POLITECNICO DI TORINO

Design of a distributed control unit for reconfigurable CNN accelerators

Corso di Laurea in Embedded system

Tesi di Laurea Magistrale

Relatore

Prof. Andrea Calimera

Nicolò Morando
matricola: 231924

Correlatore:

Dott. Valerio Tenace

Anno accademico 2017 – 2018

Abstract

Over the last few years, deep learning (DL) has evolved becoming pervasive in many scientific and industrial fields. The effectiveness of DL techniques, aided by the widespread availability of user-friendly tools developed by big ICT companies (like Google and Facebook, to name a few), is pushing the state-of-the-art in artificial intelligence, allowing Convolutional Neural Networks (CNNs) to represent a de facto standard for visual reasoning applications. CNNs are complex computational models inspired by the mechanisms that regulate the primary visual cortex of the brain, where images captured by the eyes are elaborated such to extrapolate a meaning, an information, from the surrounding environment (e.g., face recognition though feature detection). A typical CNN structure is composed of an input layer handling images for computational stages, an output layer that produces the final answer on the classification task, and several hidden layers where the feature extraction takes place. Indeed, from a functional perspective, CNNs can be divided in two main functional regions: feature extraction, and classification. The former region is where most computations take place, and it is mainly composed of a specific kind of layer: the convolutional (CONV) layer, where several multidimensional matrix-vector multiplications are carried out between input images (or feature maps) and abstract filters learned by the CNN itself. Since even the simplest CNN model contains several thousands of different filters, it is not surprising that the huge computational effort required to run DL algorithms is rapidly becoming a serious concern. Such a problem is exacerbated if we consider that most computing hardware platforms are not yet tailored to execute DL algorithms efficiently. For these reasons, a number of dedicated hardware accelerators for DL applications have been recently introduced. Being composed of several processing elements (PE) capable to carry out specific mathematical operations, those ad hoc solutions are capable to dramatically reduce execution times and the energy per operation. However, in most cases, information sharing between each PE is partially exploited, thus leaving a space for substantial performance improvements. In this thesis we present a hardware-software co-design tool called INRI, which allows to deploy a fine-tuned dataflow for specific architectures, such that superfluous data movements and power consumption are minimized. We investigate different techniques including

data reuse, smart activation/deactivation policies for PE in the idle state, and specific pixel-clustering algorithms. The performance of the proposed tool are supported by experimental results obtained with different hardware configurations running well-known CNN models, such as AlexNet, VGG-16 and ZFNet. Results demonstrate that our approach is capable to reduce the energy of CNNs by 25%, still guaranteeing an acceptable accuracy loss of 2%.

Acknowledgements

I would like to to thank Professor Andrea Calimera for giving me the opportunity to collaborate with him to this project. Another special thank to Valerio Tenace for helping and supporting me during this work .

Contents

List of Tables

List of Figures

Summary

Deep learning, in particular Convolutional Neural Network CNN, is among the most powerful and widely used techniques in computer vision. Applications vary from image classification to object detection, segmentation, Optical Character Recognition (OCR), etc. On the other side, they require a significant execution time, due to compute and memory operation. Consequently, it is difficult integrating CNN algorithms into IoT embedded systems with limited computing resources and energy supply. Most of researches tends to reduce computational CNN operation, rather than memory accesses. Introducing an off-chip memory in a device permits to store more datas, trading-off performance and energy saving. Compared to on-chip memory, accessing off-chip memory can consume up to 10x times more power consumption[1], and latency for obtaining data up to 10x times slower[1]. A possible solution could be handling the datas exploiting cloud technologies, waiting for the server returns the result, but according to CISCO, 5 quintllion bytes of datas are produced every day and by the year 2020, more than 30 billion of devices will be connected in internet [2]. In order to not congesting internet traffic, IoT devices must send datas to the network only when strictly necessary. Given the context shown, it is necessary to develop a CNN high performing system low power that executes most of the CNN operations locally. This work proposes INRI a CNN architecture able to reach high performance while reducing drastically memory access.

The architecture is based on five main modules: A CNN architecture, in particular Matrix PE [11] and Chain PE [23] both based on systolic paradigm, which the heart is the Processing Element (PE), a multiply and accumulator with registers. A dispatcher unit, charged to orchestrate the whole system operations and applying power saving technique, a 256 Kb local buffer for loading/storing datas and a Pixel Clustering module that implements an innovative algorithm for compressing images reducing drastically performance.

The system was developed mixing different program languages and techniques. Three modules (PE, Pixel Clustering and Dispatcher) were built via VHDL under QuestaSim environment and synthesised extracting informations about area, power supply and slack. The informations

obtained are integrated in a tool built in C language, able to return an accurate approximation about area, energy, power consumption and clock latency for operation of the system. The prediction was analysed simulating the system using Python with Keras frameworks, implementing three CNN architectures with pre-trained filter values: AlexNet, ZfNet and VGG16. Results demonstrate that is possible to achieve a significant memory access reduction up to 4x times lower compared to a traditional architecture without any memory access reduction techniques having an average accuracy drop around 6%.

Chapter 1

Introduction

1.1 Motivation

During last years Machine Learning has obtained the supremacy on computer vision tasks occupying more and more a significant part of our lifes. This phenomenon can be attribuited to its ability to get high accuracy, ranging from object recognition and detection. With the boosting of amount of data, it is smart to think that technological progress will be heavily influenced from smart data analysis. On the other side, Machine Learning algorithms require a significant execution time, due to compute and memory operation. Consequently, it is difficult implement CNN algorithms into IoT embedded systems with low hardware resources and energy supply. Majority of the works try to investigate how to enhance computational efficiency solutions of CNNs, leaving memory efficiency largely overlooked. Introducing an off-chip memory in a device permits to store more datas, trading-off performance and energy saving. Compared to on-chip memory, accessing off-chip memory can consume up to 10x times more power consumption[1], and latency for obtaining data up to 10x times [1]. Under these circumstances, it's necessary to develop a CNN system able to reduce power consumption, without affecting the overall performance of the system.

In this work is described INRI, a dataflow for CNN architecture that uses different techniques such as data reuse, local buffer, activation/deactivation processing element(PE) and a compression algorithm in order to achieve high performance with a contained power consumption. The hardware part of the work, was built using HDL language (in particular

VHDL) and C language. For software counterpart was used Python supported by Keras framework. The architecture is composed of a dispatcher, in charge of handling, executing and monitorate the system, a CNN based on systolic architecture that execute operations, a local buffer for storage datas and last the Pixel Clustering module used that implements an innovative compression algorithm for reducing memory access. Comparing to traditional systems, results shown that all these strategies reduce in significant way the memory access (up to 4 times lower), hence the power consumption and clock latency per operation.

1.2 Related Work

During the years, different techniques has been exploited for realising performing CNN system using different technique such as data reuse, systolic architecture (MATRIX PE[22] or CHAIN PE [23]) or proposing new dataflows (EYERISS) . Most of state state CNN solutions exploit one or more of these paradigms in order to achieve high performance.

- **weight stationary** : For optimising convolutional and filter reuse, every filter weight remains fixed in the register file (RF) inside the processing element (PE).
- **output stationary**: The aggregation of the partial sum(psum) remains static in the RF. The output feature map (ofmap) stay in the same RF for accumulation in order to reduce the psum accumulation cost.
- **no local reuse**: It uses inter-PE communication for input feature map (ifmap) reuse and psum accumulation.

All these paradigms propose interesting solutions for enhancing performance of the system, but they consider often the operation efficiency rather than also analyse memory efficiency. Considering as example the two systolic architecture introduced (MATRIX PE[11], CHAIN PE[23]). They suggest useful solutions for executing CNN operations faster, but they do not answer to the question how to access in a more efficient way to the datas to manipulate, making the off-chip memory access the bottleneck of the system. This research goes to investigate how to enhancing both memory and execute operations.

In summary, the main contributions of this work includes

- data reuse
- hardware optimizations
- test comparison among CNN
- new algorithm able to reduce data movement off-chip to on-chip (pixel clustering)

1.3 Purpose and research questions

The entire research attempt to reply to the following questions :

- How to realise a high-performance CNN architecture ?
- Is it possible to build a system both performing and low-power consuming ?
- if is it positive the previous question, how much prediction score is affected ?

