Fig. 3.5, which gives a diagram of a Cyclone V LAB that embeds ten ALMs. The LAB also includes carry chains to transfer the result of an arithmetic calculation, and register chains to transfer a sequential output from one ALM to another[3]. Finally, note the presence of *Direct Links*, which are used to link the LAB to external elements such as other adjacent LABs, external memory blocks or multiplier units. These elements are the subject of the next section.

3.1.3 Additional Resources

In recent architectures, additional resources are *hard-wired* within the FPGA. These physical components can be listed as the following:

- **Synchronous Random Access Memory (SRAM) blocks**: Ideal to store large blocks of data, the SRAMs provide up to three independent input and output ports, sharing the same memory space. In the case of Intel FPGAs, the company provides various sizes of embedded memory blocks according to the device. For instance, a Cyclone V FPGAs embed between 135 and 1220 *M10K* blocks, each delivering 10Kbits of memory.

[2]Image from http://fpgabeginners.blogspot.com/2012/08/what-is-fpga.html
[3]Within the same LAB

- **Distributed memory blocks**: In contrast with dedicated SRAM blocks which have a fixed position in the FPGA, distributed memory blocks can be implemented anywhere, and are created automatically from a set of logical resources. In the case of Intel FPGAs, these blocks –known as *MLABs*– are made up of ten ALMs, which can be configured as ten 32×2 blocks, resulting in one 32×10 equivalent dual port SRAM per MLAB. This type of memory is ideal for small memorizations (shift registers, small buffers). In Cyclone V devices, MLABs provide 640 bits of memory. In addition, it is possible to allocate up to 25% of the available logic as MLAB, providing up to 1.7Mbits of additional memory to the device.

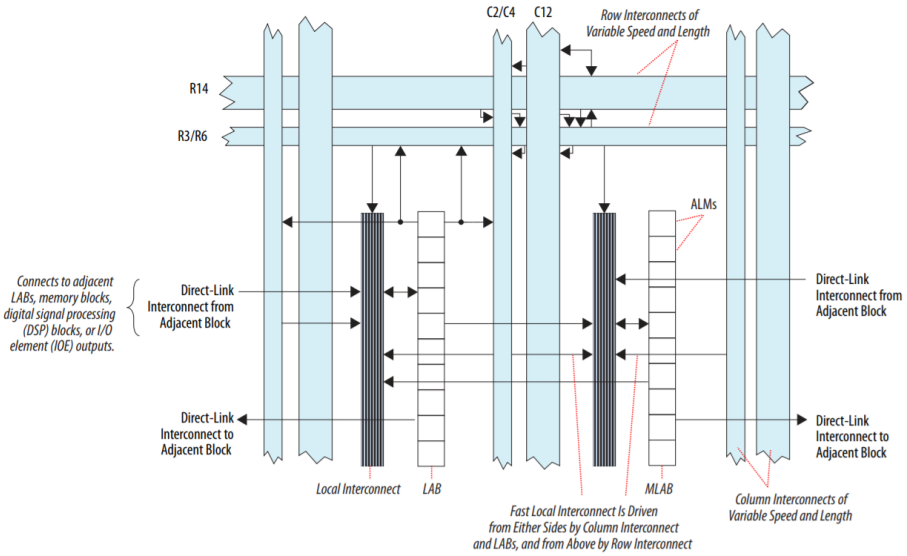
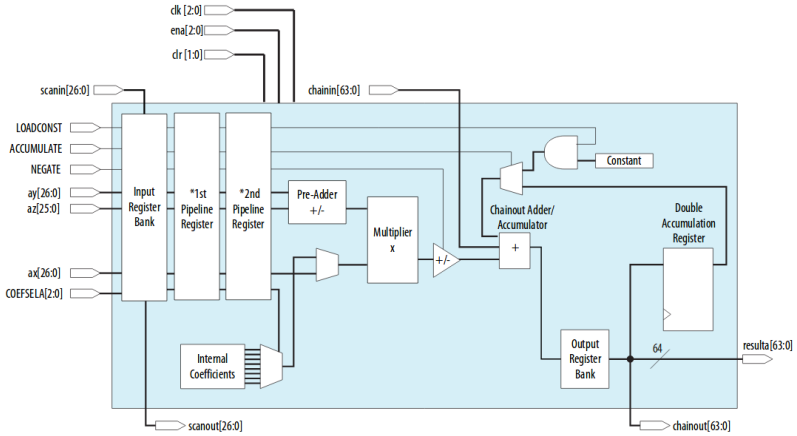


FIGURE 3.5: LAB Structure Overview in Cyclone V Devices. From [Int14a]

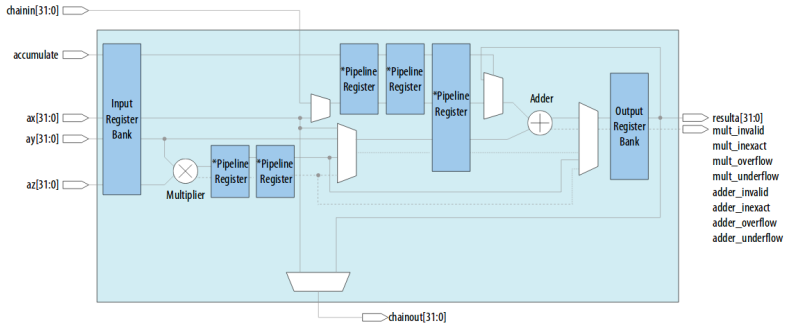
- **Digital Signal Processing (DSP) Blocks**: These hard-wired arithmetic units are optimized for computational throughput and power consumption. DSP blocks can also operate at different bit-widths and are capable to *pack* concurrent computations in a single unit. For instance, Intel FPGA DSP blocks can either implement:
  - One (27×27) bits multiplication (see Fig.3.6a)
  - Two independent (18×18) bits multiplications concurrently
  - Three independent (9×9) bits multiplications concurrently.

  Note that a number of current devices include DSPs that naively support simple precision *floating point arithmetic* (see Fig.3.6b). Thanks to DSP Blocks, high-end FPGAs such Xilinxs' Virtex Ultra-scale or Intels' Stratix 10 can peak at over 9.3 TFLOPS [Int17].

- **Hard Processor System (HPS)**: The HPS is a CPU that replaces the softcore-based co-design systems (NIOS, MicroBlaze) in a number of FPGA boards. Currently, FPGA manufacturers typically use ARM processors known for their low power consumption in embedded applications and their computation performance in datacenters. Note that Xilinx recently introduced new boards which embed ARM Mali GPUs. With this, FPGA-based systems move towards the era heterogeneous computing, which opens up many possibilities in terms of performance, flexibility, reconfigurability (hardcore can reconfigure the FPGA dynamically) and design productivity.



(A) DSP in (27×27) multiplier mode



(B) DSP in *FP*32 mode

FIGURE 3.6: Scheme of a DSP Block in a Stratix 10 FPGA. From [Int17]

3.2 From Algorithms to Hardware Architectures

3.2.1 Hardware Description Languages

In general, the algorithm to be mapped on an FPGA is described using a Hardware Description Language (HDL). HDLs, such VHSIC Hardware Description Language (VHDL) or Verilog, are specialized computer languages used to describe the structure and behaviour of digital logic circuits, and more particularly those implemented on ASICs and FPGAs.

The hardware description process is typically carried out using two different styles:

- The *behavioural* description in which the hardware is modeled based on its functionality. Thus style explicitly describes the desired *behaviour* of a given circuit.
- The *structural* description in which the digital circuit interconnects a number of building blocks (referred as components) implementing some basic behaviour. These building blocks are usually grouped in order to realize a more complex functionality.

3.2.2 FPGA Design Flow

The transcription of a source code described in an HDL into a hardware architecture follows the design flow depicted in Fig.3.7. The steps of this flow can be enumerated as:

1. Register Transfer Level (RTL) Synthesis is a process that translates the input HDL code into a RTL netlist (i.e a description of the circuit using elementary block such registers and LUTs). This phase first creates a database of all the components used

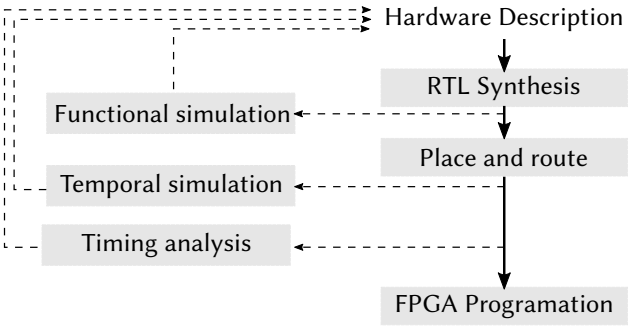


FIGURE 3.7: Design Flow in FPGA

in a given project, checking their validity (for instance the syntax and connections). Then, the synthesis tool extracts from the code macros or patterns implementing a specific behaviour (for instance a memory or a multiplication), which will be replaced by the FPGA manufacturer implementation of the functionality (for instance , the manufacturers' implementation of memorization and multiplication). FPGA manufacturers even advice to use some «coding style» to guide the synthesizer in extracting these macros. This step also optimizes the design, removing for instance the redundant or unused components.

After the synthesis is completed, it is possible to perform a functional simulation in order to check whether the described behaviour meets the expected functionalities.

2. The «*Place and route*» or *implementation* step is divided into two parts. The first one transforms the RTL representation into a logic gate level representation, taking into account the characteristics of the targeted FPGA. The second step places the resulting logic on the desired FPGA. At this stage, the post-routing simulation is possible. This step takes into consideration the characteristics of the FPGA and the placement made by the tool in order to verify that the propagation times in the FPGA do not interfere with not the proper functioning of the system. At the end of this process, a precise analysis of the time delay between the FPGA elements provides the user with a information on the maximum operating frequency of the design.

3. The *bit-stream* generation step transforms the previous representation into a configuration file (bitstream) which is used to program the connections of the targeted FPGA.

3.2.3 High Level Synthesis Tools

In FPGA designs, HDLs have the advantage of providing the best results in terms of implementation performance in terms of resources and computational throughput. In counter-part, they require a good knowledge of digital circuit design and hardware architectures [Ben10, PBMB17].

In the case of complex algorithms like deep learning, the transcription task becomes even more difficult, mainly because of the large number of computations (see sec.2.2.1) and the high variability of CNNs workloads (see sec.2.2.2). As a consequence, the transcription of complex algorithms to hardware description is a task requiring a considerable amount of development time, limiting the use of FPGAs. In response to these productivity problems, large research efforts are given towards the development of High-Level Synthesis (HLS) tools.

HLS tools are an alternative to conventional HDLs which intend to provide an easier access to FPGAs, especially for the « software programmers». This motivate a large part of the currently used HLS tools to be based on imperative languages such as C/C++,

offering familiar programming paradigms. In this context, two popular « C-like » HLS tools have been introduced: Vivado HLS, and OpenCL for FPGAs.

Vivado HLS [Xil13] is part of the Xilinx Vivado suite, and provides a programming environment similar to those available for software developers. The tool relies on *pre-compilation guidelines*, or *pragmas* to generate the RTL description of a design. Note that during the transcription process, the tool generates representations in SystemC, VHDL and Verilog, and that it is possible an HLS-designed component ca be generated separately and integrated it into a project.

OpenCL [Khr15] is an open-source framework for parallel programming on heterogeneous architectures. Programs written in OpenCL can be executed transparently on CPUs, GPUs and FPGAs. For FPGAs, OpenCL uses a master-slave model where an *OpenCL host device* controls the execution of multiple *OpenCL kernels*. On the kernel side, OpenCL abstracts away the complexities of HDL, allowing software programmers to write hardware-accelerated kernel functions in high level C/C++ code. On the host side, a host program controls and supervises the kernels using a predefined OpenCL API. Both Intel and Xilinx provide dedicated OpenCL APIs for their FPGAs with respectively the Intel SDK for OpenCL [CAD+12, Int16] and the SDaccel environment [GRT+16].

Note that OpenCL not only compiles a C code to an RTL description, but also manages the interfacing with the external memory and the communication between the host CPU and the FPGA kernel. This considerably reduces the design time, while achieving performance comparable to the traditional RTL flow, often at the price of resource utilization, especially on-chip memory [SCD+16].

The common feature between the two former tools is their inspiration from the imperative programming models. The problem is that these models, designed to run on processor-based architectures differ too much from the FPGA execution model. This difference prevents from efficiently switching from one to the other in an automatic fashion [Bou16]. One possible solution to address this problem is choosing a more adequate Model of Computation (MoC), particularly, the dataflow model.

3.3 Dataflow Model for FPGA-Based Image processing

When porting real-time vision applications on FPGA-powered platforms, the problem often boils down to finding an efficient mapping between the computational model of the formers and the execution model supported by the latters.

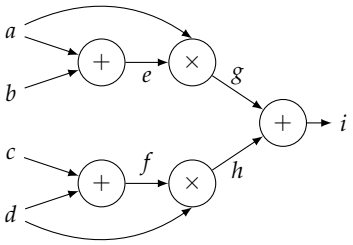
To address this mapping problem, numerous studies [BSB13, MBP+15, SBB16] advocate the use of the *dataflow Model of Computation (MoC)*. In this approach, a given algorithm is described as a graph of fundamental processing units exchanging data through unidirectional channels. As studied in [Bou16], a large variety of algorithms involved in computer vision, including CNNs, can be expressed as dataflow graphs, which significantly accelerates their execution on FPGA platforms at the price of an algorithmic reformulation effort.

3.3.1 The Dataflow Model

The foundations of the dataflow MoCs appeared in works of Sutherland *et al.* [Sut66], and were then formalized by Dennis *et al.* [DM75]. The objective of these works was to create an architecture where multiple fragments of instructions can simultaneously process streams of data. Dataflow architectures outperform conventional processor architectures in many applications which are limited by the classic bottleneck problem between the processor and the memory storing the program and data.

Programs respecting the dataflow semantics are described as DPNs. Each node of this network corresponds to a fundamental processing unit called an *actor* and each edge corresponds to a communication *FIFO* channel. Actors exchange abstract data –known as *tokens*– through these FIFOs.

The notion of time is implicit in dataflow programs, where only the concept of *causality* is relevant. Each actor follows a purely data-driven execution model wherein the *firing* (execution) is triggered only by the availability of input operands. The behaviour this actor is thus only defined by some *firing rules* that are explicitly defined by the programmer. These firing rules define the response of an actor (i.e the tokens it outputs) to a given combination of input tokens.



- step 1: $e = a + b$
- step 2: $f = c + d$
- step 3: $g = a * e$
- step 4: $h = d * f$
- step 5: $i = g + h$

FIGURE 3.8: Comparing imperative execution models and dataflow models

A comparison between the dataflow model and the conventional Von Neumann model is given in Figure 3.8 on a simple example; the computation of $i = (a + b) * a + (c + d) * d$. While the Von Neumann model executes this example in five steps, the dataflow model requires only three. Indeed, Steps 1 and 2 operate on independent data and can be performed in parallel, as well as steps 3 and 4.

3.3.2 High Level Synthesis of Dataflow Programs

The performance of dataflow architectures and their adequacy with FPGA hardware platforms motivated a number of efforts to create languages that aim at describing dataflow graphs, and generating the hardware architectures they correspond to. This is typically the case of the CAPH language, and its associated compiler.

Caph [SBB16] is a Domain Specific Language (DSL) for image processing that transcripts a given algorithm onto a digital hardware description that implements its behaviour. When compared to the HLS tools listed in sec.3.2.3, CAPH main feature is to generate a purely dataflow architecture from a DPN. This architecture naturally exploits the pipeline and parallelism exhibited by the image processing workload. Note that CAPH generates the corresponding hardware description as aVHDL code which can be implanted using an FPGA synthesis tool, as illustrated in the CAPH tool chain given in Fig.3.9.

LISTING 3.1: Caph Actors

```
actor add
 in (i1 :int<s,m> dc
     i2 :int<s,m> dc)
 out( o :int<s,m> dc)
rules
|i1:'x, i2:'y -> o:'(x+y);

actor mult
 in (i1 :int<s,m> dc
     i2 :int<s,m> dc)
 out( o :int<s,m> dc)
rules
|i1:'x, i2:'y -> o:'(x*y);
```

LISTING 3.2: Wiring Functions

```
net step1 (e,f) =
  add a b,
  add c d;

net step2 (g,h) =
  mult a e,
  mult b f;

net step3 (i) =
  add g h;
```

LISTING 3.3: (4×4) Image

```
00 01 02 03
10 11 12 13
20 21 22 23
30 31 32 33
```

To capture the concepts involved in dataflow MoC, the CAPH programming language relies on three formalisms:

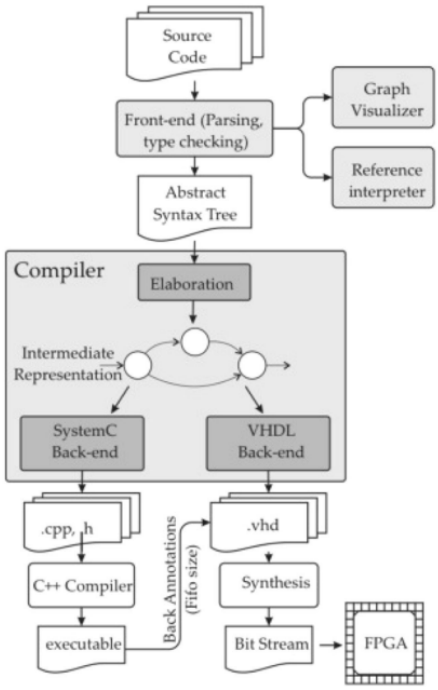


FIGURE 3.9: The CAPH tool chain Image from [SBB16]

- The first one describes the behaviour of fine-grain actors and relies and relies set of *transition rules*. These rules describe the modifications that should occur on outputs and/or local variable when receiving some values on the inputs or/and local variables. The choice of the rule to be fired is made by pattern matching, as shown in listing 3.1, which gives the CAPH implementation of addition and multiplication actors used in the example 3.8.

- The second formalism defines *wiring functions* that operate on graph edges. These are used to describe coarse-grain actors, or more generally, graph structures. In the previous example of Fig.3.8, the dataflow graph can be described using wiring functions given in listing 3.2.

- The third formalism unifies the programmers' vision of data and control flows through *structured data types*. In CAPH, tokens corresponding to actual data –for instance, pixels in a given image– and tokens referring to the structure of this data –for instance, the start of a line in the image– are uniformity represented as *tagged values*. Listing 3.3 illustrates this concept and shows how a (4×4) image that can be represented by the following stream of tokens:

  `< < 00 01 02 03 > < 10 11 12 13 > < 20 21 22 23 > < 30 31 32 33 > >`

  In this case, the red < and > tokens respectively represent the start and the end of a frame in the video stream, and the blue < and > tokens represent the start and the end of a line in an image. The advantage of this representation is that it allows the control part of the corresponding architecture to be implemented locally within the actors which exempts from synthesizing separate global controls.

To illustrate the concepts of stream processing, FPGA conception flow, and HLS tools, the following section studies the FPGA implementation of a (3×3) two dimensional convolution in VHDL and CAPH. In both cases, special interest is given to the implementation of parallelism and pipeline.

3.4 Implementation Example: Image convolution

Convolution is a recurrent operation in image processing applications (demosaicing, feature extraction). This is especially true for CNN applications, where convolution layers can be implemented as sums of 2D-convolutions, as pointed-out in sec.2.2.1.1.

This study considers a monochrome video stream `in_data` and a (3×3) convolution kernel `theta`. The input is *streamed* into a convolution block which functionality can be described by pseudo-code 3.4. In this snippet, the input data `in_data[h,w]` are acquired successively. Moreover, in sequential languages such as C/C++, the MAC operation involved is executed *sequentially* as the next iteration of the loops can only begin when the last operation in the current loop iteration is complete. As a consequence, listing 3.4 takes 9 clock cycles to output the result of one convolution.

However, in the considered implementation, we want the convolution block to operate *in the fly*, processing the data at the same rate it acquires them. To achieve this, the nested loops `loop_J` and `loop_K` have to pipelined in a way to output one result per clock cycle. This section shows how to implement this pipeline in VHDL and CAPH.

LISTING 3.4: Pseudo code of a (3×3) convolution

```
Loop_H: for (int h=0; h<H; h++){
 Loop_W: for (int w=0; w<W; w++){
  Loop_J: for (int j=0; j<3; j++){
   Loop_K: for (int k=0; k<3; k++){
    y[w][h] += x[h+j][w+k] * theta[j][k]
  }}}}
```

3.4.1 Dataflow Implementation with VHDL

As pointed-out in the last chapter, the MAC unit constitute the building block of the convolution operation. Listing 3.5 gives a VHDL implementation of a MAC unit, and more particularly the behavioural description implementing its functionality: the component operates on three inputs a, b, c, multiplies the two first and adds a third. Note that for clarity reasons, the bit-width is deliberately not expended after the accumulation process (see sec.4.4.1).

LISTING 3.5: Behavioural description of a MAC unit

```
entity CustomMAC is
  generic( BITWIDTH : integer := 4);
  port(
    clk     : in  std_logic ;                                          -- Clock signal
    reset_n : in  std_logic ;                                  -- Reset, active at low state
    enable  : in  std_logic ;                                -- Enable, active at high state
    a       : in  std_logic_vector (BITWIDTH-1 downto 0);           -- First Input operand
    b       : in  std_logic_vector (BITWIDTH-1 downto 0);          -- Second Input operand
    c_in    : in  std_logic_vector (BITWIDTH-1 downto 0);                 -- Input to add
    y       : out std_logic_vector (2*BITWIDTH-1 downto 0)                -- Output value
    );
end CustomMAC;

architecture bhv of CustomMAC is
 begin
   process(clk)
   begin
     if (reset_n = '0') then
       y <= (others => '0');

     elsif(rising_edge(clk)) then
       if (enable = '1') then
             y <= c_in + a * b;
       end if;
     end if;
   end process;
end architecture;
```

Figures 3.10 and 3.11[4] illustrate the netlists generated from listing 3.5 respectively after the synthesis and implementation steps.

[4] In both figures, the bit-width has been reduced to 4 bits for clarity reasons

During synthesis, the design is recognized as a multiplier-adder pair depicted in the generated RTL of Fig.3.10. Since the MAC unit is a very common element present in a large number of digital designs, a macro is associated to this operation in the synthesizer, leading the tool to instantiate the manufacturer implementation of the MAC. Note that one can directly instantiate the former component (for instance, the `altmult_add` in case of Intel FPGAs), though the first practice is generally preferred for code portability reasons.

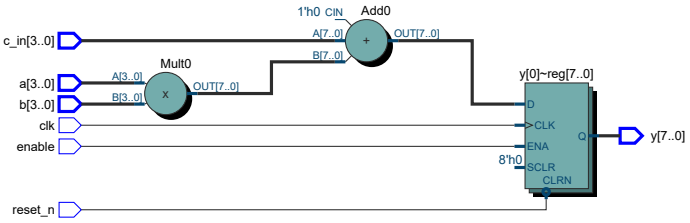


FIGURE 3.10: RTL description of the studied MAC unit

Then, during place and root, the system's inputs and outputs are automatically associated with FPGA I/O blocks (`IO_BUF` in Fig.3.11). Moreover, the multiplier-adder pair is automatically replaced by a hard DSP block (`Mult0~mac`). From this netlist, the tool infers one of the available DSP blocks and the corresponding I/O resources to implement the MAC.

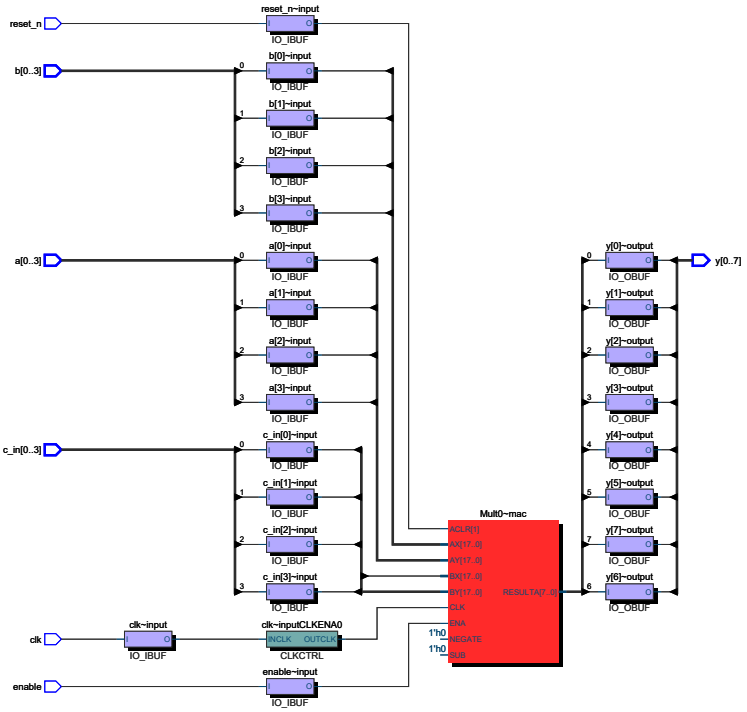


FIGURE 3.11: Post fitting representation of the studied MAC unit.

Again, the convolution algorithm benefits from significant acceleration when executed in a pipelined fashion. In VHDL, this pipeline can be implemented in two ways, as detailed in listing 3.6. The first method implicitly implements the pipeline through *behavioural* description. The second method relies on a *structural* approach and infers the previous `custom_mac` component multiple times, implementing the pipelined in an explicit fashion. In both cases, the synthesis tool chains a number of DSP blocks in a way to compute one result per clock cycle, as shown in figure 3.12.

LISTING 3.6: VHDL Implementation of Pipelined MAC Units

```
entity CustomMAC33 is
 generic(
  BITWIDTH     : integer := 8;
  KERNEL_SIZE   : integer := 3
  );
 port(
  clk     : in std_logic;                                                    -- Clock signal
  reset_n : in std_logic;                                  -- Reset, active at low state
  enable  : in std_logic;                                                    -- Enable
  x       : in pixel_array (0 to KERNEL_SIZE * KERNEL_SIZE - 1);  -- In data  (3*3*8 bits)
  theta   : in pixel_array (0 to KERNEL_SIZE * KERNEL_SIZE - 1); -- In kernel (3*3*8 bits)
  y       : out std_logic_vector (2*BITWIDTH-1 downto 0)             -- Out value (8 bits)
  );
end CustomMAC33;
--------------------------------------------------------------------------------------------
-- Behaviouaral Architecture :
architecture bhv of CustomMAC33 is
signal mac_out : prod_array (0 to KERNEL_SIZE*KERNEL_SIZE - 1) := (others=>(others=>'0'));
begin
 process(clk)
 begin
  if (reset_n = '0') then
   mac_out  <= (others => (others => '0'));
  elsif(rising_edge(clk)) then
   if (enable = '1') then
    mac_out(0)(2*BITWIDTH-1 downto 0) <= x(0) * theta(0);        -- Multiply wo/ additions
    pipelined_mac_loop : for i in 1 to KERNEL_SIZE * KERNEL_SIZE -1 loop
     mac_out(i) <= x(i) * theta(i) + mac_out(i-1)  -- Multiply than add to previous result
    end loop pipelined_mac_loop;
   end if;
  end if;
 end process;
 y <= mac_out(KERNEL_SIZE * KERNEL_SIZE-1);  -- Output is the result of the last MAC
end architecture;
--------------------------------------------------------------------------------------------
-- Structural Architecture : Instantiate the custom_mac component
architecture structural of conv33MAC is
 signal mac_out : prod_array (0 to KERNEL_SIZE*KERNEL_SIZE - 1) := (others => (others =>
      '0'));
begin
 pipelined_mac: for i in 0 to KERNEL_SIZE*KERNEL_SIZE - 1 generate
 first_mac: if i = 0 generate
  first_custom_mac_inst: custom_mac                              -- Multiply wo/ additions
  generic map( BITWIDTH=>BITWIDTH )
  port map(
   clk     => clk,
   reset_n => reset_n,
   enable  => enable,
   a       => x(i),
   b       => theta(i),
   c_in    => (others => '0'),
   y       => mac_out(i)
  );
 end generate first_mac;
 gen_mac: if i > 0 and i < KERNEL_SIZE * KERNEL_SIZE - 1 generate
  gen_custom_mac_inst: custom_mac               -- Multiply than add the previous result
  generic map( BITWIDTH=>BITWIDTH )
  port map(
   clk     => clk,
   reset_n => reset_n,
   enable  => enable,
   a       => x(i),
   b       => theta(i),
   c_in    => mac_out(i-1),
   y       => mac_out(i)
  );
 end generate gen_mac;
 last_mac: if i = KERNEL_SIZE * KERNEL_SIZE - 1 generate
  last_custom_mac_inst: custom_mac     -- Convolution Output is the result of the last MAC
  generic map( BITWIDTH=>BITWIDTH )
  port map(
   clk     => clk,
   reset_n => reset_n,
   enable  => enable,
   a       => x(i),
   b       => theta(i),
   c_in    => mac_out(i-1),
   y       => y
  );
 end generate last_mac;
 end generate pipelined_mac;
end architecture;
--------------------------------------------------------------------------------------------
```

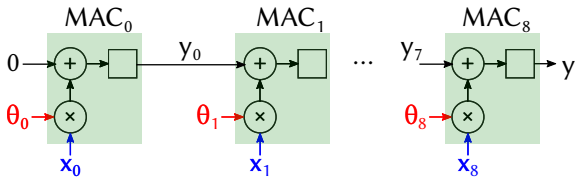


FIGURE 3.12: Scheme of a pipelined MAC for (3×3) convolutions

After implementing the pipelined MAC part, we detail how this block can be used to perform real-time image convolution. Indeed, the studied MAC block operates in a pipelined fashion but still requires all the necessary data to be made available on its inputs. Recall that (3×3) convolution operates on each (3×3) *neighbours* of a pixel, which have to be provided simultaneously. In the literature, a well known method to make this data available is the *window buffer* structure.

The window buffer, depicted in figure 3.13, relies on a number of shift registers to bufferize the input stream and extract the neighbours of each pixel. In FPGAs, these shift registers, also known as taps, can be implemented by means of distributed memory (by cascading the registers), or by means of SRAM memory blocks. In both cases, a total buffer size of $2 * W + 3$ is needed to memorize and delay the relevant pixels, where W is the width of the considered image.

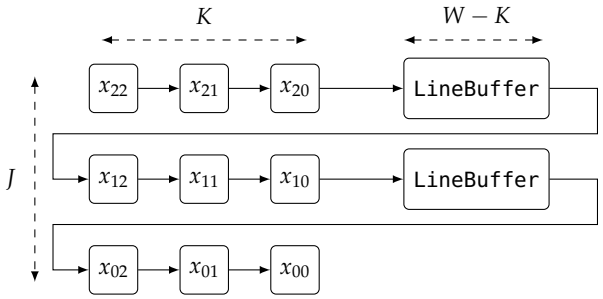


FIGURE 3.13: Structure of a (3×3) window buffer

In VHDL, the window buffer functionality can be implemented in a structural fashion by inferring the taps component three times. This is illustrated in listing 3.7 which describes the architecture of the window buffer.

Similarly to the MAC unit, the taps component is common in hardware designs and can be either implemented by describing its behaviour, or directly inferring the manufacturers' IP. In the case of Intel devices, both methods result in the inference of the `ALTSHIFT_TAPS` block.

Finally, the last conception step is to wire the `WindowBuffer33` and `CustomMAC33` components, resulting in the *top level* description of the convolution block given in 3.8. In this description, note the presence of a third component (`FlowController`) in which the structure of the output video stream is manually managed through in `out_dv` (data valid) and `out_fv` (frame valid) control signals.

LISTING 3.7: VHDL Implementation of the (3×3) Window Buffer

```
entity WindowBuffer33 is
 generic (
  IMAGE_WIDTH : integer := 320;
  DATA_WIDTH  : integer := 8
 );
  port (
  clk           : in std_logic;                                          -- Clock signal
  reset_n       : in std_logic;                                -- Reset, active at low state
  enable        : in std_logic;                                                -- Enable
  in_data       : in std_logic_vector(DATA_WIDTH-1 downto 0);    -- Input data array stream
  p00, p01, p02 : out std_logic_vector(DATA_WIDTH-1 downto 0);   -- Output 3x3 neighborhood
  p10, p11, p12 : out std_logic_vector(DATA_WIDTH-1 downto 0);
  p20, p21, p22 : out std_logic_vector(DATA_WIDTH-1 downto 0)
 );
end WindowBuffer33;

architecture structural of WindowBuffer33 is

 signal line0_pix_out  : std_logic_vector((DATA_WIDTH-1) downto 0);
 signal line1_pix_out  : std_logic_vector((DATA_WIDTH-1) downto 0);

 begin
 Taps1 : CustomTaps                                          -- First taps : Delay by a line
 generic map ( LENGTH => IMAGE_WIDTH_MAX-1, WIDTH => DATA_WIDTH)
 port map (
  clk      => clk,
  reset_n  => reset_s,
  enable   => all_valid,
  in_data  => in_data,
  out_data => line0_pix_out,
  i0 => p22 , i1 => p21, i2 => p20
 );

 Taps2 : CustomTaps                                        -- Second taps : Delay by a line
 generic map ( LENGTH => IMAGE_WIDTH_MAX-1, WIDTH => DATA_WIDTH)
 port map (
  clk       => clk,
  reset_n   => reset_s,
  enable    => all_valid,
  in_data   => line0_pix_out,
  out_data  => line1_pix_out,
  i0 => p12 , i1 => p11 , i2 => p10
 );

 Taps3 : CustomTaps                                      -- Third taps : Delay by 3 cycles
 generic map ( LENGTH => 3, WIDTH => DATA_WIDTH )
 port map (
  clk     => clk,
  reset_n => reset_n,
  enable  => enable,
  in_data => line1_pix_out,
  i0 => p02 , i1 => p01, i2 => p00
 );
 end structural;
```

LISTING 3.8: VHDL Implementation of a dataflow (3×3) Convolution block

```
entity Conv33 is
  generic (
    IMAGE_WIDTH : integer := 512;
    DATA_WIDTH  : integer := 8
    );
  port (
    clk           : in  std_logic;                                        -- Clock Signal
    reset_n       : in  std_logic;                              -- Reset, active at low state
    enable        : in  std_logic;                                        -- Enable
    in_dv         : in  std_logic;                                --Input frame is valid
    in_fv         : in  std_logic;                                -- Input data is valid
    in_data       : in  std_logic_vector(DATA_WIDTH-1 downto 0);        -- Input Pixel Value
    k00, k01, k02 : in  std_logic_vector(DATA_WIDTH-1 downto 0);       -- Convolution Kernel
    k10, k11, k12 : in  std_logic_vector(DATA_WIDTH-1 downto 0);
    k20, k21, k22 : in  std_logic_vector(DATA_WIDTH-1 downto 0);
    out_data      : out std_logic_vector(DATA_WIDTH-1 downto 0);    -- Output frame is valid
    out_dv        : out std_logic ;                                  -- Output data is valid
    out_fv        : out std_logic                                      -- Output Pixel Value
    );
end Conv33;


architecture Structural of Conv33 is
begin
  ker33(0) <= k00;                            -- Cast convolution kernel in a 9 element array
  ker33(1) <= k01;
  ker33(2) <= k02;
  ker33(3) <= k10;
  ker33(4) <= k11;
  ker33(5) <= k12;
  ker33(6) <= k20;
  ker33(7) <= k21;
  ker33(8) <= k22;

  FlowController_inst : FlowController           -- Instantiate the FlowController component
  generic map(
    IMAGE_WIDTH => IMAGE_WIDTH, DATA_WIDTH  => DATA_WIDTH
    )
  port map(
    clk             => clk,
    reset_n         => reset_n,
    enable          => enable,
    in_dv           => in_dv,
    in_fv           => in_fv,
    to_WindowBuffer => to_WindowBuffer,
    to_CustomMAC33  => to_CustomMAC33,
    out_dv          => out_dv,
    out_fv          => out_fv
    );

  WindowBuffer33_inst : WindowBuffer33                    -- Instantiate the Windows Buffer
    generic map(
      IMAGE_WIDTH => IMAGE_WIDTH, DATA_WIDTH  => DATA_WIDTH
      )
    port map(
      clk     => clk,
      reset_n => reset_n,
      enable  => to_WindowBuffer,
      in_data => in_data,
      p00     => neigh33(0), p01 => neigh33(1), p02 => neigh33(2),
      p10     => neigh33(3), p11 => neigh33(4), p12 => neigh33(5),
      p20     => neigh33(6), p21 => neigh33(7), p22 => neigh33(8)
      );

  CustomMAC33_inst : CustomMAC33                          -- Instantiate the Pipelined MAC
    generic map(
      BITWIDTH    => DATA_WIDTH, KERNEL_SIZE => 3
      )
    port map(
      clk                                 => clk,
      reset_n                             => reset_n,
      enable                              => to_CustomMAC33,
      x                                   => neigh33,
      theta                               => ker33,
      y(2*DATA_WIDTH -1 downto DATA_WIDTH) => out_data   -- Put MSBs at the output (no shift)
      );
end architecture;
```

3.4.2 Dataflow Implementation with Caph

After detailing the conception of a dataflow convolution block in VHDL, the following section shows how an HLS tool like CAPH in able to derive a similar architecture in a more productive fashion.

First, two elementary actors are used; the first one, `dp`, delays the input stream by one pixel while the second, `dl`, delays the stream by one line. These actors are very common in image processing tasks are thus already provided in the CAPH standard library.

Second, one can leverage on CAPH wiring functions to implement the window buffer: The `neigh13` function produces three wires representing the (1×3) neighbourhood of the input stream (generated by applying the `dp` actor) and the `neigh33` function produces nine wires representing the (3×3) neighbourhood. This is illustrated by Figure 3.14 where only two actors implements the (3×3) neighbourhood extraction, thanks to the wiring functions.

```
net neigh13(x) =
  x,
  dp x,
  dp (dp x);

net neigh33(x) =
  neigh13 x,
  neigh13 (dl x),
  neigh13 (dl (dl x));

net
 (o11,o12,o13),
 (o21,o22,o23),
 (o31,o32,o33)
= neigh33(i);
```

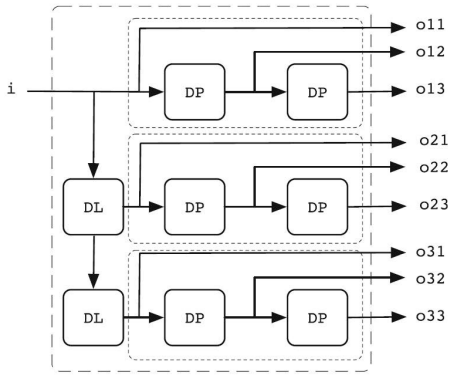


FIGURE 3.14: (3×3) Window buffer with CAPH Wiring Functions. From [SBB16]

After extracting the (3×3) neighborhood, the `mac33` actor operates on the 9 inputs and computes their dot product with the considered convolution kernel. Listing 3.9 gives the CAPH implementation of this actor[5]. Note how the pipeline is implicitly expressed line 22, thanks to which, the MAC is processed in a single clock cycle.

In the same listing, the last line simply wires the `mac33` and `neigh33` actors to implement convolution. Unlike the HDL approach, the structure of the output stream is managed when describing the actors, thanks to structured data types. As a consequence, the user is exempted from manually managing the output structure through dedicated blocks and signals, as it is the case in listing 3.8.

Finally, as a last step, the hardware description (in VHDL) is generated using the CAPH compiler. This code is then synthesized and implanted on the target FPGA using the tool chain detailed in section 3.2.2.

[5]In this listing, the convolution kernel is deliberately declared as input stream and not an input parameter. In fact, the second option specializes the convolution engine to the convolution kernel, advantaging the specialized CAPH implementation over the generic VHDL implementation

LISTING 3.9: Caph description of a pipelined MAC actor.

```
-- 2D 3x3 convolution - with explicit neighborhood generation
#include "neigh.cph"
actor mac33
  in (
      -- Input 3x3 neighborhood
      x0:signed<8> dc, x1:signed<8> dc, x2:signed<8> dc,
      x3:signed<8> dc, x4:signed<8> dc, x5:signed<8> dc,
      x6:signed<8> dc, x7:signed<8> dc, x8:signed<8> dc,
      -- Convolution Kernel
      k0:signed<8>   , k1:signed<8>   , k2:signed<8>,
      k3:signed<8>   , k4:signed<8>   , k5:signed<8>,
      k6:signed<8>   , k7:signed<8>   , k8:signed<8>
      )
  out (s:signed<8> dc)
rules
| (x0: '<, x1: '<, x2: '<, x3: '<, x4: '<, x5: '<, x6: '<, x7: '<, x8: '<) -> s:'<
| (x0: '>, x1: '>, x2: '>, x3: '>, x4: '>, x5: '>, x6: '>, x7: '>, x8: '>) -> s:'>
| (x0:'x0, x1:'x1, x2:'x2, x3:'x3, x4:'x4, x5:'x5, x6:'x6, x7:'x7, x8:'x8,
   k0: k0, k1: k1, k2: k2, k3: k3, k4: k4, k5: k5, k6: k6, k7: k7, k8: k8) ->
   s:'((k0*x0 + k1*x1 + k2*x2 + k3*x3 + k4*x4 + k5*x5 + k6*x6 + k7*k7 +k8*x8)>>8);

net (o0, o1, o2, o3, o4, o5, o6, o7, o8) = neigh233 0 x;
net r = mac33 (o0, o1, o2, o3, o4, o5, o6, o7, o8,
               k0, k1, k2, k3, k4, k5, k6, k7, k8);
```

After detailing HDL-based and CAPH based implementations, the last part of our study compares the performance of generated architectures.

3.4.3 Implementation Results

Table 3.1 compares the two implementations of convolution in terms of operating frequency and resource utilization. Both implementations consider a variable kernel operating on a 512 x 512 image, and a data width of 8 bits. The target is a low budget Intel Cyclone V `5CSEA7` FPGA, and the synthesis tool is Quartus 18.1. Code is made available online[6].

TABLE 3.1: Resource utilization and Operating frequencies of the convolution blocks

Resource	ALM	REG	DSP	M10K	MemBits	Fmax (MHz)
VHDL	95	195	5	3	10180	78.36
CAPH	1029	2059	9	11	12982	63.58

Globally, the HLS-derived architecture involves more resources, and runs slightly slower (13%) than the HDL based implementation.

In terms of logic fabric, our CAPH implementation of convolutions requires 10× more ALMs. This is due to the fact that CAPH infers inter-actor FIFOs, which, in this case, are implemented by means of registers. In turn, these registers result in the inference of ALMs. Note that this logic utilization can be significantly reduced when using the mono-actor CAPH formulation of convolutions, as detailed in [Bou16].

In terms of on-chip memory, both implementations have nearly the same requirements around 10Kbits within 20% of each other. However, there is a major difference in terms of SRAM blocks inferred:

- The VHDL implementation instantiate three M10K blocks: one for the window buffer, and two for the flow controller.
- The CAPH implementation instantiate eleven: three to implement the neighborhood extraction, and eight to implement the deep FIFOs between actors.

Finally, a substantial difference in the number of DSP blocks can be noticed; Even if the considered (3×3) convolution involve nine MAC units, the synthesizer infers five

[6] https://github.com/KamelAbdelouahab/conv33

DSPs blocks for the VHDL implementation and nine DSPs for CAPH implementation.

This is a direct result of the DSP *packing* capabilities detailed in sec.3.1.3. Thanks to DSP packing, the synthesizer is able to *pack* two MAC operations in a single DSP block when operating on 8 bits operands, as it is the case in the VHDL implementation. However, in the case of the Caph-generated architecture, a tag of two bits is inserted on each structured data to differentiate between control and data tokens. This tag expands the bit-width of the image operands by two bits, resulting in 10 bits operands for which DSP packing is unfeasible.

This can be verified in Fig.3.15, which reports the post-fitting views of the generated MAC units. On the left, note how the DSP block is inferred in « MAC mode », and how it is able to simultaneously input *four* operands of 8 bits in the case of the VHDL implementation. On the right, the DPS block is inferred as a multiplier, and is only able to input two operands of 10 and 8 bits in the case of the implementation with CAPH .

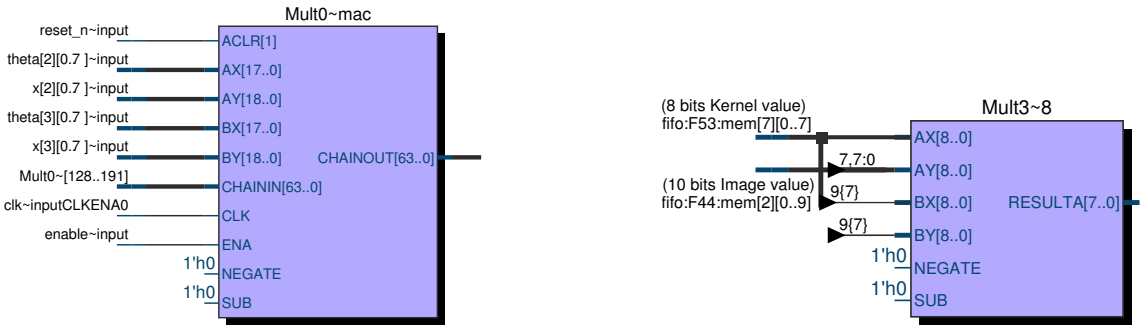


FIGURE 3.15: Post fitting view of the generated MAC unit reported by the Quartus Tool. DSP Packing is unfeasible for the figure on the right

These results corroborate our expectation: HLS greatly impacts productivity at the price of an overhead in resource utilization. Again, this overhead can be significantly reduced when reformulating the dataflow description [Bou16].

To conclude, we highlight that both implementations deliver real-time performance on an entry-level FPGA, exposing the benefits of the dataflow paradigm in the implementation of image processing tasks.

3.5 Conclusions

As demonstrated in the previous example, and more generally in this chapter, FPGAs support massive data parallelism, offering large opportunities in order to match with the real-time constraints of image processing.

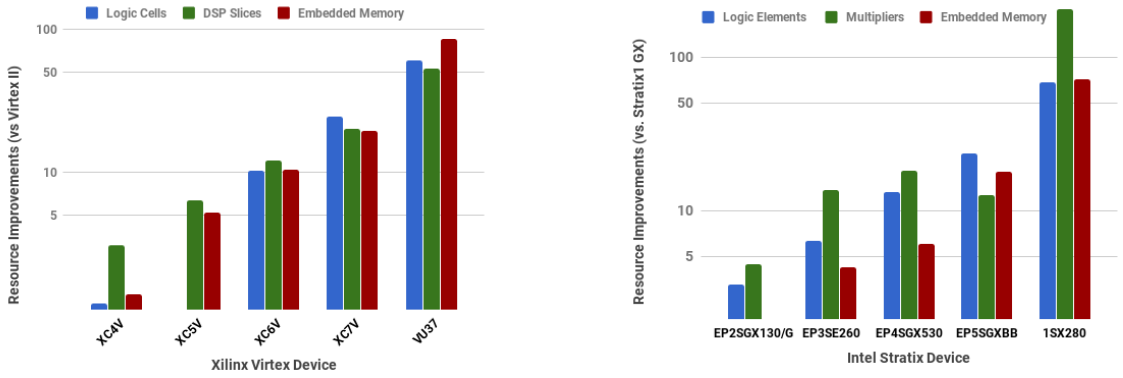


FIGURE 3.16: Evolution of resources in Xilinx and Intel FPGAs

The same advantages apply for computer vision with deep learning; While GPU implementations have demonstrated state-of-the-art computational performance, CNN acceleration is shortly moving towards FPGAs thanks to three main factors:

- First, the recent improvements in FPGA technology put reconfigurable hardware performance within striking distance to GPUs. Thanks to the increasing amount of logic, memory and computational resources (see.3.16), FPGA can now peak at 9.2 TFLOP/s [Int18b].
- Second, recent trends in CNN development increase the sparsity of CNNs and use extreme compact data types. These trends, studied in the next chapter, favourize FPGA devices which are designed to handle irregular parallelism and custom data types. As a result, next generation CNN accelerators are expected to deliver up to x5.4 better computational throughput than GPUs [NSB+17].
- Finally, the maturation of HLS tools makes FPGA development more productive, especially when designing complex applications like CNNs.

The amount and diversity of research on the subject of CNN FPGA acceleration within the last 3 years demonstrates a tremendous industrial and academic interest. As an inflection point in the development of CNN accelerators might be near, the next chapter presents a state-of-the-art of CNN inference accelerators over FPGAs.

Chapter 4

FPGA-Based Deep Learning Acceleration

As detailed in the last chapter, CNNs can benefit from a significant acceleration when running on reconfigurable hardware. This causes numerous research efforts to study FPGA-Based CNN acceleration, targeting both High Performance Computing (HPC) applications [ORK+15] and embedded devices [QWY+16].

In this chapter, we conduct a survey on methods and hardware architectures to accelerate the execution of CNNs on FPGAs. The first section lists the evaluation metrics used, then sections 4.2 and 4.3 respectively studies the computational transforms and the data-path optimization involved in recent CNN accelerators. Finally, the last section of this chapter details how approximate computing is a key in FPGA-based Deep Learning, and overviews the main contributions implementing these techniques.

4.1 Evaluation Metrics

Accelerating a CNN on an FPGA-powered platform can be seen as an optimization effort which focuses on one, or several of the following criteria:

- *Computational Throughput* ($\mathcal{T}$) : A large number of the works studied in this chapter focus on reducing the CNN execution times on the FPGA (i.e the computation latency), by improving the computational throughput of the accelerator. This throughput is usually expressed as the number of MACs an accelerator performs per second (MACS). While this metric is relevant in the case of HPC workloads, we prefer to report the throughput as the number of frames an accelerator processes per second (FPS), which suits more to the embedded vision context. The two metrics can be directly related using equation 4.1, where $\mathcal{C}$ is defined in equation 2.15, and refers to the number of computations a CNN involve in order to process a single frame:

$$\mathcal{T}_{(FPS)} = \frac{\mathcal{T}_{(MACS)}}{\mathcal{C}_{(MAC)}} \tag{4.1}$$

- *Classification/Detection Perf.* ($\mathcal{A}$): Another way to reduce CNN execution times is to trade some of their modeling performance in favour of faster execution timings. For this reason, the classification and detection metrics are reported, especially when dealing with *approximate computing* methods. As studied in sec.2.3, classification performance is usually reported as top-1 and top-5 accuracies and detection performance is reported using the mAP50 and mAP75 metrics.

- *Energy and Power Consumption* ($\mathcal{P}$): Numerous FPGA-Based acceleration methods can be categorised as either latency-driven or energy-driven. While the former focus on improving the computational throughput, the latter considers the power

consumption of the accelerator, reported in Watts. Alternatively, numerous latency-driven accelerators can be ported to low-power-range FPGAs and perform well under strict power consumption requirements.

- *Resource Utilization* ($\mathcal{R}$): When it comes to FPGA acceleration, the utilization of the available resources (LUTs, DSP blocks, SRAM blocks) is always considered. Note that the resource utilization can be correlated to the power consumption, but improving the ratio between the two is a technological problem that clearly exceeds the scope of this thesis. For this reason, both power consumption AND resources utilization metrics will be reported when available.

In the context of embedded vision, an FPGA implementation of a CNN has to satisfy to the former requirements. In this perspective, the literature provides three main approaches to address the problem of FPGA-based deep learning. These approaches mainly consists of computational transforms, datapath optimizations and approximate computing techniques, as illustrated in the chart 4.1.

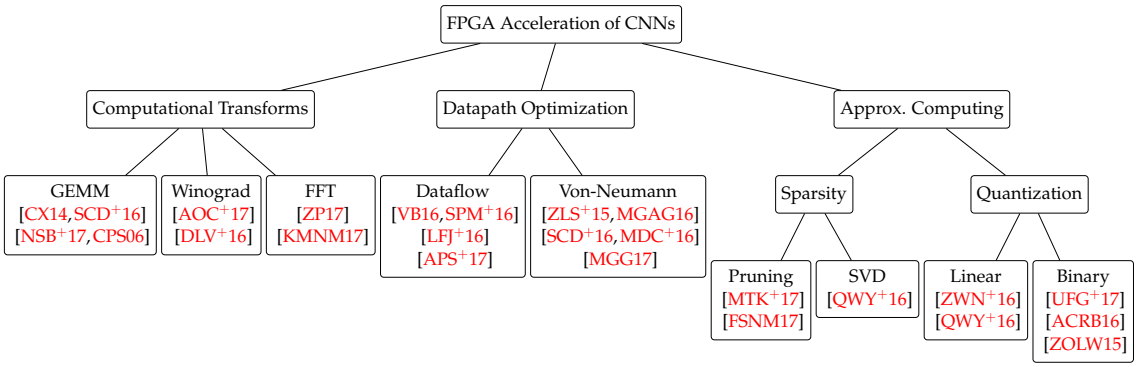


FIGURE 4.1: Main Approaches to accelerate CNN inference on FPGAs

4.2 Computational Transforms

In order to accelerate the execution of *conv* and FC layers, numerous implementations rely on computational transforms. These transforms operate on the FMs and weight arrays, and aim at vectorizing the implementations and reducing the number of operations occurring during inference.

Three main transforms can be distinguished. The *im2col* method reshapes the feature and weight arrays in a way to transform 3D-convolutions into matrix multiplications. The *FFT* method operates on the frequency domain, transforming convolutions into multiplications. Finally, in *Winograd* filtering, convolutions boil down to element-wise matrix multiplications thanks to a tiling and a linear transformation of data.

These computational transforms mainly appear in temporal architectures (see sec.2.5.1) and are implemented by means of variety of *linear algebra* libraries such OpenBLAS for CPUs[1] or cuBLAS for GPUs[2]. Beside this, various implementations make use of these transforms to efficiently map CNNs on FPGAs.

This section discusses the three former methods, highlighting their use-cases and computational improvements. For a better understanding, we recall that for each layer ℓ :

- The input feature map are represented as four-dimensional array $\boldsymbol{X}$ which the dimensions $B \times C \times H \times W$ respectively refer to the batch size, the number of input channels, the height, and the width.

[1] https://www.openblas.net/
[2] https://developer.nvidia.com/cublas

- The weights are represented as four-dimensional array $\mathbf{\Theta}$ which the dimensions $N \times C \times J \times K$ respectively refer to the depth of the output feature map, the depth of the input feature map, the vertical, and the horizontal kernel size.

4.2.1 The im2col Transformation

In CPUs and GPUs, a common way to process CNNs is to map *conv* and FC layers as General Matrix Multiplications (GEMMs). A number of studies generalize this approach to FPGA-based implementations.

For FC layers, in which the processing boils down to a matrix-vector multiplication problem, the GEMM-based implementations find its interest when processing a *batch* of FMs. As mentioned in section 2.4.1, most of the weights of CNNs are employed in the FC parts. Instead of loading these weights multiple times to classify multiple inputs, features extracted from a batch of inputs are concatenated onto a $CHW \times B$ matrix. In this case, the weights are loaded only one time per batch, as depicted in fig 4.2a. As a consequence, the former equation 2.23 –which expressed the number of memory accesses occurring on FC layers– becomes:

$$\mathcal{M}_\ell^{fc} = \text{MemRd}(\boldsymbol{\theta}_\ell^{fc}) + \text{MemRd}(\boldsymbol{X}_\ell^{fc}) + \text{MemWr}(\boldsymbol{Y}_\ell^{fc}) \tag{4.2}$$

$$= N_\ell C_\ell W_\ell H_\ell + B C_\ell H_\ell W_\ell + B N_\ell \tag{4.3}$$

$$\sim N_\ell C_\ell H_\ell W_\ell \tag{4.4}$$

As detailed in sec.2.4.2, the vectorization of FC layers is often employed in GPU implementations to increase the computational throughput while maintaining a constant memory bandwidth utilization. The same concept holds true for FPGA implementations [ZWS+16,ZFZ+16,AOC+17], which « batch » the FC layers to map them as GEMMs.

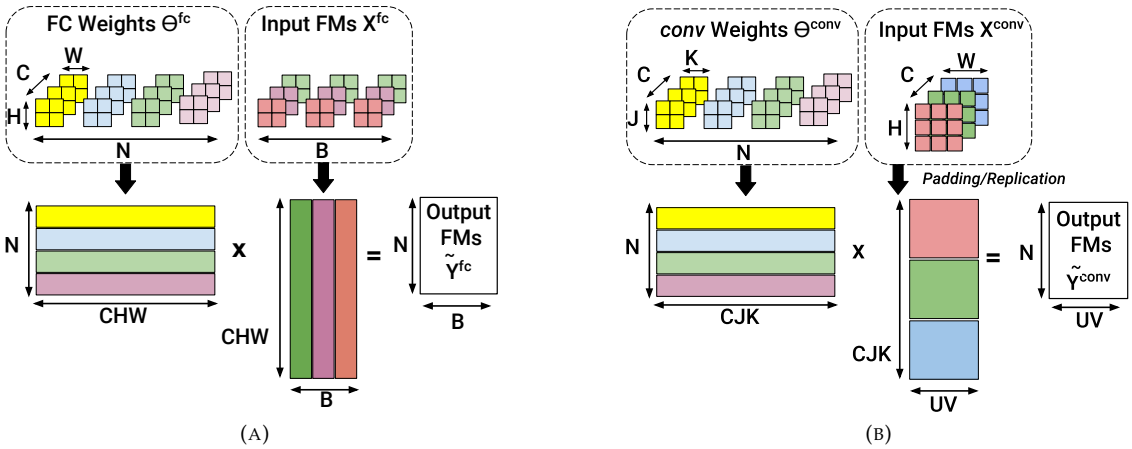


FIGURE 4.2: GEMM Based processing of: a- FC layers, b- conv layers.

3D Convolutions can also be mapped as GEMMs using the so-called *im2col* method introduced in [CPS06]. First, this method flattens all the weights of a given *conv* layer onto an $N \times CKJ$ matrix $\tilde{\Theta}$. Second, it re-arranges the input feature maps onto a $CKJ \times UV$ matrix $\tilde{X}$, squashing each Feature map to a column[3]. With these reshaped data, the output feature maps $\tilde{Y}$ are computed by multiplying of two former matrices, as illustrated in Fig 4.2b.

$$\tilde{\boldsymbol{Y}}^{\text{conv}} = \tilde{\boldsymbol{\Theta}}^{\text{conv}} \times \tilde{\boldsymbol{X}}^{\text{conv}} \tag{4.5}$$

[3]That's what the im2col name refers to: flattenning an image to a column

Suda et *al.* [SCD+16] and more recently, Zhang et *al.* [ZL17] leverage on *im2col* to derive OpenCL-based FPGA Accelerators for CNNs. However, this method introduces redundant data in the input FMs matrix which can lead to either inefficiency in storage or complex memory access patterns. As a result, and as pointed-out in [SCYE17], other strategies to map convolutions have to be considered.

4.2.2 Winograd Transform

Winograd minimal filtering algorithm, introduced in [Win80], is a computational transform that can be applied to process convolutions with a stride of 1, which is very common in CNN topologies.

This algorithm is particularly efficient when processing small convolutions (where $K \leq 3$), as advocated in [LG15]. In this work, authors outperform the throughput of the conventional *im2col* method by a factor of $\times 7.2$ when executing VGG16 on a TitanX GPU.

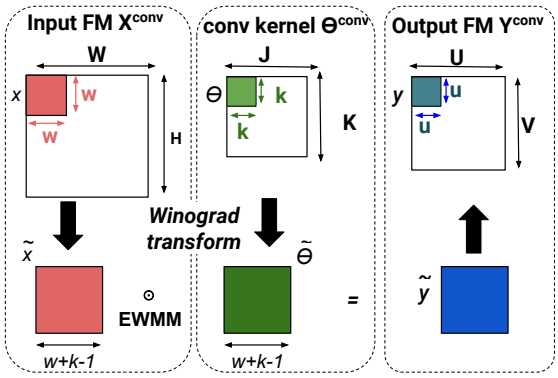


FIGURE 4.3: Winograd Filtering $F(u \times u, k \times k)$

In Winograd filtering, data is processed by blocs, referred as *tiles*, as following:

1. An input FM tile x of size $(u \times u)$ is pre-processed: $\tilde{x} = \boldsymbol{A}^T x \boldsymbol{A}$
2. In a same way, θ the filter tile of size $(k \times k)$ is transformed into $\tilde{\theta}$: $\tilde{\theta} = \boldsymbol{B}^T x \boldsymbol{B}$
3. Winograd filtering algorithm, denoted $F(u \times u, k \times k)$, outputs a tile y of size $(u \times u)$ that is computed according to equation 4.6

$$y = \boldsymbol{C}^T \left[\tilde{\theta} \odot \tilde{x}\right] \boldsymbol{C} \tag{4.6}$$

where $\boldsymbol{A}, \boldsymbol{B}, \boldsymbol{C}$ are transformation matrices defined in the Winograd algorithm [Win80] and $\odot$ denotes the Hadamard product also known as EWMM.

While a standard filtering requires $u^2 \times k^2$ multiplications, Winograd algorithm, denoted $F(u \times u, k \times k)$, requires $(u + k - 1)^2$ multiplications [Win80]. In the case of tiles of a size $u = 2$ and kernels of size $k = 3$, this corresponds to an arithmetic complexity reduction of x2.25 [LG15], and in this case, transform matrices can be written as:

$$\boldsymbol{A}^T = \begin{bmatrix} 1 & 1 & 1 & 0 \\ 0 & 1 & -1 & -1 \end{bmatrix} \quad \boldsymbol{B}^T = \begin{bmatrix} 1 & 0 & -1 & 0 \\ 0 & 1 & 1 & 0 \\ 0 & -1 & 1 & 0 \\ 0 & 1 & 0 & -1 \end{bmatrix} \quad \boldsymbol{C} = \begin{bmatrix} 1 & 0 & 0 \\ 1/2 & 1/2 & 1/2 \\ 1/2 & -1/2 & 1/2 \\ 0 & 0 & 1 \end{bmatrix} \tag{4.7}$$

Beside this complexity reduction, implementing Winograd filtering in FPGA-Based CNN accelerators has two advantages. First, transformation matrices $\boldsymbol{A}, \boldsymbol{B}, \boldsymbol{C}$ can be evaluated off-line once u and k are determined. As a result, these transforms become multiplications with the constants that can be implemented by means of LUT and shift registers, as proposed in [LLXY17].

Second, Winograd filtering can employ the loop optimization techniques discussed in section 4.3.2 to vectorize the implementation. On one hand, the computational throughput is increased when *unrolling* the computation of the Element-Wise Matrix Multiplications (EWMMs) parts over multiple DSP blocs. On the other hand, memory bandwidth is optimized using loop *tiling* to determine the size FM tiles and filter buffers.

First utilization of Winograd filtering in FPGA-Based CNN accelerators is investigated in [DLV+16] and delivers a computational throughput of 46 GOPs when executing AlexNet convolution layers. This performance is significantly improved by a factor of x42 in [AOC+17] when optimizing the data-path to support Winograd convolutions (by employing loop unrolling and tiling strategies), and storing the intermediate FM in on-chip buffers (cf sec 4.2).

The same method is employed in [LLXY17] to derive a CNN accelerator on a Xilinx ZCU102 device that delivers a throughput of 2.94 TOPs on VGG convolutional layers. The reported throughput corresponds to half of the performance of a TitanX device, with x5.7 less power consumption [Nvi15][4].

4.2.3 Fast Fourier Transform

Fast Fourier Transform (FFT) is a well known algorithm to transform the 2D convolutions into EWMM in the frequency domain, as shown in equation 4.8:

$$\textbf{conv2D}(X[c], \Theta[n,c]) = \text{IFFT}\Big(\text{FFT}(X[c]) \odot \text{FFT}(\Theta[n,c])\Big) \tag{4.8}$$

Using FFT to process 2D convolutions reduces the complexity from $O(W^2 \times K^2)$ to $O(W^2 log_2(W))$, which is exploited to derive FPGA-based accelerators to and infer CNNs [KMNM17]. When compared to standard filtering and Winograd algorithm, FFT finds its interest in convolutions with large kernel size $(K > 5)$, as demonstrated in [LG15, BKA+16]. The computational complexity of FFT convolutions can be further reduced to $O(W log_2(K))$ using the overlap-and-add Method [HR16] that can be applied when the signal size is much larger than the filter size, which is typically the case in *conv* layers $(W >> K)$. Works in [ZP17] leverage on the overlap-and-add to implement frequency domain acceleration for *conv* layers on FPGA, which results in a computational throughput of 83 GOPs for AlexNet.

4.3 Data-path Optimizations

As highlighted in sec 2.4.2, the execution of CNNs exhibit numerous sources of parallelism. However, due to the resource limitation of FPGAs devices, it might be impossible to fully exploit all the concurrency patterns, especially with the sheer volume of operations involved in deep topologies. In other words, the execution of recent CNN models can not fully be «unrolled», sometimes, not even for a single *conv* layer.

To address this problem, the general approach, advocated in state-of-the-art implementations, is to map a limited number of PEs on the FPGA. These PEs are then reused by temporally iterating data through them.

4.3.1 Systolic Arrays

Early FPGA-based accelerators for CNNs implemented systolic arrays to accelerate the 2D filtering in convolutions layers [SJC+09, FPH+09, CSJC10, GJD+14]. As illustrated in

[4] Implementation in the TitanX GPU employs Winograd algorithm and 32 bits floating point arithmetic

TABLE 4.1: Accelerators employing computational transforms

Method	Entry	Network	Comp (GOP)	Params (M)	bit-width	Desc.	Device	Freq (MHz)	Through (GOPs)	Power (W)	LUT (K)	DSP	Memory (MB)
Winograd	[DLV+16]	AlexNet-C	1.3	2.3	Float 32	OpenCL	Virtex7 VX690T	200	46	-	505	3683	56.3
	[AOC+17]	AlexNet-C	1.3	2.3	Float16	OpenCL	Arria10 GX1150	303	1382	44.3	246	1576	49.7
	[LLXY17]	VGG16-C	30.7	14.7	Fixed 16	HLS	Zynq ZU9EG	200	3045	23.6	600	2520	32.8
	[LLXY17]	AlexNet-C	1.3	2.3	Fixed 16	HLS	Zynq ZU9EG	200	855	23.6	600	2520	32.8
FFT	[ZP17]	AlexNet-C	1.3	2.3	Float 32	-	Stratix5 QPI	200	83	13.2	201	224	4.0
	[ZP17]	VGG19-C	30.6	14.7	Float 32	-	Stratix5 QPI	200	123	13.2	201	224	4.0
GEMM	[SCD+16]	AlexNet-C	1.3	2.3	Fixed 16	OpenCL	Stratix5 GXA7	194	66	33.9	228	256	37.9
	[ZFZ+16]	VGG16-F	31.1	138.0	Fixed 16	HLS	Kintex KU060	200	365	25.0	150	1058	14.1
	[ZFZ+16]	VGG16-F	31.1	138.0	Fixed 16	HLS	Virtex7 VX960T	150	354	26.0	351	2833	22.5
	[ZL17]	VGG16-F	31.1	138.0	Fixed 16	OpenCL	Arria10 GX1150	370	866	41.7	437	1320	25.0
	[ZL17]	VGG16-F	31.1	138.0	Float 32	OpenCL	Arria10 GX1150	385	1790	37.5	-	2756	29.0

figure 4.4a, systolic arrays employ *a static collection* of PEs, typically arranged in a 2-dimensional grid. These PEs operate as a co-processor under the control of a central processing unit. The configuration of systolic arrays is *agnostic* to the CNN model, making them inefficient to process large scale networks for three following reasons.

First, the static collection of PEs can only support convolutions up to a given filter size K_m, where typical values of K_m ranges from 7 in [FPH+09] to 10 in [GJD+14]. Therefore, a convolution layer (ℓ) in which $K_\ell > K_m$ is not supported by the accelerator.

Second, systolic arrays suffer from under utilization when processing layers in which the kernel size K_ℓ is much smaller then K_m. This is for instance the case in [GJD+14], where the processing 3×3 convolutions uses only 9% of DSP Blocs while the processing of these layers can be further parallelized and thus accelerated.

Finally, PEs in systolic arrays do not usually include memory caches and have to fetch their inputs from a off-chip memory. As a result, the performance of systolic arrays can rapidly be bounded by memory bandwidth of the device.

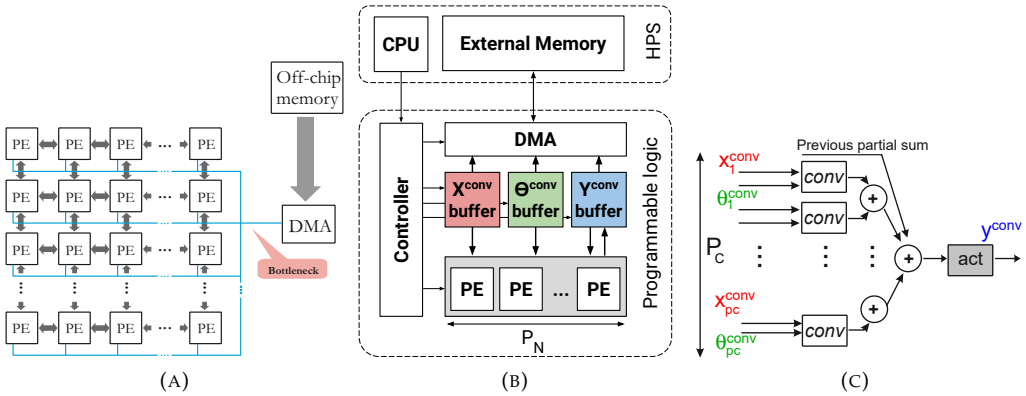


FIGURE 4.4: Generic Data-paths of FPGA-based CNN accelerators: A-Static Systolic Array. B-Dedicated SIMD Accelerator. C-Dedicated Processing Element

4.3.2 Loop Optimization in Spatial Architectures

Due to the inefficiency of systolic arrays, flexible and dedicated Spatial Architectures for CNNs were mapped on FPGAs. The general computation flow in these accelerators is illustrated in Fig.4.4b.

First, FMs and weights are fetched from DRAM to on-chip buffers, and are then *streamed* into the PEs. At the end of the PE computation, results are transferred back to on-chip buffers and, if necessary, to the external memory in order to be fetched in their turn to process the next layers. Each PE –as depicted in Fig. 4.4c– is configurable and has its own *computational* capabilities by means of DSP blocs, and its own data *caching* capabilities by means of on-chip registers.

With this paradigm, the problem of CNN mapping consists in finding the optimal architectural and temporal configuration of PEs. In other words, the best number of DSP blocs per PE, the optimal temporal scheduling of data that maximizes the computational throughput.

For convolution layers, in which the processing is described in listing 4.1, finding the optimal PE configuration comes down to a loop optimization problem [ZLS+15, QWY+16, SCD+16, MGAG16, ACFM16, MCVS17b].

LISTING 4.1: Nested Loops

```
// Lb : Batch
for (int b=0;b<B,l++){
// Ll: Layer
for (int l=0;l<L,l++){
// Ln: Y Depth
for (int n=0;n<N;n++){
// Lv: Y Columns
for (int v=0;v<V,v++){
// Lu: Y Raws
for (int u=0;u<U,u++){
// Lc: X Depth
for (int c=0;n<C;c++){
// Lj: Theta Columns
for (int j=0;j<J,j++){
// Lk: Theta Raws
for (int k=0;k<K,k++){
  Y[b,l,n,v,u] += X[b,l,c,v+j,u+k] *
       Theta[l,n,c,j,k]
}}}}}}}
```

LISTING 4.2: Loop Tiling in conv layers

```
for (int b=0;b<B,l++){
for (int n=0;n<N;n+=Tn){
for (int v=0;v<V,v+=Tv){
for (int u=0;u<U,u+=Tu){
for (int c=0;n<C;c+=Tc){
  // DRAM: Load in on-chip buffers the tiles:
  // X[l,c:c+Tc,v:v+Tv,u:u+Tu]
  // Theta [l,n:n+Tn,c:c+Tc,j,k]
  // Process on-chip tiles
  for (int tn=0;tn<Tn;tn++){
  for (int tv=0;tv<Tv,tv++){
  for (int tu=0;tu<Tu,tu++){
  for (int tc=0;tn<Tc;tc++){
  for (int j=0;j<J,j++){
  for (int k=0;k<K,k++){
    Y[l,tn,tv,tu] += X[l,tc,tv+j,tu+k] *
         Theta[l,tn,tc,j,k];
  }}}}}} // DRAM: Store output tile
}}}}
```

TABLE 4.2: Loop Optimization Parameters P_i and T_i

Parallelism	Intra layer	Inter FM	Intra FM		Inter conv.	Intra conv.	
Loop	L_L	L_N	L_V	L_U	L_C	L_J	L_K
Unroll factor	P_L	P_N	P_V	P_U	P_C	P_J	P_K
Tiling Factor	T_L	T_N	T_U	T_U	T_C	T_J	T_K

This problem is addressed by applying loop optimization techniques such *loop unrolling*, *loop tiling* or *loop interchange* to the 7 nested loops of listing 4.1. In this case, the unroll and tiling factors (*resp.* P_i and T_i) determine the number of PEs, the computational resources and on-chip memory allocated to each PE.

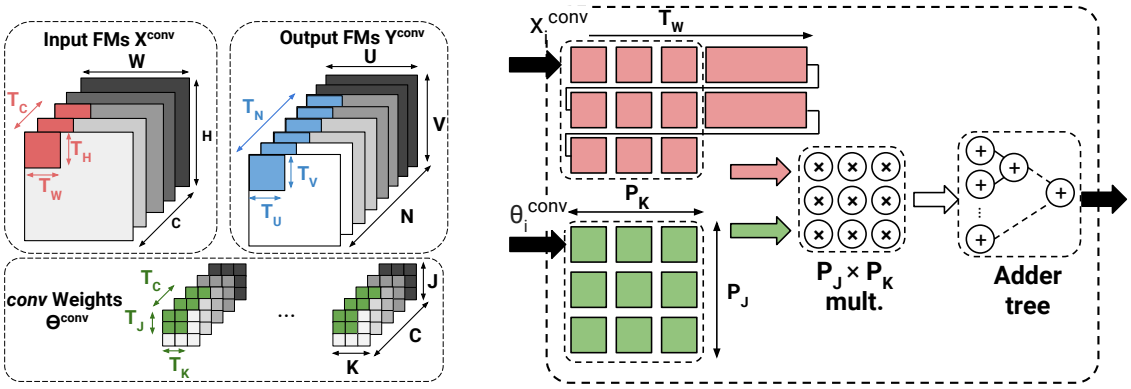


FIGURE 4.5: Loop tiling and unrolling in convolution layers

4.3.2.1 Loop Unrolling

Unrolling a loop L_i with an unrolling factor P_i ($P_i \leq i, i \in \{L, V, U, N, C, J, K\}$) accelerates its execution by allocating multiple computational resources. Each of the parallelism patterns listed in section 2.4.2 can be implemented by unrolling one of the loops of listing 4.1, as summarized in table 4.2. For the configuration given in figure 4.4c, the unrolling factor P_N sets the number of PEs. The remaining factors P_C, P_K, P_J determine the number of multipliers, as well as the size of buffer contained in each PE.

4.3.2.2 Loop Tiling

In general, the capacity of on-chip memory in current FPGAs is not large enough to store the weights and intermediate FMs of all CNN layers[5]. For example, AlexNets' convolution layers resort to 18.6 Mbits of weights, and generate a total 70.7 Mbits of intermediate feature maps[6]. In counter part, the « largest » Stratix V FPGA provides a maximum of 52 Mbits of on-chip RAM.

As a consequence, FPGA based accelerators resort to external DRAMs to store these data. As mentioned in section 2.4.3, DRAM accesses are costly in terms of energy and latency, and data caches must be implemented by means of on-chip buffers and local registers. The challenge is thus to build a data-path in a way that every data transferred from DRAM is reused as much as possible.

For *conv* layers, this challenge can be addressed by *tiling* the nested loops of listing 4.1. *Loop tiling* [DR01] divides the FMs and weights of each layer into multiple groups that can fit into the on-chip buffers. For the configuration given in figure 4.4c, the size of the buffers containing input FM, weights and output FM is set according to the tiling factors listed in table 4.2.

$$\mathcal{B}_X^{\text{conv}} = T_C \times T_H \times T_W \tag{4.9}$$
$$\mathcal{B}_\Theta^{\text{conv}} = T_N \times T_C \times T_J \times T_K \tag{4.10}$$
$$\mathcal{B}_Y^{\text{conv}} = T_N \times T_V \times T_U \tag{4.11}$$

With these buffers, the number of memory accesses occurring in *conv* layer (c.f eq.2.25) is respectively devided by $\mathcal{B}_X^{\text{conv}}$, B_Θ^{conv} and B_Y^{conv}, as expressed in equation 4.12.

$$\mathcal{M}_\ell^{conv} = \frac{C_\ell H_\ell W_\ell}{T_C T_H T_W} + \frac{N_\ell C_\ell J_\ell K_\ell}{T_N T_C T_J T_K} + \frac{N_\ell U_\ell V_\ell}{T_N T_V T_U} \tag{4.12}$$

Since the same hardware is reused to accelerate the execution of multiple conv layers with different workloads, the tiling factors are agnostic to the workload of a specific layer, as it can be noticed in the denominator of equation 4.12. As a result, the value of the tiling factors is generally set to optimize the overall performance of a CNN execution.

4.3.3 Design Space Exploration

Finding the optimal unrolling and tiling factors for a specific device is a complex problem that is generally solved using brute-force design space exploration [ZLS+15, SCD+16, AJK16, ZWS+16, MGAG16, MSC+16]. This exploration is driven by an analytical model, in which the inputs are loop factors P_i, T_i and outputs are a theoretical predictions of the computational throughput $(\mathcal{T})$, the size of buffers $(\mathcal{B})$ and the number of external memory accesses $(\mathcal{M})$. This model is parametrized by the available resources of a given FPGA platform and the workload of the considered CNN.

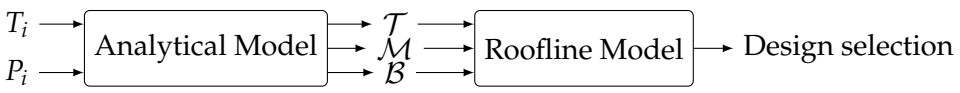


FIGURE 4.6: Design Space Exploration Methodology

[5]Exception can be made for [Mic17], where a large cluster of FPGAs is interconnected and resorts only to on-chip memory to store CNN weights and intermediate data

[6]Estimated by summing the number of outputs for each convolution layer

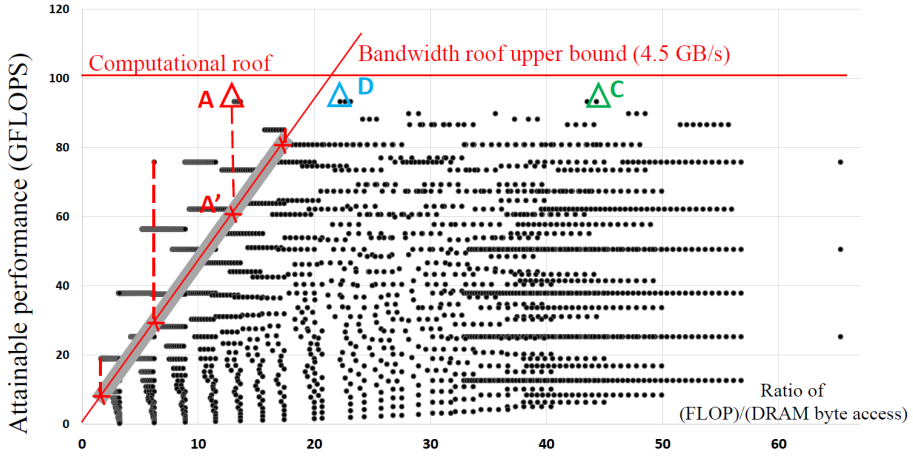


FIGURE 4.7: Design selection using the Roofline Model. Figure from [ZLS+15]

To select the feasible solutions of this optimization problem, most of the literature approaches rely on the *Roofline* method [WWP09] to accept or reject the design solutions that do not match with the maximum computational throughput or the maximum memory bandwidth of a given device.

A typical design space exploration driven by the roofline model is illustrated in figure 4.7. In this graph, each point of represents the performance of an explored solution (P_i, T_i). For a given FPGA platform, the attainable bandwidth and computational throughput are respectively reported by the diagonal and horizonal lines. Point A is an invalid solution as it is above the bandwidth roof while point A' is feasible but delivers mediocre computational throughput. Acceptable solutions are represented by points C and D, the latter being better than the former since it has lower bandwidth requirements.

4.3.4 FPGA Implementations

Employing loop optimizations to derive FPGA-based CNN accelerator was first investigated in [ZLS+15]. In this work, Zhang *et al.* report a computational throughput of 61.62 GOPs in the execution of AlexNet convolutional layers by unrolling loops L_C and L_N. This accelerator, described with Vivado HLS tools, relies on 32-bits floating point arithmetic. Works in [MSC+16] follow the same unrolling scheme and features a 16-bits fixed point arithmetic and , resulting in a x2.2 improvement in terms of computational throughput. Finally, the same unrolling and tiling scheme are employed in recent works [ZWS+16] were authors report a x13,4 improvement, thanks to a deeply pipelined FPGA cluster of four Virtex7-XV960t devices.

In all these implementations, loops L_J and L_K are not unrolled because J and K are usually small, especially in recent topologies. Works of Motamedi *et al.* [MGAG16] study the impact of unrolling these loops in AlexNet, where the first convolution layers use large 11×11 and 5×5 filters. Expanding loop unrolling and tiling to loops L_J and L_K results in a x1.36 improvement in computational throughput *vs* [ZLS+15] on the same VX485T device when using 32 floating point arithmetic. Nevertheless, and as pointed out in [MCVS17b], unrolling these loops is ineffective for recent CNN models that employ small convolution kernels.

The values of U, V, N can be very large in CNN models. Consequently, unrolling and tiling loops L_U, L_V, L_N can be efficient only for devices with high computational capabilities (i.e DSP Blocs). This is demonstrated in works of Rahman *et al.* [AJK16] that report

an improvement of $\times$1.22 over [ZLS+15] when enlarging the design space exploration to loops L_U, L_V, L_N, which comes at the price of very long exploration timing.

In order to keep data in on-chip buffer after the execution of a given layer, works of Alwani *et al.* [ACFM16] advocate the use of *fused-layer* accelerators by tiling across layer L_L. As a result, authors are able to remove 95% of DRAM accesses at the cost of 362KB of extra on-chip memory.

In all these approaches, loops L_N, L_C, L_J, L_K are unrolled in a same way they are tilled (i.e $T_i = P_i$). By contrast, the works of Ma *et al.* [MCVS17b, MKC+17] fully explore all the design variables searching for optimal loop unroll and tiling factors. More particularly, authors demonstrate that the input FMs and weights are optimally reused when unrolling only computations within a single input FM (i.e when $P_C = P_J = P_k = 1$). Tiling factors are set in way that all the data required to compute an element of Y are fully buffered (i.e $T_C = C, T_K = K, T_J = J$). The remaining design parameters are derived after a brute force design exploration. The same authors leverage on these loop optimizations to build an RTL compiler for CNNs in [MCVS17a]. To the best of our knowledge, this accelerator outperforms all the previous implementations that are based on loop optimization in terms of computational throughput.

TABLE 4.3: Accelerators employing loop optimization

Entry	Network	Comp (GOP)	Params (M)	bit-width	Desc.	Device	Freq (MHz)	Through (GOPs)	Power (W)	LUT (K)	DSP	Memory (MB)
[ZLS+15]	AlexNet-C	1.3	2.3	Float 32	HLS	Virtex7 VX485T	100	61.62	18.61	186	2240	18.4
[QWY+16]	VGG16SVD-F	30.8	50.2	Fixed 16	RTL	Zynq Z7045	150	136.97	9.63	183	780	17.5
[SCD+16]	AlexNet-C	1.3	2.3	Fixed 16	OpenCL	Stratix5 GSD8	120	187.24	33.93	138	635	18.2
[SCD+16]	AlexNet-F	1.4	61.0	Fixed 16	OpenCL	Stratix5 GSD8	120	71.64	33.93	272	752	30.1
[SCD+16]	VGG16-F	31.1	138.0	Fixed 16	OpenCL	Stratix5 GSD8	120	117.9	33.93	524	1963	51.4
[AJK16]	AlexNet-C	1.3	2.3	Float 32	HLS	Virtex7 VX485T	100	75.16	33.93	28	2695	19.5
[ZWS+16]	AlexNet-F	1.4	61.0	Fixed 16	HLS	Virtex7 VX690T	150	825.6	126.00	N.R	14400	N.R
[ZWS+16]	VGG16-F	31.1	138.0	Fixed 16	HLS	Virtex7 VX690T	150	1280.3	160.00	N.R	21600	N.R
[MSC+16]	NIN-F	2.2	61.0	Fixed 16	RTL	Stratix5 GXA7	100	114.5	19.50	224	256	46.6
[MSC+16]	AlexNet-F	1.5	7.6	Fixed 16	RTL	Stratix5 GXA7	100	134.1	19.10	242	256	31.0
[LFJ+16]	AlexNet-F	1.4	61.0	Fixed 16	RTL	Virtex7 VX690T	156	565.94	30.20	274	2144	34.8
[ACFM16]	AlexNet-C	1.3	2.3	Float 32	HLS	Virtex7 VX690T	100	61.62	30.20	273	2401	20.2
[MCVS17b]	VGG16-F	31.1	138.0	Fixed 16	RTL	Arria10 GX1150	150	645.25	50.00	322	1518	38.0
[MGG17]	AlexNet-C	1.3	2.3	Fixed 16	RTL	Cyclone5 SEM	100	12.11	N.R	22	28	0.2
[MGG17]	AlexNet-C	1.3	2.3	Fixed 16	RTL	Virtex7 VX485T	100	445	N.R	22	2800	N.R
[MCVS17a]	NiN	20.2	7.6	Fixed 16	RTL	Stratix5 GXA7	150	282.67	N.R	453	256	30.2
[MCVS17a]	VGG16-F	31.1	138.0	Fixed 16	RTL	Stratix5 GXA7	150	352.24	N.R	424	256	44.0
[MCVS17a]	ResNet-50	7.8	25.5	Fixed 16	RTL	Stratix5 GXA7	150	250.75	N.R	347	256	39.3
[MCVS17a]	NiN	20.2	7.6	Fixed 16	RTL	Arria10 GX1150	200	587.63	N.R	320	1518	30.4
[MCVS17a]	VGG16-F	31.1	138.0	Fixed 16	RTL	Arria10 GX1150	200	720.15	N.R	263	1518	44.5
[MCVS17a]	ResNet-50	7.8	25.5	Fixed 16	RTL	Arria10 GX1150	200	619.13	N.R	437	1518	38.5
[LDJ+17]	AlexNet-F	1.5	7.6	Float 32	N.R	Virtex7 VX690T	100	445.6	24.80	207	2872	37
[LDJ+17]	VGG16SVD-F	30.8	50.2	Float 32	N.R	Virtex7 VX690T	100	473.4	25.60	224	2950	47

4.4 Approximate Computing of CNN Models

Beside the computational transforms and data-path optimization, the CNN execution can be accelerated when employing approximate computing, which is known to perform efficiently on FPGAs [Mit16].