1.4 Approach and Methodology

The work proposes a high-performance and low-power convolutional neural network architecture (CNNA) exploiting systolic architecture and compression algorithms in order to improve efficiency of the system. Different experiments were executed in order to obtain the best setup with the lowest hardware implementation. They can be classified in two categories, the former finding the best trade-off among hardware resource and performance. Second, develop and test how to orchestrate the whole architecture with best performance power consumption ratio.

1.5 Scope and Limitation

The main purpose of this work is to develop a low-power performing system with a good image patterns recognition accuracy . The system obtains

an interesting prediction score around 70% (where traditional systems obtained an average around 76%) with an average memory access reduction around 60% compared to a traditional setup. It can be considered a good result, but some interesting investigation could be take place.

Considering local buffer. INRI implements SRAM local buffer 256 kb 32 bit word cell size.It uses 16 bits for data and 3 bits for implementing Pixel-Clustering algorithm.13 bits does not contain any useful informations. Differents researches demonstrate that data inputs and weights could be represented that up to 8 bit[12], without loosing significant accuracy A possible improvements could be reduce the data value to 8 bits in order to fit both data and algorithm in 16 bits.

Another limitation is data streams inside CNNA, in particular, how information pass through processing element (PE) . The communication among them happen only in vertical or horizontal way because of limitation provided by the algorithm (Pixel Clustering). A further step could a totally freedom PE communication (introducing the oblique data movement), modifying the structure.

1.6 Outline

The report is structured in the following way. Chapter 2 introduce theoretical background about machine learning, paying attention on artificial neural network, in particular convolution neural network. Chapter 3 explains the motivation behind this paper, describing briefly the stream execution. Chapther 4 shows in details which technologies were used to develop and testing the system. Chapter 5 explain in details how architecture works and on which technique is based. Chapter 6 describe the result obtained by experiment, concluding with final considerations in chapter 7.

Chapter 2

Theoretical Background

2.1 Introduction

Machine learning is a subfield of artificial intelligence (AI) that permits to a program to "understand" how to solve a specific purposes with datas received, without being explicitly instructed. A more formal definition was formulated by Tom M. Michell says “A computer program is said to learn from experience E with respect to some class of tasks T and performance measure P if its performance at tasks in T, as measured by P, improves with experience E.” [24] Machine learning algorithms can be divided into two categories supervised and unsupervised algorithms, depending on how they "learn" to make predictions

- **Supervised algorithms**: Supervised learning is the paradigm most used for machine learning. It is when the program tries to to learn the mapping function from the input (X) to the output(Y). The input are the data send by user for training/testing part, while the outcome is the prediction value. Learning method can be compared to a student learning from a teacher.

  $$Y = f(X)$$

  The purpose is to develop a mapping function that is able to predict the output variables (Y) when it receives new input data (X) . Compared to others solutions, it has the advantages that you can make a perfect decision boundary to distinguish different classes accurately, specifying by user how many classes desire to have and besides after training, it’s not necessary keep the training example in the memory,

it's necessary just the mathematical forumula "decision boundary" for classifying future inputs. On the other side, the decision might be overtrained or it tries to predict decision even through the input received was never classified into a category. Last, the training mode requires a lot of computation time. Typical supervised algorithm are linear regression, based on a linear equation where the prediction is obtained inserting parameters in the equation formed during training, random forest, that is based on a binary decision tree and support vector machines:

- **Unsupervised algorithms**: Unlike supervised learning, in unsupervised learning, the answer are not labelled and it is duty of the algorithm, to group datas correctly. It acts more closely to "true artificial intelligence" [5]. It is more complex to implement compared to supervised algorithms but it permits to the program, to solve problems never faced, building new classes.Some popular examples of unsupervised learning algorithms are k-means, based on centroid point called k used as point of reference for building a new class. Each k value is a "cluster" of element belonging to the same class or apriori algorithm .

.

2.2 CNN Overview

CNNs are complex computational models inspired by the mechanisms that regulate the primary visual cortex of the brain, where images captured by the eyes are elaborated such to extrapolate a meaning, an information, from the surrounding environment (e.g., face recognition though feature detection). A typical CNN structure is composed of an input layer handling images for computational stages, an output layer that produces the final answer on the classification task, and several hidden layers where the feature extraction takes place. Indeed, from a functional perspective, CNNs can be divided in two main functional regions: feature extraction, and classification. The former region is where most computations take place, and it is mainly composed of a specific kind of layer: the convolutional (CONV) layer, where several multidimensional matrix-vector multiplications are carried out between input images (or feature maps) and abstract filters learned by the CNN itself. Typical layers used for CNN are

convolutional, pooling, activation, normalization, fully connected and softmax. Some of the most famous CNN architectures are: LeNet, AlexNet, VGG, GoogleNet, ResNet. Figure 2.2 depicts LeNet, the first CNN architecture ever built. It was born for banks purposes, in order to recognise the digits written in the checks. The architecture receives as input, handwritten digits in the check, returning the prediction about which numbers could be.

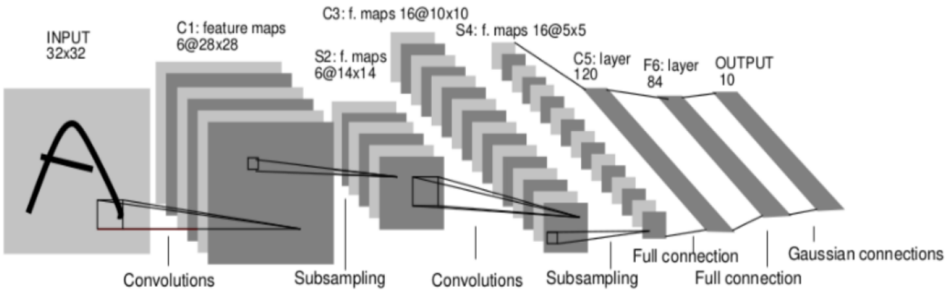


Figure 2.1: LeNet CNN architecture [14]

Nowdays, CNN architecture are more complex and filters are able to identificate curves, lines, borders going up to more complicate patterns such as eyes, bodies, etc..

2.2.1 Convolution

Considered the heaviest CNN operation (it consumes up to 90% of execution time) convolutional layers, as the name suggest, perform a convolution operation to the input received. Introducing convolution operation to the architecture, permits to the network to be deeper with lower parameters, compared to same network based only on fully connected layers and extracting patterns on the images in order to obtain useful informations of the object analysed. Figure 2.3 displays a convolution operations with an input 3x7x7 and a kernel 3x3x3. The input was applied a padding =1. The output results is a ofmap 3x3x3

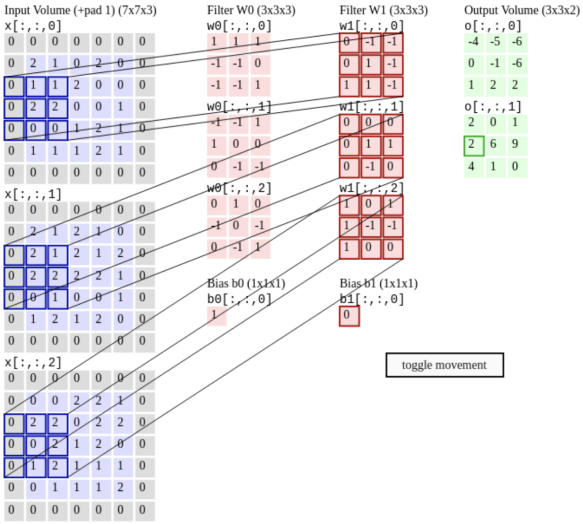


Figure 2.2: convolution on an image 3x5x5 with padding=1 with a kernel 3x3x3[16]

2.2.2 Pooling

One of the main flaw of a CNN is that requires a significant execution time, due to compute and memory operation. Introducing a layer that "cut-off" unuseful informations permits to make the system faster. For this reason, convolutional networks typically has different pooling layers, permitting to reduce the size of a certain input, producing the outputs of neurons group at one layer into a single neuron in the next layer. Pooling can be splitted in the pooling techniques: max pooling, where the output is the maximum value from a group of node, while average pooling, as name suggests, returns as output the average value from a group of node. Figure 2.4 shows an example of using Pooling resize to a 4x4 input, resizing it to 2x2. On the left is applied Max Pooling algorithm, while on the right Average Pooling one.

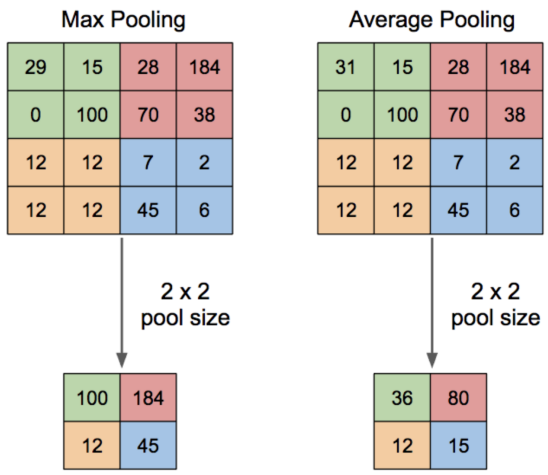


Figure 2.3: example of applications of max and average pooling [15]

2.2.3 Fully connected

Typically implemented in the last layers of CNN, fully connected layer links each node of a layer to every node in the next one. It received the input from a previous layer and returns a vector of size N, where N is the number of classes that the user decided to implement, reporting the percentage score that the input belongs to a given class.

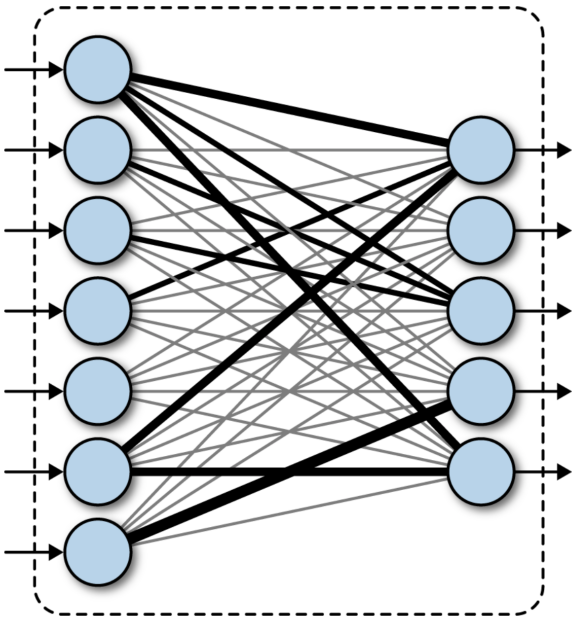


Figure 2.4: fully connected layer [17]

fully connected layer is based on artificial neural network (ANN) .

ANN are based on input layer, hidden layer (could be one or more layers) and an output layer. Figure 2.6 displays the structure of an ANN

Figure 2.7 shows more in details how a single node inside the hidden layer works. It receives the weighted sum of the inputs produced by the previous layer passing them through an activation function decided by user. The output realised passes to the node in the next layer as input. The final output is obtained by executing this procedure for all the nodes.

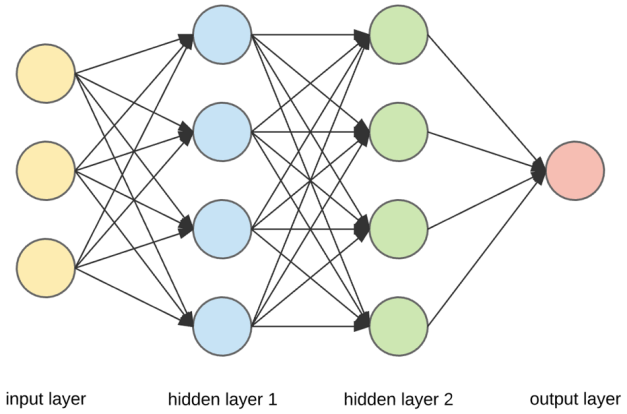


Figure 2.5: Overview of a ANN[29]

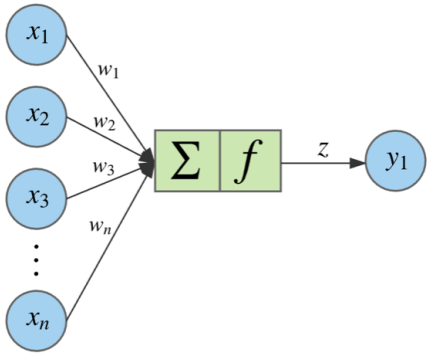


Figure 2.6: hidden layer in detail[29]

2.2.4 Padding

After a convolution, the height and the width of the new output are always lower compared to the previous input. Sometimes, in order to achieve higher accuracy, it is convenient having deep networks . Padding technique allows to control the spatial size of the output volumes, useful when it is necessary to preserve the spatial size of the input volume so the input and output width and height are the same. The method consists of surrounding the input volume with zero around the border. Figure 2.6 shows an input volume 32x32x2, where was applied a padding of 2, getting a final volume of 36x36x3

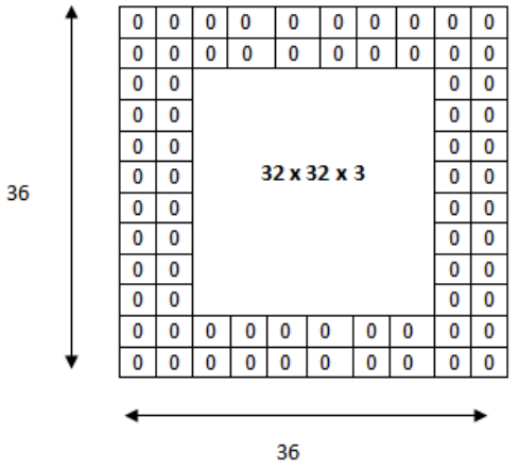


Figure 2.7: the original size is 32x32x3. After applying a padding=2 now we have 36x36x3[20]

2.2.5 Non Linearity (ReLU)

Introducing a non-linearity property in the CNN architecture, permits to obtain better performance compared to a linear one. For this reason, usually, after a convolution a Relu (Rectified Linear Unit) operation is applied. Relu is operation applied to every matrix value inputs replacing all negative values in the feature map with zero, permitting to introduce a non-linearity property in the CNN architecture.

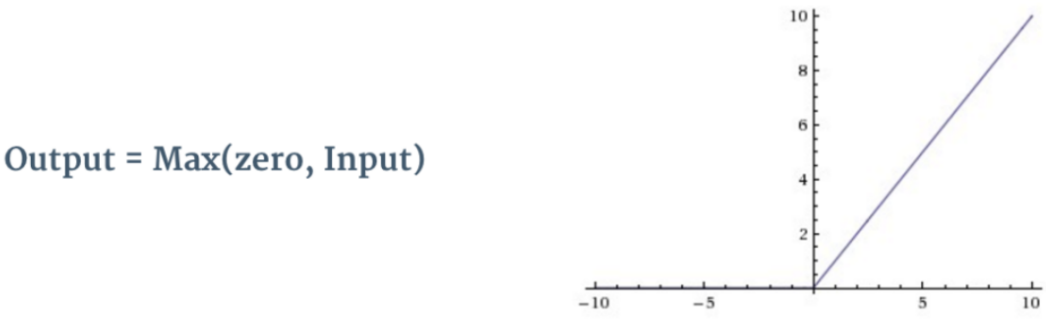


Figure 2.8: relu representation[6]

This is the most used activation function because is the one that has the closest behaviour to real neuron. In the picture 2.10 is depicted the algorithm implementation. On the left the image pre-relu processing, where the black colour stands for negative values. On the right the same image which applied ReLU algorithm, containing only positive value, hence the black colour is not more visioble.The output feature map here is also referred to as the 'Rectified' feature map.