In the methods detailed in this section, a minimal amount of the CNN accuracy is traded to improve the computational throughput or energy efficiency of the accelerator. Two main strategies are employed. The first implements approximate *arithmetic* to process the CNN layers with a reduced precision. The second aims at reducing the number of operations occurring in CNN models without critically affecting the modelling performance. Note that both approaches can resort to *fine-tuning* in order to compensate the accuracy loss introduced by approximate computing.

4.4.1 Approximate Arithmetic for CNNs

Several studies have demonstrated that the precision of both operations and operands in CNNs, and more generally in neural networks, can be reduced without critically affecting their predictive performance. This reduction can be achieved by *quantizing* either or both of the CNN inputs, weights and/or FMs using a fixed point numerical representation.

4.4.1.1 Fixed point arithmetic

In a general way, CNN models are deployed in CPUs and GPUs using the same numerical precision they were trained with, relying on *simple-precision floating point* representation. This format employs 32 bits, arranged according to the IEEE754 standard. As current FPGAs support floating operations, various implementations [ZLS+15, ACFM16, AJK16] employ such data representation.

Nonetheless, numerous studies such [AHS15, GAN+15, LTA16] demonstrate that the inference of CNNs can be achieved with a reduced precision of operands. More particularly, works in [CBD14, ZWW+17] demonstrate the applicability of Fixed Point (FxP) arithmetic to *train* and *infer* CNNs.

The FxP representation encodes numbers with a given bit-width b, using i bits for the *integer* part, and f bits for the *fractional* part ($b = i + f$). Note that value of i is selected according the desired *numerical range*, and the value of f is selected according to the desired numerical *precision*.

In the simplest version of fixed point arithmetic, all the numbers are encoded with the *same* fractional and integer bit-widths. This means that the position of the radix point is similar for all the represented numbers. In this manuscript, we refer to this representation as *static* FxP.

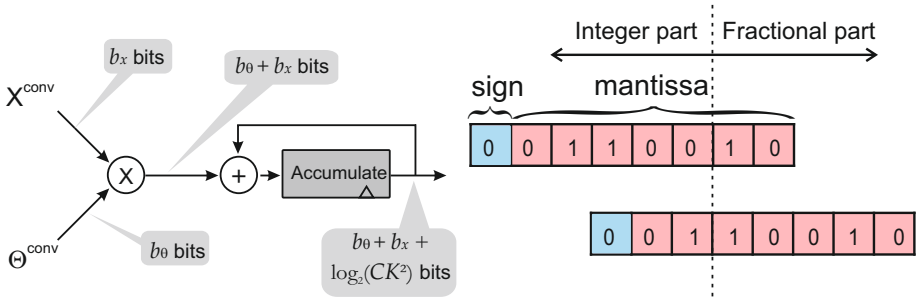
When compared to floating point, FxP is known to be more efficient in terms of hardware utilization and power consumption. This is especially true in FPGAs [DKT07], where –for instance– a single DSP block in Intel devices can either implement *one* 32bits floating point multiplication or *three* concurrent FxP multiplications of 9 bits [Int17]. This motivated early FPGA implementations, such [FMC+11, GJD+14] to employ fixed point arithmetic in deriving CNN accelerators. These implementations mainly use a 16-bits Q8.8 format , where 8 bits are allocated to the integer parts, and 8 bits to the fractional part. Note that the same Q8.8 format is used for representing the features and the weights of all the layers.

In order to prevent overflow, the former implementations also *expand* the bit-width when computing weighted-sums of convolutions. Equation 4.13 explains how the bit-width is expanded; if b_X bits are used to quantize the input FMs and b_Θ bits are used to

quantize the weights, an accumulator of b_{acc} bits is required to represent a weighted-sum of $C_\ell K_\ell^2$ elements, where:

$$b_{acc} = b_x + b_\Theta + \max_\ell \left[\log_2 \left(C_\ell K_\ell^2 \right) \right] \tag{4.13}$$

In practice, most FPGA accelerators use 48-bits accumulators, such as in [FPH+09, CSJC10].

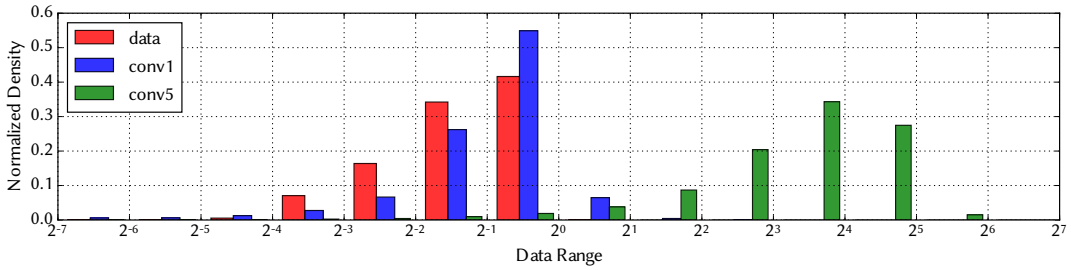


(A) bit-width of accumulators (B) Dynamic FxP

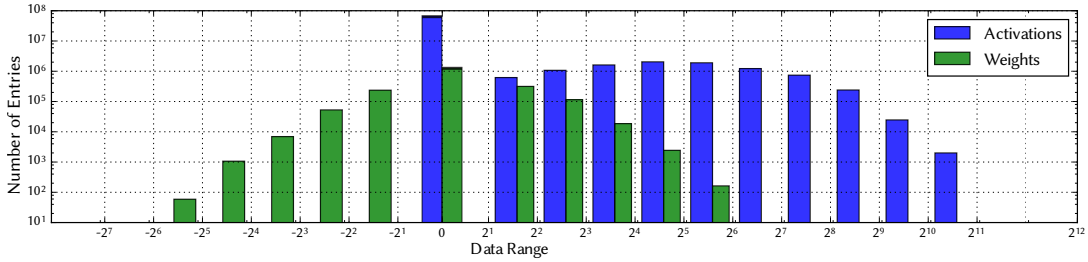
FIGURE 4.8: Fixed Point Arithmetic for CNN Accelerators

4.4.1.2 Dynamic Fixed Point for CNNs

In deep topologies, it can be observed that distinct parts of a network can have a significantly different range of data. In particular, the features of the deep layers tend to have a much larger numerical range when compared to the features of the first CNN layers.



(A) Histogram of the layer outputs



(B) Histogram of weights and activations. Inputs and weights encoded in 8 bits

FIGURE 4.9: Distribution of Alexnet activations and weights.

The histograms of Fig. 4.9a depict this phenomenon for AlexNet convolution layers[7]. While the CNN inputs (in red) are normalized take their values between 0 and 1, the outputs of the fist convolution layer (in blue) have a *wilder* numerical range, between 2^{-7} and 2^2. This is even more salient for the fifth convolution layer, where most of the outputs take their values between 2^{-1} and 2^6. The same problem appears when comparing

[7]Code made available at github.com/KamelAbdelouahab/CNN-Data-Distribution

the numerical of the CNN weights, and CNN activations. In this case, the weights are numerically much *smaller* when compared to the activations, as illustrated in figure 4.9b[8].

As a consequence, large bit-widths have to be allocated to the integer and fractional parts in order to keep a uniform precision across the network while preventing overflow. This expansion badly increases the resources requirements of a given FPGA mapping. As a result, static FxP, with its unique shared fixed exponent, is ill-suited to deep learning, as pointed out in [GMG16]

To address this problem, works in [CBD14, GMG16] advocates the use of *dynamic* FxP [Wil91][9]. In dynamic FxP, different scaling factors are used to process different parts of the network. In other words, the position of the radix point varies from one layer to another. More particularly, weights, weighted-sums and outputs of each layer are assigned distinct integer and fractional bit-widths.

The optimal values of these bit-widths (i.e the ones that deliver the best trade-off between accuracy loss and computational load) for each layer can be derived after a profiling processes, performed by dedicated frameworks that supports FxP. Among these frameworks, Ristretto [GMG16] and FixCaffe [GWC+17] are compatible with Caffe while TensorFlow natively supports 8 bits computations. Most of these tools can *fine-tune* a given CNN model to improve the accuracy of the quantized network. Figure 4.10 illustrates the improvements brought by dynamic FxP. One may note how the inference of Alexnet is possible using 6 bits in dynamic FxP, while classic fixed point requires 15 bits to deliver the same accuracy.

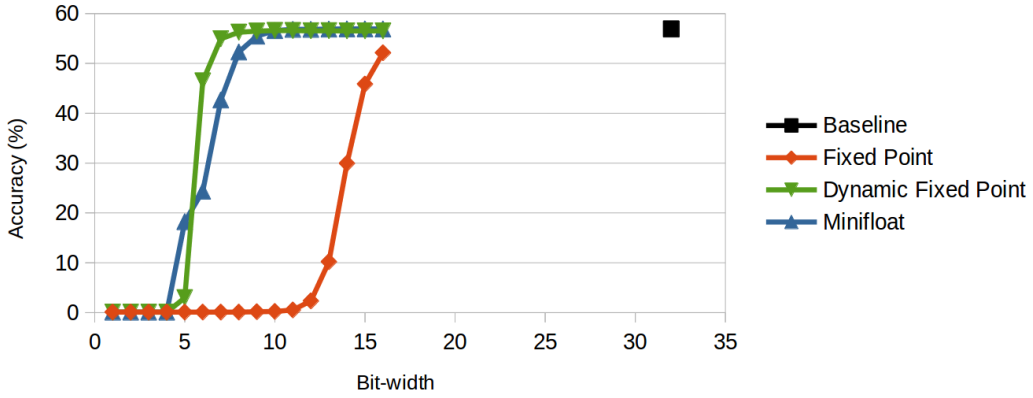


FIGURE 4.10: AlexNet top1 accuracy for various FxP representations, from [Gys16]

4.4.1.3 FPGA Implementations

The FPGA-Based CNN Accelerator proposed in [SCD+16] is build upon this quantification scheme and employs different precisions to represent the FM, convolution kernels and FC weights with *resp.* 16,8,10 bits. Without fine-tuning, authors report a drop of 1% in classification accuracy of AlexNet. In a same way, Qiu *et al.* employ FxP to quantize the VGG network with respectively 8 bits for the weights, 8 bits for activations and 4 bits for FC layers, resulting in 2% of accuracy drop. In all these accelerators, dynamic quantization is supported by means of data shift modules [QWY+16]. Finally, the accelerator in [MGG17] rely on the Ristretto framework [GMG16] to derive an AlexNet model wherein the data is quantized in 16 bits with distinct integer bit-widths per layer[10].

[8] This figure deliberately multiplies the weights and activations by a scale factor of $2^7 - 1$ to emulate an 8 bits quantization.

[9] An other approach to address this problem is to use custom floating point representations, as detailed in [AOC+17]

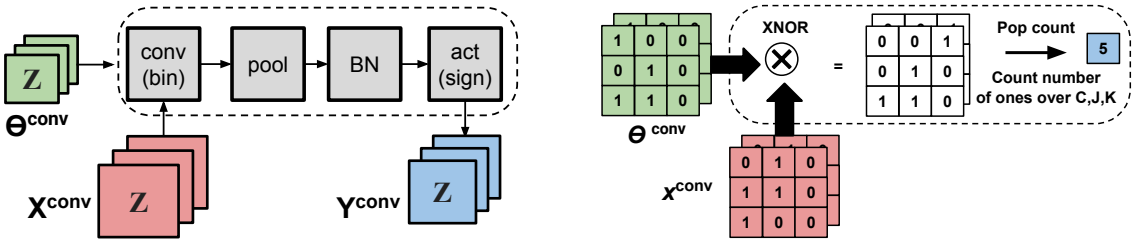
[10] Since the same PEs are reused to process different layers, the same bit-width is used with a variable radix point for each layer

4.4.1.4 Extreme quantification with Binary and pseudo-Binary Nets

Training and inferring CNNs with *extremely compact data representations*, is an area that is recently gaining a lot of research interest. Early works of Courbariaux *et al.* in BinaryConnect [CBD15] demonstrate to feasibility of training neural networks using *binary* weights i.e weights with either a value of $-\theta$ or θ encoded in 1 bit. BinaryConnect lowers the bandwidth requirements of a network by a factor of x32 at the price of an accuracy loss, evaluated at 19.2% on ImageNet[11].

The same authors go further in their investigations in [HCS+16] and propose Binary Neural Networks (BNNs) that represent both feature maps and weights with only 1 bit. In these networks, negative values are represented as 0 while positive values are represented as 1.

BNNs greatly simplify the processing of convolutions, boiling-down the computations of MACs into bitwise XNOR operations followed by a pop-count (see Fig.4.11b). Moreover, authors use the *sign* function as activation and apply Batch normalization before applying of the activation, which reduces the information lost during binarization (see Fig. 4.11a. In turn, a higher drop in classification accuracy occurs when using BNNs, evaluated at 29.8% for ImageNet. This accuracy drop is than lowered to 11% by Rastegari *et al.*, using different scale factors for binary weights (i.e $-\theta_1$ or $+\theta_2$)



(A) Processing Pipeline (B) Binary Convolutions

FIGURE 4.11: Binary Neural Networks

Beside BNNs, *Pseudo-Binary Networks*, such DoReFa-Net [NFS17] and Quantized Neural Networkss (QNNs) [HCS+18] reduce the accuracy drop to 6.5% when employing a slightly expanded bit-width (2 bits) to represent the intermediate FMs. Similarly, in Trained Ternary Quantization (TTQ) [ZHMD17], weights are constrained to three values (2 bits) $-\theta_1, 0, -\theta_2$, but FM are represented in a 32bits float scheme. As a consequence, the efficiency gain of TTQ is not as high as in BNNs. In turn, TTQ achieves comparable accuracy on ImageNet, within 0.7% of full-precision.

In FPGAs, BNNs benefit from a significant acceleration as the processing of "binary" convolutions can be mapped on XNOR gates followed by a pop count operation, as depicted in figure 4.11b. Furthermore, and as suggested in [NSB+17], pop count operation can be implemented using lookup tables in a way that convolutions are processed only with logical elements. Thus, the DSPs blocs can be used to process the batch norm calculation (eq 2.10, which can be formulated as a linear transform in order to reduce the number of operations. This approach is followed in the implementation of [ZSZ+17] to derive an FPGA-Based accelerator for BNNs that achieves 207.8 GOP/s while only consuming 4.7 W and 3 DSP Blocs to classify the Cifar10 dataset.
For the same task, works in [UFG+17, FUG+17] use a smaller network configuration[12] and reaches a throughput of 2.4 TOP/s when using a larger Zynq 7Z045 Device with

[11] When compared to an exact 32 Bits implementation of AlexNet

[12] The network topology used in this work involves 90% less computations and achieves 7% less classification accuracy on Cifar10

11W Power consumption.
For ImageNet classification, Binary Net implementation of [LYL+17] delivers an overall throughput 1.9 TOP/s on a Stratix V GSD device. In all these works, the first layer is not binarized to achieve better classification accuracy. As pointed-out in [LYL+17], the performance in this layer can be improved when using a higher amount of DSP blocs. Finally, an accelerator for TTQs is proposed in [PBP+17] and achieves a peak performance of 8.36 TMAC/s when classifying the Cifar10 data-set with a 2-bit precision.

4.4.2 Reduced Computations

In addition to approximate arithmetic, several studies attempt to the reduce the number of operations involved in CNNs. For FPGA-Based implementations, two main strategies are investigated: *weight pruning*, which increases the *sparsity* of the model, and *low-rank approximation* of filters, which reduces the number of multiplications occurring in the inference.

4.4.2.1 Weight Pruning

As highlighted in [LWF+15], CNNs as over-parametrized networks and a large amount of the weights can be removed –or *pruned*– without critically affecting the classification accuracy. In its simplest form, pruning is performed according to the magnitude such as the lowest values of the weights are truncated to zero [HPTD15]. In a more recent approach, weights removal is driven by energy consumption of a given node of the graph, which is 1.74x more efficient than magnitude-based approaches [YCS17]. In both cases, pruning is followed by a fine-tuning of the remaining weights in order to improve the classification accuracy. This is for instance the case in [HMD16], where pruning removes respectively 53% and 85% of the weights in AlexNet *conv* and FC layers for less then 0.5% accuracy loss.

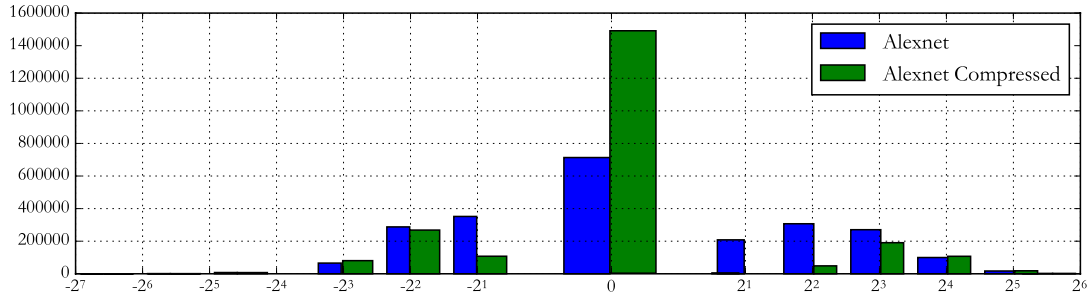


FIGURE 4.12: Histogram of conv weights in a compressed Alexnet model

4.4.2.2 Low Rank Approximation

Another way to reduce the computations occurring in CNNs is to maximize the number of *separable filters*. A 2D-separable filter, denoted θ^{sep}, has a unitary rank[13], and can be expressed as two successive 1D filters ($\theta^{\text{sep}}_{J\times 1}$ then $\theta^{\text{sep}}_{1\times K}$). Filter decomposition reduces the number of multiplications from $J \times K$ to $J + K$. This is illustrated in the example Fig.4.13, where the 3×3 averaging filter is separable, and can thus be decomposed into two successive one-dimensional convolutions.

The same concept expands to 3D-convolutions, where a separable filter requires $C + J + K$ multiplications instead of $C \times J \times K$ multiplications.

[13] Meaning that rank $(\theta^{\text{sep}}) = 1$

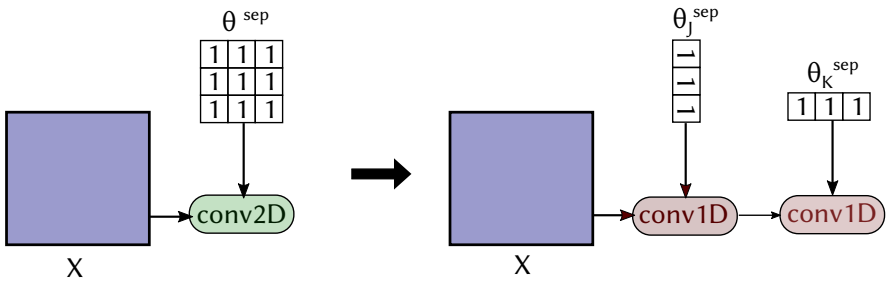


FIGURE 4.13: Example of a separable filter

Nonetheless, only a small proportion of CNN filters are separable. To increase this proportion, a first approach is to force the convolution kernels to be separable by penalizing *high rank filters* when training the network [STR+15]. Alternatively, and after the training, the weights $\mathbf{\Theta}$ of a given layer can be approximated into a small set of r *low rank filters*. In this case, $r \times (C + J + K)$ multiplications are required to process a single 3D-convolution.

Finally, when implementing im2col methods to process convolutions as GEMMs (c.f sec. 4.2.1), computations can be reduced by decomposing the weight matrix $\tilde{\mathbf{\Theta}}^{fc}$ through Single Value Decomposition (SVD). In a similar way to pruning, low rank approximation or SVD is followed by a fine-tuning in order to counterbalance the drop in classification accuracy.

4.4.2.3 FPGA Implementations

In FPGA Implementations, SVD is applied on FC layer to significantly reduce the number of weights, such as in [QWY+16], where authors derive a VGG16-SVD model that achieves 87.96% accuracy on ImageNet with 63% less parameters.

Alternatively, one can take advantage of the numerous research efforts given to accelerate Sparse GEMM on FPGA [DRM14]. In this case, the challenge is to determine the optimal format of matrices that maximizes the chance to detect and skip zero computations, such compressed sparse column (CRC) or compressed sparse row (CSR) formats[14]. Based on this, Sze *et al.* [SCYE17] advocates the use of the CRC to process CNNs. Indeed, this format requires lower memory bandwidths when the output matrix is smaller then the input, which is typically the case in CNNs where $N < CJK$ in Fig 4.2b.

However, this efficiency of CRC format is only valid for extremely sparse matrices (typically with $\leq 1\%$ of non zeros), while in practice, pruned CNN matrices are not that sparse (typically, $\leq 4 - 80\%$ of non zeros). Therefore, works in [NSB+17] propose a *zero skip scheduler* that identifies zero elements skips them in the scheduling of the MAC processing. As a consequence, the number of cycles required to compute the sparse GEMM is reduced. For AlexNet layers, the zeros skip scheduler results in a 4x speedup. The same authors project a throughput of 12 TOP/s for pruned CNNs in the next Intel Stratix10 FPGAs, which outperforms and the computational throughput of state-of-the-art GPU implementations by 10%.

[14] These format represents a matrix by three one-dimensional arrays, that respectively contain nonzero values, row indices and column indices

TABLE 4.4: Accelerators employing Approximate arithmetic

AxC	Entry	Dataset	Comp (GOP)	Params (M)	bit-width In/Out	bit-width FMs	bit-width θ^{conv}	bit-width θ^{FC}	Acc (%)	Device	Freq (MHz)	Through. (GOPs)	Power (W)	LUT (K)	DSP	Memory (MB)
FP32	[ZL17]	ImageNet	30.8	138.0	32	32	32	32	90.1	Arria10 GX1150	370	866	41.7	437	1320	25.0
FP16	[AOC+17]	ImageNet	30.8	61.0	16	16	16	16	79.2	Arria10 GX1150	303	1382	44.3	246	1576	49.7
	[MCVS17b]	ImageNet	30.8	138.0	16	16	8	8	88.1	Arria10 GX1150	150	645	N.R	322	1518	38.0
DFP	[MCVS17a]	ImageNet	30.8	138.0	16	16	16	16	N.R	Arria10 GX1150	200	720	N.R	132	1518	44.5
	[ZL17]	ImageNet	30.8	138.0	16	16	16	16	N.R	Arria10 GX1150	370	1790	N.R	437	2756	29.0
	[ZSZ+17]	Cifar10	1.2	13.4	20	2	1	1	87.7	Zynq Z7020	143	208	4.7	47	3	N.R
	[UFG+17]	Cifar10	0.3	5.6	20/16	2	1	1	80.1	Zynq Z7045	200	2465	11.7	83	N.R	7.1
BNN	[LYL+17]	MNIST	0.0	9.6	8	2	1	1	98.2	Stratix5 GSD8	150	5905	26.2	364	20	44.2
	[LYL+17]	Cifar10	1.2	13.4	8	8	1	1	86.3	Stratix5 GSD8	150	9396	26.2	438	20	44.2
	[LYL+17]	ImageNet	2.3	87.1	8	32	1a	1	66.8	Stratix5 GSD8	150	1964	26.2	462	384	44.2
	[PBP+17]	Cifar10	1.2	13.4	8	2	2	2	89.4	Xilinx7 VX690T	250	10962	13.6	275	N.R	39.4
TNN	[PBP+17]	SVHN	0.3	5.6	8	2	2	2	97.6	Xilinx7 VX690T	250	86124	7.1	155	N.R	12.2
	[PBP+17]	GTSRB	0.3	5.6	8	2	2	2	99.0	Xilinx7 VX690T	250	86124	6.6	155	N.R	12.2

TABLE 4.5: Accelerators employing pruning and low rank approximation

Reduc.	Entry	Dataset	Comp (GOP)	Params (M)	Removed Param. (%)	bit-width	Acc (%)	Device	Freq (MHz)	Through. (GOPs)	Power (W)	LUT (K)	DSP	Memory (MB)
SVD	[QWY+16]	ImageNet	30.8	50.2	63.6	16 Fixed	88.0	Zynq 7Z045	150	137.0	9.6	183	780	17.50
Pruning	[FSNM17]	Cifar10	0.3	13.9	89.3	8 Fixed	91.5	Kintex 7K325T	100	8620.7	7.0	17	145	15.12
	[NSB+17]	ImageNet	1.5	9.2	85.0	32 Float	79.7	Stratix 10	500	12000.0	141.2	N.R	N.R	N.R

4.5 Conclusions

Numerous studies leverage on FPGA computational power to implement the feed-forward propagation of CNNs. After studying the accelerators available on the literature, two conclusions can be drawn:

- The first one is related to automation of the mapping process: Deriving FPGA accelerators calls for a holistic exploration of the design space of both CNN hyper-parameters (sec. 4.4.2), arithmetic precision (sec. 4.4.1) and Hardware parameters (sec. 4.3.2). This process calls for either tool-flows to automatically Map CNNs on FPGAs, or models to estimate performance and energy of CNN mappings before synthesis.
- The second is related to the FPGA devices used. In fact, little research interest is given to « embedded » FPGAs and a majority of state-of-the-art accelerators target high-end devices. This is generally done by inferring a limited number of processing elements, time-sharing the computations across these PEs. Such Time-multiplexed architectures do not match with the streaming nature of CNNs and deliver poor real-time performance when ported to resource constrained FPGAs. This is especially true for the FPGAs integrated in smart camera nodes, where the computational resources are up to 50 times less important than in high-end devices.

To address the two former points, the next chapters discuss a tool-flow to efficiently map CNNs on FPGAs. This tool-flow rely on coarse-grain optimizations at the model level, and fine-grain optimizations at the architectural level, both described in the remaining parts of this manuscript.

Chapter 5

Model-Based Optimization of CNN Mappings on FPGAs

In order to make real-time embedded vision feasible on reconfigurable hardware, this chapter focuses on optimizations that operate at the level of the CNN and its execution model. These optimizations are agnostic to the FPGA technology used and can be combined with other *low-grain* approaches operating at the RTL levels that will be discussed in chapter 6.

The proposed optimizations take the form of « rules » that are derived after mapping CNNs with variable configurations. By configuration, we refer to the CNN hyper-parameters (Depth, kernel-size, number of activations), and to execution models (Dataflow *vs* Von-Neumann). Moreover, this chapter studies the effects of pruning and quantization on the performance of an FPGA implementation.

Comparing the performance of multiple CNN mappings on FPGAs is a time consuming task that calls for automated tools to generate the hardware of a given CNN. In the sequel, HADDOC, a tool-chain to map CNN graphs on FPGAs is proposed. The first part of this chapter (sec.1,2,3) discusses the model of computation HADDOC relies-on and introduces the principles of *direct hardware mapping* of CNNs on FPGAs.

Then, section 4 details the design space exploration process with HADDOC and derives first rules of efficient hardware mapping of CNNs. Finally, the last section explores a novel parameters in CNNs: the number of views.

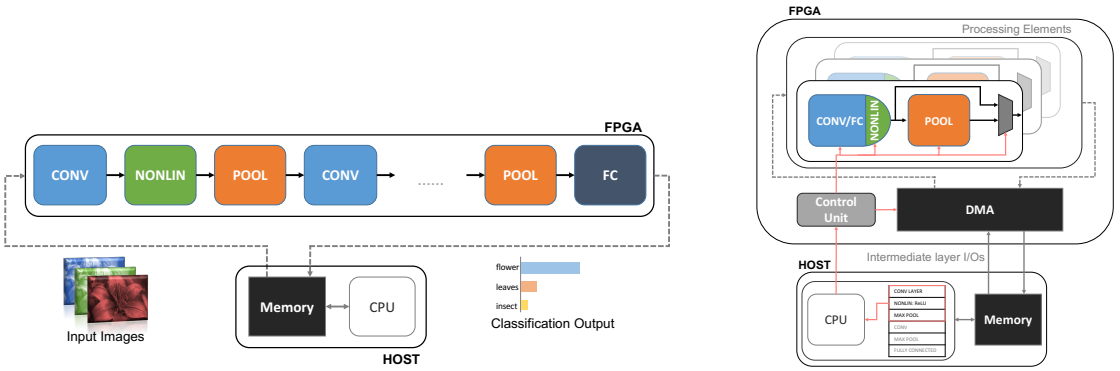
5.1 Models of Computation for CNN inference on FPGAs

In general, two different paradigms exist in deriving custom hardware accelerators, particularly in the case of CNN acceleration. Figure 5.1 depicts these paradigms. On one side of the spectrum, Von Neumann based Models are widely present in the literature, and support a large variety of CNN workloads. On the other side, dataflow accelerators directly map the operations involved in a given CNN, delivering better performance at the price of a lower flexibility.

5.1.1 The Von Neumann paradigm

As detailed in chapter 4, numerous FPGA-based CNN accelerators comprise a computation engine and a control block. The computation engine takes the form of a systolic array or a collection of processing elements, and execute the CNN operations sequentially in a time-multiplexed fashion. The model of computation in such accelerators is inspired by Von Neumann architectures, as illustrated in Fig.5.1b. This architecture involves a fixed template scaled according to the available FPGA resources and controlled by a software, usually hosted in a CPU.

With this scheme, the CNN is described as a sequence of micro-instructions that are executable by a static architecture. In this case, the same bit-stream can target multiple



(A) Dataflow Based (B) Von Neumann Based

FIGURE 5.1: Models of Architecture for CNN Inference on FPGA. From [VKB18]

CNNs, granting high flexibility to the accelerator. In turn, this flexibility comes at the price of inefficiencies resulting from the complex control mechanisms and the high variability of CNNs workloads.

5.1.2 The Dataflow Paradigm

As detailed in section 3.3, image processing workloads generally benefit from a significant acceleration when running on *dataflow* architectures. This acceleration comes in exchange of an algorithmic reformulation effort by the programmer. However, in the case of deep learning based computer vision, CNN are algorithms that «naturally» match with the dataflow semantics.

This is especially true for the feed-forward propagation, which is a typical streaming workload wherein the execution is purely data-driven. Indeed, the inference of a CNN (detailed in sec.2.2) can be described as a graph where the nodes represent fundamental computations occurring of the feature maps. This execution layout is in contrast with Von Neumann execution models and, as highlighted in section 2.5.3, a given CNN accelerator can easily be memory-bounded if it has to fetch every instruction from memory [SCYE17].

The advantages of such a model for CNN inference are clear.

- First, it catches all the parallelism patterns detailed in section 2.4.2 and grants high computational throughput that delivers real-time performances.
- Second, it exempts from external memory accesses as the streams of tokens locally stay on the FIFOs, next to the processing capabilities of the actors
- Finally, the dataflow graph can be seen as an intermediate representation that stays between the mathematical description of the CNN and its hardware implementation. This representation can thus be used to derive a hardware implementation of the CNN.

These advantages motivated numerous research efforts to consider the dataflow paradigm for the CNN inference problem. The next section reviews the main contributions.

5.1.3 Dataflow-Inferred CNN Accelerators

The dataflow-based implementation of CNNs was first investigated in [FMC+11] where Farabet *et al.* describe NeuFlow, an acceleration engine for CNNs relying on a dataflow model. NeuFlow transforms the CNN graph into a set of dataflow instructions, each

being described as a hardware configuration of 2D-processing elements called *Processing tiles* implemented on FPGA. The execution of the graph is carried out by sequencing the instructions through the Processing tiles.

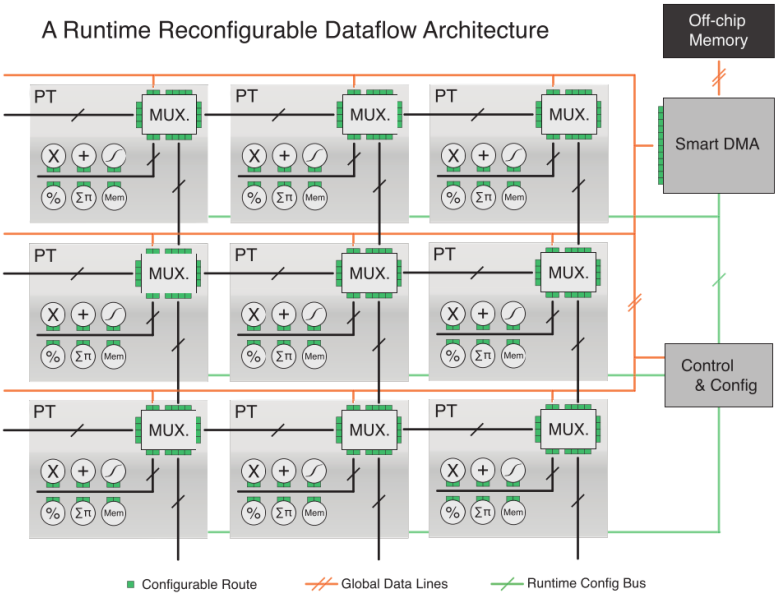


FIGURE 5.2: Hardware Architecture of NeuFlow [FMC+11]

A special case of dataflow, called Static Data-Flow (SDF) [LM87], is a paradigm in which the number of tokens produced and consumed by each actor can be specified *a priori*, which is the case in CNN inference. SDF is employed in [VB16, VB17] to optimize the mapping of CNN graphs on FPGAs. In this works, actors of the dataflow graph correspond to the *layers* of the CNN. This graph is then modelled as a topology matrix Γ where each row corresponds to an arc and each column corresponds to a node (layer). Thus, an element $\Gamma(a, \ell)$ in this matrix specifies the rate of the data that flows from layer ℓ along the arc a.

Moreover, authors define elementary building blocks such sliding windows, convolution banks and memory I/O units, in a way that each column of Γ (i.e each layer of the CNN) is to be mapped to a set of hardware building blocks that implement its functionality. To find the optimal configuration of these building blocks, design space exploration is achieved by decomposing and *transforming* the topology matrix. Finally, the study in [VB16] exploits the dynamic reconfiguration capabilities of FPGAs and features a partitioning of the DPN into subgraphs, with one generated bit-stream per subgraph. Nevertheless, this approach requires the reconfiguration of the FPGA whenever data has to enter a different subgraph, which can add a substantial reconfiguration time overhead.

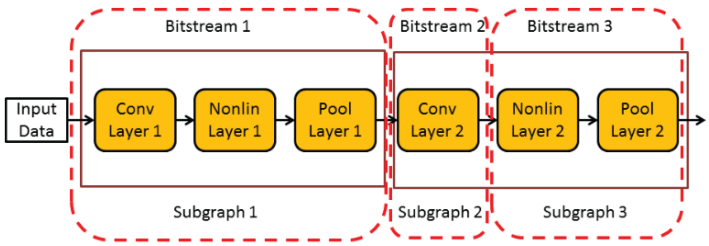


FIGURE 5.3: Graph Partitioning in FPGAConvNet [VB16]

5.2 Direct Hardware Mapping of CNNs on FPGAs

In all the previously evoked approaches, FPGA resources are time-shared to provide a tunable trade-off between resource utilization and throughput performance (see sec.4.3). This « time-multiplexing» calls for memory accesses which, even with the help of a dma, limits the final speedup [FMC+11].

To overcome this limitation, a *direct* mapping of the CNN onto the physical resources of the FPGA is proposed. The so-called Direct Hardware Mapping (DHM) consists of *physically* mapping the DPN on the target device. Each actor of the graph becomes a computing unit with its specific instance on the FPGA, and each edge of the graph becomes a signal.

5.2.1 DHM of convolution layers

As stated in section 2.4.1, convolution layers are the most computationally intensive parts of a CNN. To accelerate the execution of these layers, the proposed DHM approach fully « unrolls » the processing of a given *conv* layer ℓ by:

- Physically Mapping the N_ℓ three-dimensional convolutions in a concurrent fashion, as illustrated in Fig. 5.4a.
- Physically Mapping each of the former 3D-convolutions as C_ℓ concurrent two-dimensional convolutions, as shown in Fig 5.4b.
- Implementing each 2D-convolution in a pipelined fashion, using $J_\ell \times K_\ell$ multipliers, as shown in Fig 5.4c.

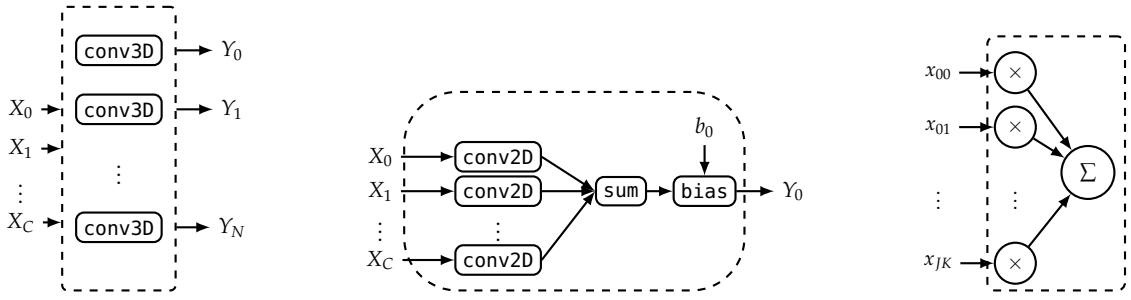


FIGURE 5.4: The 3 levels of DHM implementation of CNN entities

With this method, the DHM approach fully unrolls the processing in a way to produce N_ℓ token per clock cycle. This corresponds to mapping $\mathcal{R}m_\ell$ multipliers per layer, as shown in the following equation:

$$\mathcal{R}m_\ell = N_\ell C_\ell J_\ell K_\ell \tag{5.1}$$

The advocated approach has two main benefits:

- First, it grants a high computational throughput that depends only on the length of input streams (i.e the input video resolution), and the frequency of the design F, as expressed in equation 5.2.

$$\mathcal{T}_{(\text{FPS})} = \frac{F}{H_0 W_0} \tag{5.2}$$

- Second, it exempts from any external memory access. Streams of tokens are produced on the fly at each actor and are transferred to the next actor through FIFOs channels.
- The major downside is related to the flexibility of the accelerator. With the proposed DHM, a CNN mapping can rapidly be bounded by the available resources present in a given FPGA. For instance, if the graph of a given layer (or network) involves more multiplications than the number of multipliers available on an FPGA, the dataflow mapping is not feasible. Tactics to overcome this downside are discussed in the next chapter.

5.2.2 Use-cases of DHM

Because of the previously evoked argument, direct hardware mapping of CNN graphs is advocated for networks and layers that involve a *low number of computation on a high number of data*. For this kind of workloads, DHM requires less resources and can process a high number of inputs at a high computational throughput, without being bounded by resources availability.

By contrast, a Von Neumann paradigm is more suitable for workloads involving a high number of computation on a small amount of data. In this case, there is less data to move in and out the processing elements when iterating the computations. Consequently, Von Neumann based accelerators spends less timings on memory access, allowing real-time performance on « time-multiplexed » architectures.

Thus, different paradigms have to be used according to the workload of a given layer. To characterize this workload, the Computation to Communitcation (CTC) metric is defined, and corresponds to the ratio between the number of resources to be mapped (eq.5.1) and the amount of data to be manipulated (eq.2.25). For each *conv* layer, this ratio can be written as:

$$CTC_\ell = \frac{\mathcal{R}m_\ell}{\mathcal{M}_\ell} \tag{5.3}$$

$$= \frac{N_\ell C_\ell J_\ell K_\ell}{C_\ell H_\ell W_\ell + N_\ell C_\ell J_\ell K_\ell + N_\ell U_\ell V_\ell} \tag{5.4}$$

Figure 5.5 reports the CTC ratio for various layer of multiple CNNs[1]. The higher the CTC is, the lower the dataflow paradigm is suitable. In particular, one may note that the first layers of CNNs manipulate a high number of inputs/outputs while involving a small number of resources[2]. It it thus preferable to implement these layers in a dataflow fashion. By contrast, notably because of the sub-sampling layers, mid-range and last layers manipulates a smaller amount of data while involving more resources. They are thus good candidates for a time-multiplexed implementation.

It is also worth to mention that CNN detectors such YOLOv2 operate on images with a higher resolution when compared to CNN classifiers (416×416 vs 227×227). Dataflow architectures are thus expected to perform more efficiently for these workloads.

5.2.3 The case of FC Layers

Fully connected layers is a typical example of workloads involving a large number of computations on a small amount of data. According to the results of [Bou16], implementing FC layers in a fully pipelined fashion consumes 47% of the resources allocated

[1]The code reproducing this chart is made available at https://github.com/KamelAbdelouahab/CNN-Workload

[2]Exception made for Alexnet first conv layer as it involves 11×11 convolutions

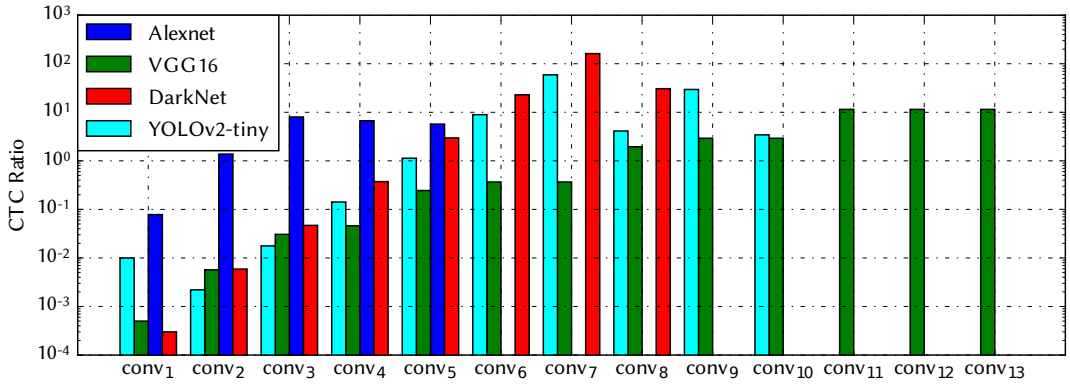


FIGURE 5.5: CTC Ratio in Popular CNNs. Y axis in logarithmic scale

to a given mapping despite representing 3% of the total computational workload. As a consequence, models of computation other then the dataflow MoC have to be considered to implement of the fully connected parts of a CNN.

In the upcoming experiments, the processing of FC layers is formulated as as a vector-matrix multiplication (cf. sec.4.2.1). With this approach, one can leverage on the advances of a large number of works addressing the problem of matrix multiplication on FPGA, according to the matrix dimension [KDC12, SQH+18], the matrix sparcity [ZP05, DRM14], and the resource/energy constrains [JCP02, JCP05]. Moreover, one can directly use the optimized matrix multiplication IP cores made available by FPGA manufacturers [DFK+07, Int14b].

The downside of this approach is that it requires a data transfer between the feature extraction block that computes the *conv* layers, and the matrix-multiplication block that computes the FC layers. Fortunately, the transferred data volume corresponds to vector of few hundreds of elements representing the features extracted from each frame. For example, in the case of Alexnet, the dimension of this vector is

$$C_{conv5} * H_{conv5} * W_{conv5} = 43264$$

This corresponds to 1.29 MBs of bandwidth in a 30 FPS video frame.

5.3 Direct Hardware Mapping with CAPH

In the upcoming experiments, the hardware description of a given CNN is derived by CAPH *HLS tool*. The choice of using CAPH is motivated by the high productivity it grants when prototyping dataflow-based architectures on reconfigurable hardware, as studied in section 3.3.