Even through ReLU is the activation layer with the closest neuron behaviour, it has several flaws. As displayed in figure 2.9, ReLU is not zero-centered, it is ubounded and could happen that ReLu neurons goes into states in which they become inactive for every inputs receiving, with no gradients flow backward through the neuron

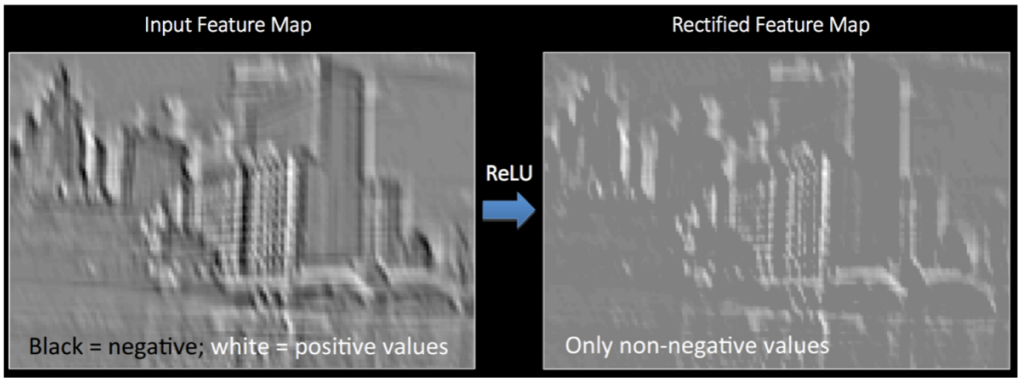


Figure 2.9: image which applied relu[6]

2.3 CNN architecture

Interested in ANN started almost 80 years ago, after experimental works start understanding how mammalian visual cortex works, permitting to scientists to build model similar to biological neural networks.[19]

From this scenario, Convolutional Neural Network raised up. A typical CNN structure is composed of an input layer handling images for computational stages, an output layer that produces the final answer on the classification task, and several hidden layers where the feature extraction takes place. Indeed, from a functional perspective, CNNs can be divided in two main functional regions: feature extraction, and classification. The former region is where most computations take place, and it is mainly composed of a specific kind of layer: the convolutional (CONV) layer, where several multidimensional matrix-vector multiplications are carried out between input images (or feature maps) and abstract filters learned by the CNN itself. Every years differents CNN architectures challenge in a competition each other in order to obtain the highest score prediction. This competition is the ImageNet Large Scale Visual Recognition Challenge (ILSVRC), based on ImageNet project. The ImageNet project is database containing over 14 million images used for training or predicting patterns for CNN . In the 2011, Before the spreading of CNN, the best prediction score obtained in a top-5 rank was to 26.2%. The following year, AlexNet reached a 15.3% top-5 rank score.

2.3.1 AlexNet

AlexNet [25] (fig2.9) is a CNN architecture come into the limelight for being the 2012 ImageNet[31] LSVRC-2012 [28] competition winner by a large margin (15.3% VS 26.2% (second place) top-5 error rates). The network was trained for 6 days using two GTX 580 3Gb GPUs. It is based on 5 convolutional layers and 3 fully connected layers. Relu is applied after every convolutional and fully connected layer and padding layer is applied before the first and the second fully connected layer. The network has 62.3 million parameters, and needs 1.1 billion computation for completing the whole CNN operations. Convolution layers are only 6% of all operations, but it consumes 95% of the whole computation. Going in details, figure 2.9 show the overall architecture. The first layer is based of a convolution with 96 filters 3x11x11 with a stride=4. The outputs

produced go to the second layer, which applies first a max pooling and then a 256x96x5x5 convolution. Similar behaviour for the third layer, differing just for the filter size of 384x256x3x3. The last two convolution layers have the same weight size 384x384x3x3. After completing this part, the output is "flatted" passing through the two fully connected layer, giving finally the result.

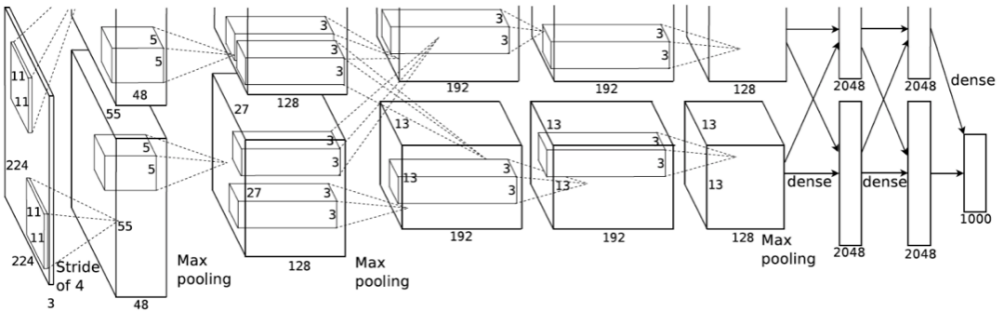


Figure 2.10: overall AlexNet architecture [25]

2.3.2 ZfNet

ZfNet[26](fig 2.10) is CNN that won the competition ILSVRC 2013 [30] reaching a top-5 error rate of 14.8%. It has almost the same structure of AlexNet, tweaking the hyper-parameters of AlexNet and adding more CNN layers.

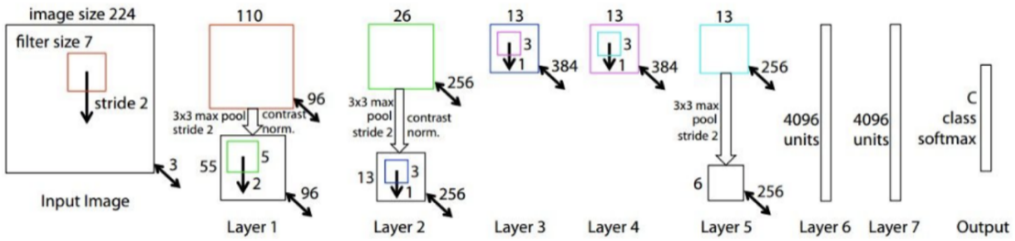


Figure 2.11: overall ZfNet architecture [26]

2.3.3 VGG16

Another important CNN architecture is VGG16 [27](fig 2.11). It based on a long chain of convolution layer (13), with size 3x3. It reaches 70,5% Top-1 Accuracy and 90 % Top-5 accuracy. VGGNet is built on 16 convolutional layers using a uniform architecture. It executes 3×33 times 33×3 convolutions and 2×22 times 22×2 pooling requiring about 140 million parameters, resulting difficult to handle.Nevertheless, it is currently the most preferred choice in the community for extracting features from images. Picture 2.11 shows the whole architecture

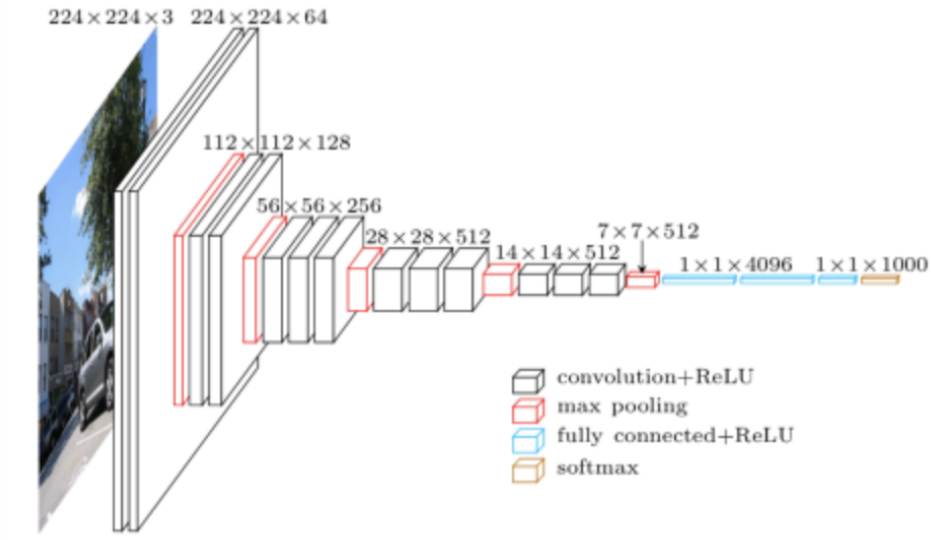


Figure 2.12: Vgg19 architecture [27]

Chapter 3

Hardware architecture for CNN

In CNN architecture, accessing in memory off-chip and computing operations require considerable execution time. Most of the works mainly focus on the computational efficiency of CNNs, without being interested in memory efficiency of CNNs. Compared to on-chip memory, accessing off-chip memory can consume up to 10x times power consumption[1], occupying the 30% of the whole execution time for completing a certain task[10].Consequently, it is difficult integrating CNN algorithms into IoT embedded systems with limited hardware resources. A possible solution could be elaborate datas on cloud, but according to CISCO, 5 quintllion bytes of datas are produced every day and by the year 2020, more than 30 billion of devices will be connected in internet [2]. So in order to not congesting internet traffic, in cloud computing can not be the solution. Datas have to be manipulate locally, but it means execute a significant memory transaction on/off-chip and viceversa, degrading both power and timing performance of the system. Considering a traditional system, based on Von Neumann architecture, memory off-chip accesses degrade the overall performance of the system. The CPU must wait for DRAM data, often wasting clock cycle because the operation cannot be executed if not received the data from off-chip memory. A possible solution could be logic in memory, a memory with some combinational logic associated with each storage element. This is an interesting solution, but it' s still a new technology not tested enough. After having analysed this context carefully, it

was opted to reduce memory access via compression algorithm and utilising systolic paradigms for CNN architecture. In particular was opted to study two architectures: PE CHAIN an MATRIX PE. They're both based on systolic architecture, where the primitive module is the processing element (PE).

Processing Element is based on a multiply and accumulator (MAC) 4 registers ,3 inputs and 2 outputs. It makes a multiplication among 'a' and 'b', and then with the value obtained makes an addition with 'x'. 'a' will take two clock cycle for set out as output, instead the sum result only one. Figure 3.1 shows an PE hardware structure.

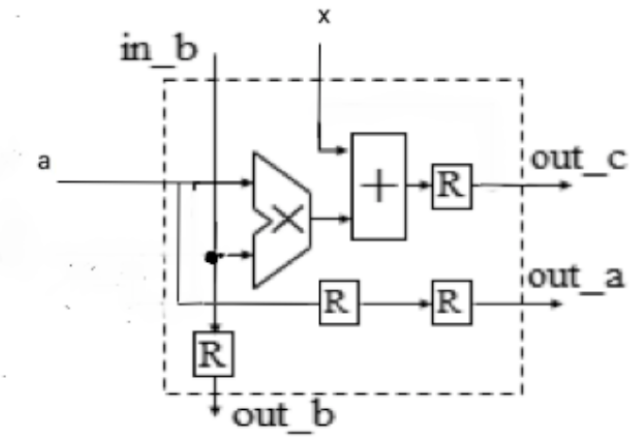


Figure 3.1: PE logic block rappresentation

The PE can be setup in 2 different setups: "normal mode" and "passing mode". The former is activated when PE is in charge of executing a certain CNN operation, viceversa the PE is set in a "idle mode", and it just passes the data received to the next PE.

Many systolic architectures were analysed, but because of Pixel Clustering algorithm nature (convolution kernel window must slide on the input row by row). the choice was to implement 2 predefined structure : PE CHAIN an MATRIX PE. The former is useful when good performance is not the primary target but rather than power consumption, viceversa Matrix PE is advised.

3.1 CHAIN PE

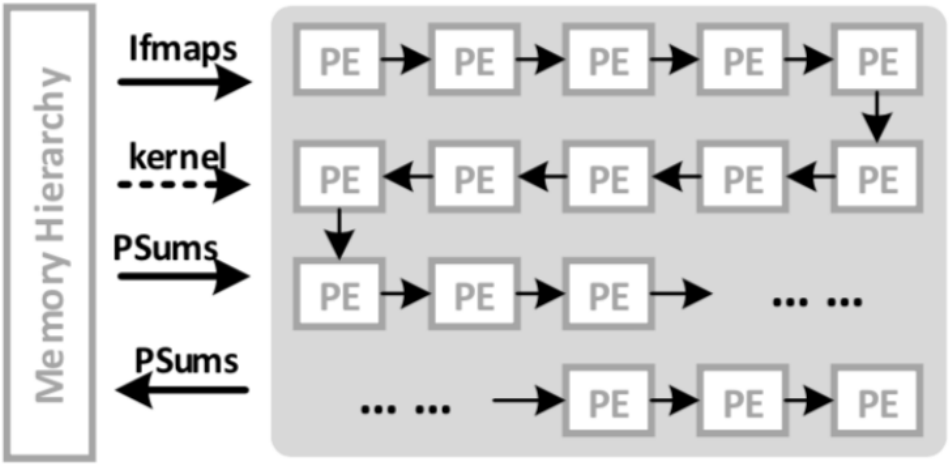


Figure 3.2: PE logic block rappresentation[23]

Chain PE [23] is 1D chain architecture developed in order to improve the energy efficiency of the system. As the name suggests, PEs organization forms a chain, as depicted in figure 4.3. Chain PE is controlled by INRI Dispatcher that setups the environment to work correctly The dataflow execution is like this :

1. INRI Dispatcher sets up the CHAIN NN to CNN parameters received by the user.
2. It computes how many kernel fit in the PEs, activating the useful ones and switching to "passing mode" the others.
3. IINRI Dispatcher start charging kernel into PEs
4. The CNN operation desired begins.

Compared to a traditional CNN, chain_NN has following advantages:

1. Good energy efficiency compared to the others systolic architecture.
2. It just required KxK PEs to work correctly (where K is the kernel size).

3. It is highly reconfigurable permitting to obtain high performance for different CNN parameter.

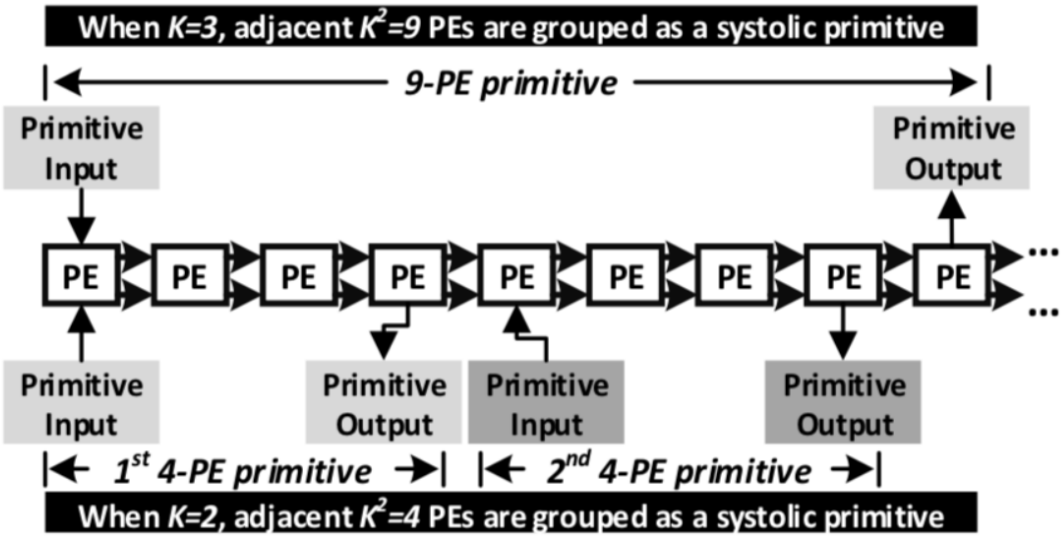


Figure 3.3: An example of how a 1D chain architecture is divided into cascaded systolic primitives for various kernel size K [23]

In Fig. 3.3, the PE chain is splitted into 1D primitives according to the kernel size. The upper side displays when 1D primitive contains 9 PEs (K=3) while the lower one K=2. The first PE of the chain receives the input from INRI Dispatcher, while the last one send the result to the Pixel Clustering module. For this research, it was opted to use a systolic chain composed of 121 PEs. Table 3.1 shows different possible kernel size combinations that can be obtained having 121 PEs.