5.3.1 CAPH Formulation of CNNs

To describe a Convolutional Neural Network as a dataflow graph, a set of actors have to be specified first, each actor being defined according to its functionality in the CNN.

Based on the topology of popular CNN models, Bourrasset et *al.* [Bou16] proposed a CAPH implementation of common CNN actors. To build the complex dataflow graphs involved in CNNs, authors follow an ascendant conception flow that relies on the wiring functions of the CAPH tool to build high semantics actors from the lower semantic ones (see sec.3.3.2). In the case of convolution and pooling layers, the main actors are:

convLayer : This actor describes the behaviour of a *conv* layer and implements the 3D-convolutions as a sum of 2D-convolutions (see eq. 2.4). This is done by wiring concurrent `conv2DJK`[3] blocks with `sum`, `bias` and `ReLU` actors. These respectively sums, adds the bias, and applies a ReLU function to the inputs streams.

conv2DJK : Performs a two-dimensional $J \times K$ convolution on input streams. Authors implement this 2D-convolution as a *mono-actor*, meaning that it is described using a large number of firing rules instead of wiring fine grain actors. While this implementation suffers from a low expressibility, it grants a better computational throughput and a lower resource utilization[4].

PoolJK : Sub-samples the input stream by applying $J \times K$ max-pooling. This actor is built by wiring `PoolH` and `PoolV` actors, which respectively sub-samples the stream in the horizontal (*resp.* vertical) dimension.

Listing 5.1 gives an example of a CAPH description of a CNN using the formerly defined actors. The corresponding DPN is illustrated in figure 5.6.

LISTING 5.1: Example of a CAPH Description of a CNN

```
-- Layer conv1: Extracts Three Feature Maps C=1, K=3, N=3
net(c10, c11, c12) = convs conv233c rep3 weights_conv1 bias_conv1 i;
-- Apply ReLU Activation
net(r10, r11, r12) = map relu (c10, c11, c12);
-- Layer pool1: 2x2 Max-pooling
net(p10,p11,p12) = map (pool 2 2) (r10,r11,r12);
-- Layer conv2: Extracts Five Feature Maps C=3, K=3, N=5 and applies ReLU
net(c20,c21,c22,c23,c24) =
convlayer conv233c weights_conv2 bias_conv2 sum3 relu ((p10,p11,p12),
                                                       (p10,p11,p12),
                                                       (p10,p11,p12),
                                                       (p10,p11,p12),
                                                       (p10,p11,p12));
-- Layer pool2: 2x2 Max-pooling
net(p20,p21,p22,p23,p24) = map (pool 2 2) (c20,c21,c22,c23,c24);
```

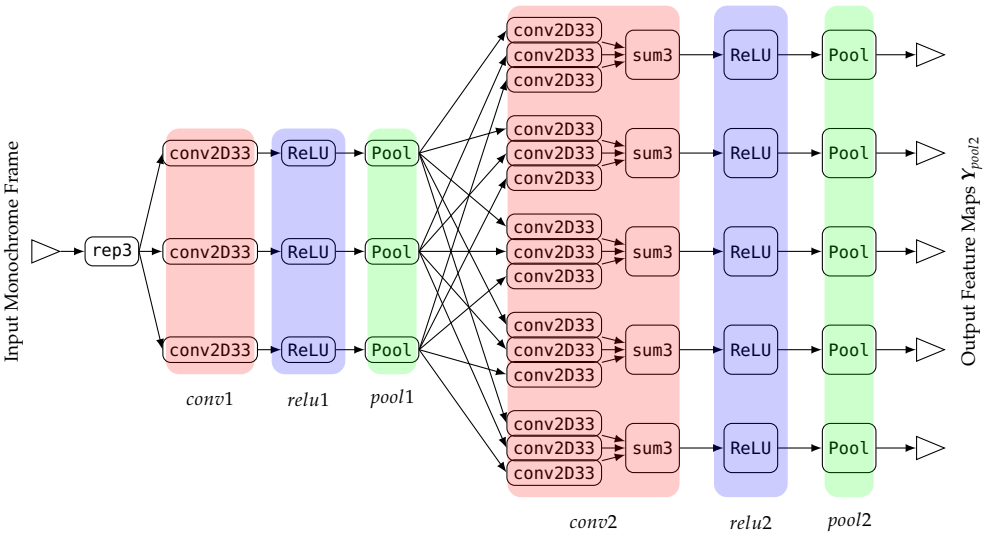


FIGURE 5.6: Dataflow Graph of the Network described in 5.1

In the same work [Bou16], the author highlights that implementing a high level functionality by wiring low grain actors can badly impact the performance and resource utilization of a given FPGA mapping. However, the proposed approach grants a higher productivity, fastening the prototyping and exploration process. The studies proposed in the next sections of this chapter are all based on the CAPH implementation of CNNs discussed above.

[3] Where J and K represent the size of the convolution kernel

[4] As highlighted in section 3.4.2

5.3.2 Automated Hardware Generation

Based on the CAPH Implementation of CNN actors detailed above, this section introduces the HADDOC tool. It automatically transforms a CNN description into a synthesizable hardware description thanks to the CAPH language.

The Conception flow of HADDOC is illustrated in figure 5.7. Starting with a CNN designed and trained using Caffe [JSD+14], HADDOC generates the corresponding DPN described as a CAPH network. HADDOC also extracts the weights from the pre-trained Caffe network and, if specified by the user, quantizes these weights into a desired fixed point-representation[5]. Second, the CAPH compiler is used to create a hardware description of the generated DPN. This hardware description, which is platform and constructor independent, is finally mapped on a given device using the adequate synthesis Tool.

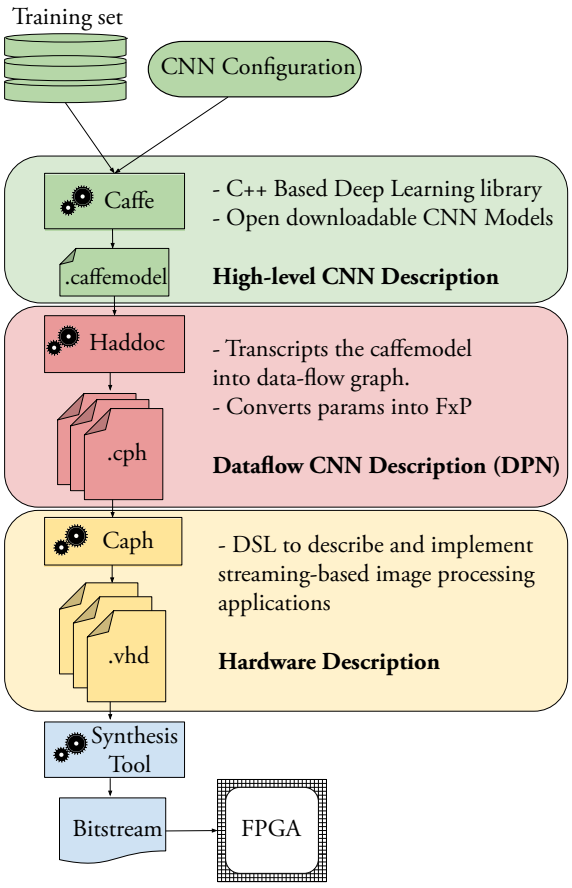


FIGURE 5.7: Haddoc Conception flow for FPGA Mapping of CNNs

HADDOC automates and fastens the generation of dataflow accelerators. In the sequel, this tool is used to explore the performance of CNN mappings in the embedded space. Through this study, we want to clearly quantify the efficiency improvements[6] brought by methods like quantization or pruning to a given FPGA mapping.

5.4 Design Space Exploration

As detailed in the last chapter, deriving efficient CNN mappings on FPGAs often comes down to finding the best trade-off between computational throughput, classification accuracy and energy consumption. Finding this trade-off calls for an exploration in large

[5]Haddoc1 code is made available https://github.com/KamelAbdelouahab/haddoc1
[6]In terms of CNN classification accuracy and FPGA resource utilization

space of design parameters such the CNN depth (L) , the topology (N, K), and the arithmetic precision (b).

5.4.1 Depth and Convolution Kernel Size

This first example of design space exploration considers four CNNs designed for OCR applications. These networks are based on a LeNet5 CNN, which includes a small number of convolutional layers interspersed with sub-sampling layers, as detailed in table 5.1. This table also reports the number of multipliers to be mapped on the device ($\mathcal{R}m_\ell$), and the computational workload of each layer ($\mathcal{C}_\ell$)

TABLE 5.1: Topology of the Studied LeNet Implementations: For each layer ℓ, N, C, J refers to the number and dimensions of 3D filters, U to feature maps' width width, $\mathcal{R}_m$ to the number of multipliers and $\mathcal{C}$ computational workload

Impl.	Layer	N	C	J	U	Act, pool	$\mathcal{R}m_\ell$	$\mathcal{C}_\ell$ (KMAC)
I1	conv1	20	1	5	24	Pool	500	288.00
	conv2	50	20	5	8	Pool	25000	1600.00
	ip1	500	800	1	1	ReLU	-	400.00
	ip2	10	500	1	1	Softmax	-	5.00
I2	conv1	20	1	3	26	Pool	180	121.68
	conv2	50	20	3	11	Pool	9000	1089.00
	ip1	270	800	1	1	ReLU	-	408.38
	ip2	10	500	1	1	Softmax	-	2.70
I3	conv1	6	1	3	26	TanH+Pool	54	36.50
	conv2	16	6	3	11	TanH+Pool	864	104.5
	conv3	120	8	3	5	TanH+Pool	17280	155.5
	ip1	84	480	1	1	TanH	-	22.68
	ip2	10	84	1	1	Softmax	-	0.84
I4	conv1	6	1	3	26	ReLU+Pool	54	36.50
	conv2	8	6	3	11	ReLU+Pool	432	52.27
	conv3	60	8	3	3	ReLU	4320	38.88
	ip1	420	540	1	1	ReLU	-	226.80
	ip2	10	420	1	1	Softmax	-	4.20

The first CNN is the standard implementation of LeNet available in Caffe[7]. It includes two convolutional layers using (5×5) kernels interspersed with sub-sampling layers[8]. The second implementation $I2$ considers a similar topology, but replaces the previous (5×5) kernels by (3×3) filters, which theoretically reduces the utilized FPGA resources.

The third implementation is our porting on Caffe of the original LeNet5 described in [LBBH98][9]. It includes three convolution layers interspersed with sub-sampling and TanH non-linearities. Note that this implementation involves less convolutions per layer than the one proposed by Caffe, but includes a third computationally intensive *conv* layer. Finally, $I4$ is network that involve the same number of layers as $I3$, but extracts twice less FMs on the second and third layers, significantly reducing the number of multipliers mapped on the device.

[7] https://github.com/BVLC/caffe/blob/master/examples/mnist/lenet.prototxt

[8] Caffe implementation of Lenet does not include use activation functions after the convolutions, but works surprisingly well on MNIST dataset...

[9] We slightly modified the network to use 3x3 kernels on 28x28 frames instead of 5x5 kernels on 32x32 frames

For the four implementations, the number of neurons involved the FC layers is set according to the dimension of the features extracted by the last *conv* layer. In particular, we set the number of neurons in a way that maintains the computational workload of classifier as constant as possible, around 400KMACs.

5.4.1.1 Training

The studied CNNs are first implemented in the Caffe framework and trained on 60K examples of the MNIST handwritten digit database. The remaining 10K examples of the dataset are used to validate the trained model and evaluate its classification performance on MNIST. This classification performance is reported as an Top1 accuracy rate in table 5.2.

(A) (B)

FIGURE 5.8: Differences between MNIST and USPS databases

Moreover, and to insure the trained networks generalize well on unseen «real-life» examples, we evaluate the CNNs' top1 accuracy on 1500 entries of the USPS dataset which were never been seen by the models. Note that USPS is more difficult to solve than MNIST, as depicted by the samples of figures 5.8b and 5.8a.

5.4.1.2 Accuracy Evaluation

As it can be seen in table 5.2, the training is successful as each of the four implementations achieves more than 98.8% accuracy on MNIST. Particularly, this table highlights how the deep networks ($I3$, $I4$) generalize better on USPS when compared to shallow networks with two layers ($I1$, $I2$).

It can also be observed that reducing the filter size degrades the performance on USPS by only 1.79%, while theoretically reducing the resources by a $\times 2.7$ factor. Similarly, reducing the number of feature maps decreases the classification performance by few percents (-0.05% on MNIST and -6.36% on USPS), but significantly improves the efficiency of the mapping ($\times 3.68$ less resources utilization expected).

These results suggest that reducing the depth on a network degrades its classification performance, especially on unseen data samples. By contrast, lowering the size of convolution filters and reducing the dimensions of feature maps are approaches that can both be considered. To validate this suggestions, the next section focuses on the mapping of the networks on FPGA devices.

TABLE 5.2: Accuracy of the studied networks on MNIST and USPS

Impl.	$\mathcal{R}_m$	$\mathcal{C}$(MMAC)	$\mathcal{A}_{\text{MNIST}}$	$\mathcal{A}_{\text{USPS}}$
I1	25500	1.88	98.83 %	70.87 %
I2	9180	1.21	99.01 %	69.08 %
I3	18198	0.44	98.96 %	85.38 %
I4	4806	0.17	98.81 %	76.21 %

5.4.1.3 FPGA Implementation Results

To map the proposed networks, a Stratix `5SGSED8N3F45I4` device is selected for prototyping purposes, mainly related to the availability of DSPs and I/Os. Moreover, a fixed

point format of 8 bits is applied to both the weights and activations. This quantization doesn't effect the classification accuracy as it will be detailed in the next section.

The FPGA implantations use Caph-2.9.0 and Quartus-18.0. The corresponding post-fitting reports are summarized in table 5.3.

TABLE 5.3: Post-fitting Results of the CNNs mapped with Haddoc

Network	Actor	ALM[10]	Registers	DSP	Freq (MHz)
I1	conv	245618 (93.6%)	191K	694	N.R
	pool	2850 (1.1%)	1K	0	
	fifo	119078 (45.4%)	211K	0	
	total	**334885 (127.6%)**	**344.2**	**694**	
I2	conv	98830 (37.7%)	44K	1004	59.87
	pool	3020 (1.2%)	1K	0	
	fifo	79355 (30.2%)	145K	0	
	total	**181296 (69.1%)**	**190.9**	**1004**	
I3	conv	207654 (79.1%)	9K	2115	N.R
	pool	5216 (2.0%)	3K	0	
	fifo	164859 (62.8%)	299K	0	
	total	**377730 (152.9%)**	**401.0**	**2115**	
I4	conv	105924 (40.4%)	49K	1421	61.46
	pool	3574 (1.4%)	1K	0	
	fifo	87885 (33.5%)	162K	0	
	total	**197493 (75.3%)**	**214.0**	**1421**	

It can first be noticed that mapping of $I1$ and $I3$ requires more resources than what is available on the selected device, which calls for further optimizations that will be addressed in the next chapter. The other implementations of LeNet are synthesisable on the FPGA and operate at frequency around $61MHz$. This corresponds to 33 FPS on $720p$ monochrome videos streams according to eq.5.2[11].

As expected, using (3×3) convolution filters significantly reduces the resources instantiated, granting $\times 1.8$ savings in terms of ALMs and registers. Reducing the dimension of the extracted FMs brings even more savings, lowering the utilization of the logic blocks, registers and DSPs blocks by respectively $\times 2$, $\times 1.8$ and $\times 1.5$.

Table 5.3 also shows how the actors involved in the processing of *conv* layers (namely, `conv2D33`, `sum`, `bias`, `ReLU`) are –as expected– the most resources-hungry parts of a given mapping. What wasn't expected though is the high hardware cost of the *FIFOs*. Generated by CAPH between each separate actors, FIFO channels result in a large overhead that represent up to 31% of the ALMs and 76% of the registers of a given mapping.

This overhead explains why the implementation $I4$ requires as much resources as $I2$ despite involving twice less computations. Indeed, adding a third convolutional layer causes CAPH to generate an additional stage of FIFOs between the layers, which in turn, badly impacts the efficiency of the FPGA Mapping.

5.4.2 Quantization vs Pruning

The last study has shown how a given CNN should be deep enough to provide tolerable generalization performance, and how the kernel size K can be decreased to positively

[10] Percentage given against the total number of ALMs available on the device: 262400

[11] The actor model currently defined in the CAPH compiler constrains the actor frequency to be twice bigger than the input data frequency. This constrain divides the computational throughout by two

impact the efficiency of the FPGA mapping. In this part, the remaining parameters (N, b) are explored. While similar works already addressed this so-called « pruning vs quantization » problem [LM97, FTX17], none of them focused on the impacts of these methods on an FPGA implementation. A tool such HADDOC make this study possible by automating the hardware generation process, and thus, the design space exploration.

5.4.2.1 Methodology and Experimental Setup

The proposed automated design space exploration relies on an iterative method that generates the corresponding CNN hardware architecture for each network topology and data representation size (see algorithm 1).

At each iteration, The CNN performances and hardware utilization are monitored by triggering the proposed tool chain depicted in Figure 5.9. It consists of using Caffe to train a network with a selected topology, then HADDOC to transcript the Caffe model into multiple DPNs with variable bit-widths. CAPH processes these DPNs and uses its SystemC backend to perform a functional simulation that evaluates the Top1 accuracy of the FPGA implementation. In parallel, CAPH also generates the hardware description of the CNN whose hardware utilization is monitored using the Quartus II synthesizer.

Algorithm 1: Design Space Exploration

```
Set Boundaries of Design Space N1min, N1max, N2max, N3max, bwmin, bwmax;
for N1 ∈ [N1min, N1max] do
    for N2 ∈ [N1max, N2max] do
        for N3 ∈ [N2max, N3max] do
            Caffe    : Train
            for bw ∈ [bwmin, bwmax] do
                Haddoc : Generate DPN
                Caph   : Generate Hardware Description and SystemC Files
                SystemC: Evaluate Accuracy
                Quartus : Evaluate DSP
            end
        end
    end
end
```

As a proof of concept, this method is tested to explore the design space of CNNs for OCR applications. Following the results of the previous section, Three convolutional layers are trained. Each employs (3×3) filters and is followed by ReLU activation and max-pooling.

In this experiment, choice is made to monitor the DSP blocks utilization, and to set, in the most resource-hungry case, a limited number of 5 outputs for the first layer, 10 for the second layer, and 14 for the third layer. This topology offers reasonable classification accuracy on the MNIST database (98.7%) and can be expanded at the price of higher exploration timings.

Moreover, reducing the number of FMs extracted at each layer can be seen as in our case as a pruning of the CNN weights. In this study, the pruning is performed in a « brute force » manner by successively removing the activations of a given layer and re-training the network. However, magnitude-driven [HPTD15] or energy-driven pruning [YCS17] (see sec.4.4.2) can also be considered and integrated into the proposed exploration toolchain.

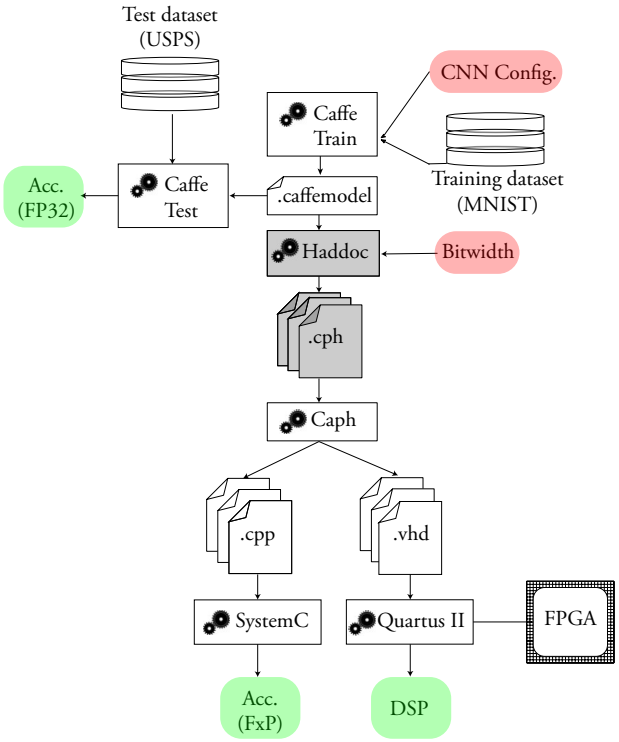


FIGURE 5.9: Design Space Exploration With Haddoc

The other explored parameter is the bit-width b. Weights and FMs can have a maximum size of 7 bits, which results in a -0.1% loss of accuracy on MNIST when compared to a floating-point reference. In contrast, a minimum of 3 bits were used to represent the parameters, which was the weakest precision usable to have acceptable classification rates. With these design space «boundaries» , 76 networks, with 5 different data representation are explored (A total of 380 combinations). SystemC processed the 1500 entries of USPS a rate of 66.6 classifications per second while the synthesis tool takes an average of 6 minutes to estimate the number of DSP blocks required. Thus, a configuration is explored every 8.5 minutes[12].

TABLE 5.4: Remarkable Configurations: C1 is the most efficient, C2 has the lowest hardware utilization and C3 is the more accurate

Conf.	N_1	N_2	N_3	b	$\mathcal{A}$ USPS	$\mathcal{A}$ MNIST	DSP	Ratio
C1	4	6	8	5	64.8%	98.3 %	161	0,40%
C2	3	5	7	3	48.7%	82.4 %	140	0,34%
C3	4	8	12	7	73.2%	99.7 %	428	0.17%

5.4.2.2 Implementation Results

A few remarkable configurations –among the 350 explored networks– are reported in tab 5.4. The most efficient one is C1: it delivers the best trade-off between hardware cost and classification accuracy. Table 5.5 gives the post-fitting reports of C1. This mapping uses 161 of the 1963 DSP blocks available in the Stratix V device and 20 % of the available logic fabric. It also maintains a classification accuracy of 64.8% on USPS at a rate of 57.93 MHz per pixel, which corresponds to 31 classifications per second on $720p$ video stream.

[12]Results achieved with an Intel i7-4700K CPU on 32x32 Frames

Therefore, C1 could be implemented on a lower-end device with less logic resources and DSP blocks.

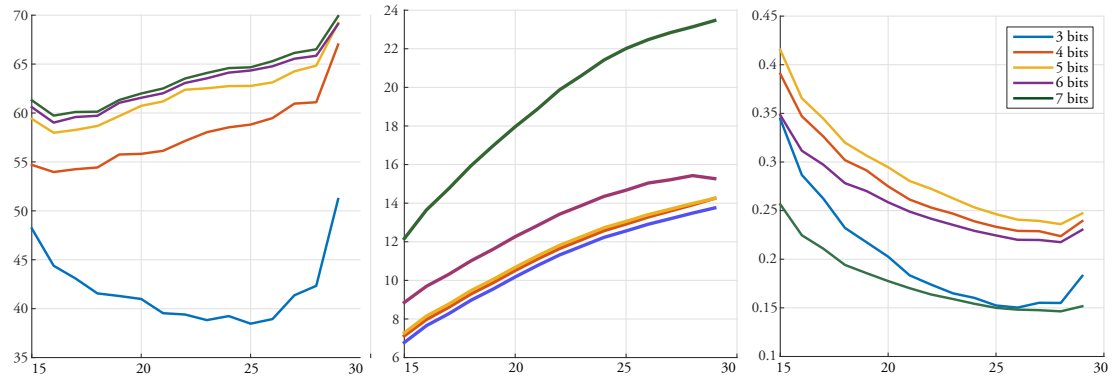
C2 is the configuration with the lowest number of neurons and data representation size, thus, its mapping has the lowest DSP usage. Finally, C3 is the configuration that delivers the greatest classification accuracy. This configuration is among the ones with the highest number of FMs and bit-widths considering the design space boundaries established above.

TABLE 5.5: C1 implementation features on a Stratix-V device

Logic utilization (in ALMs)	53,779 / 262,400 (20 %)
Total RAM Blocks	109 / 2,567 (4 %)
Total DSP Blocks	161/1963 (8 %)
Frequency	57.93 MHz
Classification rate(on 720p streams)	31 FPS

5.4.2.3 Exploration Results

In this section, the performances of the explored configurations are analyzed. On one hand, when only network topology is explored while the bit-width is maintained constant, both of classification accuracy and DSP utilization increase linearly with the number of feature maps as shown in figures 5.10a and 5.10b.



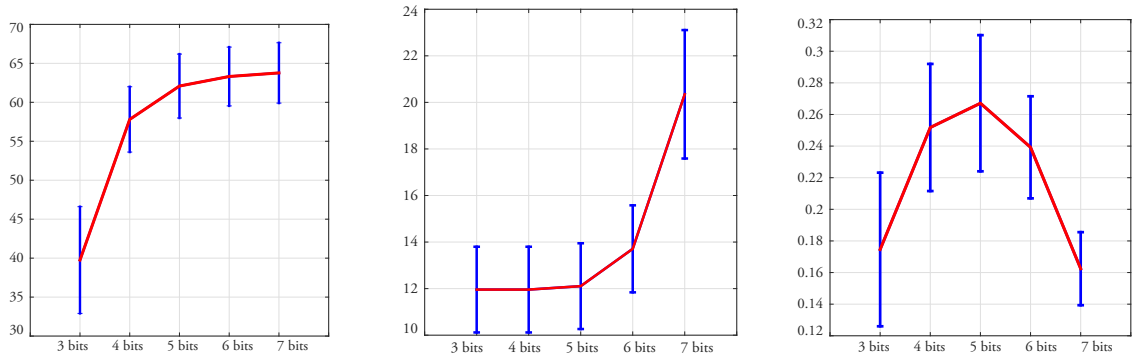
(A) Accuracy (%) (B) DSP Blocks (%) (C) Acc/DSP Ratio(%)

FIGURE 5.10: Design space exploration of CNN topologies. Values on the abscissa axis reports the total Number of FMs extracted

On the other hand, and to monitor the effects of numerical rounding on the efficiency of the FPGA mapping, figures 5.11a and 5.11b are plotted. The first ones illustrates how the mean classification accuracy grows with numerical precision before saturating at a bit-width of 5, meaning that a precision of 5 bits is sufficient enough to maintain tolerable classification and generalization performance for the studied OCR application.

The second figure 5.11b depicts how the DSP utilization grows quadratically with the bit-width. More particularly, it shows that using compact bit-widths lower than 5 bits brings little to no savings in terms of DSP utilization. This is due to the fact that current FPGAs and synthesis tools can not natively pack such low bit-width computations in a single DSP block, and motivates the use of logic resources to map this kind of computations[13]. Alternatively, one can consider that a 5 bit precision delivers –for this application– the best trade-off between classification accuracy and resource utilization, as highlighted in figure 5.10c.

[13]Another solution would be to «manually» force a DSP block to pack 4 bits operands, as proposed in [WZY17]



(A) Accuracy (%) (B) DSP Blocks (%) (C) Acc./DSP Ratio(%)

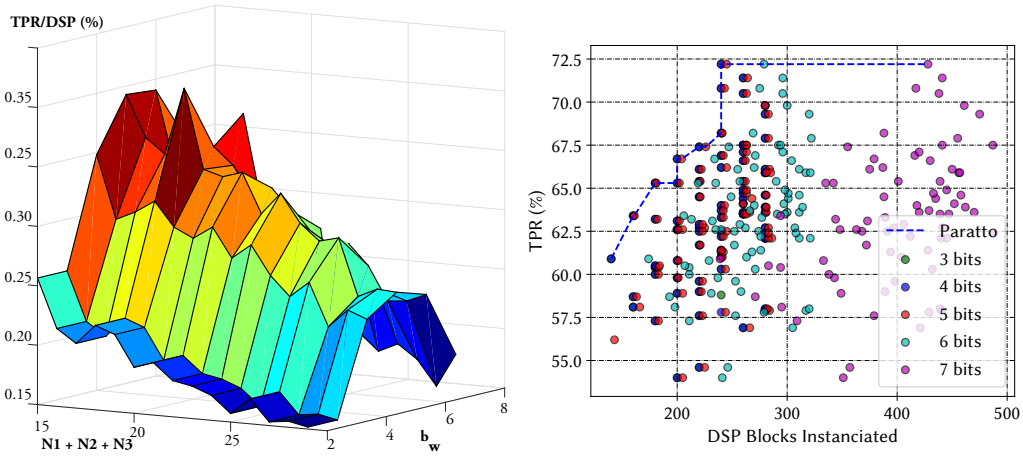
FIGURE 5.11: Design space exploration of parameter bit-widths

5.4.3 Holistic Exploration Results

To understand the joint effects of pruning and quantization on a given FPGA mapping, figures 5.12a and 5.12b reports the performance and resource utilization of all the explored configurations.

The first figure considers the ratio between accuracy and DSP utilization of the implementations. It shows that the most efficient configurations can be obtained after exploring *both* the topologies and data representations. Particularly, the presence of maximums in this plot suggests that using optimization algorithms such gradient descent can be considered to drive the exploration[14], which could significantly fasten the process.

In Fig. 5.12b, efficiency in terms of Pareto-optimal is considered. Similarly, the most efficient configurations (i.e the ones laying on the Pareto front plotted in dashed blue), have a significantly different bit-width and topology. Finally, the same plot shows how the proposed exploration flow can be used to derive CNNs configurations that respect given constraints in terms of accuracy or DSPs block utilization.



(A) Ratio between accuracy and DSP Utilization (B) Paretto Fronts

FIGURE 5.12: Results of the Holistic Design Space Exploration

[14] In contrast with the proposed approach which relies on a brute force exploration

5.5 Multi-view CNNs

To improve the efficiency of embedded deep learning on reconfigurable hardware, the previous sections first considered the models of computation, then the accuracy to resource utilization trade-off through approximate computing. In all the studies, the deep learning algorithm operates on a *unique* input, generally a video stream of a given scene or a single picture if a given object.

But what happens when CNNs operate on multiple inputs, typically, multiple images giving different perspectives of a given object? In this section, the so-called Multi-View CNNs (MVCNNs) are detailed, focusing on the efficiency improvements they can bring. In fact, the concept of MVCNNs is already addressed in the literature and contribution of this work is to adapt it to the smart camera context. To do so, the first subsection investigates the feasibility of MVCNNs for a small number of perspectives while the second subsection to reduces the computational workload of MVCNNs through graph adjustments.

5.5.1 Related Work

The concept of MVCNN was introduced by Su *et al.* [SMKLM15] for 3D-shape recognition applications. In this work, authors design a CNN that inputs 12 perspectives of a given object in order to retrieve its class and 3D-shape, as illustrated in figure 5.13.

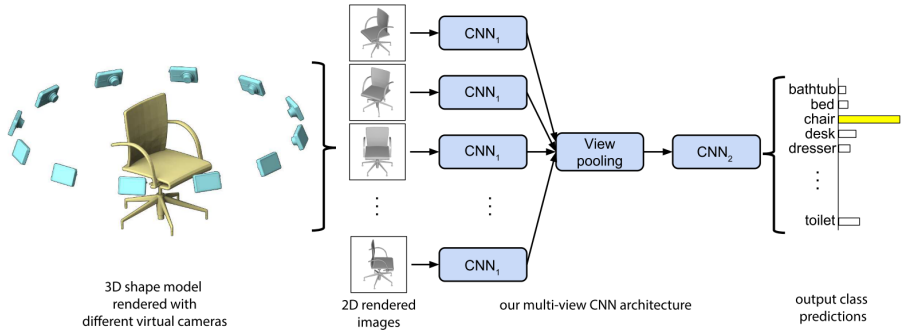


FIGURE 5.13: Multi-view CNN for 3D Shape Recognition. Image from [SMKLM15]

As depicted by the same figure, a *view-pooling* layer is introduced in this kind of networks to fuse the feature maps computed from different perspectives. This layer is quite similar to max-pooling layers (see sec.2.2.1) except it returns the maximum value of the feature maps across the number of views and not across a given neighborhood[15].

The computation of this layer is detailed in equation 5.5, where the first dimension of tensor $\boldsymbol{X}^{VP}$ is no longer the batch-size B, but the number of views n_v. Note that in this section, the concept of batch size is replaced by the number of views for clarity purposes, meaning that feature maps involved before the application of view-pooling have a dimension $(n_v \times C \times W \times H)$, and the feature maps post view-pooling layers have a dimension $(1 \times C \times W \times H)$

$$\begin{aligned} &\forall \{c,h,w\} \in [1,C] \times [1,H] \times [1,W] \\ &\boldsymbol{Y}^{\text{VP}}[0,c,h,w] = \max_{v \in [1:n_v]} \left(\boldsymbol{X}^{\text{VP}}[v,c,h,w] \right) \end{aligned} \tag{5.5}$$

[15] as it is the case in max-pool layers

Authors train and evaluate the accuracy of the proposed MVCNN on ModelNet40. ModelNet40 is a database of 12K three-dimensional 3D models from 40 common categories, few of them depicted in figure 5.14. For this dataset, authors report the following results:

- A baseline implementation of AlexNet, trained on ImageNet1K, achieves 83.0 % of Top1 accuracy on the test set of ModelNet.
- When fine-tuned on the training set of ModelNet, Alexnet, with a single perspective, delivers 85.1% Top1 Accuracy.
- This Top1 accuracy grows to 88.6% when averaging the predictions of twelve AlexNets, each operating separately on a perspective.
- The proposed MVCNNs with view-pooling (trained on ImageNet1K without fine-tuning) delviers an accuracy of 88.1% on the test set of ModelNet. This accuracy grows to 88.9% when fine-tuning the multi-view network on the ModelNet training set.

These results show how the accuracy of CNNs classification tasks can be further improved when considering multiple views of the image. This is even more true according to our experiments, where the ResNet50 CNN, introduced two years after the MVCNNs results were published, delivers a 87.1% Top1 accuracy on ModelNet with fine-tuning. In this particular case, the accuracy improvement brought by multi-view (3.8%) is higher than the improvements brought by topology optimization (2%).

FIGURE 5.14: Examples of entries in the ModelNet40 database

5.5.2 Efficiency Improvements

The previously discussed work focuses only on improving the performance of 3D-shape recognition and reconstitution, giving no interest to efficiency or computational workload.

In fact, before the view-pooling layer is involved, all the layers are replicated across the number of views. Consequently, the studied « MVCNN AlexNet » with $n_v = 12$ views[16] replicates the first layers n_v times, causing the computational workload to increase by a factor of $\times n_v$. As a result, the proposed network increases the computations from $0.66GMAC$ to $2GMAC$ in order to deliver 3.8% more accuracy.

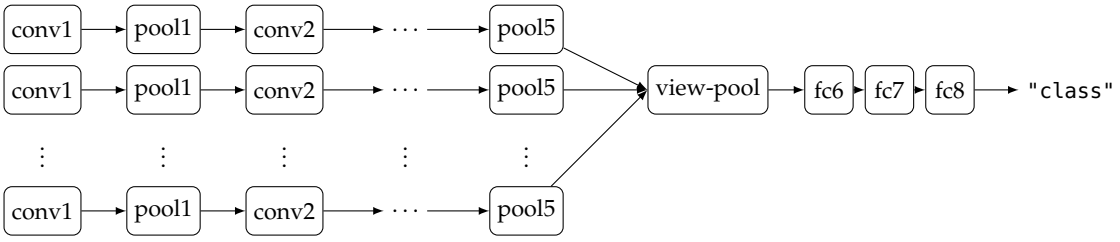


FIGURE 5.15: Graph of a Multi-view Alexnet as proposed in [SMKLM15]

In the context of a smart camera, such multi-view CNNs are not adequate solutions because, mainly for two reasons:

[16] where the view-pool layer is placed just before the FC layers

- First, the number of perspectives ($n_v = 12$), and their arrangement (360° around the object) is not a «practical» design solution.
- Second, the workload of MVCNN –as described in [SMKLM15]– clearly exceeds the computational capabilities of smart camera nodes.

The first advocated approach to lower the computational workload is to reduce the number of views. Indeed, implementing an MVCNN with a limited number of views, for instance three, from slightly different angles is an acceptable design solution for a multi-view smart camera with three image sensors. Moreover, and since the computational workload grows linearly with the number of views, an MVCNN with 3 views is expected to involve $\times 12/3 = 4$ times less computations *vs.* a CNN with twelve views .

To check if MVCNNs still maintain their accuracy improvements, the experiments of [SMKLM15] are re-conducted on Alexnet (referred as MVA), with a variable number of views. The accuracy of the derived networks is then compared to CNNs and MVCNNs where the number of FMs per layer is divided by two (compared to the baseline implementation of Alexnet). This is done by adapting the Caffe implementation of MVCNNs[17] so that it can support a number of views n_v ranging from 1 to 12. In all the considered networks, training and inference are achieved on multiple *adjacent* views of a given object of the ModelNet40 dataset[18].

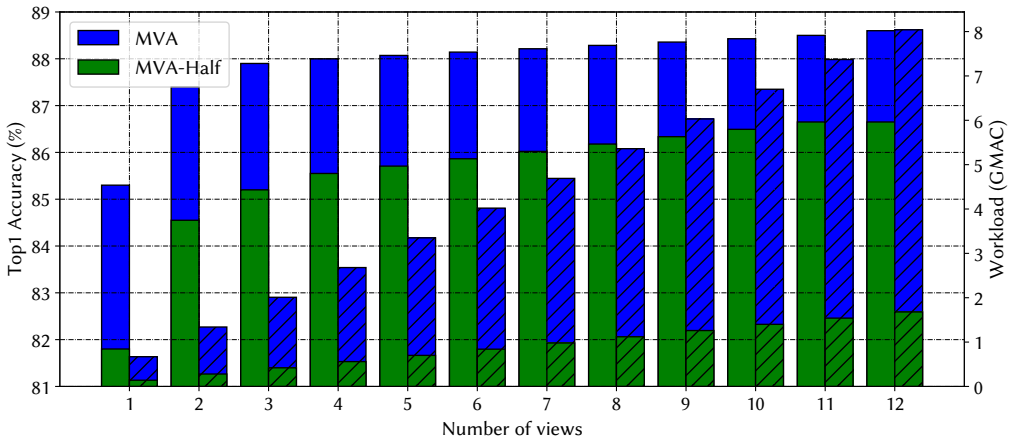


FIGURE 5.16: Top1 Accuracy and Computational Workload of Muti-view CNNs. The hashed parts represent the computational Workload

The results of these experiments are reported in figure 5.16 where the workload is plotted in hashed bars on the right axis and Top1 accuracy is plotted in a plain color on the left axis. The blue columns refer to Multi-View AlexNets while the green columns refer to MVCNNs with a halved number of features maps. In these plots, on may note the following trends:

- An MVCNN with three adjacent perspectives delivers +2.6% more accuracy than a single view CNN, which is 0.8% better that a ResNet50 with 80% less computations.
- When compared to a twelve-view MVCNN, a three-view MVCNN delivers −0.7% less accuracy, but is 4× less computationally intensive.
- In the single view case, halvening the number of FMs (MVA1-Half) decreases the accuracy by 3.5% while in the multi view case, it decreases accuracy by −2.0% (MVA12-Half). This suggests that providing additional perspectives to a CNN compensates the accuracy loss due to pruning.

[17] https://github.com/suhangpro/mvcnn/tree/master/caffe

[18] Acknowledgments to Gautier Claisse for providing the GTX 1080 GPU used for training the MVCNNs

To sum up, the so-called MVA-Half3 with 3 views network has similar accuracy to the original single view AlexNet while involving 0.09G less MAC operations. To decrease the computational workload even more, the second contribution of this work is the place view-pooling layer just after the `pool1` layer of the network, as depicted in figure 5.17.

In this case, only the computations of the first layer `conv1` are duplicated, greatly reducing the workload of MVCNNs. This comes at the price of a loss of -0.6% (*resp.* -0.8%) in the classification accuracy of MVA3 (*resp.* MVA3-Half).

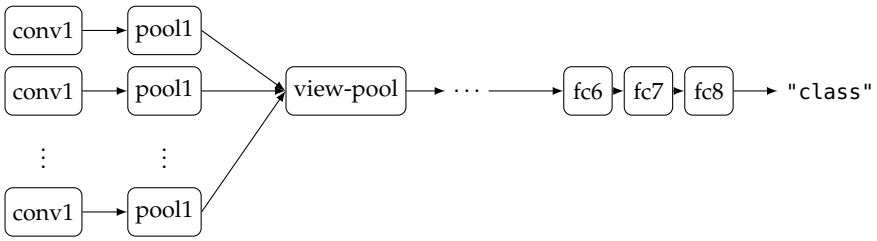


FIGURE 5.17: Multi-view Alexnet with view-pooling after the pool1 layer

The results of all the MVCNN investigations are summarized in Fig.5.18. As demonstrated, multi-view networks improve the computational efficiency of CNNs by increasing their accuracy rates at the price of a low computational overhead. Particularly, these improvements outperform the enhancements brought by topology optimization alone. In addition, the former experiments hint that MVCNNs can be more resilient towards pruning when compared to conventional CNNs.

However, note that all the results reported in this section are achieved *for ModelNet dataset*, which motivates further investigations related to the generalization performance of MVCNNs, and their implementation of multi-view smart camera nodes.

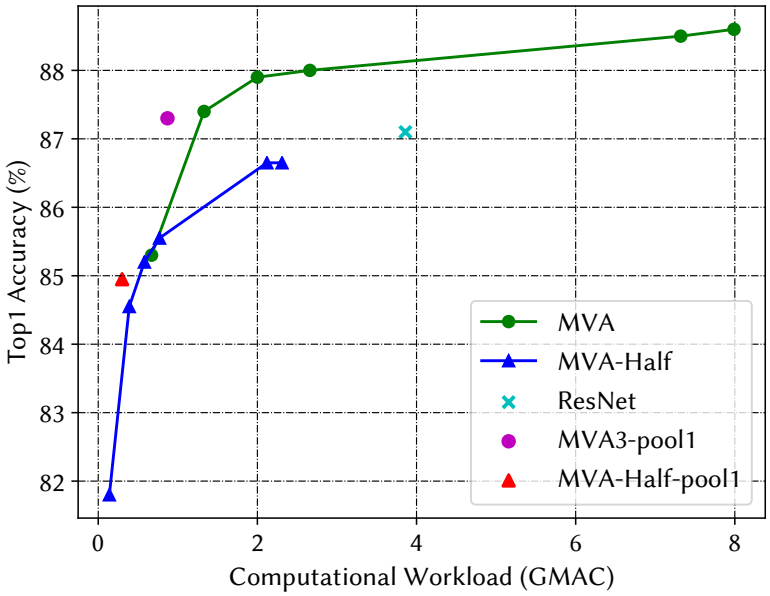


FIGURE 5.18: Accuracy to Workload trade-off for MVCNNs

5.6 Conclusions and perspectives

This chapter studied model-based methods that aim at improving the efficiency of CNN mappings on FPGAs. First, the nuances between the Von Neumann and the dataflow paradigms have been presented. It has been pointed-out that two paradigms have to be considered according to the workload and nature of a CNN layer.

The second section of this chapter focused on the hardware mapping of CNNs on FPGAs with the dataflow MoC. A first tool-flow, built-upon the CAPH language, have been proposed. As pointed-out, relying on HLS greatly improves the design productivity, but comes at the price of a large overhead, especially in terms on logic utilization. This problem will be addressed in the next chapter.

The proposed tool-flow have then been exploited to explore the design space of CNN hyper-parameters. This study focused on the depth, the quantization scheme, and the CNN topology, evaluating their impact on the mappings' efficiency. The results advocated the necessity of a holistic approach, where all the parameters are tweaked to optimize the mapping according to the considered constraints.

Finally, a novel CNN-hyper parameter have been investigated: the number of views. Indeed, the last section demonstrated how the CNN efficiency can be further improved when processing multiple perspectives of a given object, opening future research directions in the area of multi-view smart camera nodes.