Table 3.1

Kernel size	# of PEs primitive	# of active primitives	# of active PE	Efficiency
3x3	9	13	117	96%
4x4	16	7	112	92 %
7x7	49	2	98	81%
9x9	81	1	81	66%
11x11	121	1	121	100%

For implementing 1d chain architecture, every primitive for convolutions are are represented as 1D implementation.This is obtained using pipeline technique, forming a chain of multiply-accumulate operations (MAC) depicted in Fig. 4.5 designing a systolic architecture called 1D systolic primitive. 1D systolic primitives are based on a group of KxK identical PEs displayed in Fig. 4(a). The KxK PEs have the same size of a convolutional kernel window. Every PE, before start the "real" operations, has charged by INRI Dispatcher with a specific kernel weight.

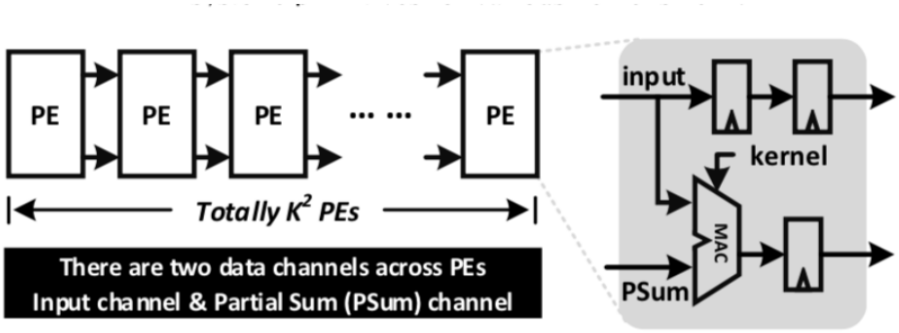


Figure 3.4: Each pipeline stage forms a basic process engine (PE). The cascading PEs form a 1D systolic architecture for a 2D convolution operation[23]

Thus, this type of architecture are able to pass and reused both input data and partial sums during convolution, without accessing external memory. Meanwhile, it has a fixed input bandwidth requirement and a constant output delay regardless of the kernel size. Therefore, this 1D systolic primitive can not only decrease the memory accesses, but also can be designed according the performance and kernel size desired. Concluding, This architecture is easy to build and implementing , but it presents some flaws. The more the matrix kernel is big and the less inefficient is. in fact for example considering a convolution with a matrix kernel 3x3 and stride 1, the 33% of the clock cycle bring useful information. Doing the same thing, but with a matrix kernel 11x11, only 1 clock cycle to 11 bring useful information, reducing the efficentess of the system.

3.2 PE Matrix

This architecture [11] is developed in order to achieve very high performance. In order to get it, it needs (KxK)xN. where K is the size of the matrix kernel and N is equal to the M-K+1, where M is the heigh of the matrix input.

Here below a picture of the system:

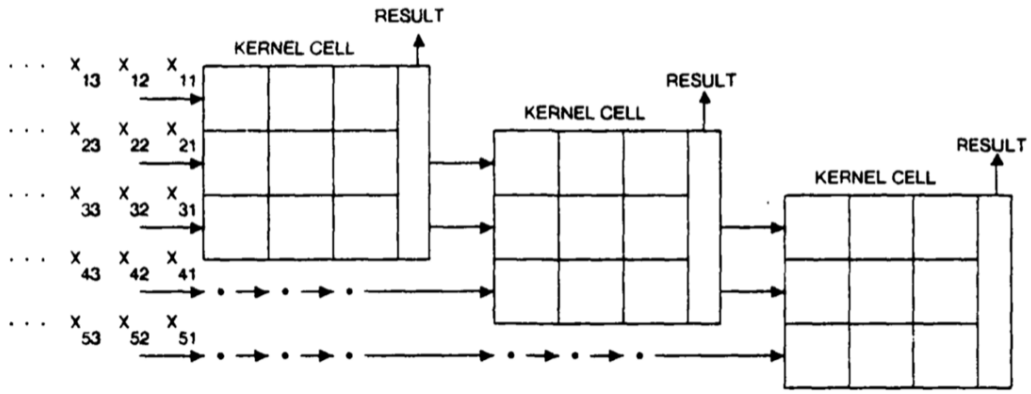


Figure 3.5: PE Matrix overview [11]

Explaining in details, the CNN evaluated based its architecture using a systolic one, where the basic element is the PE. As the name suggest, the PEs are positioned forming a matrix, where a PE communicates with every module around it. The input value stays in a the PE for two clock cycle passing from the two registers, while the partial sum remains just for one clock.The inputs enter in the PE and it is executed the multiplication with the filter value previously stored by INRI Dispatcher.The result multiplication obtained is summed with the partial result received. Meanwhile, the value stored in the first register goes in the second one, while the value stored in the second register become the output of the PE. Then, the row-interface cell handles the whole partial sums produced making and addition and subsequently sending to Pixel Clustering module. Given the area required for implementing this kind of architecture (first AlexNet layer would require 25894 PE) ,was opted to use a simplified version of it, inserting only a single matrix PE 11x11

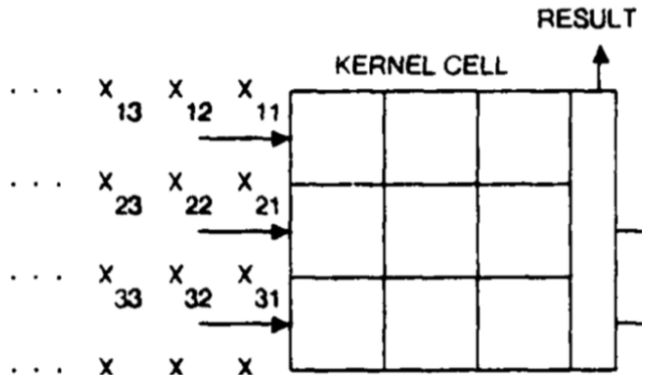


Figure 3.6: LowPE Matrix overview [11]

. Performance are affected but not so much as it can expected for one reason: often in the convolution the stride is not equal to one, so we have different primitive matrix PE that waste power computational producing not useful results.This is avoided using only one primitive matrix PE.

Chapter 4

Hardware software co-design for CNN

Memory off-chip accesses, are most of the times, the bottleneck of high performing systems.

The microprocessor speed grows much faster compared the rate of improvement in DRAM speed.It can not be considered as solution downstreaming the CPU frequency because it would stil worse the performance. Picture 4.1 shows CPU and DRAM performance growing through the years. CPU has an exponential growth, while DRAM a linear one.

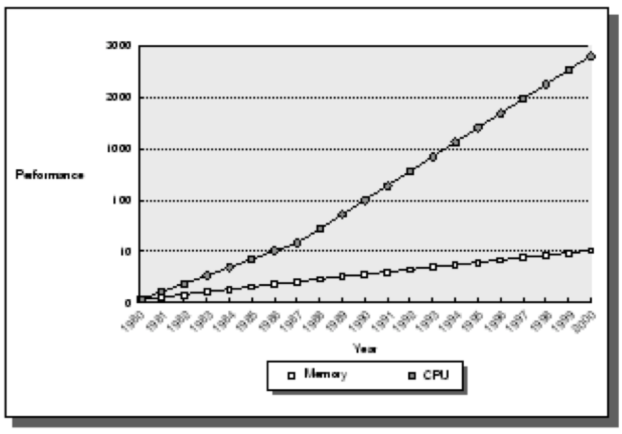


Figure 4.1: growth comparison among CPU and DRAM [8]

To overcome this gap , new technologies are arising, such for example logic in memory, a memory with some combinational logic associated

with each storage element. This is an interesting solution, but it is still an immature technology, and for INRI development was preferred to use traditional CMOS facilities.

The building and testing of the project required the adoption of different programs and code languages. The first idea was to develop the whole system using HDL programming, in particular VHDL, but during the first experiment, was noticed that syntesis of the whole system required long time. For this reason was opted to mix both hardware and software techniques for simulating the system. Starting from the beginning, the essential parts of architecture (PE, Pixel Clustering module and the dispatcher) was written using HDL language (VHDL) using Questasim 10.4 IDE, testbenched for observing the correct behaviour, synthesized them and extracted informations about their main value (area occupied, power and timing necessary to work correctly). After gathered the main informations from the synthesis, it was developed a tool written in C using XCode able to provide the energy, power consumption and the clock latency per operation of the system. After completed the hardware simulation part, concerning the prediction part, it was built a tool written in Python with Keras framework, feed with inputs data by ImageNet, a very large dataset used for official CNN competition. Figure 4.2 shows the overall architecture

.