Chapter 6

Architectural Optimizations of CNN Mappings on FPGAs

In the previous chapter, the study focused on *coarse grain* optimizations that occur at the level of the CNN model and its hyper-parameters. With this aim in mind, a first tool-flow have been proposed, and relied on the CAPH HLS tool to fasten the hardware generation process and automate the design space exploration.

These hardware architectures, derived by CAPH, constitute the baseline contribution of this thesis. However, they suffer from:

- A large overhead in the resource utilization, particularly in terms of logic resources. This overhead prevents numerous CNN networks layers from a DHM implementation, especially in resource-constrained FPGAs.
- An impossibility to *fine-tune* the CNN actors and perform low-level tweaks to the architecture.

To overcome the shortcomings listed above, the next studies by-pass the CAPH layer and rely on a full RTL description of CNNs. This approach sacrifices the productivity of HLS in favour of flexibility in optimizing *low grain* architectural parameters. The main objective of these optimizations is to reduce the resource utilization, especially the logic fabric allocated to convolution layers. Indeed, when looking at the experiments of sec. 5.4.1, the number of ALMs is the first resource that prevents direct hardware mapping.

With this objective in mind, low-level « tactics » to directly map CNN graphs on « embedded FPGAs » are given. First, the problem of the FIFO overhead is addressed. Then, a first optimization of the convolution engines is given. These two tactics significantly improve the utilization of memory blocks and registers.

The third section goes further in the fine-grain optimization process, and points-out how Single Constant Multiplication (SCM) greatly reduces the resource utilization of convolution actors in a given mapping. Section 4 focuses on the adder parts and studies the trade-off between resource utilization and computational throughout. Finally, the last sections leverage on the proposed tactics to derive an improved version of the HADDOC tool. In this version, resource utilization, throughput, and modeling performance of CNN implementations can be tweaked to meet with the imposed constraints.

6.1 FIFO channels in Dataflow Inferred CNNs

6.1.1 FIFO channels in the Dataflow paradigm

In the dataflow model, the execution is purely *data-driven* as each actor is trigged only by the availability of input operands. It is thus critical to make sure that these operands are perfectly synchronized, and are made available to the actor at the correct clock cycle.

To depict this concept, lets consider the dataflow graph of figure 6.1. This example involves three actors to compute $r_i = d_i + a_i - b_i * c_i$, and illustrates how streams `a` and `d` have to be delayed by respectively one and two clock cycles to correctly perform the previous operation. The corresponding dataflow architecture of this graph –such the one derived by CAPH– implements theses delays by means of two First-In First-Out (FIFO) channels.

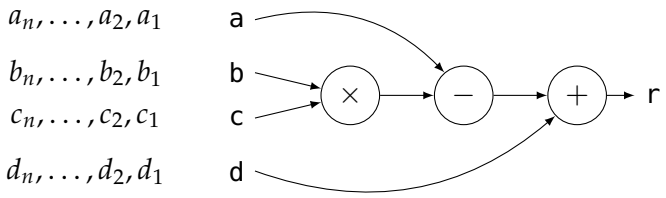
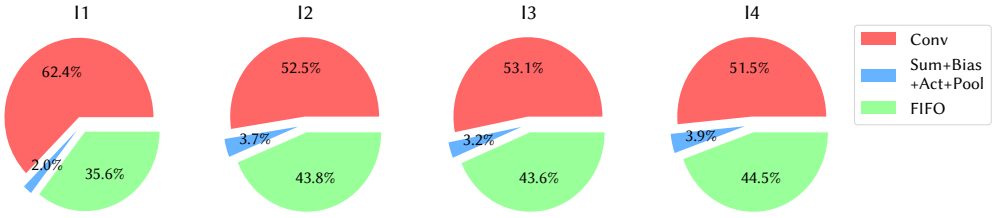
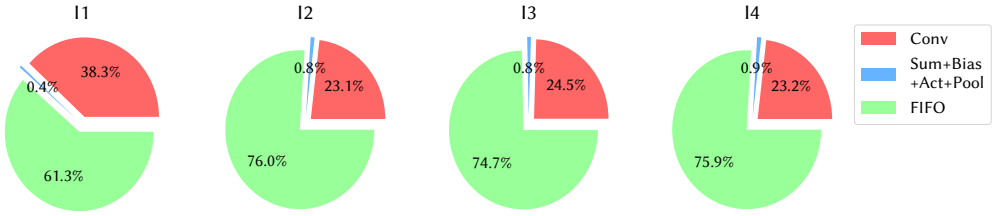


FIGURE 6.1: Example of a dataflow graph: streams a and d are respectively delayed by one and two clock cycles by means of FIFO channels

The former concept also holds true for dataflow inferred CNNs where FIFO channels are rooted between each dependent actors. However, because of the large number of actors involved in CNN inference, FIFOs result in large resource utilization. This overhead is illustrated in Fig.6.2, where FIFO channels use up to 44% of the logic fabric and 76% of the registers allocated to the mappings. Resource utilization grows even more with the number of layers, making the implantation of some deep CNNs impossible on current embedded FPGA platforms.



(A) ALMs



(B) Register

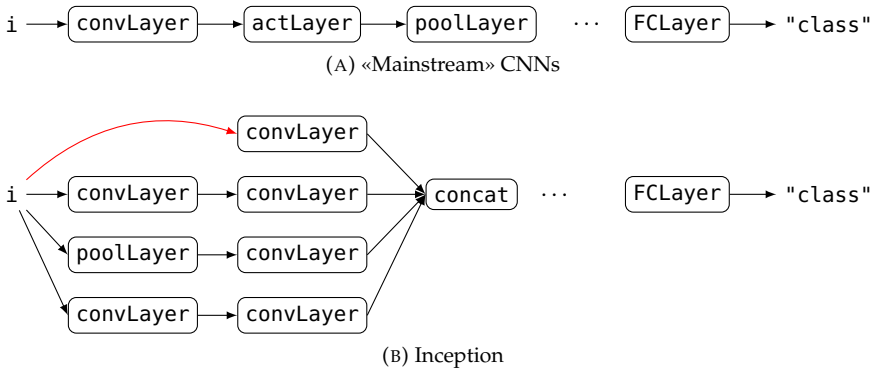
FIGURE 6.2: Resource utilization per actor for the explored LeNet implementations

6.1.2 All we need is signals!

When it comes to the inference of CNNs, most of topologies correspond to dataflow graphs where the tokens are « naturally » synchronized. Indeed, a large number of CNN

layers showcase a regular structure wherein tokens (i.e feature maps) do not bypass any stage.

This structure is illustrated in figure 6.3a, and depicts the DPNs generated from «mainstream» CNNs such LeNet, AlexNet, VGG or DarkNet. In these networks, stream synchronization occurs by itself given the nature of the graphs. As a result, FIFO channels are **not** needed in these networks, an can thus be replaced by *direct signals*, which require much less resources to be mapped.



(A) «Mainstream» CNNs

(B) Inception

FIGURE 6.3: Patterns Involved in the DPN representation of CNNs.

Note that for some networks like GoogleNet or ResNets, the presence of inception modules and shortcut connections (see sec. 2.2.2) prevents from entirely removing the FIFOs. This is illustrated in fig. 6.3b where the red edge of the graph have to be delayed. This delay is explicitly formulated by instantiating a FIFO with the appropriate depth.

6.1.3 Validation

To study the effect of FIFO removal on FPGA mappings, the exploration of sec. 5.4.1 is re-conducted using FIFOs and direct signals. Table 6.1 reports the resource utilization of LeNet networks before and after this replacement. It can first be noticed that number of allocated registers decreases by respectively 18%, 43.6%, 43% and 50%. In turn, these savings in terms of registers reduce the amount of inferred logic blocs by 34.4% and 39.4% in the case of $I2$ and $I4$ implementations.

Despite these savings, the other implementations of LeNet ($I1$ and $I4$) still do not fit on the device because of a lack in logic elements or DSP blocks. On a side note, when replacing FIFOs by signals, we notice that the synthesis tool automatically infers the all the available DSPs (1963) then maps the remaining multiplications using the logic fabric. This explains why the number of ALMs grows in the case of $I1$ and $I3$.

TABLE 6.1: Resource Utilization of the previously studied LeNet mappings

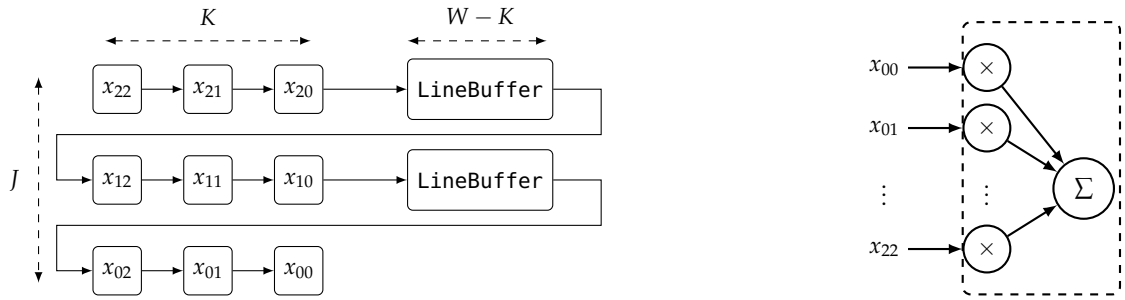
	With FIFOs			With Signals		
	ALM	Reg	DSP	ALM	Reg	DSP
$I1$	334 K	344 K	694	370 K	279 K	985
$I2$	181 K	190 K	1004	118 K	107 K	1004
$I3$	377 K	401 K	2115	432 K	227 K	1963
$I4$	197 K	214 K	1421	119 K	106 K	1421

6.2 Memory-Efficient Window Buffers

6.2.1 Motivation

As studied in chapter 4, the literature provides multiple approaches to accelerate the computation of convolutions. In the proposed DHM approach, the processing of 3D-Convolutions is formulated as a sum of concurrent 2D-Convolutions. Each of the 2D-Convolution engines is fully pipelined and its architecture can be divided into 2 parts:

- A *Window Buffer (WB)*: This component relies on on-chip memories to grant a simultaneous access to the $J \times K$ neighbors of each entry of the input stream (i.e feature map or input image). The architecture of the window buffer, depicted in figure 6.4a, extracts the neighborhood $[x_{00}, \ldots, x_{JK}]$ on the fly at each clock-cycle and concurrently provides them to the second block.
- A *Dot-Product (DP) block*: Multiplies the extracted neighborhood with the convolution kernels then sums the partial products. Since all the input operands (neighborhoods and kernels) are made available thanks to the window buffer, the dot-product operation can be performed in a single clock-cycle by using $J \times K$ multipliers. (See Fig.6.4b).



(A) Windows Buffer (B) Parallel Dot-Product

FIGURE 6.4: Hardware Architecture of a Pipelined 2D-Convolution engine (J=K=3)

Combining the window buffer and the parallel dot-product block fully exploits the intra-kernel parallelism of CNNs discussed in sec. 2.4.2. As a consequence, the structure described in fig. 6.4 performs one 2D-convolution per clock cycle. The DHM approach instantiate this structure $C \times N$ times, highly increasing the computational throughput at the price of resources utilization.

Indeed, scaling this approach to a full convolution layer, which involves hundreds of convolutions, leads to map hundreds of window buffers on the FPGA resources. Particularly, memory resources such registers and on-chip buffers are highly impacted by the high number of window buffers. For a given layer ℓ, the number of window buffers and memory they require can be written as the following equation, where b denotes the width of the buffer (i.e the bit-width of the value it stores):

$$\text{WinBuff}_\ell = C_\ell * N_\ell \tag{6.1}$$

$$\text{MemBits}(\text{WinBuff}_\ell) = b * \text{WinBuff}_\ell * [W_\ell * (J_\ell - 1) + K_\ell - 1] \tag{6.2}$$

$$= b * C_\ell * N_\ell * [W_\ell * (J_\ell - 1) + K_\ell - 1] \tag{6.3}$$

As the number of memory bits grows, the resource utilization of a given mapping (in terms of logic fabric and memory blocks) increases. This is illustrated by figure 6.5 which reports the number of ALM and SRAM blocks allocated to a growing number of

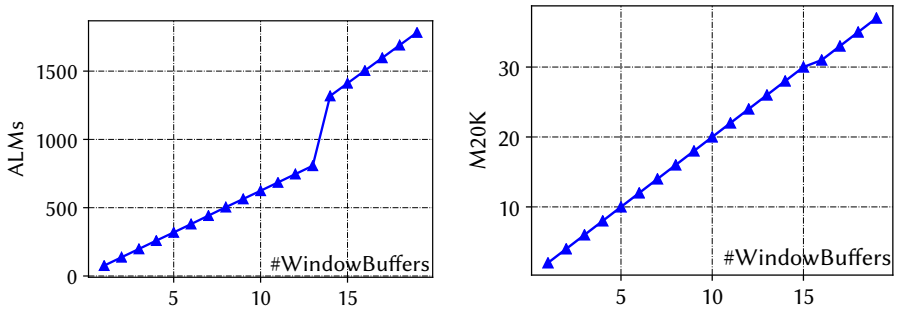


FIGURE 6.5: FPGA Resources instantiated when mapping Window Buffers.

window buffers. This experiment is achieved on an Intel Stratix V `5SGXMABN3` device[1] for $W = 227$ and $J = K = 3$. It can be noticed that the synthesis tool instantiates M20K blocks to map the window buffers, but more importantly, adds a *glue* of logic resources to map these RAM blocks. For a large number of buffers –which is typical in the case of CNNs–, this overhead of ALMs grows, and can represent a non-negligible percentage of the resources. For instance, in our early attempts to map AlexNet first layer, we noticed that 10% of the ALMs were allocated to the windows buffer parts.

6.2.2 Factorizing The Window Buffers

Figure 6.6a illustrates the dataflow graph of convolution layers as proposed previously in this manuscript. The graph also explicitly depicts how 3D-convolutions are implemented as a sum of concurrent 2D-convolutions, each 2D-Convolution being mapped as a combination of a window buffer and dot-product block. Notice in this graph how multiple window buffers operate on *the same input streams*.

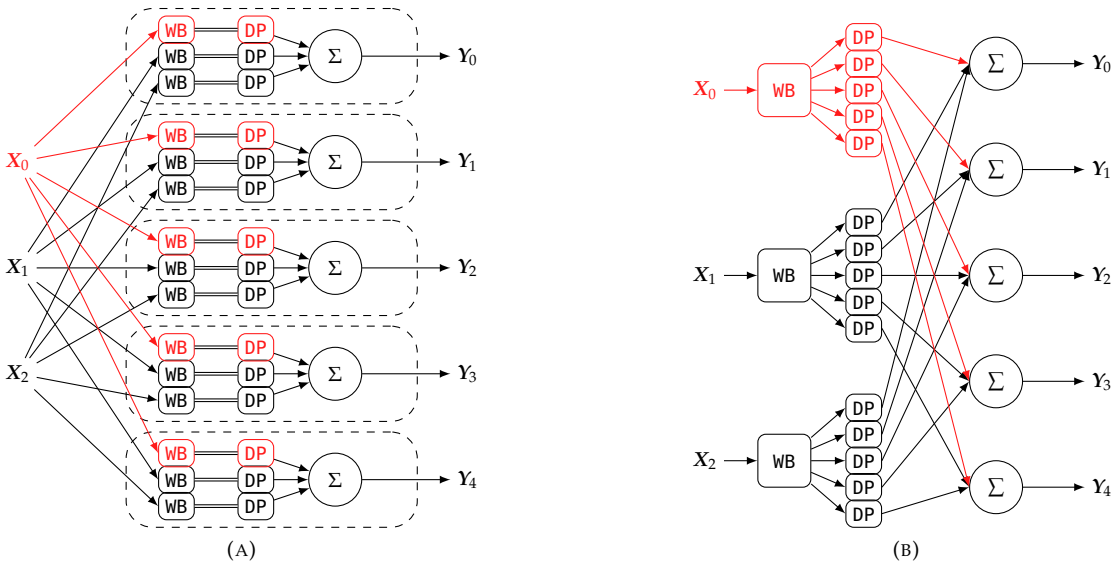


FIGURE 6.6: DPN of a *conv* layer with and without window buffer factorization

Indeed, the previously proposed implementation of 3D convolutions can be considered as an inefficient, since it allocates multiple hardware instances to perform the same task on the same inputs. To address this inefficiency, the graph is reformulated in Fig. 6.6b in a way to *factorize* the window buffers. In this case, the memory footprint of a given layer is divided by a factor N_ℓ. As a consequence, the number of memory needed to map window buffers becomes:

[1]This device is selected because it has the largest number of I/O available

$$\text{MemBits}(\text{WinBuff}_\ell) = b * C_\ell * [W_\ell * (J_\ell - 1) + K_\ell - 1] \tag{6.4}$$

6.2.3 Validation

To demonstrate the improvements of this reformulation, lets consider the case of AlexNet first *conv* layer ($W = 227, N = 96, C = 3, K = 11$) with FMs and weights quantized to $b = 6$ bits. This layer is mapped on a Stratix V `5SGXMABN3` device with and without factorized buffers. Note that in both cases, FIFO channels are replaced by direct signals. The resource utilization in terms of logic and memory blocks of both mappings is reported in table 6.2

TABLE 6.2: Logic Fabric and Memory Resources Allocated to Map Alexnets' first layer

	ALMs	Memory (Bits)	M20K
WinBuffers wo/ factorization	10208	3939840	194
Total `WB+DP+sum`	132039	3939840	194
WinBuffers w/ factorization	929	41040	30
Total `WB+DP+sum`	122760	41040	30

While the «non-factorized» implementation of 3D-convolutions requires $3.9MBits$ of on-chip storage, the factorized version requires $41.04KBits$. This saving in terms of on-chip memory reduces the number of SRAM blocks by factor of $\times 6.4$, lowering the overall logic utilization by 8%.

One can also note that the synthesis tool infers one SRAM block per line-buffer instead of sharing the block across multiple line buffers. In this case, the width of input feature maps, and thus the resolution of input streams, can be increased without affecting the resource utilization, up to an image width of 2560 pixels (size of an M20K SRAM block).

Finally, the savings brought by the proposed factorization are expected to scale with the number of features maps produced by a layer, as illustrated by the projections of figure 6.7[2].

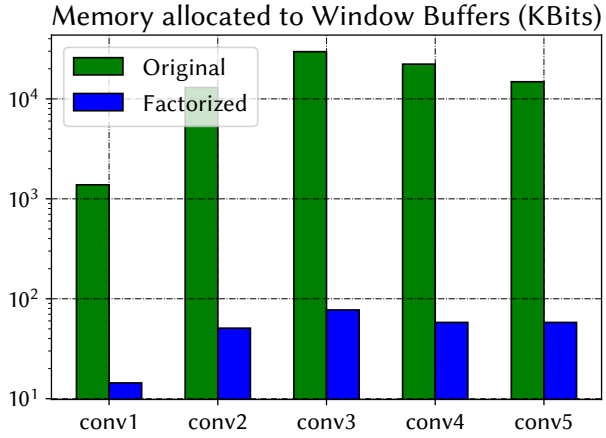


FIGURE 6.7: Theoretical Memory utilization of Window Buffers in AlexNet *conv* layers.

After optimizing the registers and memory blocks, the remaining parts of this chapter focus on the *computational* resources allocated to CNN mappings. The two following sections respectively focus on the *multiplications* than the *accumulations* occuring in 3D convolutions.

[2]Plotted with equations 6.3 and 6.4

6.3 Convolutions with Single Constant Multiplications

6.3.1 Motivation

In all the previous experiments, the direct hardware mapping considered relatively simple CNN layers while targeting high-end FPGA platforms. This is motivated by the need of abundant resources (ALMs and DSPs) for prototyping purposes. However, when addressing state-of-the-art CNN layers, such AlexNet or VGG, the following problem appears: The number of multipliers required to map these layers greatly exceeds the number of multipliers available on current FPGA platforms.

To illustrate this problem, one may consider the case of AlexNets $conv1$ ($N = 96, C = 3, J = K = 11$). To directly map this layer, 34848 multipliers are needed according to equation 5.1. However, and to the best of our knowledge, the « largest » FPGA currently available on the market delivers 5760 hardwired DSP blocks[3].

To address this problem, one can rely on the *DSP packing* capabilities of FPGAs evoked in sec.3.1.3. Recall that a single DSP blocks to either implement:

- One (27×27) bits multiplication, or
- Two independent (18×18) bits multiplications concurrently, or
- Three independent (9×9) bits multiplications concurrently.

However, and even with DSP Packing, the available number of $9bits$ multipliers ($5760 \times 3 = 17280$) is still not enough to map AlexNets first layer with a $9bits$ precision. Moreover, and as pointed-out in sec. 5.4.2.3, DSP packing has its limitations when using very compact bit-widths lower that 5 bits, while the literature provides methods to infer CNNs with much more compact bit-widths.

Finally, the same problem holds true when addressing smaller networks and low-end FPGAs. Table 6.3 reports the requirements of popular CNN layers in terms of multipliers and nuances them with the number of DSP blocks available of common FPGA platforms.

TABLE 6.3: Multipliers in Popular CNN layers

Layer	$\mathcal{R}m$	FPGA	Available Multipliers
LeNet conv1	500	Cyclone V	336
LeNet conv2	2500	SEA6	
VGG conv1-1	1728	Arria 10	4554
VGG conv1-2	36864	GX900	
AlexNet conv1	34848	Stratix 10	17280
AlexNet conv2	307200	GX2800	

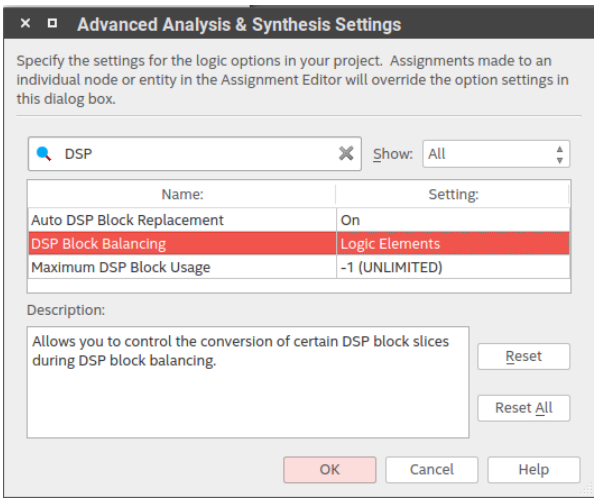
6.3.2 Single Constant Multiplication on FPGA

6.3.2.1 Multiplication with logic resources

As discussed above, the number of multipliers currently available on FPGAs clearly limits the complexity of implementable CNN. To overcome this limitation, the proposed solution is to implement multiplications using *logic resources* instead of DSP blocks. In this case, the resulting circuitry relies on AND gates and of half-adders to perform multiplications [Int04].

[3]Number of multipliers present on an Intel Stratix 10 GX2800 [Int18b]

From a technical side, one can either modify the synthesizer settings (Fig.6.8a), or use *dedicated attributes* in the hardware description of the component (Fig.6.8b) in order to infer logic resources instead of hardwired multipliers



(A) Through the Synthesizer settings (Quartus II)

```
entity LogicMult is
port (
 clk   : in  std_logic;
 x     : in  std_logic_vector(3 downto 0);
 theta : in  std_logic_vector(3 downto 0);
 y     : out std_logic_vector(7 downto 0)
 );
 attribute use_dsp48 : string;
 attribute use_dsp48 of
 LogicMult : entity is "no";
end LogicMult;

architecture RTL of LogicMult is
begin
  process(clk)
  begin
  if (rising_edge(clk)) then
    y <= theta * x;
  end if;
  end process;
end architecture;
```

(B) Through VHDL Attribute

FIGURE 6.8: Implementing multiplications with logic resources in FPGAs

When implementing arithmetic operations with logic resources, the cost of a fixed point multiplier varies as the square of the precision of its operands while the cost of an adders varies as a linear function of the precision [DKT07]. As a consequence, the amount of logic fabric inferred to map a given convolution grows quadratically with the bit-widths of the operands and weights.

As a result, the proposed solution can be applied on extremely compact bit-widths such in QNNs or TTQ (see sec. 4.4.1.4), but scales badly when considering a larger numerical precision that exceeds 4 bits. Indeed, the high amount of logic resources needed in this case prevents from directly mapping a CNN. A possible solution to this problem is discussed next.

6.3.2.2 Constant Multiplies

To reduce the footprint of the mapping, we take advantage of the fact that the convolution weights – and hence one operand of each multiplication – are constants derived from an off-line training stage. These multipliers can thus be specialized to their constants.

In the literature, this method is often referred as Single Constant Multiplication (SCM) [VP07], and has been successfully applied to minimize the footprint of Finite Impulse Response (FIR) filters in signal processing tasks [dDL00, Kon17, Wal17]. More importantly, FPGA synthesis tools employ their own SCM methods to derive area-optimized arithmetic circuits. Indeed, the circuitry of a constant multiplier in is specialized according to the value of its constant multiplicand. The synthesis tool automatically performs low-level optimizations, and more particularly:

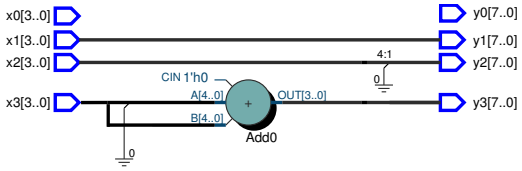
- It removes the circuit in case of a multiplication by 0
- It replace the multiplier by a direct signal in case of a multiplication by 1
- It transforms the multiplier into a shift-register in case of a multiplication by a power of 2.

This can be verified with the example of Fig.6.9, which gives the hardware description of four constant multipliers parametrized by their multiplicands $t0 = 0$, $t1 = 1$, $t2 = 2$, $t3 = 3$.

```
entity ConstMult is
generic(
  t0   : in unsigned(3 downto 0) := "0000";
  t1   : in unsigned(3 downto 0) := "0001";
  t2   : in unsigned(3 downto 0) := "0010";
  t3   : in unsigned(3 downto 0) := "0011"
);
port (
  x0,x1,x2,x3 : in  unsigned(3 downto 0);
  y0,y1,y2,y3 : out unsigned(7 downto 0)
);
end ConstMult;

architecture str of ConstMult is
begin
  y0 <= x0 * theta0;
  y1 <= x1 * theta1;
  y2 <= x2 * theta2;
  y3 <= x3 * theta3;
end architecture;
```

(A) VHDL Description



(B) Generated RTL Representation

FIGURE 6.9: Example of a constant multiplier implementation on an FPGA

One may note how the synthesis tool removes the multiplication circuit in the case of `t0`, and how it directly wires the output to the input when the multiplicand is equal to one. In the case of `t2`, the synthesizer simply shifts the output by one bit. Finally, when the multiplicand takes the values of three, the synthesis tool shift the input by one bit then infers an adder in a way to map the operation $(3 * x)$ as $(2 * x + x)$

In counterpart, the major downside of this approach is that it completely locks the architecture of the accelerator, greatly limiting its flexibility. Relying on the direct hardware mapping with SCM requires to re-synthesise the hardware design whenever the CNN topology or weights are changed.

Still, CNN mappings benefit from a significant reduction in resource utilization thanks to SCM. Indeed, CNN layers have large percentage of zero-valued, one-valued and power-of-two-valued parameters, as summarized in table 6.4. The reduction is even greater when considering the *sparse* CNNs discussed in section 4.4.2, where the sparse weights can represent up to 63% of total convolution kernels. Thus, DHM can remove up to 63% of the multipliers in the FPGA implementation of these networks.

TABLE 6.4: Statistics on convolution kernels in popular CNNs

Network	Alexnet	DeepComp.	SqueezeNet	VGG16	VGG16	VGG16
Layer	conv1	conv1	conv1	conv1-1	conv1-2	conv2-1
Weights	34848	34848	14112	1728	36864	73728
Null(%)	36.84	38.32	22.67	5.56	44.66	57.15
Pow2(%)	26.94	26.21	39.34	18.98	26.25	20.64

6.3.3 Validation

To quantify the impact of SCM, Table 6.5 reports the resource utilization of the previously studied LeNet5-I1 implementation on an embedded Intel Cyclone V `5CGXFC9E7`

device. In this experiment, a 5-bit precision is selected and three multiplication schemes are studied. In the first result, only DSP blocks are used to infer all CNN multiplications. The resulting hardware requires 72× the available resources of the device. The second case features multipliers based on logic elements and requires 3.8× the available logic. Finally, using tailored *constant* multipliers reduces resources by a factor of 8.6×, fitting the CNN accelerator onto an Intel Cyclone V embedded FPGA.

TABLE 6.5: Impact of SCM in the mapping of LeNet5-I1

	DSP-based	Logic-based	SCM
Logic Usage (ALM)	NA	433500 (381%)	50452 (44%)
DSP Block usage	24480 (7159 %)	0 (0%)	0 (0%)

6.4 Accumulation with Pipelined Adders

6.4.1 Motivation

After studying the optimization of the multiplications, this section addresses the problem of additions in CNN mappings.

Recall that with DHM, the computation of dot-products is fully unrolled. Consequently, the $(C \times J \times K)$ partial products resulting from each 3D convolution have to be *accumulated* in a parallel fashion. This concurrent accumulation is achieved by means of an adder with $n_{opd} = CJK$ inputs, referred in this manuscript as a Multiple Operand Adder (MOA).

By default, synthesis tools chain multiple binary adders[4] to generate a MOA, each binary adder being generated using combinatorial logic. In its most naive implementation, the architecture of a MOA can be described using the VHDL loop given in Fig. 6.10. The hardware generated from the former description has a « cascaded » structure as illustrated in the same figure.

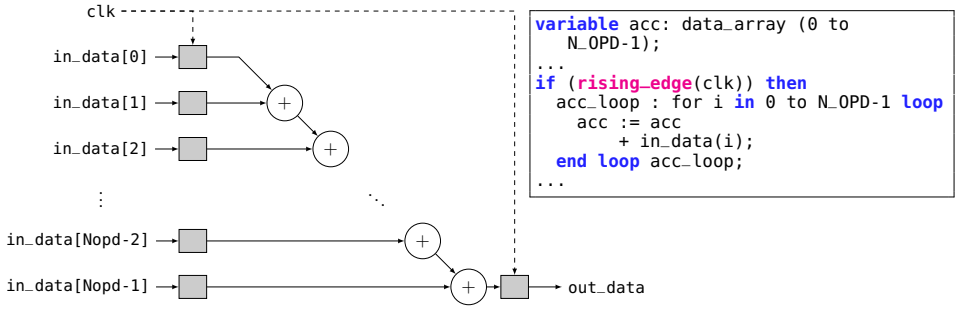


FIGURE 6.10: Implementation a MOA by cascading binary adders

Fig. 6.10 shows how the cascaded structure infers $n_{opd} - 1$ binary adders to implement a n_{opd}-input MOA. Consequently, the resource utilization in expected to grow linearly with n_{opd}.

The same figure also shows that the cascaded MOA generates a critical path which is function of $n_{opd} - 1$. Thus, the operating frequency is expected to decrease as $O(1/n_{opd})$. These expectations are confirmed by Fig.6.11 which reports the performance of the cascaded MOA and its resource utilization for a variable number of inputs[5]. One may note how the maximal operating frequency of the design decreases with the number of inputs,

[4]Binary adders refer to adders with TWO operands and NOT adders with a 1-bit operand

[5]Experiments performed on a an Intel 5CSEMA5F31 device. The maximum frequency is constrained to 50 MHz.

which results in a low computational throughput when the number of operands becomes large.

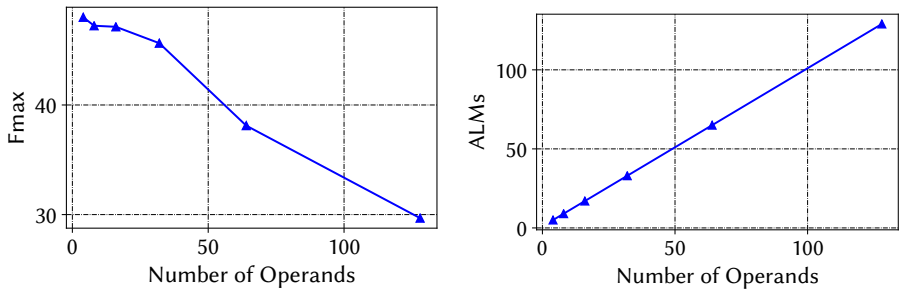


FIGURE 6.11: Performance of the «cascaded» MOA for a variable number of inputs

The problem is that for state-of-the-art CNNs, C, J, K, and thus the number of operands of MOAs is indeed important. In the case of AlexNet, adders can input up to 1774 operands. Consequently, the MOA architecture described in fig.6.10 bottlenecks the mapping, greatly limiting the operating frequency and computational throughput.

6.4.2 Pipelined Adders

A first solution to the problem of low operating frequencies is to *Pipeline* the MOAs. This method places a register after *each* binary adder, and can be simply implemented by using VHDL signals instead of variables, as highlighted in Fig. 6.12.

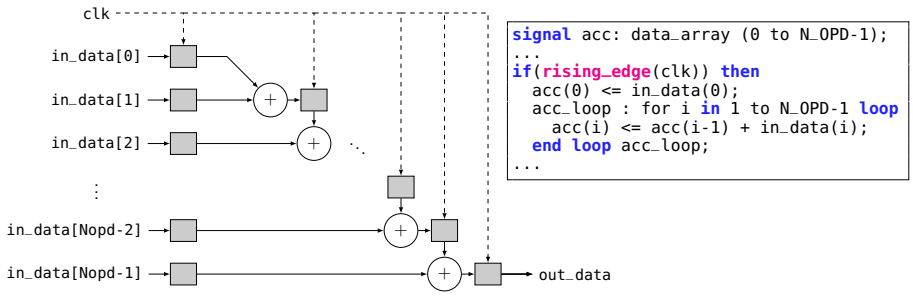


FIGURE 6.12: Hardware Architecture of a Pipelined MOA. Registers are interleaved between each adder stage

However, the drawback of pipeline is the additional *resource utilization* it generates. Figure 6.13 compares the resources utilizations and the maximum frequencies of the pipelined and unpipelined adders. As expected, the pipeline requires a number of registers that grows linearly with the number of operands. In the selected device, these registers are implemented by means of ALMs, which explains why the logic resources increase twice rapidly than in the unpipelined implementation. In turn, the pipelined adder maintains a constant operating frequency (arround 50 MHz) independently from the number of operands. Indeed, the pipeline trades resource efficiency for throughput improvements .

6.4.3 Adder Trees

In order to improve the operating frequency, another solution would be to reduce the critical path. This can be achieved using the *tree* structure illustrated in figure 6.14a.

The tree MOA generates a number of stages, each of them summing the inputs two by two. Consequently, the number of inputs is halfened after each stage, until the addition is completed. For an n_{opd}-MOA, the total number of stages required to complete the

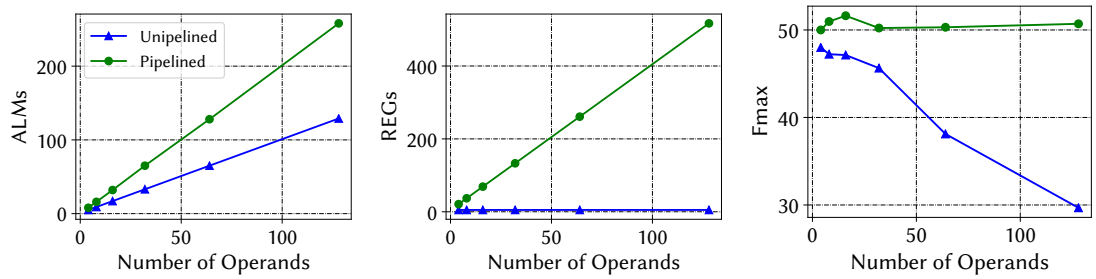
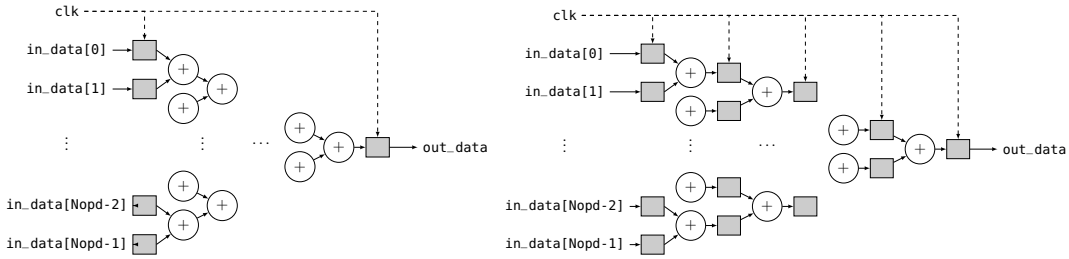


FIGURE 6.13: Performance of the pipelined «cascaded» adder



(A) unpipelined (B) Pipelined

FIGURE 6.14: Implementation of a MOA with the tree structure

addition is $\log_2(n_{opd})$. As a result, the critical path varies with of $\log_2(n_{opd})$ and the maximum frequency with $O(1/\log_2(n_{opd}))$.

$$\text{Number of Adders} = n_{opd} - 1 \tag{6.5}$$

$$\text{Depth of Cascade MOA} = n_{opd} - 1 \tag{6.6}$$

$$\text{Depth of Tree MOA} = ceil(\log_2(n_{opd})) \tag{6.7}$$

In addition, and similarly to the cascade structure, the adder tree can also be pipelined to increase its operating frequency (see Fig.6.14b). Figure 6.15 compares the resources utilizations and operating frequencies of pipelined and unpipelined adder *trees*.

Similarly to the cascaded structure, the pipeline results in an overhead in the resource utilization, but maintains the operating frequency at a constant level ($\sim 50MHz$). Of course, without the pipeline, the frequency decreases with the number of operands. Figure 6.16 summarizes the performances of tree and cascade structures. It particularly shows how the tree adder outperforms the cascade structure when working with resource constrained devices. Indeed, the tree structure delivers a better operating frequency for similar resource utilization in this case.

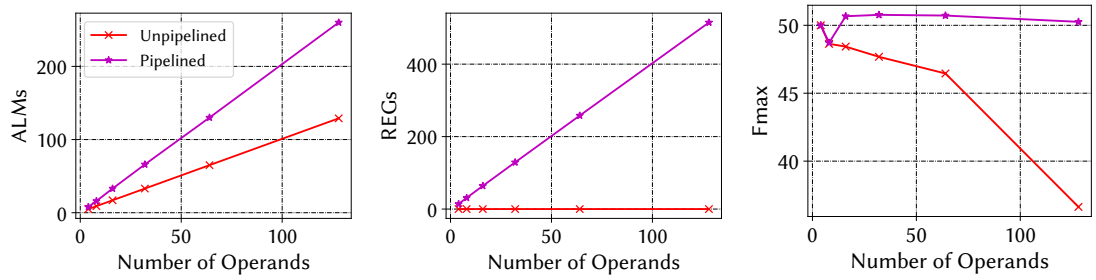


FIGURE 6.15: Performance of a tree MOA Structure

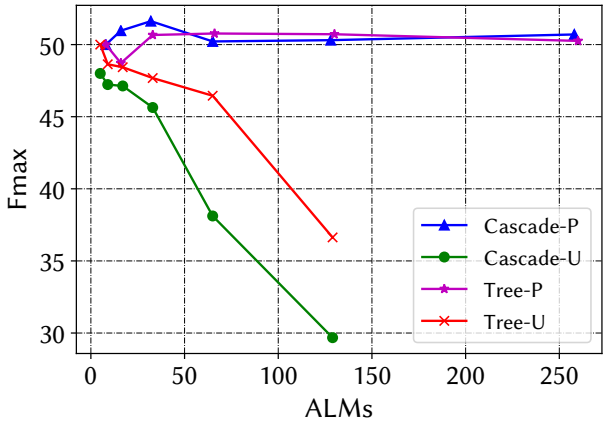


FIGURE 6.16: Frequency and hardware utilization of the studied adders

6.4.4 Resource to Throughput Trade-off

The trade-off between ALM utilization and Fmax can be controlled further by partially pipelining the hardware architecture. For the tree structure, this approach inserts registers *periodically* between a variable amount of adder stages. To control the periodicity (i.e the number of stages between two successive registers), the s_{reg} parameter is introduced, and ranges from 1 to ceil$(\log_2(n_{opd}))$:

- When $s_{reg} = 1$, the adder is fully pipelined (as in fig.6.14b) and both the frequency and resource utilization are expected to be at their maximum.
- By contrast, when $s_{reg} = \text{ceil}(\log_2(n_{opd}))$, no registers are inserted between adders (such in fig.6.14a), and consequently, the frequency and resource utilization are expected to be at their minimum.

To study the impact of the s_{reg} parameter, the following experiment considers a tree MOA of 128 inputs of 4 bits. Figures 6.17 respectively reports the evolution of ALMs, registers and *Fmax* with a variable s_{reg}. As expected, resource utilization and operating frequency decrease with the stride of register. However, in our case of a 128-input adder, a value of $s_{reg} = 2$ delivers the best ratio between the operating frequency and the resource utilization, as pointed-out in Fig.6.18.

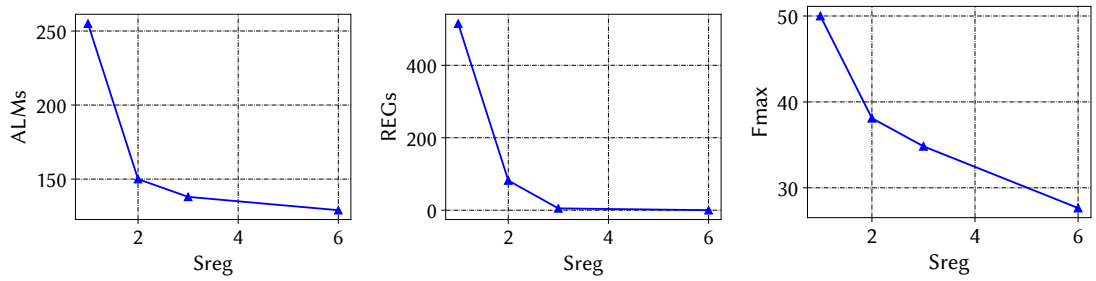


FIGURE 6.17: Impact of Pipeline on Resources Utilization and frequency of MOAs

More generally, the previous studies have demonstrated that DHM can balance between resource utilization and computational throughput. This trade-off is selected according to the resource available on a given FPGA platform, the workload of the CNN, and the real-time constraints.

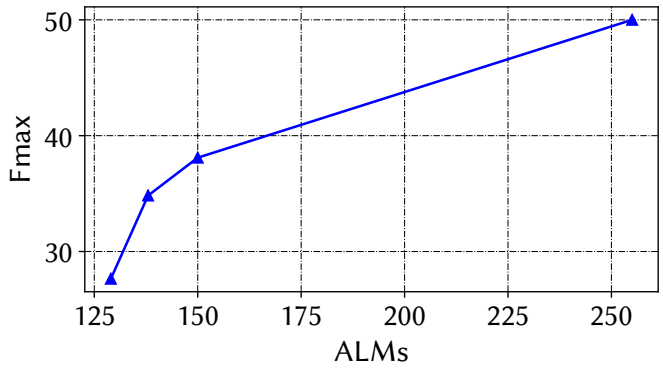


FIGURE 6.18: Frequency to ALM trade-off using the s_{reg} parameter

6.5 Implementation Results

After discussing tactics optimizing the low-grain components of CNN mappings, this section reports the implementation results with HADDOC2. When compared to the first version, the major difference in HADDOC2 is the description of CNN actors, and the extraction of the CNN weights.

- Instead of relying on the CAPH HLS tool, Haddoc2 uses a set of components written in VHDL-2008 to directly transcript a CNN Network onto a VHDL top-level file.
- To map multiplications as SCMs, CNN weights are rounded than *hard-coded* as *generic* VHDL parameters in a configuration file.

LISTING 6.1: Caffe description

```
name: "LeNet"
...
layer {
  name: "conv2"
  type: "Convolution"
  bottom: "pool1"
  top: "conv2"
  param {
    lr_mult: 1
  }
  param {
    lr_mult: 2
  }
  convolution_param {
    num_output: 50
    kernel_size: 3
    stride: 1
    weight_filler {
      type: "xavier"
    }
    bias_filler {
      type: "constant"
    }}
}
...
```

LISTING 6.2: Generated VHDL code

```
...
architecture RTL of lenet is
...
conv2: convLayer
  generic map(
    BITWIDTH     => BITWIDTH,           -- bw
    IMAGE_WIDTH  => CONV2_IMAGE_WIDTH, -- H,W
    KERNEL_SIZE  => CONV2_KERNEL_SIZE, -- J,K
    NB_IN_FLOWS  => CONV2_IN_SIZE,     -- C
    NB_OUT_FLOWS => CONV2_OUT_SIZE,    -- N
    KERNEL_VALUE => CONV2_KERNEL_VALUE,-- Theta
    BIAS_VALUE   => CONV2_BIAS_VALUE   -- beta
  )
  port map(
    clk      => clk,
    reset_n  => reset_n,
    enable   => enable,
    in_data  => pool1_data,           -- X_conv
    in_dv    => pool1_dv,
    in_fv    => pool1_fv,
    out_data => conv2_data,           -- Y_conv
    out_dv   => conv2_dv,
    out_fv   => conv2_fv
  );
...
```

The output of HADDOC2 is a platform independent VHDL code that can be implemented on the FPGA device using the adequate synthesis tool. Examples of inputs and outputs of are given in listings 6.1 and 6.2. Note that the tool and the library of VHDL components are detailed in appendix B, and are made available online[6].

[6] https://github.com/KamelAbdelouahab/haddoc2

6.5.1 Direct Hardware Mapping of Networks

As a first proof of concept, FPGA-based accelerators for three simple networks, namely LeNet5-I1[7], SVHN [NW11][8] and CIFAR10 [Kri09][9], are implemented with HADDOC2. Table 6.6 details the topology of these CNNs and the shares of their zero-valued, one-valued and power-of-two-valued parameters. Note that CIFAR10 and SVHN share the same topology but have different kernel values, which is useful to study the impact of these values on the FPGA mapping.

TABLE 6.6: Experimental Setup: Topology, weights stats and top1 accuracy

	LeNet5-I1			Cifar10			SVHN		
Layer parameters	*N*	*C*	*K*	*N*	*C*	*K*	*N*	*C*	*K*
conv1+pool+TanH	20	1	5	32	3	5	32	3	5
conv2+pool+TanH	50	20	5	32	32	5	32	32	5
conv3+pool+TanH	–	–	–	64	32	5	64	32	5
top1-FP32 (%)	98.96			76.63			87.54		
selected bit-width	3			6			6		
top1-FxP (%)	98.32			73.05			86.03		
zero parameters(%)	88.59			33.78			37.14		
pow2 parameters(%)	0.05			31.20			27.55		
other (%)	11.36			35.02			35.31		

For each network, the quantization scheme is selected after studying its impact on the top1 accuracy. To do so, the three networks are first trained on respectively MNIST, Cifar10 and SVHN datasets. Than, the *Ristretto* framework [GMG16] is used to explore their top1 accuracy for variable bit-widths. The results of this exploration are reported in Fig. 6.19 and shows how how a 3-bit representation can be chosen for LeNet5 without affecting classification accuracy (resp. 6-bit representation for SVHN and CIFAR10).

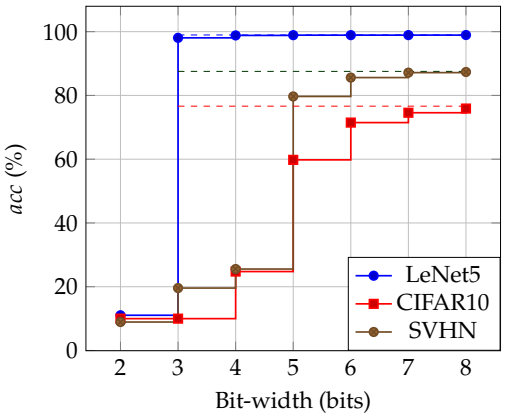


FIGURE 6.19: Evolution of top1-accuracy vs bit-width for the studied CNNs. The dashed lines refer to accuracy of the baseline 32-bits floating point model.

Tables 6.7-a and 6.7-b respectively detail post-fitting results on two embedded FPGA platforms: the Intel Cyclone V `5CGXFC9E7` and the Xilinx Kintex 7 `XC7Z045FBG`. To the best of our knowledge, these numbers are the first to demonstrate the applicability of a DHM-based approach for the implementation of CNNs on embedded FPGAs.

The three hardware accelerators fit onto the embedded devices with no off-chip memory requirement, the reported memory footprint corresponding to line buffers used by

[7]Caffe model available at `https://github.com/BVLC/caffe/tree/master/examples/mnist`
[8]Caffe model available at `https://github.com/alexvking/neural-net-house-number-recognition`
[9]Caffe model available at `https://github.com/BVLC/caffe/tree/master/examples/cifar10`

the factorized window buffers. Moreover, the three mappings operate at more than 45 FPS, granting real-time performances on 720p video streams. Finally, and as expected when using DHM, the logic utilization grows with the number of non-null weights, as it will be discussed in the last section of this chapter.

TABLE 6.7: Resource Utilization of the generated mappings:
a- On an Intel Cyclone V FPGA, b- On a Xilinx Kintex 7 FPGA.

		LeNet5-I1	Cifar10	SVHN
a	Logic Elements (ALMs)	8067 (7%)	51276 (45%)	39513 (35%)
	DSP Blocks	0 (0 %)	0 (0%)	0 (0%)
	Block Memory Bits	176 (1%)	15808 (1%)	10878 (1%)
	Frequency	65.71 MHz	63.89 MHz	63.96 MHz
	FPS (*720p/3 mul-sc*)	54	52	53
b	Slices	25031 (11%)	172219 (79%)	136675 (63%)
	DSP Blocks	0 (0%)	0 (0%)	0 (0%)
	LUTs as Memory	252 (1%)	1458 (2%)	1552 (1%)
	Frequency	59.37 MHz	54.17 MHz	54.49 MHz
	FPS (*720p/3 mul-sc*)	49	44	45

6.5.2 Direct Hardware Mapping of CNN Layers

When it comes to large-scale CNNs like AlexNet or VGG, the high amount of resources needed to implement these networks *entirely* makes the DHM approach not feasible on current FPGA platforms, not even the highest-end ones. However, thanks to the tactics discussed above, separately mapping each layer of these CNNs is yet possible. Table 6.8 reports the post-fitting results of the mapping of AlexNet, VGG, and YOLOv3 layers on a Terasic DE5-Net board[10] with a 6bits precision[11]. Note that we focus on the mapping of the *first* layers of these networks since their CTC ratio makes them suitable for a direct hardware mapping, as argued in sec. 5.2.2.

TABLE 6.8: Resource utilization of AlexNet, VGG and YOLOv3 first layers

Network	Layer	ALMs	Registers	M20K
AlexNet	conv1	122760 (52%)	159587	30 (1%)
DeepComp	conv1	123512 (53%)	155626	30 (1%)
SqueezeNet	conv1	74394 (32%)	153152	18 (<1%)
VGG16	conv1-1	13 554 (6%)	15545	6 (<1%)
VGG16	conv1-2	123 559 (59%)	149103	988 (39%)
VGG16	conv2-1	176 033 (84%)	221300	494 (19%)
YOLOv3-tiny	conv1	6835 (3%)	2793	2 (<1%)
YOLOv3-tiny	conv2	28468 (14%)	39168	8 (<1%)

As shown in table 6.8, DHM is indeed feasible for large-scale CNN layers. AlexNets' first layer, which involves the highest number of multiplications due to the (11×11) convolutions, is the most resource-hungry layer, requiring more than 123K of ALMs. Interestingly, the hardware utilization of this layer could not be reduced further when considering the compressed version of AlexNet proposed in [HMD16][12]. Indeed, the

[10] Embeds an Intel Stratix V 5SGXEA7 FPGA
[11] Which grants a tolerable accuracy according to [HMD16, NYFS18]
[12] Referred As DeepComp in the table

compressed and original *conv*1 layers share nearly the same sparsity, around 36% zero-valued weights. In place, we found that using SqueezeNet [IMA+16], and its (7×7) filers, was a better way to maintain AlexNet accuracy while lowering the resource utilization, resulting in 40% less utilization of the ALMs.

By contrast to AlexNet, the first layers of VGG16 and YOLOv3-tiny have a low CTC and, consequently, a small footprint. These layers can even be ported to embedded FPGAs featuring a lower amount of available resources and power consumption.

6.5.3 OCR on the DreamCam with Haddoc2

To validate the concepts and tactics discussed in the last two chapters, this section details the implementation of a CNN-based OCR system on the DREAMCAM smart camera. The DREAMCAM [BB14], illustrated in Fig. 6.20 is a smart camera development platform in which the processing core is an Intel Cyclone III FPGA. The DreamCam is also *modular*, and can be equipped with a large panel of:

- **Sensors**: mainly an MT9 or an E2V (1280×720) image sensor, a gyroscope, an accelerometer or a Global Positioning System
- **Communication interfaces**: USB-2.0 or Gigabit Ethernet

The Cyclone III FPGA is also connected to 6×1 MBytes of SRAM memory blocks wherein each memory block has private data and address buses allowing up to six processes to simultaneously access the external memory.

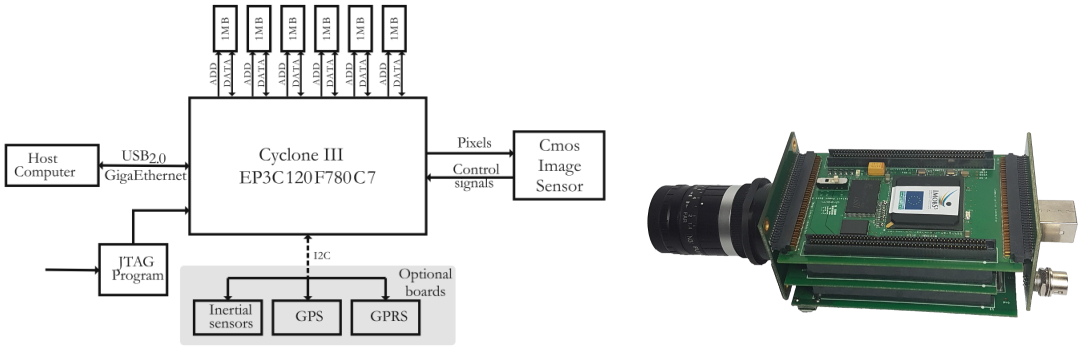


FIGURE 6.20: The DreamCam Smart Camera [BB14]

Moreover, the DREAMCAM is compatible with the GPSTUIO tool-chain [CHB+16], which manages the inclusion of the sensor and communication IPs, and gives user-friendly software interface to parametrize and display the processed video flows.

In the following demonstration, a frame depicting 100 hand-written digits is sent to the FPGA through USB. In this FPGA are mapped the three convolution layers of LeNet5-I4 (see tab.5.1)[13] as well as the activation and pooling parts. The extracted features maps ($\boldsymbol{Y}^{conv3}$) are than transfered to a CPU where the processing of the fully connected and softmax layers occur. Fig. 6.21 describes this setup:

- The red parts of the scheme are hardware instances generated by Haddoc2
- The green part is the hardware and software glue generated by GPSTUDIO
- The blue parts are implemented in software using the Caffe library.

[13] This implementation is selected because it delivers a tolerable generalization performance on USPS as pointed-out in tab. 5.2

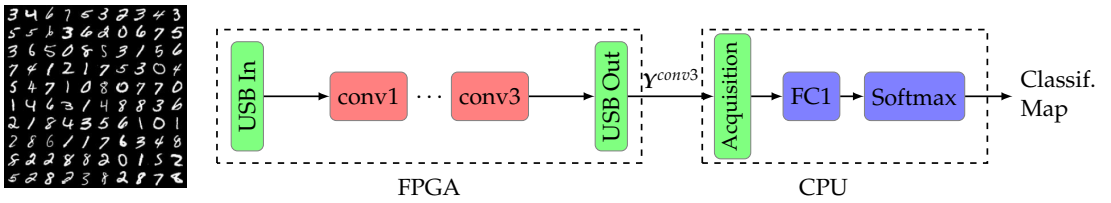
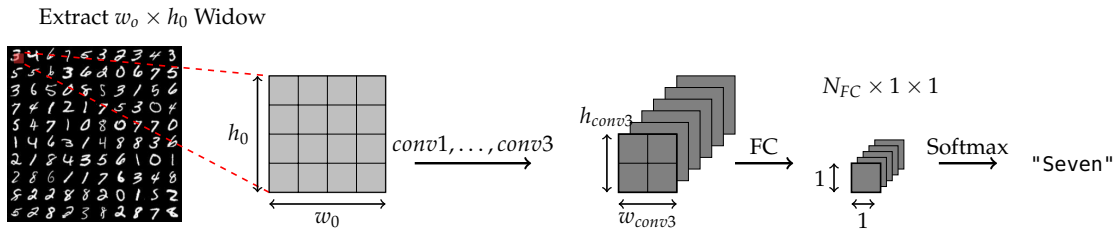


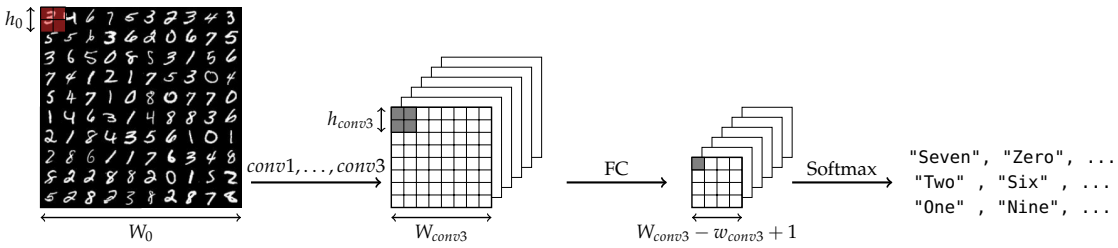
FIGURE 6.21: Demonstration Setup

6.5.3.1 Classification without Sliding Windows

A key advantage of ConvNets is their ability to work on large images without resorting to *sliding windows*.



(A) With a Sliding Window



(B) Without Sliding Window

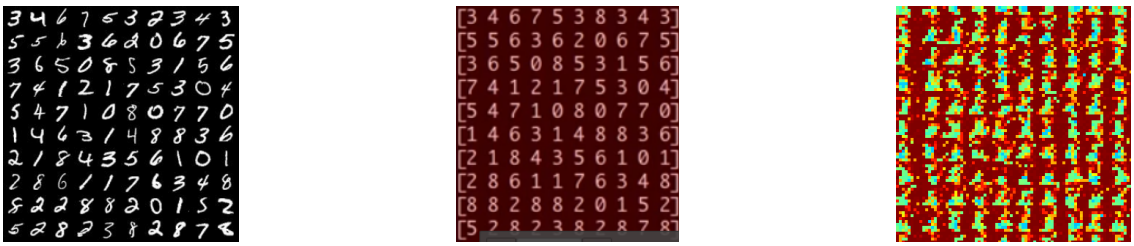
FIGURE 6.22: Deploying CNNs with and without Sliding Windows

Indeed, conventional classifiers in computer vision are used to rely on a sliding window block that extracts a given $(w_0 \times h_0)$ window on the input image, performs the classification on this window, than iterates this process across all the possible positions in the original image. Instead, CNNs can directly operate on a large input $(W_0 \times H_0)$ simply by computing convolutions at each layer over the entire image. This concept is depicted in figures. 6.22 where the output of the CNN is no longer a vector N_{FC} (i.e a single class), but a map of classifications on each position of the image.

To compute the FC and Softmax results, the weights are shared across multiple positions of $(\boldsymbol{Y}^{conv3})$. In this case, the processing of an fully connected layer can be seen as a convolution where $K_{FC} = w_{conv3}$ and $J_{FC} = h_{conv3}$.

6.5.3.2 Implementation Results

Figures 6.23 give the inputs and outputs of the studied OCR system. As it can be seen in fig.6.23b, the proposed hardware implementation of LeNet5 with a 3 bits precision correctly classifies 98 of the 100 digits processed. Note that this figure only shows the classfications where the confidence (i.e the output of the softmax function) exceeds 0.5. This confidence map is depicted in fig.6.23c. As expected, the confidence rate is high (green) in the regions where the digits are present, and low (red) in the interval between digits.



(A) Input Frame (B) Classif. map (C) Confidence map

FIGURE 6.23: Classification results

For the FPGA resource utilization, the complete design uses respectively 87% and 69% of the logic and memory available on the Cyclone III device. Most of the logic resource are allocated to the last convolution layer *conv*3, which involves the highest number of computations. By contrast, most of the RAM block of the FPGA are allocated to the USB controller and only a small part of these memories are allocated to the CNN mapping, thanks to the factorized window buffer discussed in sec.6.2. Tables 6.9 summarize the resource utilization of the mapping.

TABLE 6.9: Resource Utilization of the OCR system on the DreamCam Platform

(A) Resource Summary

Logic utilization (in Logic Elements)	103854 / 119088 (87 %)
Total Memory Bits	2.14 / 3.98 MBits (54 %)
Total M9K RAM Blocks	298 / 432 (69 %)
Total 9-bit Multipliers	0 / 576 (0 %)
Frequency	53.42 MHz

(B) Utilization per Entity

Entity	LE	MemBits	M9K
`conv1`	769	1112	1
`pool1`	5512	0	0
`conv2`	10253	4896	6
`pool2`	7820	0	0
`conv3`	61219	6303	20
`pool3`	15546	152	3
`USB`	2280	2130944	268

As reported in the same table, the proposed OCR system operates at a maximum frequency of 53.42 MHz, which corresponds to a theoretical throughput 671 FPS on the processed (282×282) frames. However n practice, this frame rate drops to $\sim$ 83 FPS, the main bottleneck being the bandwidth of the USB2.0 used. Notice that the maximum frequency of the design slightly varies with the input resolution and more particularly in image width, as depicted in Fig.6.24.

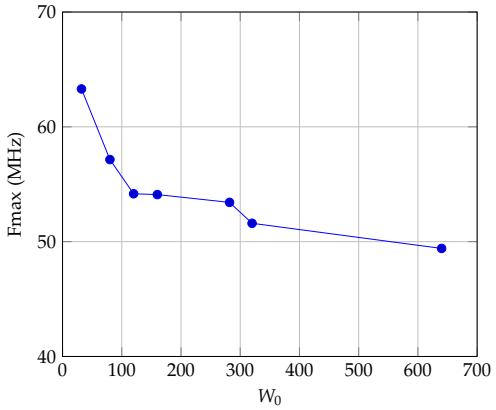


FIGURE 6.24: Evolution of Fmax with the input Resolutions

6.6 Modeling CNN Mappings

As pointed-out in the implementation results, the resource utilization of a given CNN mapping is correlated to the shares ofe zero-valued and the power-of-two valued weights. This suggest that the footprint of CNN mappings can be reduced further by tweaking these parameters.

However, tweaking the CNN weights and finding an optimal design that trades between accuracy and resource utilization calls for an exploration in a large design space. While the design space exploration of CNN *hyper-parameters* has been the subject of chapter 5, the design space exploration we discuss in this chapter occurs on the *parameters* them-selves (i.e the values of Θ).

This makes the design space even larger. With a larger space, exploration process takes more time to be completed. Particularly, the process of *synthesizing* the generated hardware on FPGA –and thus reporting the resource utilization– is the most critical. Indeed, the synthesis of CNN accelerators on FPGAs is a task that can take up to several hours to be completed. For instance, the mapping results reported in table 6.8 took up to 20 hours to be synthesized by the Quartus II tool [14].

A solution to fasten the exploration process is to mathematically *model* the resource utilization according to the values of the weights. This is what this section aims to: derive a model that *estimates* the resources allocated to a given mapping without resorting to synthesis tools. *In fine*, the model discussed here would be integrated in an exploration tool-chain, accelerating the exploration process.

6.6.1 Resource Utilization by Entity

To see which of the CNN actors impacts resource utilization the most, table 6.10 gives –for each of the layers studied in sec.6.5.2– the ALMs used by each entity. As expected, most of the logic fabric is allocated to the dot-product parts of the mapping (The SCMs and MOAs). Indeed, multipliers and adders use more than 95% of the allocated ALMs. Consequently, the modeling will focus on these two parts, and will mainly consider the logic resources (i.e ALMs) allocated to the mapping. Note that this study can be extended to DSPs (in the case of non-SCM implementations) and SRAM blocks.

TABLE 6.10: ALMs Used by Entity

Layer	SCMs	MOAs	Window Buffers
AlexNet-conv1	30491 (24.84 %)	89786 (73.14 %)	929 (0.76 %)
DeepComp-conv1	33542 (27.07 %)	87505 (70.62 %)	959 (0.77 %)
SqueezeNet-conv1	21139 (27.00 %)	51909 (66.30 %)	549 (0.70 %)
VGG16-conv1-1	3649 (26.92 %)	8798 (64.92 %)	174 (1.28 %)
VGG16-conv1-2	51071 (41.33 %)	66810 (54.07 %)	5212 (4.22 %)
VGG16-conv2-1	78691 (44.70 %)	92143 (52.34 %)	4301 (2.44 %)
YOLOv3-tiny-conv1	1117 (16.34 %)	5305 (77.62 %)	286 (4.18 %)

On a side note, tab.6.10 also shows a surprising result: the largest portion of the resources is allocated to map the *adder* parts and not the multiplier parts. Despite being less complex than multipliers, the MOA components takes up to 77% of the ALMs inferred to map a layer, even with the optimization discussed in sec.. This problem is addressed in the next chapter of this manuscript.

[14] Running on an Intel i7-4770 CPU with 16 GB of RAM

6.6.2 ALM utilization model

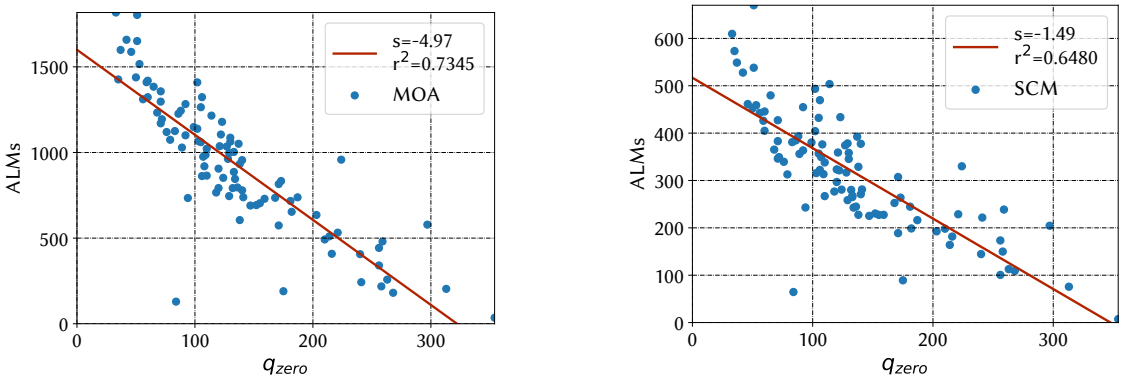
In the following section, we aim at deriving a *linear model* that quantifies the resource utilization of SCM and MOA blocks. The inputs of this linear model are metrics (or features) extracted from the CNN weights, which can accurately estimate the resource utilization when combined.

In an naive approach, the first metrics taken into account are the number of zero-valued and power-of-two-valued weights in a given kernel (resp. noted q_{zero} and q_{pow2}). Moreover, a third metric q_{bit1} that measures the numbers of «bits-set-to-one» in a given kernel is added. Indeed, our experiments have shown that synthesis tools tend to infer more resources when the number of bits-set-to-one in the constant multiplicand is above average.

With the previously listed features, the linear model can be written in eq.6.8, where α_{zero}, α_{pow2} and α_{bit1} are coefficients that can be estimated using linear regression. The proposed study uses the Python implementation of Linear Regression made available in the `sklearn` library. The following experiments can be reproduced [15].

$$\text{Model}_1 : \quad \hat{\mathcal{R}}_{ALM}(n) = \alpha_{zero} * q_{zero}[n] + \alpha_{pow2} * q_{pow2}[n] + \alpha_{bit1} * q_{bit1}[n] \tag{6.8}$$

Clearly, the allocated resources decrease with the portions of zero valued and power-of-two valued weights in a kernel. This is confirmed by Fig 6.25 which plots the logic resources allocated to each of the 96 three-dimensional convolutions involved in AlexNets' first layer. For each 3D-kernel (i.e each point in the scatter plot), the ALMs and number of zero-valued weights is reported.



(A) MOA (B) SCM

FIGURE 6.25: Resource utilization of 3D-convolutions *vs.* zero-valued weights

However, the three former features alone can not accurately model the resource utilization, especially of the one of the MOA parts. Indeed, the MSE of the estimation remains too high, exceeding 10%. To improve the modeling quality, a fourth feature q_{dyn} is introduced, and is directly correlated to the *numerical dynamic* of the kernels.

$$\text{MSE} = \text{mean}\left[(\hat{\mathcal{R}}_{ALM}(n) - \mathcal{R}_{ALM}(n))^2\right] \tag{6.9}$$

Recall that the accumulation of partial products in DHM is achieved with a MOA that inputs multiple operands with variable bit-widths. To map this adder, the synthesis tool generates a circuit which the complexity is correlated to the number of inputs, but more importantly to their numerical dynamic range.

[15] https://github.com/KamelAbdelouahab/Multi-Operand-Adder

This concept is illustrated by the following example: Let x be an input vector and $w = [2, 0, 18, 256]$ a constant weight vector. Both of x and w are quantized to 8 bits and let's study the mapping of the dot-product $x * w$ with SCM and MOA:

- The multiplication of x by the first coefficient can be implemented with a shift register and the resulting partial product $p[0] = x[0] * w[0]$ requires $8 + mcl(2) = 9$ bits to be represented, where $\text{mcl}(x) = \max(\text{ceil}(log_2(x)))$.
- The multiplication by the second coefficient is skipped and does not generate any partial product.
- The multiplication by the third coefficient requires $8 + \text{mcl}(18) = 13$ bits to be represented.
- The multiplication by the last coefficient is implemented by means of shift register and the partial product requires $8 + \text{mcl}(256) = 16$ bits to be represented.
- Finally, the accumulation of these partial terms is achieved with a MOA that inputs respectively 9, 13 and 16 bits. The circuitry of this adder has a complexity which function of the number of partial products and their dynamic. This dynamic, expressed in bits, can be written in equation 6.11, where b_x is the bit-width of x, and $\text{mcl}(\theta[c, j, k])$ is the minimal number of bits required to encode a constant $\theta[c, j, k]$.

$$\text{Model}_2 : \quad \hat{\mathcal{R}}_{ALM}(n) = \alpha_{dyn} * q_{dyn}[n] \tag{6.10}$$

$$q_{dyn}[n] = \sum_{c=0}^{C-1} \sum_{j=0}^{J-1} \sum_{k=0}^{K-1} [b_x + \text{mcl}(\Theta[n, c, j, k])] \tag{6.11}$$

Figures 6.26 illustrate how the resource utilization increases with q_{dyn}. As it can be seen, the q_{dyn} feature delivers better r-squared values when compared to q_{zero}, and is thus more accurate in modeling the CNN resources. This last result is corroborated by tables 6.11 which report the r-squared values of all the linear models using the four features previously introduced. Each model is tested on the first layer of AlexNet, SqueezeNet and DeepComp.

TABLE 6.11: R-Squared Values and Estimation Error of the proposed Linear Models

(A) MOA R2

	Alexnet	Squeezenet	DeepCmp.
Model1	0.680	0.682	0.692
Model2	0.776	0.711	0.778
Model3	0.814	0.737	0.810

(B) SCM R2

	Alexnet	Squeezenet	DeepCmp.
Model1	0.599	0.509	0.617
Model2	0.726	0.591	0.748
Model3	0.801	0.690	0.833

(C) MOA MSE

	Alexnet	Squeezenet	DeepCmp.
Model1	5.59e-02	1.44e-02	5.35e-02
Model2	3.93e-02	1.31e-02	3.87e-02
Model3	3.25e-02	1.19e-02	3.29e-02

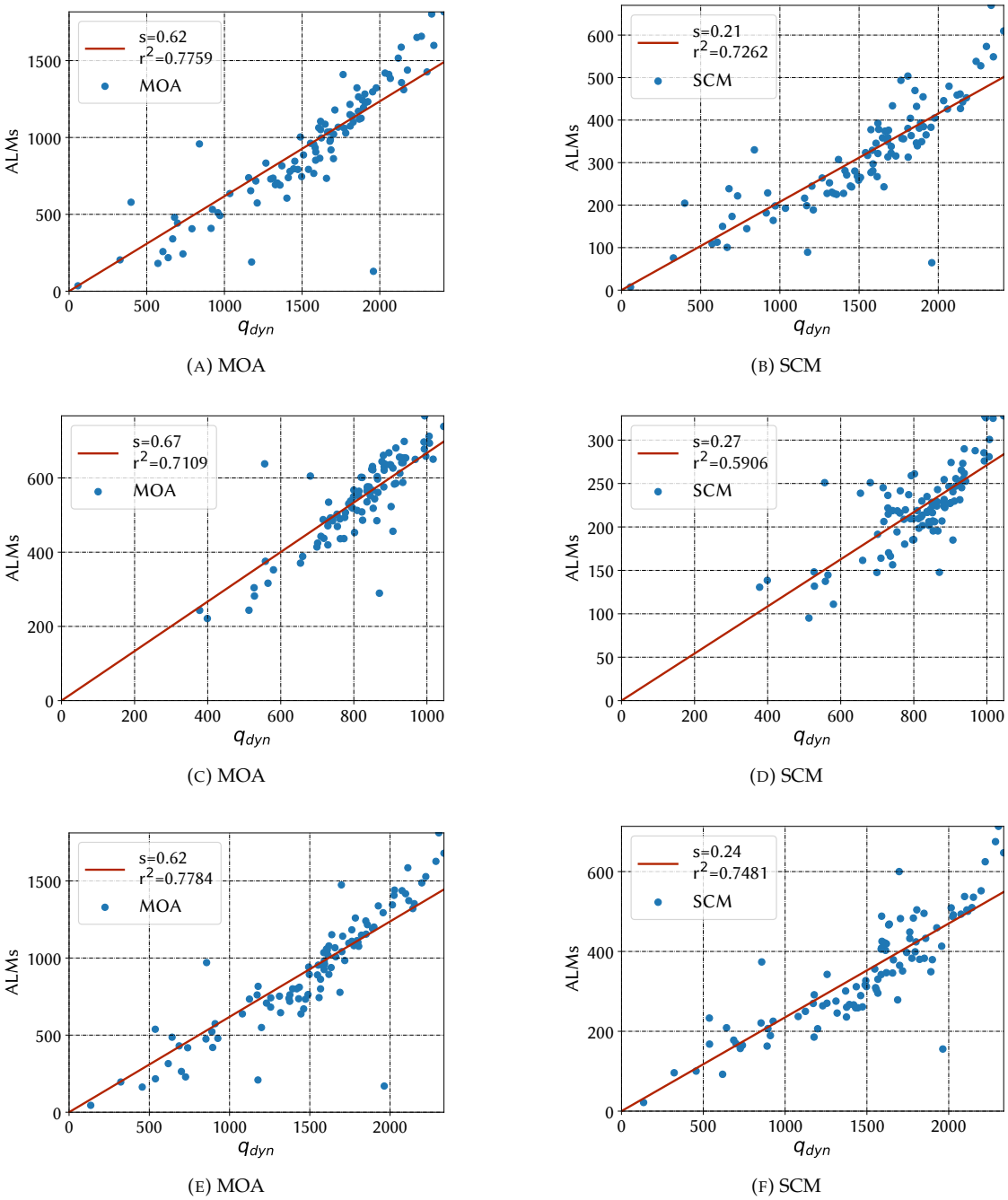
(D) SCM MSE

	Alexnet	Squeezenet	DeepCmp.
Model1	6.15e-02	2.14e-02	5.83e-02
Model2	4.20e-02	1.78e-02	3.84e-02
Model3	3.05e-02	1.35e-02	2.55e-02

The last column of tables 6.11 stands for the third model which combines the four features to predict the logic utilization as formulated in equation 6.12. For all the tested

networks, Model3 delivers the most accurate estimation of resources utilization for both multi-operand adders and constant multipliers.

$$\text{Model}_3 : \hat{\boldsymbol{R}}_{ALM}(n) = \alpha_{qdyn} * q_{qdyn}[n] + \alpha_{zero} * q_{zero}[n] + \alpha_{pow2} * q_{pow2}[n] + \alpha_{bit1} * q_{bit1}[n] \tag{6.12}$$



(A) MOA (B) SCM

(C) MOA (D) SCM

(E) MOA (F) SCM

FIGURE 6.26: Resource utilization of 3D-convolutions *vs.* q_{dyn}

6.7 Conclusions and Perspectives

This chapter has investigated tactics that optimizes the *direct hardware mapping (DHM)* of CNNs on FPGAs for embedded vision applications. It has demonstrated that current embedded FPGAs provide enough hardware resources to support this approach. To demonstrate DHM, the HADDOC2 tool has been introduced and used to automatically generate platform-independent CNN hardware accelerators from high level CNN descriptions.

The optimizations discussed in the last two chapters open new opportunities in terms of hardware implementations of CNNs. An interesting perspective would be to extend DHM principles to ASIC technologies as well as Binary Neural Networks.

A first future direction of this works would be to rely on the resource utilization model proposed in sec.6.6 to predict the CNN resource *during the training*. In this case, the CNN can be jointly optimized toward accuracy and low resource utilization.

Another future direction would be to investigate the feasibility of MCM for CNNs. MCM demonstrated significant improvements in terms of resources consumption when mapping 1D-convolutions on FPGAs [PM14], and its major advantage over SCM is its ability to *reuse* some parts of the circuitry of a given constant adder to implement another. For instance, MCM can use the result of a previous multiplication of $x \times 4$ to implement the multiplication of $x \times 5$. As a result, it grants resources savings that are function of the values of the kernels, but also function of the *relation* between these values.

Chapter 7

Negative Results on Optimizing Direct Hardware Mapping

This chapter details experiments for optimizing direct hardware mapping of CNNs that were promising but did not conduct to performance improvements. More particularly, we address the problem pointed out in tab.6.10, which is the high resource utilization of adder trees in direct CNN mappings.

To solve the problem of adders, the first sections of this chapter explore two different strategies: *serial adders* using local time multiplexing of additions, and *approximate adders* trading off accuracy for resources. The third section is more general and explores the feasibility of *stochastic arithmetic* in direct hardware mapping. For the three studies, code reproducing the experiments is made available online[1][2].

7.1 Serial Adders

7.1.1 Motivation

FPGA devices –and more particularly the DSP blocks they embed– can run at a peak frequency which is much higher when compared to the rate at which data and feature maps are acquired by a given CNN layer. Given this, it is possible to replace clusters of binary adder trees by a serial accumulator that runs in a different (higher) clock domain. In other words, serialization trades a clusters of n_c binary adders that operate at a frequency f_0 for a *single accumulator* that operates at a frequency $f_c > f_0$. This accumulator runs $\times n_c$ times faster than the remaining parts of the mapping (eq.7.1). One may note that using serial adders, the CNN is not any more fully using « direct hardware mapping ». To make serialization possible, a parallel-to-serial register (serializer) is required at the input the accumulator, as shown in Figure 7.1.

$$f_c = n_c * f_0, \quad 0 \leq n_c \leq n_{opd} \tag{7.1}$$

In recent FPGA devices, DSP blocks can peak at $f_c =\sim 200$ MHz while a $720p@30FPS$ video stream can be acquired at a frequency of $f_0 = 27.6$ MHz. Serialization can thus replace about ≈ 6 inputs of a Multiple Operand Adder (MOA) by a serializer-accumulator pair, which would theoretically reduce the footprint of a MOA by a factor of $n_c - 1 \approx 5$ under the hypothesis that serializers have a simpler, negligible circuitry when compared to MOAs.

[1] Serialized and approximate adders: https://github.com/KamelAbdelouahab/Multi-Operand-Adder
[2] Stochastic Computing: https://github.com/KamelAbdelouahab/SC

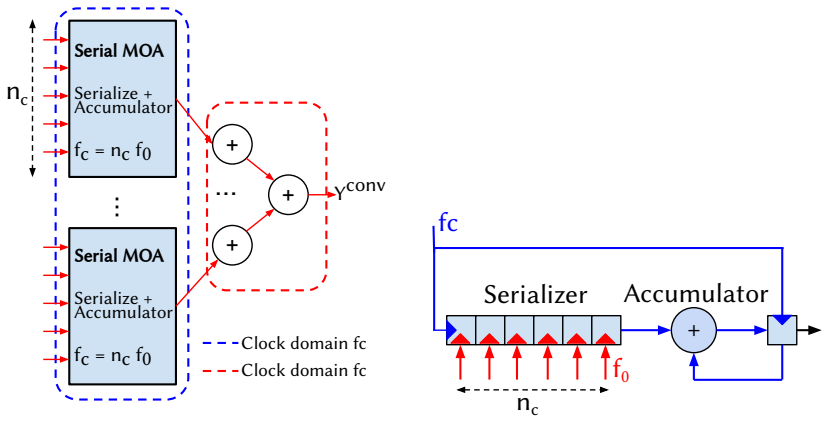


FIGURE 7.1: Architecture of a serial MOA. Each Serializer-Accumulator Pair replaces a cluster of adders in the MOA

7.1.2 Experiments and Results

In order to study the impact of serialization on an MOA, the Serializer/accumulator pair of Figure 7.1 is implemented in VHDL[3]. Figure 7.2 reports the logic utilization (in terms of ALMs) of both the serializer, the accumulator and the serial adder for variable cluster sizes. These results are compared to the logic utilization of the standard cascade structure of a MOA detailed in sec.6.10.

This figure shows a very unexpected result: the resource utilization of the serializer/accumulator pair exceeds the resources used by a «cascaded» implementation of a MOA. This is the result of the costly logic fabric required by the serializer part, displayed in Figure 7.2, which grows linearly with the number of parallel inputs. The overhead of serializers thus invalidates the approach.

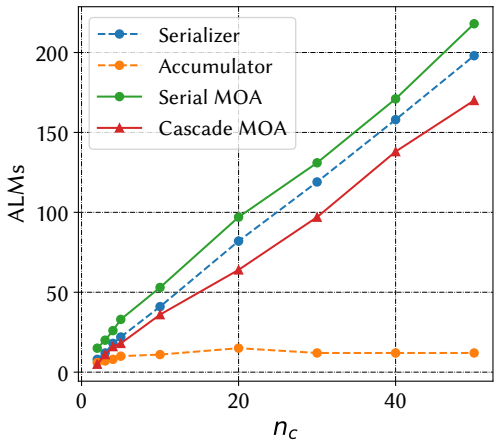


FIGURE 7.2: Comparison of the Logic resources used by a serialized and fully pipelined implementation of a MOA: The serializer results in a large resource overhead

[3]Synthesized on an Intel Stratix V 5SGXEA7 FPGA using Quartus 16.0. The bit-width of operands is 8 bits

7.2 Approximate Adders

7.2.1 Motivation

As highlighted in sec.4.4, deep CNNs are over-parametrized networks that tolerate by nature a degree of approximate computing and multiple state-of-the-art publications demonstrate the resiliency of CNNs towards compact bit-width arithmetic [GAN+15, WLW+16]. This hints that CNNs may support others types of approximate computing techniques such as approximate adders.

These adders, which use is limited to fault-tolerant applications, are known to deliver higher speed and power efficiency than exact operators [JHL15]. In order to solve the challenge of MOA footprint reduction for CNNs, we leverage on the low resource utilization of the LOA approximate adders [MAFL10].

As illustrated in Figure 7.3, a LOA divides a b-bit adder into two sub-adders. The first one is an approximate b'-bit sub-adder that computes the sum of least-significant bits by using a bit-wise OR operation. The second is an exact $(b - b')$-bit sub-adder that processes the most-significant bits using full adders. An extra AND gate is used to generate the carry-in signal for the exact adder part.

As pointed-out in the study of [JHL15], LOA is the slowest yet most *area efficient* approximate adder, making it the best candidate to reduce the footprint. In the Multi-Operand case, area saving may be achieved by replacing the exact binary adders present in the tree with approximate adders such the LOAs.

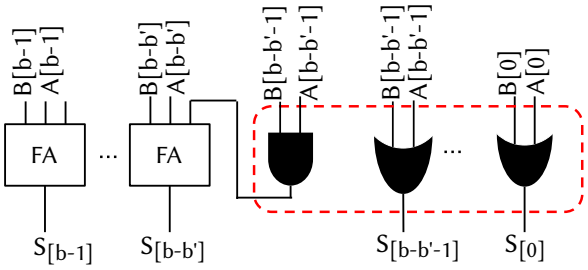


FIGURE 7.3: Hardware structure of a Lower-part OR approximate adder (LOA). Approximate parts in the red box. Each LOA Replaces a Binary Adder in the Tree of figure 6.10

7.2.2 Experiments and results

In order to investigate the feasibility of approximate LOA adders in CNN implementations, two experiments are conducted: the first studies the impact of the approximation on the accuracy of the adder . The second is hardware-oriented and focuses on the resources savings of LOA adders on FPGAs. Both studies use the following metrics:

- The « approximation » is quantified using the *approximation ratio*. It is defined as the number of approximated bits per total bit-width b/b'. A ratio of 0% corresponds to an exact adder while a ratio of 50% means that half of the bits of a given addition have been approximately processed using OR gates.
- The « accuracy » of the adder is quantified using the MRED metric. If $S = A + B$ is the result of an exact addition of A and B, and $\hat{S}$ the result of an approximate addition with same operands. The error distance is defined as:

$$MRED(S, \hat{S}) = mean\left(\frac{|\hat{S} - S|}{S}\right). \tag{7.2}$$

- As usual, the number of ALMs quantifies the logic resource utilization.

The evolution of the MRED metric when varying bit-widths and approximation ratios is illustrated in Figure 7.4. In terms of accuracy, using lower-part OR Adders results in a relatively small error ($< 10\%$ MRED for 8bits adders), which suggests that they might be exploited to derive energy-efficient CNN accelerators.

However, when it comes to FPGA resources utilization, our experiments show that no area saving can be achieved on an FPGA when using LOAs. Indeed, the number of ALMs remains surprisingly constant, independently from the number of bits processed by an OR gate. This is explained by the fact that modern FPGA devices employ complex logical modules (such as Intel ALMs) which already embed a hard-wired full adder, as explained in section 3.1.1. This logical module either implements a full adder in the case of exact adders, or implements an OR gate using the LUT it includes in the case of the approximate LOA adder. As a consequence, current FPGA and related hardware synthesizers do not benefit from approximate computing when targeting MOA adders. Note that these results have been observed on both Intel and Xilinx FPGAs.