It is composed from these modules:

- Two Pixel Clustering
- Dispatcher
- CNN architecture
- Local buffer

The system works in the following way:

1. The first step is retrieving data from the memory off chip. Datas pass through pixel clustering module that apply the algorithm.
2. The dispatcher knowing in advance the type of CNN architecture from the data received by user chose place the data in the right order for it

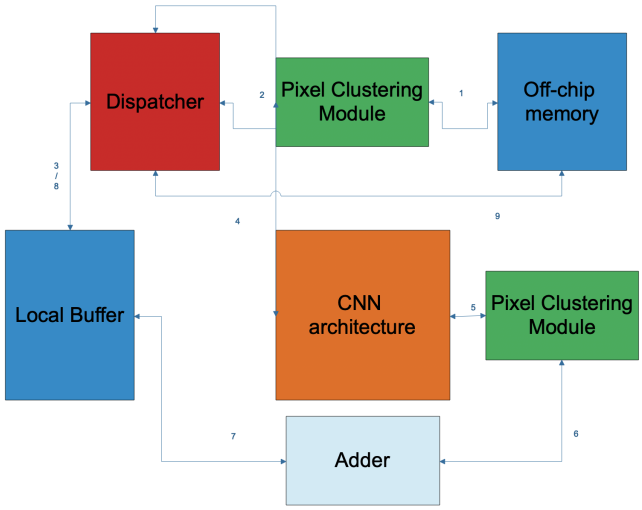


Figure 4.2: INRI design representation

3. based on operation requested it activate/deactivated the PE, in order to improve energy efficeness

4. Then, starts the operation desired (the most of the test was done using convolution operation because of complexity, able to reach the 90% of overall performance of the operations).

5. Later, datas pass again to pixel clustering module, before stored in a second local buffer. Completed it, they re sent again to the memory.

4.1 Dispatcher

The first CNN built, was LeNet-5 in 1998. It was used by several banks in order to hand-written digits on checks. [14] Compared to 20 years old, the CNN complexity raised of different orders of magnitude(today a CNN architectures can reach 100 million parameters, LeNet-5 compared to it, it has 60k parameters [13]), permitting to reach high accuracy prediction. This spreading happens for two main reasons: first the increasing of computational power has allowed to complete complex task in acceptable amount of time and second the lowering cost of memory, allowing to store a huge amounts of data. For this reason, in the actual CNN, it

is necessary to implement a module that orchestrates the whole operations. For example, a bad accessing ram management leads to degrade the speed of the system, and if it doesn't respect the DRAM idle time, it could provide an unexpected value conducting the operation to return the result wrong. Another case could be leave activated some modules even if not used. Considering a PE that just passes information to another PE without computing any operation, it will consume the same even through is not necessary.

Starting from this postulates, it was introduce an hardware dispatcher in the system: INRI Dispatcher. Its duty is to handle the whole operations, outlining when read/write in local buffer/memory, which CNN operation execute, which PE activate and so on.

In order to achieve this enhancement, the Dispatcher is supported by a local buffer and CNN systolic architecture. Thanks to this optimizzation, is possible to achieve high performance, reducing power consumption.

It was realised using HDL language and simulated developing a tool.

4.1.1 Hardware

The hardware realization, starts with coding the module using VHDL under Questa Sim-64 10.6 program. After completing it, was build differents testbenches for confirming the correct behaviour. Then, it was synthesised with CMOS045_SC_14_CORL_LS_bc_1.05V_105C library using design compiler. The INRI Dispatcher ,to set environment correctly, needs to receive two macro groups inputs from user: information about CNN structure and data to parse. The former is about the number of PEs that compose the system, local buffer size, CNN operation desired, filter size and the type of CNN architecture chosen, while the latter is the image with its size. After gathered all these informations, the dispatcher starts doing optimizzation, computing how many matrix filters can fit in the PEs, inserting in each of them, the filter values withdrawn from off-chip . then the PEs not exploited for any computation, change their status in "passing mode", where their duty is only passing the value to the next PE .Next step, it's withdraw data inputs from DRAM and place them correctly in the local buffer. After setting up the environment, the dispatcher launches the operation desired and feeds the CNN architecture with datas. The dispatcher had to know in advance t which clock cycle contains useful partial sum to store in local buffer or not. These steps are

repeated until the operation is completed. Figure 4.1 shows an example of convolution operation. The dispatcher sets the environment charging the filter value in PEs and stores the input datas inside local buffer.It launches the convolution and waits for the right clock for getting the partial value to store it in the on-chip memory. These steps continue until all filter kernel are parsed

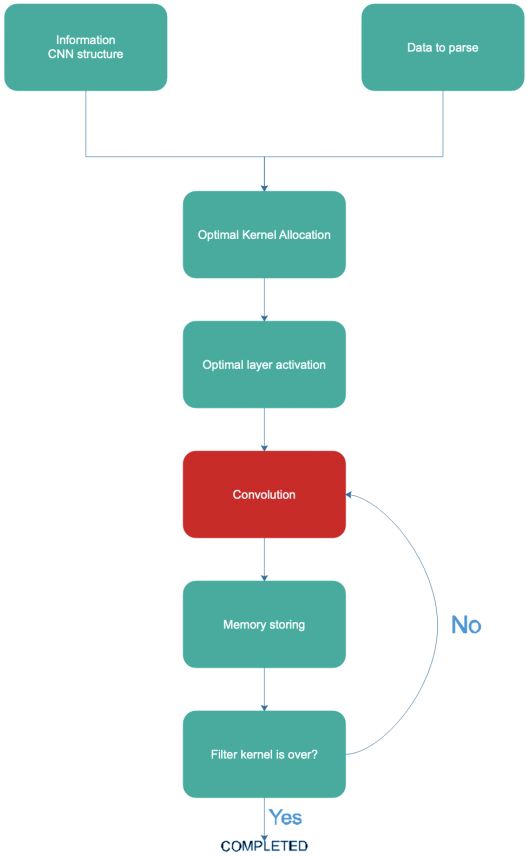


Figure 4.3: example of dispatcher execution flow

4.1.2 Software

Given the complexity of the whole system, it wasn't possible to synthesised it, because it would have required too much time. For this reason, it was opted to build a simulation tool that returns the main informations (area occupied, power consumed, clock latency for completing an operation an energy) about architecture. The software is called "Performance

Extractor" and it was built using C language

It works receiving two macro groups informations from user: information about CNN structure and Hardware architecture. The former is about the number of PE that compose the system, the local buffer size , the CNN operation desired, while the latter is the input size, kernel size and channel, stride and padding. Based on these inputs, the tool is able to elaborate it and report information about time latency, area occupied and power consumed. Figure 4.2 shows a dataflow execution of the software. This tool doesn't give any information about the prediction score.

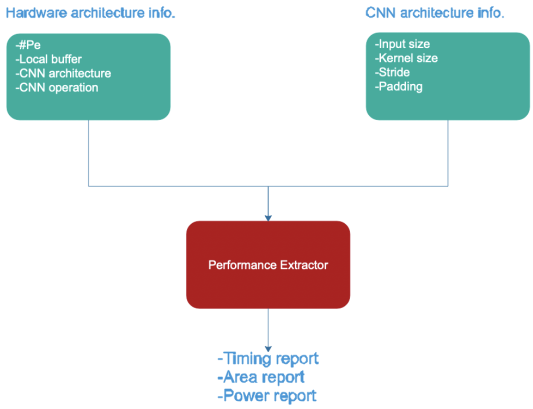


Figure 4.4: Software dataflow

4.2 Pixel Clustering

Reducing memory space requirement is important to many applications, in particular for embedded systems, permitting to drastically reduce the memory latency hence the energy consumption.