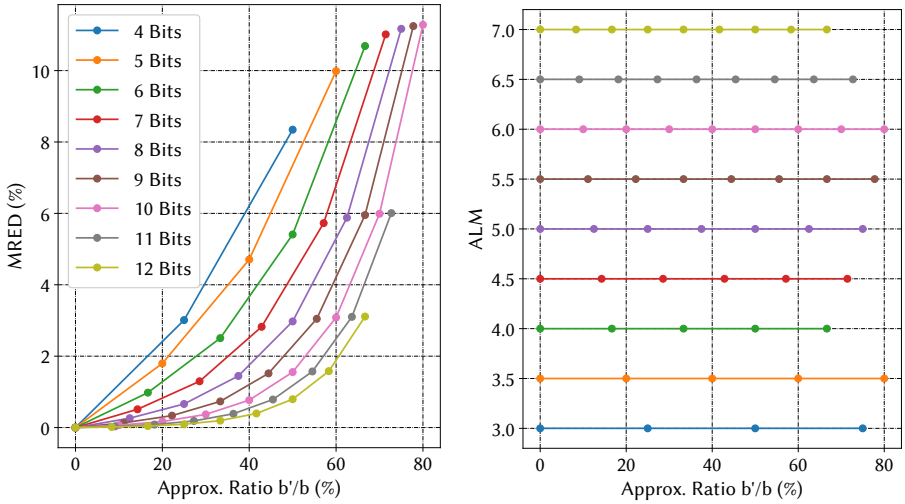


FIGURE 7.4: Error Rates and logic utilization of LOAs for variable bit-widths and approximation ratios.

7.3 Stochastic arithmetic

After addressing the problem of adders, the last section of this manuscripts studies the feasibility of stochastic arithmetic for CNN direct hardware mapping.

SC [Von56] is a processing technique that represents values as streams of random bits with a specific mean value. One of the advantages of SC is its ability to implement complex computations with simple bit-wise operations on the streams. As a results, stochastic computing has been successfully applied in numerous image processing algorithms [BC01, LL11], and their hardware implementations [AH13, AH14, Ala15].

7.3.1 Sequence generation and Arithmetic operations

In SC, a given number x is represented as a random sequence s_x . In the basic «unipolar» format, the number of ones appearing in s_x determines the value of x. In other words, the numerical value of a given number x is the ratio between the number of ones appearing in s_x and the length of this sequence. Generally, this ratio is expressed as the *expected value* of s_x, which can be written as:

$$x = \mathbb{E}[s_x] = \frac{\text{Number of ones in a sequence } s_x}{\text{Length of sequence } s_x} \tag{7.3}$$

From a hardware point of view, the stochastic stream s_x is created using a random number generator and a comparator, the random numbers being usually generated by a LFSR. Fig.7.5 illustrate the LFSR-comparator pair which, hereafter is referred to as a SNG.

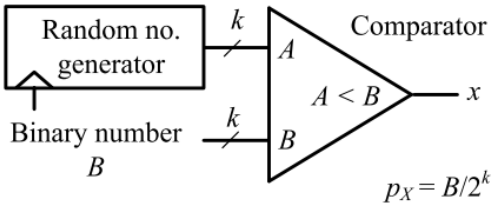
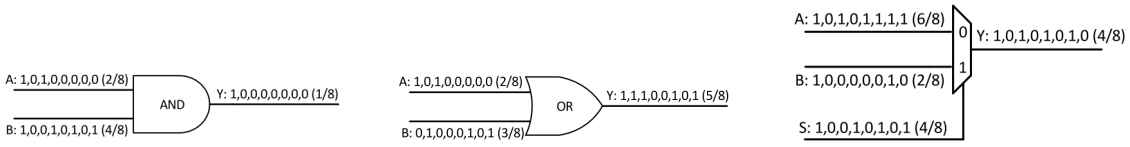


FIGURE 7.5: Stochastic number generator using an LFSR, from [Ala15]

As stated above, SC implements complex computations with simple bit-wise operations. In the case of multiplication, two « binary » numbers a and b can be multiplied in the stochastic domain by simply using a single AND gate, as depicted in fig 7.6a.

$$a * b = \mathbb{E}[s_a] * \mathbb{E}[s_b] = \mathbb{E}[s_a \wedge s_b] \tag{7.4}$$



(A) Multiplication (B) Addition with OR (C) Addition with MUX

FIGURE 7.6: Circuits implementing stochastic computing arithmetic, from [ALPO+15]

When it comes to addition, two methods can be identified: scaled adders or OR gates. On one hand, for OR gates, it is known that the expected value of $s_a \vee s_b$ can be written as:

$$\mathbb{E}[s_a \vee s_b] = \mathbb{E}[s_a] + \mathbb{E}[s_b] + \mathbb{E}[s_a \wedge s_b] \tag{7.5}$$

When the $\mathbb{E}[s_a \wedge s_a]$ term is close to 0 (i.e when the stochastic streams are generated by disjoint SNGs) the OR gate functions as an approximate adder.

On the other hand, scaled adders rely on a *multiplexer* to sum two streams s_a and s_b. It is known that the output s_c of a 2-1-multiplexer with a selection bit (sel) is:

$$s_c = MUX(s_a, s_b) = (s_a \wedge sel) \vee (s_b \wedge \bar{sel}) \tag{7.6}$$

When considering eq.7.5, the expected value of the s_c stream can thus be written as:

$$\mathbb{E}[s_c] = \mathbb{E}[s_a \wedge sel] + \mathbb{E}[s_b \wedge \bar{sel}] + \mathbb{E}[(s_a \wedge sel) \wedge (s_b \wedge \bar{sel})]) \tag{7.7}$$

Since $\mathbb{E}[sel \wedge \bar{sel}]$ is always null, the former expression becomes:

$$s_c = \mathbb{E}[s_a \wedge sel) + \mathbb{E}[s_b \wedge \bar{sel}] \tag{7.8}$$

In the equation above, note that when the probability of stochastic stream sel is 0.5, the output of the MUX has an expected value of:

$$c = \frac{\mathbb{E}[s_a] + \mathbb{E}[s_b]}{2} \tag{7.9}$$

This corresponds to $(a + b)/2$, as illustrated in Fig.7.6c.

As in the precious discussion, stochastic computing benefits from a great simplicity when implementing addition and multiplication circuits. In turn, SC remains an approximate computing technique which is error-prone and suffers from three major downsides:

- First, the errors generated by stochastic arithmetic circuits are function of the length of the stochastic steams. In fact, stochastic circuits have to operate on extremely long sequences to deliver tolerable accuracy. At similar precision, a conventional binary representation of b bits number in the binary domain corresponds to a sequence of 2^b bits in the stochastic domain. For instance, a stochastic sequence of 512 bits is required to accurately represent an 9 bits number.
- Second, the transition between the binary and the stochastic domain calls for dedicated hardware blocks (SNGs and binary counters), which can generate a large overhead when the number of operands increases.
- Third, the errors generated by stochastic arithmetic circuits are also function of the statistical independence between the stochastic sequences. The errors of stochastic circuits are minimal when the SNGs are uncorrelated and increases when the SNGs are correlated. This generally prevents from sharing the same random number generator between two SNGs.

Yet, the advantages of SC motivated numerous research efforts to investigate its feasibility for neural network inference, as discussed in the next section.

7.3.2 Stochastic computing and neural network inference

Early works of [ALPO+15, RLL+17, KKY+16] demonstrate the applicability of stochastic arithmetic to accelerate neural network inference. More particularly, Ardakani *et al.* [ALPO+15] propose an FPGA accelerator to classify the MNIST dataset, where multiplications are processed using AND gates and activation functions (TanH) are implemented using Finite State Machine (FSM). Such an implementation delivers a computational throughput of 15.44 TOP/s with a miss-classification rate of 2.40% on MNIST,

which is comparable to the accuracy of « conventional » approximate computing techniques on this dataset (see Sec.4.4).

The work of Kim *et al.* [KKY+16] addresses the problem of long bit sequences by exploiting the *progressive precision* characteristics of stochastic computing. Progressive precision allows stochastic circuits to have a precision that increases gradually with the length of the sequence. Authors rely on this characteristic to implement multiplications that process 32 bits to make a decision, and, if processing fails, continues processing the next 32 bits, up to a 512 bits sequence. As a result authors are able to trade off between the accuracy of the classification and the latency of the processing.

7.3.3 Stochastic Computing for CNN DHM

The studies described above demonstrate how stochastic computing is a processing approach that improves the efficiency of neural network implementations. It particularly suggests that the direct hardware mapping can significantly benefit from SC, especially in reducing the footprint of the accelerator.

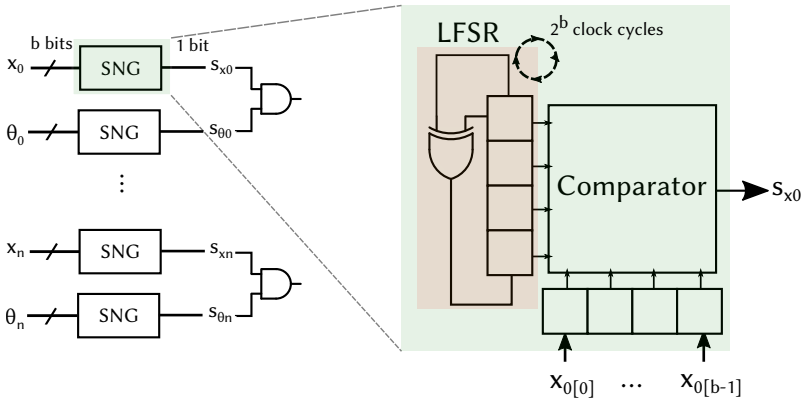


FIGURE 7.7: Hardware Architecture of the stochastic multiplier block

In order to study the feasibility of stochastic computing in the DHM context, we design stochastic multipliers that are integrated into the previously described dot product engines (see Fig.B.3). Thus, the stochastic multipliers replace the conventional « binary » multipliers introduced in section 6.3 with the objective to reduce the mappings' footprint. Note that this study of multipliers can simply translated to stochastic adders, simply by considering OR gates in place of AND gates.

As evoked in the previous section, stochastic circuits call for dedicated SNGs to be used for each data. Ideally, each stochastic multipliers requires its own SNG, with a proper random number generator, to insure that the generated stochastic streams are independent. Thus, each binary multiplier is replaced by an SNG-Adder pair, as shown in Fig.7.7. This figure also depicts the architecture of the SNG which uses an LFSR that generates random numbers as described in [GK88]. This LFSR is initialized with a parametrizable random seed and generates pseudo-random numbers at each clock cycle.

In the following experiment, 32 multipliers (binary and stochastic) are mapped on an Intel `5SGXMABN3F45I4` device[4]. These multipliers are mapped with a variable bit-width to study the evolution of resource utilization and operating frequency. The results of this study are given in Fig.7.8.

Clearly the stochastic multiplier has a simpler circuitry when compared to the conventional one, which results in resources savings. These savings grow with the bit-width

[4]This device is selected because it delivers the highest amount of IOs

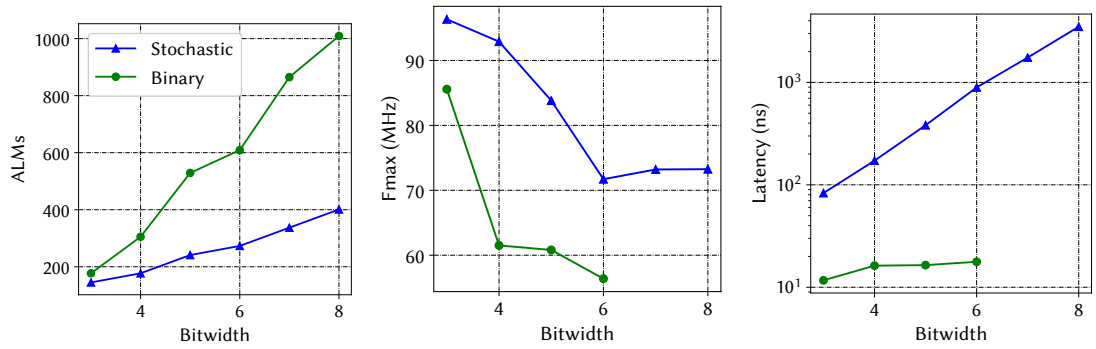


FIGURE 7.8: Performance of Stochastic and Conventional Multipliers

and can be reduced by a factor of $\times 2.5$ ALMs for 8 bits multiplications. This is explained by the fact that the resource utilization of the stochastic multiplier depends mainly on the utilization of its SNG (shift register + comparator), which is –at a similar bit-width– less complex than a conventional multiplier. Moreover, and since the stochastic arithmetic involves less combinatorial logic, the stochastic multiplier peaks at higher frequencies, and operates up to 53% faster than the binary circuit.

However, when it comes to the latency of the computation, and thus the « real » computational throughout, conventional binary multipliers largely outperform the stochastic ones. Indeed, one may recall that SC requires a stochastic stream of 2^b to encode an $b-$bits number. Consequently, the SNG needs 2^b clock cycles to generate the stream and perform the multiplication. This latency greatly impacts the throughput, even if the stochastic multiplier operates at higher frequencies.

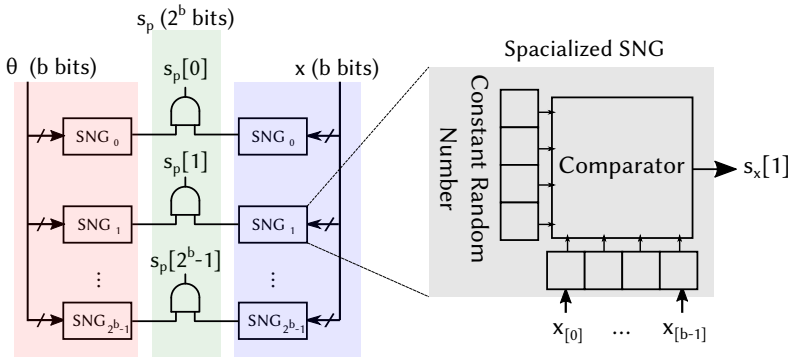


FIGURE 7.9: Hardware Architecture of a parallel stochastic multiplier

A possible solution to decrease the latency is to infer a large number of SNGs, each of them set at a different state. With this method, 2^b SNGs generate 2^b random numbers simultaneously, allowing the stochastic multiplier to operate in a single clock cycle. Moreover, and since the SNGs involve random number generators with a single state, they can be specialized and benefit from a simpler circuitry. This is depicted in Fig.7.9, where each random number generator of the SNG is replaced by a simpler combinatorial logic, specializing the LFSR block of figure 7.8.

However, and even with the help of the specialized number generators, the resource utilization of of this multiplier is too high, mainly because of the large number of stochastic number generators (see Fig.7.10). This last figure also gives the latency and resource utilization of the studied stochastic multiplier blocks, and compares them to the conventional binary multipliers. On one hand of the spectrum, the standard serial SC-Multiplier (in blue) generates the lowest resource utilization at the price of poor computational performance. On the other hand, the parallel SC multiplier operates with a low latency at

the price of excessively high resource utilization. The conventional binary multipliers are the most efficient, involving a lower resource utilization.

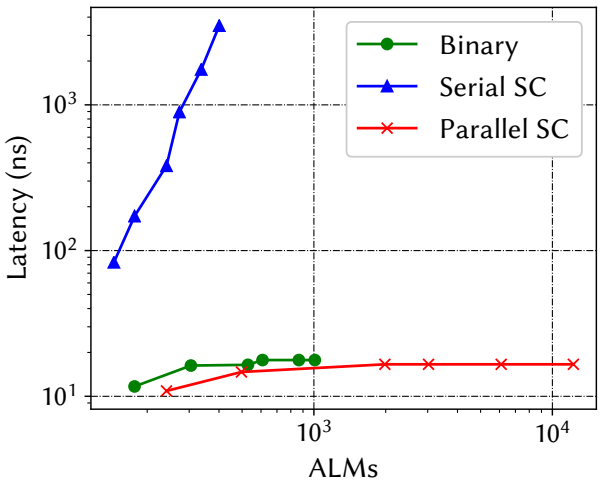


FIGURE 7.10: Latency and resource utilization of the studied multipliers

As demonstrated in the two previous studies, stochastic number generators constitute a problem in the context of SC-based inference of neural networks; SNGs result in either a high processing latency or in a large hardware overhead. This conclusion agrees with the study of [KKY+16] where authors found that SC circuits perform $\times 4.61$ times faster when no SNG is implemented. The same study also reports that at a similar footprint, SC performs $\times 1.53$ times slower compared with a « conventional » 9-bit fixed-point, which also matches with our results.

7.4 Conclusions

This chapter has introduced the challenge of multi-operand adder footprint reduction when implementing a CNN with direct hardware mapping on an FPGA. Three potential solutions have been studied, respectively relying on serialization of adders, approximate computing, and stochastic computing.

Though originally promising, these solutions have proven ineffective with current FPGA architectures that do not lend themselves well to adder approximation and serialization. On one hand, the serialization of a cluster of adders does not reduce the footprint since the serializers require too many logic elements. On the other hand, the approximated adder is also ineffective, due to the structure of the logic blocks. **These conclusions motivate for introducing new specialized DSP blocks in FPGAs, implementing large adders fully in hardware.**

Finally, this chapter demonstrated that the stochastic arithmetic results in either a large computational latency or a large resource overhead, preventing its deployment in DHM-based implementations.

Chapter 8

Conclusions and Perspectives

8.1 Conclusions

This thesis addressed the problem of CNN mappings on FPGAs in the context of embedded smart cameras. Mainly, it has demonstrated that a dataflow-based approach and a direct hardware mapping are adequate solutions to meet the real-time constraints of FPGA-based smart camera networks. Indeed, these paradigms naturally exploit the *streaming* nature of CNN workloads, and fully take advantage of the large parallelism CNNs exhibit. In counterpart, direct hardware mapping results in large utilization of the hardware resources, making its implementation in resource-constrained devices a challenging task.

To address this challenge, the first contributions of this work are oriented towards adapting the structure of CNN workloads and the numerical precision they involve. As advocated in numerous studies, deep learning applications are resilient to approximate computing, and can be pruned [MTK+17] or quantized [GPMG18] without critically impacting their reliability. As demonstrated in chapter 5, pruning and quantization greatly impact the footprint of a given FPGA implementation, making the direct hardware mapping possible. In this context, we have proposed a design space exploration methodology, capable of deriving CNN topologies and arithmetic precisions according to the application and considered FPGA device. To conclude the model-based optimization part, multi-view CNNs have been introduced, opening new perspectives in terms of multi-view smart camera nodes.

In the next chapters, the manuscript has focused on the fine-grain optimization of the generated hardware architectures. In this scope, chapter 6 has started by demonstrating how a majority of the FIFO components, mapped between CNN actors, can be safely removed without affecting the behaviour of a dataflow CNN mapping. Then, a study of specialized multipliers and pipelined adder trees has been provided. These « tactics » result in a significant reduction of FPGA resource requirements, making direct hardware mapping feasible on « embedded » FPGAs. To support this demonstration, the deployment of a CNN-based OCR system on a Cyclone III smart camera has been detailed in chapter 6

Finally, the work described in this manuscript have brought to light an unexpected feature of FPGA-based CNN mappings: the high resource utilization of the adders. Indeed, after specializing the multipliers, up to 75 % of the resource inferred by a given mapping are allocated to the adder logic. Factually, CNN inference involves the addition of a large number of operands, which synthesizers implement by cascading logical resources. All our efforts to overcome this problem –discussed in chapter 7– were unfruitful, opening a new challenge in the area of FPGA-based CNN acceleration.

8.2 Perspectives and future directions

The prospects of this thesis are mainly oriented towards improving the efficiency, and reducing the resource utilization of the generated accelerators.

In this perspective, a promising direction is to investigate the performance of the DHM approach for binary neural networks, especially with the improvement in reliability they recently demonstrated [RCB17].

Furthermore, one can imagine a CNN fine-tuning process driven by DHM results. Similarly to the studies that explore the quantization during training [BPF+18], the value of the weights can be updated so that they jointly improve the modeling power and implementation efficiency. Note that early stages of this investigation are given in sec.6.6, and aim at modeling the resource utilization of a given mapping according to the bit-width, the topology, and the value of the convolution kernels.

Additionally to these opportunities, the efficiency improvements brought by MVCNNs motivate their implementation on multi-view smart cameras [YEBM02], where the first layers of the CNN are mapped on an FPGA chip at nearest of the sensor. This concept is corroborated by the fact that the first layers are usually more suitable for a dataflow mapping, while the last layers better fit a Von Neumann execution paradigm, as demonstrated in section.5.2.2.

Finally, this last point highlights the relevance of heterogeneous computing in the case of CNN inference; Even if this manuscript advocates the use of fine-grain FPGA devices, the addition of different hardware substrates, such as GPUs or many-cores, to a smart camera is currently a solution to be considered, especially with the emergence of low-power "embedded" GPUs.

Bibliography

[ABC+16] Martin Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, Xiaoqiang Zheng, and Google Brain. TensorFlow: A System for Large-Scale Machine Learning. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation - OSDI '16*, 2016.

[ABM13] Edoardo Ardizzone, Alessandro Bruno, and Giuseppe Mazzola. Saliency Based Image Cropping. In *Proceedings of the International Conference on Image Analysis and Processing - ICIAP '13*. Springer, Berlin, Heidelberg, 2013.

[ACFM16] Manoj Alwani, Han Chen, Michael Ferdman, and Peter Milder. Fused-layer CNN accelerators. In *Proceedings of the Annual International Symposium on Microarchitecture - MICRO '16*, volume 2016-Decem, 2016.

[ACRB16] Renzo Andri, Lukas Cavigelli, Davide Rossi, and Luca Benini. YodaNN: An ultra-low power convolutional neural network accelerator based on binary weights. *Proceedings of the IEEE Computer Society Annual Symposium on VLSI - ISVLSI '16*, 2016-Septe, 2016.

[AH13] Armin Alaghi and John P. Hayes. Survey of Stochastic Computing. *ACM Transactions on Embedded Computing Systems*, 2013.

[AH14] Armin Alaghi and John P Hayes. Fast and Accurate Computation using Stochastic Circuits. In *Proceedings of the Design, Automation & Test in Europe Conference & Exhibition - DATE '14*. IEEE, 2014.

[AHS15] Sajid Anwar, Kyuyeon Hwang, and Wonyong Sung. Fixed point optimization of deep convolutional neural networks for object recognition. In *Proceedings of the IEEE International Conference on Acoustics, Speech and Signal Processing - ICASSP '15*, 2015.

[AJK16] Rahman Atul, Lee Jongeun, and Choi Kiyoung. Efficient FPGA acceleration of Convolutional Neural Networks using logical-3D compute array. In *Proceedings of the Design, Automation & Test in Europe Conference & Exhibition - DATE '16*, Dresden, Germany, 2016. IEEE.

[Ala15] Armin Alaghi. *The Logic of Random Pulses : Stochastic Computing*. PhD thesis, University of Michigan, 2015.

[ALPO+15] Arash Ardakani, Francois Leduc-Primeau, Naoya Onizawa, Takahiro Hanyu, and Warren J. Gross. VLSI Implementation of Deep Neural Network Using Integral Stochastic Computing. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 25(10), 2015.

[AOC+17] Utku Aydonat, Shane O'Connell, Davor Capalija, Andrew C. Ling, and Gordon R. Chiu. An OpenCL(TM) Deep Learning Accelerator on Arria 10.

In ACM, editor, *Proceedings of the ACM/SIGDA International Symposium on Field-Programmable Gate Arrays - FPGA '17*, Monterey, California, USA, 2017. ACM.

[APBS18] Kamel Abdelouahab, Maxime Pelcat, François Berry, and Jocelyn Sérot. Accelerating CNN inference on FPGAs: A Survey. Technical report, Université Clermont Auvergne, 2018.

[APS+17] Kamel Abdelouahab, Maxime Pelcat, Jocelyn Serot, Cedric Bourrasset, and Francois Berry. Tactics to Directly Map CNN graphs on Embedded FPGAs. *IEEE Embedded Systems Letters*, 2017.

[BB14] Merwan Birem and François Berry. DreamCam: A modular FPGA-based smart camera architecture. *Journal of Systems Architecture*, 60(6), jun 2014.

[BC01] Bradley D. Brown and Howard C. Card. Stochastic neural computation I: Computational elements. *IEEE Transactions on Computers*, 50(9), 2001.

[Ben10] Khaled Benkrid. Reconfigurable Computing in the Multi-Core Era. In *Proceedings of the International Workshop on Highly Efficient Accelerators and Reconfigurable Technologies - HEART '10*, 2010.

[BKA+16] Jeremy Bottleson, Sungye Kim, Jeff Andrews, Preeti Bindu, Deepak N. Murthy, and Jingyi Jin. ClCaffe: OpenCL accelerated caffe for convolutional neural networks. In *Proceedings of the IEEE International Parallel and Distributed Processing Symposium - IPDPS'16*, 2016.

[Bot10] Leon Bottou. Large-Scale Machine Learning with Stochastic Gradient Descent. In *Proceedings of International Conference on Computational Statistics - COMPSTAT'10*. Springer, 2010.

[Bou16] Cedric Bourrasset. *High level synthesis of dataflow programs for image processing on FPGA-based smart camera. Application to machine learning*. PhD thesis, Universite Blaise Pascal, Clermont-Ferrand, feb 2016.

[BPF+18] Michaela Blott, Thomas Preusser, Nicholas Fraser, Giulio Gambardella, Kenneth O'Brien, and Yaman Umuroglu. FINN-R: An End-to-End Deep-Learning Framework for Fast Exploration of Quantized Neural Networks. *ACM Transactions on Reconfigurable Technology and Systems*, sep 2018.

[BSB13] Cedric Bourrasset, Jocelyn Serot, and Francois Berry. FPGA-based smart camera mote for pervasive wireless network. In *Proceedings of the International Conference on Distributed Smart Cameras - ICDSC'13*, 2013.

[CAD+12] Tomasz S. Czajkowski, Utku Aydonat, Dmitry Denisenko, John Freeman, Michael Kinsner, David Neto, Jason Wong, Peter Yiannacouras, and Deshanand P. Singh. From OpenCL to high-performance hardware on FPGAS. In *Proceedings of the International Conference on Field Programmable Logic and Applications - FPL '16*. IEEE, aug 2012.

[CB16] Lukas Cavigelli and Luca Benini. Origami : A 803 GOp/s/W Convolutional Network Accelerator. *IEEE Transactions on Circuits and Systems for Video Technology*, 8215(c), 2016.

[CBD14] Matthieu Courbariaux, Yoshua Bengio, and Jean-Pierre David. Training deep neural networks with low precision multiplications. *arXiv e-print*, dec 2014.

[CBD15] Matthieu Courbariaux, Yoshua Bengio, and Jean-Pierre David. BinaryConnect: Training Deep Neural Networks with binary weights during propagations. In *Advances in Neural Information Processing Systems - NIPS'15*, 2015.

[CES16] Yu-Hsin Chen, Joel Emer, and Vivienne Sze. Eyeriss: A Spatial Architecture for Energy-Efficient Dataflow for Convolutional Neural Networks. In *Proceedings of the International Symposium on Computer Architecture - ISCA '16*, 2016.

[CHB+16] Sebastien Caux, Edouard Hendrickx, François Berry, Maxime Pelcat, and Jocelyn Serot. Demo GPStudio. In *Proceedings of the International Conference on Distributed Smart Cameras - ICDSC'16*, New York, New York, USA, 2016. ACM Press.

[Cis17] Cisco. Cisco Visual Networking Index: Global Mobile Data Traffic Forecast Update, 2016–2021 White Paper. *Cisco*, 2017.

[CPC16] Alfredo Canziani, Adam Paszke, and Eugenio Culurciello. An Analysis of Deep Neural Network Models for Practical Applications. *arXiv e-print*, may 2016.

[CPS06] Kumar Chellapilla, Sidd Puri, and Patrice Simard. High Performance Convolutional Neural Networks for Document Processing. In *Proceedings of the International Workshop on Frontiers in Handwriting Recognition - FHR'06*. Suvisoft, oct 2006.

[CSJC10] Srimat Chakradhar, Murugan Sankaradas, Venkata Jakkula, and Srihari Cadambi. A Dynamically Configurable Coprocessor for Convolutional Neural Networks. *ACM SIGARCH Computer Architecture News*, 38(3), jun 2010.

[CX14] Jason Cong and Bingjun Xiao. Minimizing computation in convolutional neural networks. In *Proceedings of the International Conference on Artificial Neural Networks - ICANN '14*. Springer, 2014.

[DBSM07] Fabio Dias, Francois Berry, Jocelyn Serot, and Francois Marmoiton. Hardware, Design and Implementation Issues on a Fpga-Based Smart Camera. In *Proceedings of the International Conference on Distributed Smart Cameras - ICDSC'07*. IEEE, sep 2007.

[dDL00] Florent de Dinechin and Vincent Lefevre. Constant multipliers for FPGAs. *Parallel and Distributed Processing Techniques and Applications*, 2000.

[DDL+18] Li Du, Yuan Du, Yilei Li, Junjie Su, Yen Cheng Kuan, Chun Chen Liu, and Mau Chung Frank Chang. A Reconfigurable Streaming Deep Convolutional Neural Network Accelerator for Internet of Things. *IEEE Transactions on Circuits and Systems I: Regular Papers*, 65(1), 2018.

[DDS+09] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. Imagenet: A large-scale hierarchical image database. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition - CVPR '09*. IEEE, 2009.

[DFC+15] 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 *Proceedings of the International Symposium on Computer Architecture - ISCA '15*, jun 2015.

[DFK+07] Nirav Dave, Kermin Fleming, Myron King, Michael Pellauer, and Muralidaran Vijayaraghavan. Hardware acceleration of matrix multiplication on a Xilinx FPGA. In *Proceedings of ACM and IEEE International Conference on Formal Methods and Models for Co-Design, MEMOCODE'07*, 2007.

[DKT07] Jean-Pierre David, Kassem Kalach, and Nicolas Tittley. Hardware Complexity of Modular Multiplication and Exponentiation. *IEEE Transactions on Computers*, 56(10), oct 2007.

[DLV+16] Roberto DiCecco, Griffin Lacey, Jasmina Vasiljevic, Paul Chow, Graham Taylor, and Shawki Areibi. Caffeinated FPGAs: FPGA Framework For Convolutional Neural Networks. In *Proceedings of the International Conference on Field-Programmable Technology - FPT '16*, 2016.

[DM75] Jack B Dennis and David P Misunas. A Preliminary Architecture for a Basic Data-flow Processor. In *Proceedings of the International Symposium on Computer Architecture - ISCA '75*. ACM, 1975.

[DR01] Steven Derrien and Sanjay Rajopadhye. Loop tiling for reconfigurable accelerators. In *Proceedings of the International Conference on Field Programmable Logic and Applications - FPL '01*, volume 2147. Springer, 2001.

[DRM14] Richard Dorrance, Fengbo Ren, and Dejan Marković. A scalable sparse matrix-vector multiplication kernel for energy-efficient sparse-blas on FPGAs. In *Proceedings of the ACM/SIGDA International Symposium on Field-Programmable Gate Arrays - FPGA '14*, New York, New York, USA, 2014. ACM.

[EH94] J.G. Eldredge and B.L. Hutchings. RRANN: a hardware implementation of the backpropagation algorithm using reconfigurable FPGAs. In *Proceedings of the IEEE International Conference on Neural Networks - ICNN'94*, volume 4. IEEE, 1994.

[EP18] Amir Erfan Eshratifar and Massoud Pedram. Energy and Performance Efficient Computation Offloading for Deep Neural Networks in a Mobile Cloud Computing Environment. In *Proceedings of the Great Lakes Symposium on VLSI - GLSVLSI'18*, GLSVLSI '18, New York, NY, USA, 2018. ACM.

[EVW+10] Mark Everingham, Luc Van Gool, Christopher K. I. Williams, John Winn, and Andrew Zisserman. The Pascal Visual Object Classes (VOC) Challenge. *International Journal of Computer Vision*, 88(2), jun 2010.

[FFFP06] Li Fei-Fei, Rob Fergus, and Pietro Perona. One-shot learning of object categories. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 28(4), 2006.

[FMC+11] Clement Farabet, Berin Martini, Benoit Corda, Polina Akselrod, Eugenio Culurciello, and Yann LeCun. NeuFlow: A runtime reconfigurable dataflow processor for vision. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition - CVPR '11*. IEEE, jun 2011.

[FPH+09] C Farabet, C Poulet, J Y Han, Y LeCun, David R. Tobergte, and Shirley Curtis. CNP: An FPGA-based processor for Convolutional Networks. In *Proceedings of the International Conference on Field Programmable Logic and Applications - FPL '09*, 2009.

[FSNM17] Tomoya Fujii, Simpei Sato, Hiroki Nakahara, and Masato Motomura. An FPGA Realization of a Deep Convolutional Neural Network Using a Threshold Neuron Pruning. In *Proceedings of the International Symposium on Applied Reconfigurable Computing - ARC'16*, volume 9625, 2017.

[FTX17] Peter Forsyth, Raphael Tang, and Zehao Xu. An Empirical Study of Pruning and Quantization Methods for Neural Networks. Technical report, University of Waterloo, 2017.

[FUG+17] Nicholas J Fraser, Yaman Umuroglu, Giulio Gambardella, Michaela Blott, Philip Leong, Magnus Jahre, and Kees Vissers. Scaling Binarized Neural Networks on Reconfigurable Logic. In *Proceedings of the Workshop on Parallel Programming and Run-Time Management Techniques for Many-core Architectures and Design Tools and Architectures for Multicore Embedded Computing Platforms - PARMA-DITAM'17*. ACM, 2017.

[GAN+15] Suyog Gupta, Ankur Agrawal, Pritish Narayanan, Kailash Gopalakrishnan, and Pritish Narayanan. Deep Learning with Limited Numerical Precision. In *Proceedings of the International Conference on Machine Learning - ICML '15*, 2015.

[GDDM14] Ross Girshick, Jeff Donahue, Trevor Darrell, and Jitendra Malik. Rich Feature Hierarchies for Accurate Object Detection and Semantic Segmentation. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition - CVPR '14*, 2014.

[GGS+17] Urban Gregor, Krzysztof J. Geras, Ebrahimi Kahou Samira, Ozlem Aslan, Wang Shengjie, Abdelrahman Mohamed, Matthai Philipose, Matt Richardson, and Caruana Rich. Do Deep Convolutional Neural Networks need to be deep and convolutional ? In *Proceedings of the International Conference on Learning Representations - ICLR'17*, 2017.

[Gir15] Ross Girshick. Fast R-CNN. In *Proceedings of the IEEE International Conference on Computer Vision - ICCV '15*, 2015.

[GJD+14] Vinayak Gokhale, Jonghoon Jin, Aysegul Dundar, Berin Martini, and Eugenio Culurciello. A 240 G-ops/s mobile coprocessor for deep neural networks. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition - CVPR '14*, jun 2014.

[GK88] P. K. Gupta and R. Kumaresan. Binary Multiplication with PN Sequences. *IEEE Transactions on Acoustics, Speech, and Signal Processing*, 36(4), apr 1988.

[GMG16] Philipp Gysel, Mohammad Motamedi, and Soheil Ghiasi. Hardware-oriented Approximation of Convolutional Neural Networks. In *arXiv preprint*, 2016.

[GPMG18] Philipp Gysel, Jon Pimentel, Mohammad Motamedi, and Soheil Ghiasi. Ristretto: A Framework for Empirical Study of Resource-Efficient Inference in Convolutional Neural Networks. *IEEE Transactions on Neural Networks and Learning Systems*, 2018.

[GRT+16] Giulia Guidi, Enrico Reggiani, Lorenzo Di Tucci, Gianluca Durelli, Michaela Blott, and Marco D. Santambrogio. On How to Improve FPGA-Based Systems Design Productivity via SDAccel. In *Proceedings of the IEEE International Parallel and Distributed Processing Symposium - IPDPS'16*. IEEE, may 2016.

[Guo17] Tian Guo. Towards Efficient Deep Inference for Mobile Applications. *arXiv preprint*, 2017.

[GWC+17] Shasha Guo, Lei Wang, Baozi Chen, Qiang Dou, Yuxing Tang, and Zhisheng Li. FixCaffe: Training CNN with Low Precision Arithmetic Operations by Fixed Point Caffe. In *Proceedings of the International Workshop on Advanced Parallel Processing Technologies - APPT '17*. Springe, aug 2017.

[GWK+17] Jiuxiang Gu, Zhenhua Wang, Jason Kuen, Lianyang Ma, Amir Shahroudy, Bing Shuai, Ting Liu, Xingxing Wang, Li Wang, Gang Wang, Jianfei Cai, and Tsuhan Chen. Recent Advances in Convolutional Neural Networks. *Pattern Recognition*, 2017.

[Gys16] Philipp Gysel. *Ristretto: Hardware-Oriented Approximation of Convolutional Neural Networks*. PhD thesis, University of California, 2016.

[HCS+16] Itay Hubara, Matthieu Courbariaux, Daniel Soudry, Ran El-Yaniv, and Yoshua Bengio. Binarized neural networks. In *Advances in Neural Information Processing Systems - NIPS'16*, feb 2016.

[HCS+18] Itay Hubara, Matthieu Courbariaux, Daniel Soudry, Ran El-Yaniv, and Yoshua Bengio. Quantized Neural Networks: Training Neural Networks with Low Precision Weights and Activations. *Journal of Machine Learning Research*, sep 2018.

[HLM+16] 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. *Proceedings of the International Symposium on Computer Architecture - ISCA '16*, 16, 2016.

[HMD16] Song Han, Huizi Mao, and William J. Dally. Deep Compression - Compressing Deep Neural Networks with Pruning, Trained Quantization and Huffman Coding. *Proceedings of the International Conference on Learning Representations - ICLR'16*, 2016.

[Hor14] Mark Horowitz. Computing's energy problem (and what we can do about it). In *Proceedings of the IEEE International Solid-State Circuits - ISSCC '14*. IEEE, feb 2014.

[HPFA07] Stephan Hengstler, Daniel Prashanth, Sufen Fong, and Hamid Aghajan. MeshEye. In *Proceedings of the International conference on Information processing in sensor networks - IPSN '07*, New York, New York, USA, 2007. ACM Press.

[HPTD15] Song Han, Jeff Pool, John Tran, and William J Dally. Learning both Weights and Connections for Efficient Neural Network. In *Advances in Neural Information Processing Systems - NIPS'15*, 2015.

[HR16] Tyler Highlander and Andres Rodriguez. Very Efficient Training of Convolutional Neural Networks using Fast Fourier Transform and Overlap-and-Add. *arXiv preprint*, 2016.

[HS15] Kaiming He and Jian Sun. Convolutional Neural Networks at Constrained Time Cost. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition - CVPR '15*, 2015.

[HSS12] G. Hinton, N. Srivastava, and K. Swersky. A separate, adaptive learning rate for each connection. Slides of Lecture Neural Networks for Machine Learning. Technical report, University of Toronto, 2012.

[HW62] David H Hubel and Torsten N Wiesel. Receptive fields, binocular interaction and functional architecture in the cat's visual cortex. *The Journal of physiology*, 160(1), 1962.

[HZRS16] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep Residual Learning for Image Recognition. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition - CVPR '16*, 2016.

[IMA+16] Forrest N. Iandola, Matthew W. Moskewicz, Khalid Ashraf, Song Han, William J. Dally, and Kurt Keutzer. SqueezNet: AlexNet accuracy with 50x fewer parameters and 0.5MB Model Size. *arXiv e-print*, arXiv:1602, 2016.

[Int04] Intel FPGA. Implementing Multipliers in FPGA Devices. Technical report, Altera, 2004.

[Int14a] Intel FPGA. *Cyclone V Device Handbook*, volume 1. 2014.

[Int14b] Intel FPGA. Floating-Point IP Cores User Guide. Technical report, 2014.

[Int16] Intel FPGA. The Intel FPGA SDK for Open Computing Language (OpenCL), 2016.

[Int17] Intel FPGA. Intel Stratix 10 Variable Precision DSP Blocks User Guide. Technical report, Intel FPGA, 2017.

[Int18a] Intel FPGA. Cyclone V Device Overview. Technical report, 2018.

[Int18b] Intel FPGA. Intel Stratix 10 Product Table. 2018.

[IS15] Sergey Ioffe and Christian Szegedy. Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift. In Francis Bach and David Blei, editors, *Proceedings of the International Conference on Machine Learning - ICML '15*, volume 37, Lille, France, 2015.

[JCP02] Ju-wook Jang, Seonil Choi, and Viktor K Prasanna. Area and Time E ffi cient Implementations of Matrix Multiplication on FPGAs. In *Proceedings of the International Conference on Field-Programmable Technology - FPT'02*, 2002.

[JCP05] Ju Wook Jang, Seonil B. Choi, and Viktor K. Prasanna. Energy- and time-efficient matrix multiplication on FPGAs. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 2005.

[JHL15] Honglan Jiang, Jie Han, and Fabrizio Lombardi. A Comparative Review and Evaluation of Approximate Adders. In *Proceedings of the Great Lakes Symposium on VLSI - GLSVLSI'15*. ACM Press, 2015.

[JSD+14] Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross Girshick, Sergio Guadarrama, and Trevor Darrell. Caffe: Convolutional Architecture for Fast Feature Embedding. In *Proceedings of the ACM International Conference on Multimedia - MM'14*, 2014.

[KB14] Diederik P. Kingma and Jimmy Ba. Adam: A Method for Stochastic Optimization. In *Proceedings of the International Conference on Learning Representations - ICLR'15*, dec 2014.

[KDA+16] Ivan Krasin, Tom Duerig, Neil Alldrin, Vittorio Ferrari, and Sami Abu-El-Haija. Openimages: A public dataset for large-scale multi-label and multi-class image classification, 2016.

[KDC12] Srinidhi Kestur, John D. Davis, and Eric S. Chung. Towards a universal FPGA matrix-vector multiplication architecture. In *Proceedings of the IEEE International Symposium on Field-Programmable Custom Computing Machines - FCCM'12*, 2012.

[Khr15] Khronos. OpenCL: The open standard for parallel programming of heterogeneous systems, 2015.

[KKY+16] Kyounghoon Kim, Jungki Kim, Joonsang Yu, Jungwoo Seo, Jongeun Lee, and Kiyoung Choi. Dynamic Energy-accuracy Trade-off Using Stochastic Computing in Deep Neural Networks. In *Proceedings of the Annual Conference on Design Automation - DAC '16*, number 1, New York, NY, USA, 2016. ACM.

[KMNM17] Jong Hwan Ko, Burhan Ahmad Mudassar, Taesik Na, and Saibal Mukhopadhyay. Design of an Energy-Efficient Accelerator for Training of Convolutional Neural Networks using Frequency-Domain Computation. In *Proceedings of the Annual Conference on Design Automation - DAC '17*, 2017.

[Kon17] Moller Konrad. *Run-time Recongurable Constant Multiplication on Field Programmable Gate Arrays*. PhD thesis, Kassel University, 2017.

[Kri09] Alex Krizhevsky. Learning Multiple Layers of Features from Tiny Images. Technical report, Technical Report, University of Toronto, 2009.

[KSGH12] Alex Krizhevsky, Ilya Sutskever, Hinton Geoffrey E., and Geoffrey E Hinton. ImageNet Classification with Deep Convolutional Neural Networks. In *Advances in Neural Information Processing Systems - NIPS'12*, 2012.

[KTB15] Matthias Kümmerer, Lucas Theis, and Matthias Bethge. Deep Gaze I: Boosting Saliency Prediction with Feature Maps Trained on ImageNet. In *Proceedings of the International Conference on Learning Representations - ICLR'15*, 2015.

[LAE+16] Wei Liu, Dragomir Anguelov, Dumitru Erhan, Christian Szegedy, Scott Reed, Cheng-Yang Fu, and Alexander C. Berg. SSD: Single Shot MultiBox Detector. In *Proceedings of the European Conference on Computer Vision - ECCV'16*. Springer, 2016.

[LBBH98] Y LeCun, L Bottou, Y Bengio, and P Haffner. Gradient Based Learning Applied to Document Recognition. In *Proceedings of the IEEE*, 1998.

[LBD+90] Yann LeCun, Bernhard E Boser, John S Denker, Donnie Henderson, Richard E Howard, Wayne E Hubbard, and Lawrence D Jackel. Handwritten digit recognition with a back-propagation network. In *Advances in Neural Information Processing Systems - NIPS'90*, 1990.

[LBH15] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. Deep learning. *Nature*, 521(7553), 2015.

[LCY13] Min Lin, Qiang Chen, and Shuicheng Yan. Network In Network. *arXiv preprint*, arXiv:1312, dec 2013.

[LDJ+17] Zhiqiang Liu, Yong Dou, Jingfei Jiang, Jinwei Xu, Shijie Li, Yongmei Zhou, and Yingnan Xu. Throughput-Optimized FPGA Accelerator for Deep Convolutional Neural Networks. *ACM Transactions on Reconfigurable Technology and Systems*, 10(3), 2017.

[LFJ+16] Huimin Li, Xitian Fan, Li Jiao, Wei Cao, Xuegong Zhou, and Lingli Wang. A high performance FPGA-based accelerator for large-scale convolutional neural networks. In *Proceedings of the International Conference on Field Programmable Logic and Applications - FPL '16*. IEEE, aug 2016.

[LG15] Andrew Lavin and Scott Gray. Fast Algorithms for Convolutional Neural Networks. *arXiv e-print*, arXiv: 150, sep 2015.

[LL11] Peng Li and David J. Lilja. Using stochastic computing to implement digital image processing algorithms. In *Proceedings of the IEEE International Conference on Computer Design - ICCD '11*, 2011.

[LLXY17] Liqiang Lu, Yun Liang, Qingcheng Xiao, and Shengen Yan. Evaluating fast algorithms for convolutional neural networks on FPGAs. In *Proceedings of the IEEE Annual International Symposium on Field-Programmable Custom Computing Machines - FCCM '17*, 2017.

[LM87] Edward A Lee and David G Messerschmitt. Synchronous data flow. In *Proceedings of the IEEE*, 1987.

[LM97] Tomas Lundin and Perry Moerland. Quantization and Pruning of Multilayer Perceptrons: Towards Compact Neural Networks. Technical report, Dalle Molle Institute for Perceptual Artificial Intelligence, 1997.

[LMB+14] Tsung-Yi Lin, Michael Maire, Serge Belongie, James Hays, Pietro Perona, Deva Ramanan, Piotr Dollár, and C. Lawrence Zitnick. Microsoft COCO: Common Objects in Context. In *Proceedings of the European Conference on Computer Vision - ECCV'14*. Springer, 2014.

[LMH04] M. Leeser, S. Miller, and Haiqian Yu. Smart Camera Based on Reconfigurable Hardware Enables Diverse Real-Time Applications. In *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines - FCCM'04*. IEEE, 2004.

[LSD15] Jonathan Long, Evan Shelhamer, and Trevor Darrell. Fully Convolutional Networks for Semantic Segmentation. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition - CVPR '15*, 2015.

[LTA16] Darryl Lin, Sachin Talathi, and V Annapureddy. Fixed Point Quantization of Deep Convolutional Networks. In *Proceedings of the International Conference on Machine Learning - ICML '16*, 2016.

[LWF+15] Baoyuan Liu, Min Wang, Hassan Foroosh, Marshall Tappen, and Marianna Pensky. Sparse Convolutional Neural Networks. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition - CVPR '15*, 2015.

[LYL+17] Shuang Liang, Shouyi Yin, Leibo Liu, Wayne Luk, and Shaojun Wei. FP-BNN: Binarized Neural Network on FPGA. *Neurocomputing*, oct 2017.

[MAFL10] H R Mahdiani, A Ahmadi, S M Fakhraie, and C Lucas. Bio-Inspired Imprecise Computational Blocks for Efficient VLSI Implementation of Soft-Computing Applications. *IEEE Transactions on Circuits and Systems I: Regular Papers*, 57(4), apr 2010.

[Mag17] Luca Maggiani. *Heterogeneous Smart Cameras: towards the Internet of Reconfigurable Things*. PhD thesis, Scuola Superiore SantAnna, 2017.

[MBP+15] Luca Maggiani, Cedric Bourrasset, Matteo Petracca, Francois Berry, Paolo Pagano, and Claudio Salvadori. HOG-Dot: A Parallel Kernel-Based Gradient Extraction for Embedded Image Processing. *IEEE Signal Processing Letters*, 2015.

[MCVS17a] Yufei Ma, Yu Cao, Sarma Vrudhula, and Jae-sun Seo. An automatic RTL compiler for high-throughput FPGA implementation of diverse deep convolutional neural networks. In *Proceedings of the International Conference on Field Programmable Logic and Applications - FPL '17*. IEEE, sep 2017.

[MCVS17b] Yufei Ma, Yu Cao, Sarma Vrudhula, and Jae-sun Seo. Optimizing Loop Operation and Dataflow in FPGA Acceleration of Deep Convolutional Neural Networks. In *Proceedings of the ACM/SIGDA International Symposium on Field-Programmable Gate Arrays - FPGA '17*, 2017.

[MDC+16] Paolo Meloni, Gianfranco Deriu, Francesco Conti, Igor Loi, Luigi Raffo, and Luca Benini. Curbing the Roofline : a Scalable and Flexible Architecture for CNNs on FPGA. In *Proceedings of the ACM International Conference on Computing Frontiers - CF '16*, Como, Italy, 2016.

[MGAG16] Mohammad Motamedi, Philipp Gysel, Venkatesh Akella, and Soheil Ghiasi. Design space exploration of FPGA-based Deep Convolutional Neural Networks. In *Proceedings of the Asia and South Pacific Design Automation Conference - ASPDAC'16*, jan 2016.

[MGG17] Mohammad Motamedi, Philipp Gysel, and Soheil Ghiasi. PLACID: A Platform for FPGA-Based Accelerator Creation for DCNNs. *ACM Transactions on Multimedia Computing, Communications, and Applications*, 13(4), sep 2017.

[Mic17] Microsoft. Microsoft unveils Project Brainwave for real-time AI, 2017.

[Mit16] Sparsh Mittal. A Survey of Techniques for Approximate Computing. *ACM Computing Surveys*, 48(4), mar 2016.

[MKC+17] Yufei Ma, Minkyu Kim, Yu Cao, Sarma Vrudhula, and Jae-sun Seo. End-to-end scalable FPGA accelerator for deep residual networks. In *Proceedings of the IEEE International Symposium on Circuits and Systems - ISCAS '17*. IEEE, may 2017.

[MM16] Bert Moons and Verhelst Marian. A 0.3–2.6 TOPS/W precision- scalable processor for real-time large-scale ConvNets. In *IEEE Symposium on VLSI Circuits*, number June, 2016.

[MSC+16] Yufei Ma, Naveen Suda, Yu Cao, Jae Sun Seo, and Sarma Vrudhula. Scalable and modularized RTL compilation of Convolutional Neural Networks onto FPGA, 2016.

[MTK+17] Pavlo Molchanov, Stephen Tyree, Tero Karras, Timo Aila, and Jan Kautz. Pruning Convolutional Neural Networks for Resource Efficient Learning. *arXiv preprint*, 2017.

[NFS17] Hiroki Nakahara, Tomoya Fujii, and Shimpei Sato. A fully connected layer elimination for a binarizec convolutional neural network on an FPGA. In *Proceedings of the International Conference on Field Programmable Logic and Applications - FPL '17*. IEEE, sep 2017.

[NSB+17] Eriko Nurvitadhi, Suchit Subhaschandra, Guy Boudoukh, Ganesh Venkatesh, Jaewoong Sim, Debbie Marr, Randy Huang, Jason OngGee-Hock, Yeong Tat Liew, Krishnan Srivatsan, and Duncan Moss. Can FPGAs Beat GPUs in Accelerating Next-Generation Deep Neural Networks? In *Proceedings of the ACM/SIGDA International Symposium on Field-Programmable Gate Arrays - FPGA '17*, 2017.

[Nvi15] Nvidia. GPU-Based Deep Learning Inference: A Performance and Power Analysis. *White Paper*, 2015.

[Nvi16] Nvidia. Nvidia Tesla P100 GPU Architecture. *White Paper*, 2016.

[Nvi17] Nvidia. Nvidia Tesla V100 GPU Architecture. *White Paper*, (v1.1), 2017.

[NW11] Yuval Netzer and Tao Wang. Reading digits in natural images with unsupervised feature learning. In *Advances in Neural Information Processing Systems - NIPS'11*, 2011.

[NYFS18] Hiroki Nakahara, Haruyoshi Yonekawa, Tomoya Fujii, and Shimpei Sato. A Lightweight YOLOv2. In *Proceedings of the ACM/SIGDA International Symposium on Field-Programmable Gate Arrays - FPGA '18*, New York, New York, USA, 2018. ACM Press.

[OJU+16] Francisco Ortega, Jose M. Jerez, Daniel UrdaMunoz, Rafael LuqueBaena, and Leonardo Franco. Efficient Implementation of the Backpropagation Algorithm in FPGAs and Microcontrollers. *IEEE Transactions on Neural Networks and Learning Systems*, 27(9), sep 2016.

[ORK+15] Kalin Ovtcharov, Olatunji Ruwase, Joo-young Kim, Jeremy Fowers, Karin Strauss, and Eric Chung. Accelerating Deep Convolutional Neural Networks Using Specialized Hardware. *White paper*, feb 2015.

[PBMB17] Maxime Pelcat, Cedric Bourrasset, Luca Maggiani, and Francois Berry. Design productivity of a high level synthesis compiler versus HDL. In *Proceedings of the International Conference on Embedded Computer Systems: Architectures, Modeling and Simulation - SAMOS'16*, 2017.

[PBP+17] Adrien ProstBoucle, Alban Bourge, Frédéric Pétrot, Hande Alemdar, Nicholas Caldwell, and Vincent Leroy. Scalable High-Performance Architecture for Convolutional Ternary Neural Networks on FPGA. In *Proceedings of the International Conference on Field Programmable Logic and Applications - FPL '17*, jul 2017.

[Per17] Hugh Perkins. Deep CL: OpenCL library to train deep convolutional neural networks, 2017.

[PM14] Yu Pan and Pramod Kumar Meher. Bit-Level Optimization of Adder-Trees for Multiple Constant Multiplications for Efficient FIR Filter Implementation. *IEEE Transactions on Circuits and Systems I: Regular Papers*, 61(2), feb 2014.

[QWY+16] Jiantao Qiu, Jie Wang, Song Yao, Kaiyuan Guo, Boxun Li, Erjin Zhou, Jincheng Yu, Tianqi Tang, Ningyi Xu, Sen Song, Yu Wang, and Huazhong Yang. Going Deeper with Embedded FPGA Platform for Convolutional Neural Network. In *Proceedings of the ACM/SIGDA International Symposium on Field-Programmable Gate Arrays - FPGA '16*, New York, NY, USA, 2016. ACM.

[RCB17] Manuele Rusci, Lukas Cavigelli, and Luca Benini. Design Automation for Binarized Neural Networks: A Quantum Leap Opportunity? *arXiv preprint*, nov 2017.

[RCLC17] Xukan Ran, Haoliang Chen, Zhenming Liu, and Jiasi Chen. Delivering Deep Learning to Mobile Devices via Offloading. In *Proceedings of the Workshop on Virtual Reality and Augmented Reality Network*, VR/AR Network '17, New York, NY, USA, 2017. ACM.

[RDGF16] Joseph Redmon, Santosh Divvala, Ross Girshick, and Ali Farhadi. You Only Look Once: Unified, Real-Time Object Detection. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition - CVPR '16*, 2016.

[RDS+14] Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, Alexander C. Berg, and Li Fei-Fei. ImageNet Large Scale Visual Recognition Challenge. *International Journal of Computer Vision*, 115(3), sep 2014.

[RF18] Joseph Redmon and Ali Farhadi. YOLOv3: An Incremental Improvement. Technical report, University of Washington, apr 2018.

[RHGS17] Shaoqing Ren, Kaiming He, Ross Girshick, and Jian Sun. Faster R-CNN: Towards Real-Time Object Detection with Region Proposal Networks. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 2017.

[RHR+17] Samyam Rajbhandari, Yuxiong He, Olatunji Ruwase, Michael Carbin, Trishul Chilimbi, Samyam Rajbhandari, Yuxiong He, Olatunji Ruwase, Michael Carbin, Trishul Chilimbi, Samyam Rajbhandari, Yuxiong He, Olatunji Ruwase, Michael Carbin, Trishul Chilimbi, Samyam Rajbhandari, Yuxiong He, Olatunji Ruwase, Michael Carbin, and Trishul Chilimbi. Optimizing CNNs on Multicores for Scalability, Performance and Goodput. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems - ASPLOS'17*, volume 51. ACM, apr 2017.

[RLL+17] Ao Ren, Ji Li, Zhe Li, Caiwen Ding, Xuehai Qian, Qinru Qiu, Bo Yuan, and Yanzhi Wang. SC-DCNN: Highly-Scalable Deep Convolutional Neural Network using Stochastic Computing. *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems - ASPLOS'17*, 2017.

[RWA+16] 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 *Proceedings of the International Symposium on Computer Architecture - ISCA '16*. IEEE, 2016.

[SAWM08] Mohammed A-Megeed Salem, Markus Appel, Frank Winkler, and Beate Meffert. FPGA-based Smart Camera for 3D wavelet-based image segmentation. In *Proceedings of the International Conference on Distributed Smart Cameras - ICDSC'08*. IEEE, sep 2008.

[SBB16] Jocelyn Serot, François Berry, and Cedric Bourrasset. High-level dataflow programming for real-time image processing on smart cameras. *Journal of Real-Time Image Processing*, 12(4), dec 2016.

[SCD+16] Naveen Suda, Vikas Chandra, Ganesh Dasika, Abinash Mohanty, Yufei Ma, Sarma Vrudhula, Jae-sun Seo, and Yu Cao. Throughput-Optimized OpenCL-based FPGA Accelerator for Large-Scale Convolutional Neural Networks. In *Proceedings of the ACM/SIGDA International Symposium on Field-Programmable Gate Arrays - FPGA '16*, 2016.

[SCYE17] Vivienne Sze, Yu-Hsin Chen, Tien-Ju Yang, and Joel Emer. Efficient Processing of Deep Neural Networks: A Tutorial and Survey. *Proceedings of the IEEE*, 105(12), dec 2017.

[Sho94] Richard G Shoup. Parameterized convolution filtering in a field programmable gate array. In *Proceedings of the International Workshop on Field Programmable Logic and Applications on More FPGAs.*, 1994.

[SJC+09] Murugan Sankaradas, Venkata Jakkula, Srihari Cadambi, Srimat Chakradhar, Igor Durdanovic, Eric Cosatto, and Hans Peter Graf. A Massively Parallel Coprocessor for Convolutional Neural Networks. In *Proceedings of the IEEE International Conference on Acoustics, Speech and Signal Processing - ICASSP '17*. IEEE, jul 2009.

[SLJ+15] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. Going Deeper with Convolutions. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition - CVPR '15*, 2015.

[SMKLM15] Hang Su, Subhransu Maji, Evangelos Kalogerakis, and Erik Learned-Miller. Multi-view Convolutional Neural Networks for 3D Shape Recognition. In *Proceedings of the IEEE International Conference on Computer Vision - ICCV '15*. IEEE, dec 2015.

[SPM+16] Hardik Sharma, Jongse Park, Divya Mahajan, Emmanuel Amaro, Joon Kyung Kim, Chenkai Shao, Asit Mishra, and Hadi Esmaeilzadeh. From high-level deep neural models to FPGAs. In *Proceedings of the International Symposium on Microarchitecture - MICRO '16*, 2016.

[SQH+18] Junzhong Shen, Yuran Qiao, You Huang, Mei Wen, and Chunyuan Zhang. Towards a Multi-array Architecture for Accelerating Large-scale Matrix Multiplication on FPGAs. In *Proceedings of the International Symposium on Circuits and Systems - ISCAS'18*. IEEE, may 2018.

[STR+15] Amos Sironi, Bugra Tekin, Roberto Rigamonti, Vincent Lepetit, and Pascal Fua. Learning separable filters. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 37(1), 2015.

[Sut66] Ivan Sutherland. *Online graphical specification of procedures*. PhD thesis, MIT, 1966.

[SZ14] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv preprint*, arXiv:1409, 2014.

[Tay06] Michael Taylor. CSE 548: Computer Architecture. Dataflow Computers. Technical report, University of Washington, 2006.

[TKTH18] Lucas Theis, Iryna Korshunova, Alykhan Tejani, and Ferenc Huszár. Faster gaze prediction with dense networks and Fisher pruning. *arXiv e-print*, jan 2018.

[UFG+17] Yaman Umuroglu, Nicholas J Fraser, Giulio Gambardella, Michaela Blott, Philip Leong, Magnus Jahre, and Kees Vissers. FINN: A Framework for Fast, Scalable Binarized Neural Network Inference. In *Proceedings of the ACM/SIGDA International Symposium on Field-Programmable Gate Arrays - FPGA '17*, 2017.

[VB16] Stylianos Venieris and Christos Bouganis. FpgaConvNet: A Framework for Mapping Convolutional Neural Networks on FPGAs. In *Proceedings of the IEEE Annual International Symposium on Field-Programmable Custom Computing Machines - FCCM '16*, 2016.

[VB17] Stylianos Venieris and Christos Bouganis. Latency-Driven Design for FPGA-based Convolutional Neural Networks. In *Proceedings of the International Conference on Field Programmable Logic and Applications - FPL '17*, 2017.

[VKB18] Stylianos I. Venieris, Alexandros Kouris, and Christos-Savvas Bouganis. Toolflows for Mapping Convolutional Neural Networks on FPGAs. *ACM Computing Surveys*, 51(3), jun 2018.

[Von56] J Von Neumann. Probabilistic logics and the synthesis of reliable organisms from unreliable components. *Automata Studies*, 1956.

[VP07] Yevgen Voronenko and Markus Püschel. Multiplierless multiple constant multiplication. *ACM Transactions on Algorithms*, 3(2), may 2007.

[Wal17] E. Walters. Reduced-Area Constant-Coefficient and Multiple-Constant Multipliers for Xilinx FPGAs with 6-Input LUTs. *Electronics*, 6(4), 2017.

[Wan17] Dong Wang. PipeCNN: An OpenCL-based FPGA Accelerator for Convolutinal Neural Networks. In *Proceedings of the International Conference on Field-Programmable Technology - FPT '17*, 2017.

[WBSS04] Z. Wang, A.C. Bovik, H.R. Sheikh, and E.P. Simoncelli. Image Quality Assessment: From Error Visibility to Structural Similarity. *IEEE Transactions on Image Processing*, 13(4), apr 2004.

[Wil91] D. Williamson. Dynamically scaled fixed point arithmetic. In *Proceedings of the IEEE Pacific Rim Conference on Communications, Computers and Signal Processing Conference*. IEEE, 1991.

[Win80] Shmuel Winograd. *Arithmetic complexity of computations*, volume 33. Siam, 1980.

[WLW+16] Jiaxiang Wu, Cong Leng, Yuhang Wang, Qinghao Hu, and Jian Cheng. Quantized Convolutional Neural Networks for Mobile Devices. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition - CVPR '16*, 2016.

[Woo14] Josh Woodhouse. Big, big, big data: higher and higher resolution video surveillance, 2014.

[WOWL15] Lijun Wang, Wanli Ouyang, Xiaogang Wang, and Huchuan Lu. Visual Tracking with Fully Convolutional Networks. *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition - CVPR'15*, 2015.

[WWP09] Samuel Williams, Andrew Waterman, and David Patterson. Roofline: An insightful visual performance model for multicore architectures. *Communications of the ACM*, 52(4), apr 2009.

[WZY17] Yi Wang, Mingxu Zhang, and Jing Yang. Exploiting Parallelism for Convolutional Connections in Processing-In-Memory Architecture. In *Proceedings of the Annual Conference on Design Automation - DAC '17*, 2017.

[Xil13] Xilinx. *Introduction to FPGA Design with Vivado High-Level Synthesis*, volume 998. 2013.

[YCS17] Tien-Ju Yang, Yu-Hsin Chen, and Vivienne Sze. Designing Energy-Efficient Convolutional Neural Networks using Energy-Aware Pruning. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition - CVPR '17*, 2017.

[YEBM02] Jc Yang, M Everett, C Buehler, and L McMillan. A real-time distributed light field camera. *The Eurographics Association*, 2002.

[ZFZ+16] Chen Zhang, Zhenman Fang, Peipei Zhou, Peichen Pan, and Jason Cong. Caffeine: Caffeine: Towards uniformed representation and acceleration for deep convolutional neural networks. In *Proceedings of the International Conference on Computer-Aided Design - ICCAD '16*, New York, New York, USA, 2016. ACM.

[ZHMD17] Chenzhuo Zhu, Song Han, Huizi Mao, and William J. Dally. Trained Ternary Quantization. In *Proceedings of the International Conference on Learning Representations - ICLR'17*, dec 2017.

[ZL17] Jialiang Zhang and Jing Li. Improving the Performance of OpenCL-based FPGA Accelerator for Convolutional Neural Network. In *Proceedings of the ACM/SIGDA International Symposium on Field-Programmable Gate Arrays - FPGA '17*, 2017.

[ZLS+15] Chen Zhang, Peng Li, Guangyu Sun, Yijin Guan, Bingjun Xiao, and Jason Cong. Optimizing FPGA-based Accelerator Design for Deep Convolutional Neural Networks. In *Proceedings of the ACM/SIGDA International Symposium on Field-Programmable Gate Arrays - FPGA '15*, FPGA, 2015.

[ZOLW15] Rui Zhao, Wanli Ouyang, Hongsheng Li, and Xiaogang Wang. Saliency detection by multi-context deep learning. *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition - CVPR '15*, 2015.

[ZP05] Ling Zhuo and Viktor K. Prasanna. Sparse Matrix-Vector multiplication on FPGAs. In *Proceedings of the ACM/SIGDA International Symposium on Field-Programmable Gate Arrays - FPGA '05*, New York, New York, USA, 2005. ACM.

[ZP17] Chi Zhang and Viktor Prasanna. Frequency Domain Acceleration of Convolutional Neural Networks on CPU-FPGA Shared Memory System. In *Proceedings of the ACM/SIGDA International Symposium on Field-Programmable Gate Arrays - FPGA '17*, 2017.

[ZSZ+17] Ritchie Zhao, Weinan Song, Wentao Zhang, Tianwei Xing, Jeng-Hau Lin, Mani Srivastava, Rajesh Gupta, and Zhiru Zhang. Accelerating Binarized Convolutional Neural Networks with Software-Programmable FPGAs. In *Proceedings of the ACM/SIGDA International Symposium on Field-Programmable Gate Arrays - FPGA '17*, 2017.

[ZWN+16] 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. *arXiv e-print*, 2016.

[ZWS+16] Chen Zhang, Di Wu, Jiayu Sun, Guangyu Sun, Guojie Luo, and Jason Cong. Energy-Efficient CNN Implementation on a Deeply Pipelined FPGA Cluster. In *Proceedings of the International Symposium on Low Power Electronics and Design - ISLPED '16*, 2016.

[ZWW+17] Shuchang Zhou, Yuzhi Wang, He Wen, Qinyao He, and Yuheng Zou. Balanced Quantization: An Effective and Efficient Approach to Quantized Neural Networks. *Journal of Computer Science and Technology*, 32, 2017.

Publications

Journals

- K. Abdelouahab, M. Pelcat, J. Sérot and F. Berry (2017) *«Tactics to Directly Map CNN graphs on Embedded FPGAs»*, IEEE Embedded Systems Letters.

Conference Proceedings

- J. Bonnard, K. Abdelouahab, M. Pelcat and F.Berry (2018) *Real-time Embedded Object Classification with FPGA-based Distributed Multi-View CNNs*, Submitted to the Design Automation Conference - DAC'19 (submitted)
- K. Abdelouahab, M. Pelcat, and F. Berry (2018) *«The Challenge of Multi-Operand Adders in CNNs on FPGAs, And How NOT to Solve It!»*. Proceedings of the International Conference on Embedded Computer Systems: Architectures, Modeling and Simulation - SAMOS'18.
- K. Abdelouahab, M. Pelcat, and F. Berry (2017) *« PhD Forum: Why TanH can be a Hardware Friendly Activation Function for CNNs»*. Proceedings of the 11th International Conference on Distributed Smart Cameras - ICDSC'17.
- K. Abdelouahab, C. Bourrasset, M.Pelcat, J. Sérot, J.C.Quinton and F. Berry (2016) *«A Holistic Approach for Optimizing DSP Block Utilization of a CNN implementation on FPGA»*. Proceedings of the 10th International Conference on Distributed Smart Cameras - ICDSC'16.

Book Chapters

- K. Abdelouahab, M. Pelcat and F. Berry (2018) *«Accelerating CNN inference on FPGAs: A Survey»*, Deep Learning in Computer Vision: Theories and Applications (submitted).

Appendix A

Topology of Popular CNN Models

The following tables detail the topology of the CNNs studied in this manuscript. For each network, the convolution and fully connected layers, with their dimension, are listed. The presence of activation, batch normalization or sub-sampling is denoted in the last column of the tables. This last column also gives the order of the layers following each convolution.

TABLE A.1: Object Detectors

Network	Layer	C	N	J × K	U × V	Act, pool, BN
RCNN	conv1	3	96	11x11	55x55	ReLU+Pool+BN
	conv2	48	256	5x5	27x27	ReLU+Pool+BN
	conv3	256	384	3x3	13x13	ReLU
	conv4	192	384	3x3	13x13	ReLU
	conv5	192	256	3x3	13x13	ReLU+Pool
	fc6	9216	4096	-	-	ReLU
	fc7	4096	4096	-	-	ReLU
	fc-rcnn	4096	200	-	-	-
Yolov2-tiny	conv1	3	16	3x3	416x416	BN+ReLU+Pool
	conv2	16	32	3x3	208x208	BN+ReLU+Pool
	conv3	32	64	3x3	104x104	BN+ReLU+Pool
	conv4	64	128	3x3	52x52	BN+ReLU+Pool
	conv5	128	256	3x3	26x26	BN+ReLU+Pool
	conv6	256	512	3x3	13x13	BN+ReLU+Pool
	conv7	512	1024	3x3	12x12	BN+ReLU
	conv8	1024	512	3x3	12x12	BN+ReLU
	conv9	512	425	1x1	12x12	-

TABLE A.2: Classifiers

Network	Layer	C	N	J × K	U × V	Act, pool, BN
LeNet5	conv1	1	20	5x5	24x24	Pool
	conv2	20	50	5x5	8x8	Pool
	fc1	800	500	-	1x1	ReLU
	fc2	500	10	-	1x1	Softmax
Cifar 10	conv1	3	32	5x5	32x32	Pool+ReLU
	conv2	32	32	5x5	16x16	ReLU+Pool
	conv3	32	64	5x5	8x8	ReLU+Pool
	fc1	1024	10	-	1x1	Softmax
AlexNet	conv1	3	96	11x11	55x55	ReLU+BN+Pool
	conv2	48	256	5x5	27x27	ReLU+BN+Pool
	conv3	256	384	3x3	13x13	ReLU
	conv4	192	384	3x3	13x13	ReLU
	conv5	192	256	3x3	13x13	ReLU+Pool
	fc6	9216	4096	-	1x1	ReLU
	fc7	4096	4096	-	1x1	ReLU
	fc8	4096	1000	-	1x1	Softmax
VGG16	conv1-1	3	64	3x3	224x224	ReLU
	conv1-2	64	64	3x3	224x224	ReLU+Pool
	conv2-1	64	128	3x3	112x112	ReLU
	conv2-2	128	128	3x3	112x112	ReLU+Pool
	conv3-1	128	256	3x3	56x56	ReLU
	conv3-2	256	256	3x3	56x56	ReLU
	conv3-3	256	256	3x3	56x56	ReLU+Pool
	conv4-1	256	512	3x3	28x28	ReLU
	conv4-2	512	512	3x3	28x28	ReLU
	conv4-3	512	512	3x3	28x28	ReLU+Pool
	conv5-1	512	512	3x3	14x14	ReLU
	conv5-2	512	512	3x3	14x14	ReLU
	conv5-3	512	512	3x3	14x14	ReLU+Pool
	fc6	25088	4096	-	1x1	ReLU
	fc7	4096	4096	-	1x1	ReLU
	fc8	4096	1000	-	1x1	Softmax
Darknet	conv1	3	16	3x3	256x256	BN+ReLU+Pool
	conv2	16	32	3x3	128x128	BN+ReLU+Pool
	conv3	32	64	3x3	64x64	BN+ReLU+Pool
	conv4	64	128	3x3	32x32	BN+ReLU+Pool
	conv5	128	256	3x3	16x16	BN+ReLU+Pool
	conv6	256	512	3x3	8x8	BN+ReLU+Pool
	conv7	512	1024	3x3	4x4	BN+ReLU+Pool
	conv8	1024	1000	3x3	6x6	ReLU+Pool+Softmax

Appendix B

Direct Hardware Mapping with Haddoc2

This appendix describes the blocks introduced in the HADDOC2 tool. This tool relies on a hierarchical construction of the CNN actors; Similarly to the first version, the low-grain actors (multipliers, adders, line buffers) are described in a *behavioral* VHDL and constitute the building blocks of coarse-grain actors (Dot-product blocks, Window Buffers, layers ...) which are described in *structural* VHDL, and listed next.

B.1 Convolution Layers

The convolution layer block implements the processing of eq.2.3 by mapping N three-dimensional convolutions. Clearly, the `convLayer` block is parametrized by the number of 3D-kernels (N) and their dimension $(C \times J \times K)$, but also the value of these kernels (Θ), following the principles of SCM described in sec.6.3. Figure B.1 explains how the `convLayer` operates:

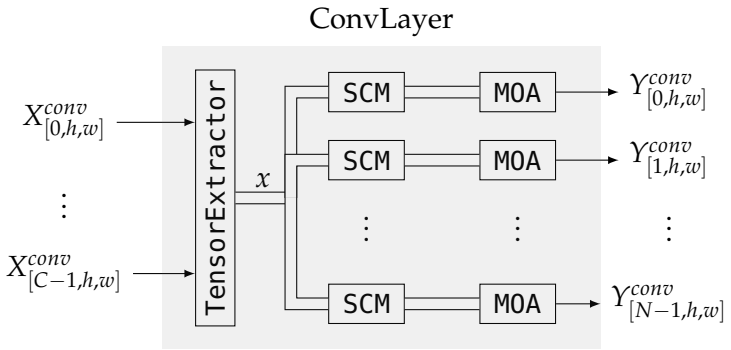


FIGURE B.1: Implementation of a Convolution Layer in Haddoc2

The Tensor Extractor block extracts the $(C \times J \times K)$ neighbors of each input $X^{conv}_{[c,h,w]}$. The output of this block, denoted x in fig.B.1, is built by instantiating the `WindowBuffer` structure C times, following the technique discussed in sec.6.2. A diagram of the `TensorExtractor` block and its hardware description is given in Fig.B.2.

The Single Constant Multiplication (SCM) part multiplies each entry of tensor x by each weight of $\Theta[c, j, k]$. The `scm` component is instantiated N times per layer, and generates $(C \times J \times K)$ constant multipliers per 3D-convolution. This corresponds to $(N \times C \times J \times K)$ constant multipliers per layer.

The Multiple Operand Adder (MOA) part sums the partial products resulting from the constant multiplications. As depicted in B.1, each layer generates N adders, and each

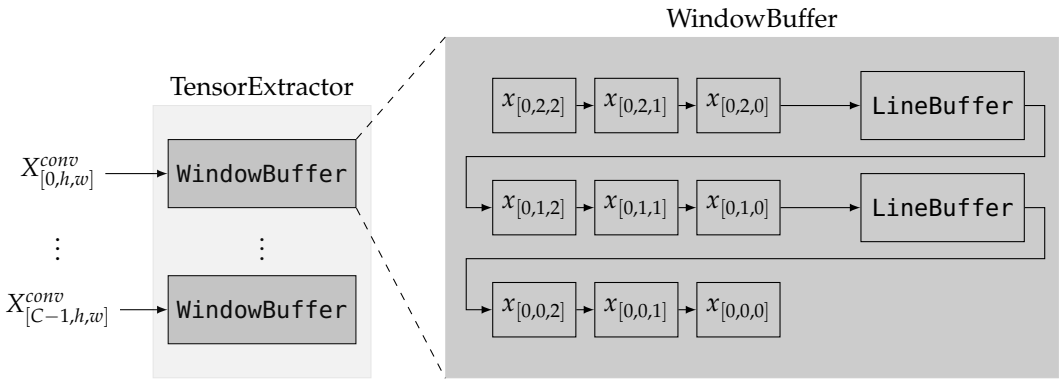


FIGURE B.2: Hardware Architecture of the Tensor Extractor

of them inputs $(C \times J \times K)$ operands. In its most « naive » form, the `MOA` component is implemented as a binary adder tree, as depicted in fig. B.3.

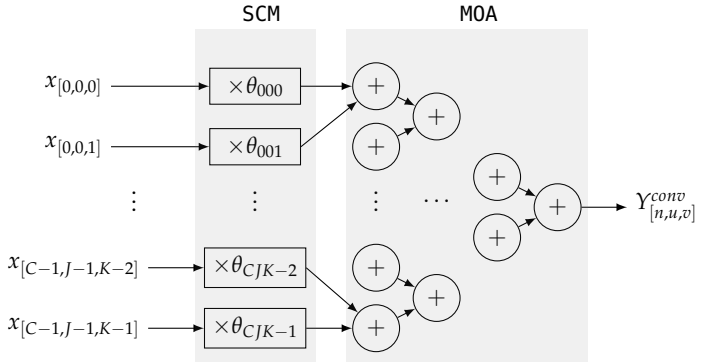


FIGURE B.3: Hardware Architecture of SCM and MOA parts

B.2 Pooling Layers

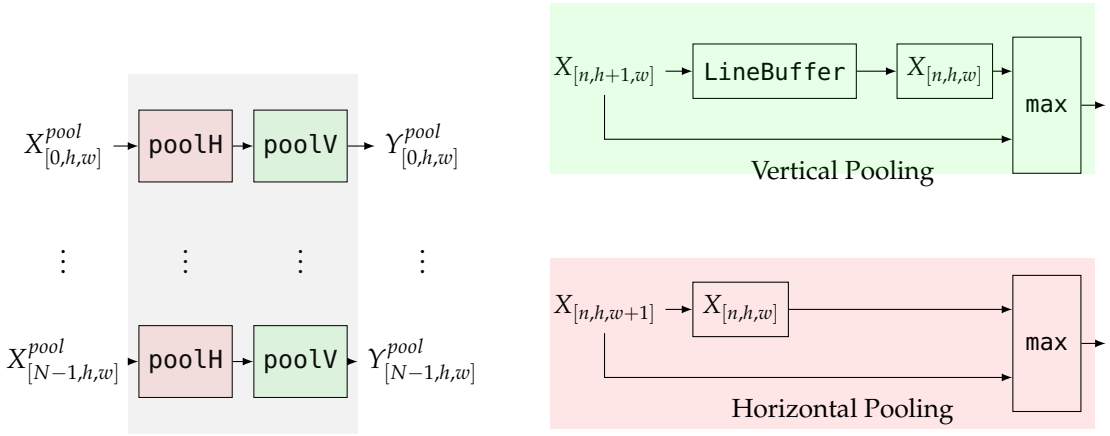
Similarly to the CAPH implementation, sub-sampling layers are built by combining the vertical and horizontal pooling blocks. The `PoolH` block computes the maximum (or average) of K adjacent inputs $(X_{[c,h,w]}, \ldots X_{[c,h,w+K-1]})$. It is followed by the `PoolV` block which relies on a `LineBuffer` to compute the vertical maximum of $(X_{[c,h,w]}, \ldots, X_{[c,h+K-1,w]})$. Fig. B.4b illustrate the architecture of the `PoolH` and `PoolV` blocks in the case where $K = 2$.

B.3 Activation Layers

ReLU: The ReLU function is the simplest activation to implement as it requires only a comparator and a multiplexer to be mapped. Note however that the block implementing this layer, namely `RELULayer`, is parametrized by a `shift` value that can be used to implement –by means of shit-registers– the « leaky ReLU » function used in some CNN layers:

$$\text{Leaky-ReLU}(x) = \frac{\text{ReLU}(x)}{\alpha} \tag{B.1}$$

Sigmoid and TanH Layers: HADDOC2 relies on a piece-wise approximation to implement the Sigmoid and hyperbolic tangent functions. These piece-wise functions either threshold, or apply a or a linear transformation to their inputs, as depicted in Fig. B.5a.

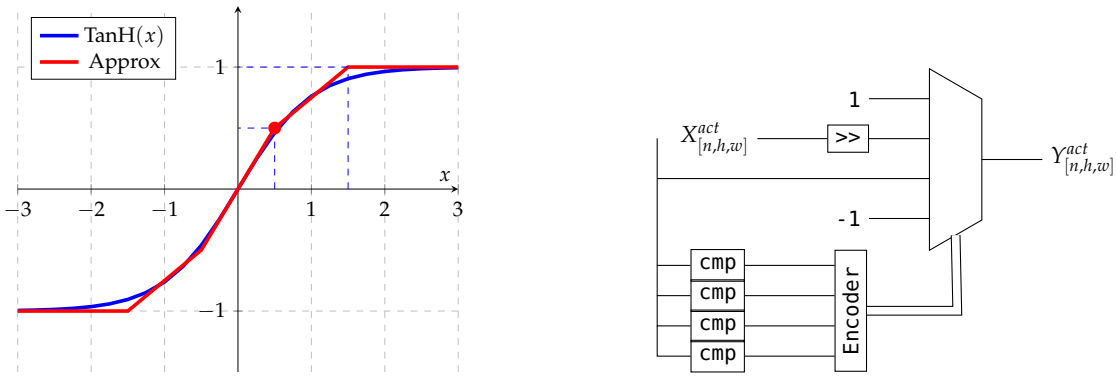


(A) Pool layer block

(B) (2×2) Vertical and Horizontal Pooling

FIGURE B.4: Implementation of a Pooling Layer in Haddoc2

When compared to an «exact» implementations using lookup tables, the key advantage of the proposed approximations is their low hardware footprint. For instance, the piece-wise implementation of the TanH requires only four comparators, one multiplexer and one shift register, as illustrated in fig.B.5b.



(A) TanH function and its approx.

(B) Hardware Implementation

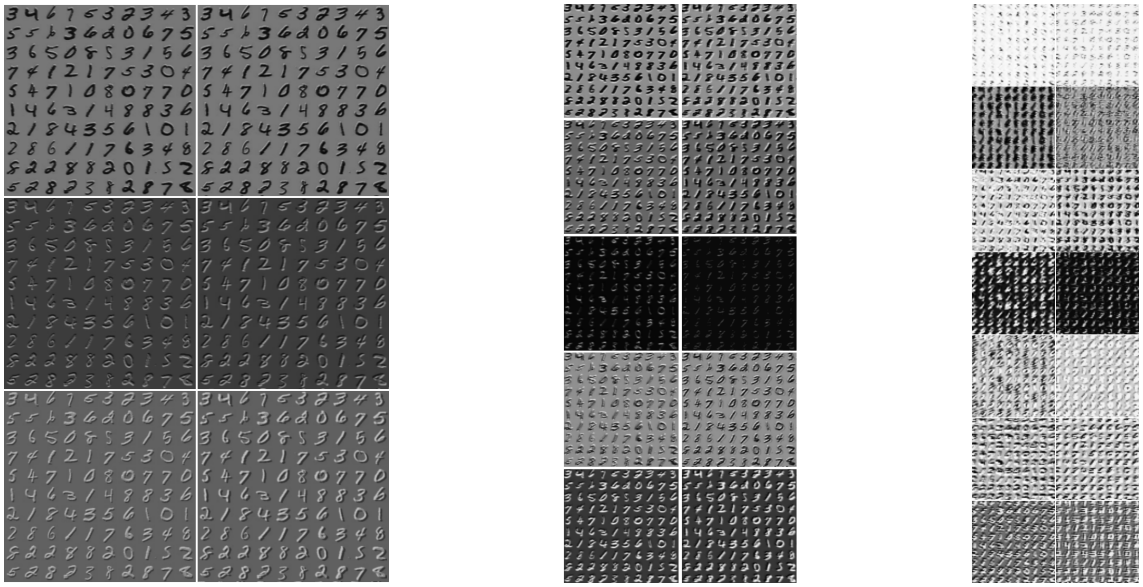
FIGURE B.5: TanH Function: Approximation and Implementation

B.3.1 Feature Map similarity

Fig. B.6 gives an example of feature maps extracted by HADDOC2 and compares them to features extracted by Caffe. Despite being slightly different, the feature maps deliver the same classification rate in both hardware and software implementations. In this example, the implementation of sec.6.5.3 is considered.

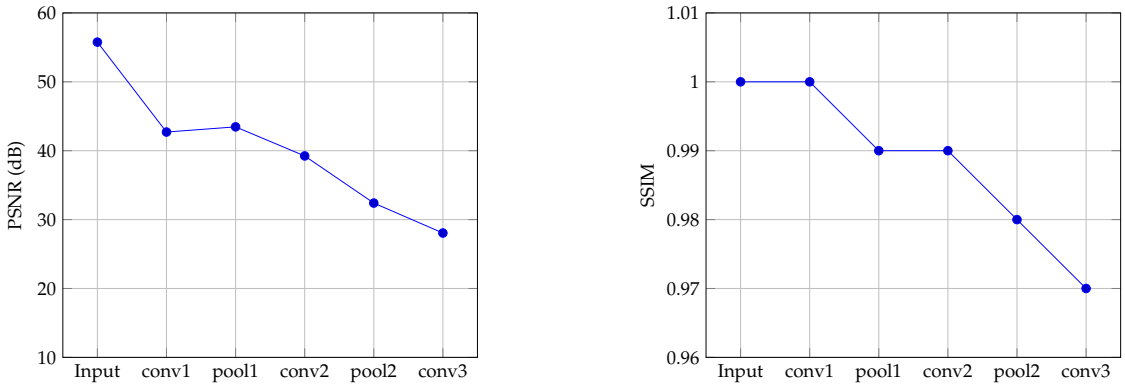
Figure B.7 quantifies the similarity of hardware-extracted and software-extracted feature maps in terms of Peak signal-to-noise ratio (PSNR) and Structural SIMilarity (SSIM). While the former metric estimates absolute errors, the latter is a perception-based model that considers structural information of an image [WBSS04].

On the left figure, the PSNR drops after each layer, mainly because of the rounding and approximated TanH function. Surprisingly, on the right figure, the SSIM stays on a high level at over 0.97. This high structural similarity may explain why CNNs deliver the same classifications in hardware and software, despite the differences between the two feature maps.



(A) conv1 (B) conv2 (C) conv3

FIGURE B.6: Intermediate Feature Maps: Hardware-computed on the left, software-computed on the right



(A) PSNR (B) SSIM

FIGURE B.7: Similarity between Hardware and Software extracted Features