CNNs manipulate a huge quantity of datas, coming from input data, weight parameters and activations as an input propagates through the network.For example, ResNet network has the 50-layer that handles about 26 million weight datas, computing about 16 million activations in the forward pass. If not applied any optimizations, huge quantity of memory is necessary, making the system more expensive, bigger and power hungry. To overcome these issues, it was developed a compression algorithm able to reduce up to 4x times the memory access, without affecting in significant

way the prediction of the system. The module was produce in hardware and simulated using Python. It was called "Pixel Clustering". The idea behind the algorithm is that given an image, the pixels contained in a certain row, the ones close each other , under a certain threshold, have the same value. Exploiting this postulate developing a tailored architecture, it is possible to drastically reduce the input size, leaving almost unchanged the image, hence reducing memory access.

4.2.1 Hardware

The hardware realization, starts with coding the module using VHDL under Questa Sim-64 10.6 program. After completing it, was build differents testbenches for confirming the correct behaviour. Then, it was synthesised with CMOS045_SC_14_CORL_LS_bc_1.05V_105C library using design compiler . For making this module compatible with INRI, was slightly modified the architecture and the way reading data . Values are stored inside on 32 bit but only 19 bits are effectively used. 16 bit contain the data value while the last three ones are used to indicate how many times is repeated. Figure 4.3 illustrate how it is implemented in a memory word cell.

Figure 4.5: word cell rappresentation

Starting from the beginning, the module receives the input data, setting it as "primary value". Based on decision user, the module will build a range J, where J= primary value$\pm T$. The next inputs are approximated to the "primary value" if their value are included inside J range. The other parameter set by user is G. With this parameter we define how many times at most we can inglobate the values under J range.

The operation completes when a value goes out J or has been reached the maximum elements for a group. the module returns as output the data indicating how many times repeat it. This operation is performed when we access the very first time in off-chip memory and when the CNN operation is performing a convolution.

Figure 4.4 depicts a real application of Pixel Clustering. In this case was opted to set G=4 and T=3. In this example, exploiting this stratagem, the memory access is 15x times reduced.

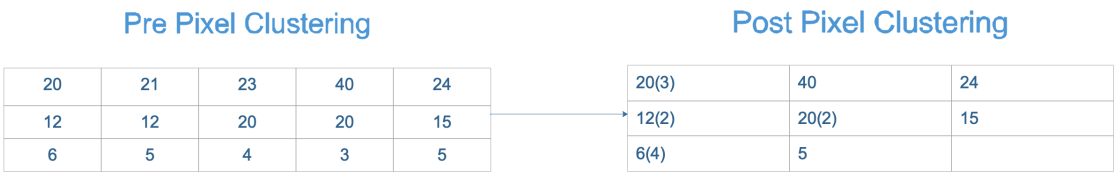


Figure 4.6: Pixel Clustering application

4.2.2 Software

Given the complexity of the whole system, it wasn't possible to synthesised it, because it would have required too much time. For this reason, it was opted to build a simulation tool of the module that returns the image compressed and memory access savings. The software is called "Pixel Clustering " and it was built using Python language. It receives as inputs the grouping and threshold value for the clustering and the input to apply the algorithm. The program reads the datas and apply the clustering. It returns the data compressed and the memory access savings. Image 4.5 displays an example of the algorithm application. The image on the left is without any compression technique. User send as input the image on the left,setting the g=4 and k=5 in this case. The program produces as output the image on the right . The image obtained was feed on CNN AlexNet returning a prediction score of 79% that the picture contains a cat (the original one reach a score of 86% including the same sentence), but lowering 4x times memory access.

4.3 CNN simulator framework

During the first phases of the work, it was noticed that wasn't possible to realise the whole system via hardware, because of the amount required of time requested for synthesis. The system analysis was splitted in two parts: one concerning performance analysing power consumption, clock

Figure 4.7: On left the original picture, on the right the same image with pixel clustering applied with cluster =4and threshold = 5. Even through they look like the same picture, the second image feed as input in INRI architecture reduces the memory access 4 times lower.

latency per operation and overall area of the system and the second one about a performance prediction. From hardware analysis the first step was developed PE, INRI Dispatcher and Pixel Clustering modules.

For every module was done the following steps:

- **HDL module coding** : Each module was written in VHDL simulated using Questa Sim-64 10.6.
- **Testing the module behaviour**: For every module was developed a testbench in VHDL in order the test the correct working. The program used was Questa Sim-64 10.6.
- **Synthesis & Constraints** : Every module was synthesized using Design Vision with CMOS045_SC_14_CORL_LS_bc_1.05V_105C library, setting the clock constraint to 8 ns.
- **Extracting information** : with the commands “report_area”,”report_timing”, “report_power” was extracted respectively information about area size, slack violation and power/energy consumption

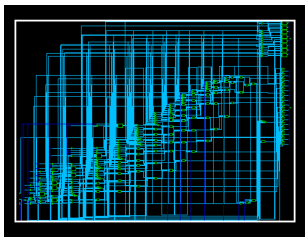
In table 4.1 are shown the information extracted:

After gathering all the informations of the macro blocks , in order to analysing the performance of the complete system was developed a tool using C language.

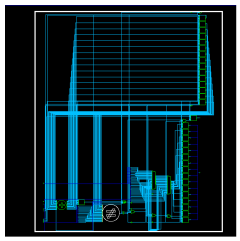
The program returns as outputs the power consumption, the energy and the clock latency for completing the operation of the system.

Table 4.1: Synthesis results

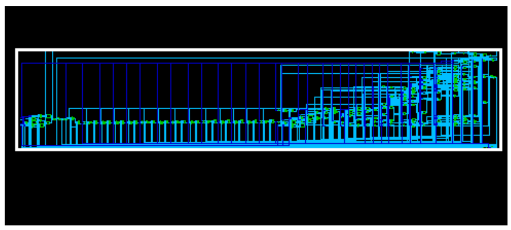
	report power	report area	slack
PE	312e2 mW	704,188802	5,18ns
INRI DISPATCHER	512e2 mW	1539,1455	3,5ns
Pixel Clustering Module	2,48e2 mW	455,4648	5,4ns



(a) PE RTL design



(b) INRI dispatcher RTL design



(c) pixel clustering RTL design

Figure 4.8

About Prediction, it was developed a tool that returns a top 1/3/5 CNN architecture and then it makes a percentage score comparison with an image without any modification and its counterpart with Pixel Clustering algorithm applied. The tools is "Result Comparison" and it is written in Python. It was chosen Python because of its good integration with machine learning frameworks.

The CNN architecture implemented was AlexNet, VGG19 and ZfNet.

Chapter 5

Results

5.1 Experimental Setup

The experiments proposed trying to demonstrate the possible performance improvements achievable using INRI architecture concerning memory access, clock latency and power savings without loosing significant percentage in prediction. The INRI hardware configuration used for the test is 256kb local buffer SRAM, 121 PE and 125 Mhz frequency. Summarising, the topics of interest covered with experiments are:

- Memory access
- Accuracy drop
- Prediction score drop
- Power savings

It was analysed both INRI with different setup Pixel Clustering algorithm configurations (grouping values equal to 4,7 and threshold 5,7,15) and traditional system without any reduction technique.Charts results obtained are an average score among the three CNN architecture VGG16, AlexNet and ZfNet, including all their layers. Completing the part about memory transaction, was investigated the accuracy compared to a traditional system. It was analyse how INRI top1 prediction score differs from a traditional architecture, with same inputs, filter and CNN architecture. The INRI configuration used is grouping values equal 4 and 7 and for each of them threshold values equal 5,7,15. Terminated it, was also tested

the INRI prediction score. Then was effectuated a test concerning power consumption, making an analysis with traditional system and INRI.

5.1.1 Dataset

For testing INRI, the dataset chosen was ImageNet, using more than 10k image of different randomly classes. ImageNet contains over 14 million of images labelled splitted on 20 thousand categories [31]. It was opted to using these database because is the most used for CNN challenges.

5.1.2 Program languages and frameworks

For the hardware simulation, it was used C language with XCode IDE. Pixel clustering and result comparison was implemented using Python. It was chosen because it's a very powerful scripting language and it's extremely supported for machine learning. Concerning about CNN and prediction parts, it was simulated using Keras.

Keras is a powerful framework for Python able to execute some of the most famous CNN pre trained architecture model such as VGG-18, GoogleNet, AlexNet. Besides is also possible to build its own CNN architecture. For this reason it was chosen to be used in the CNN implementation.

The Keras installation is the same one for every Operating System, just open Pyhton terminal and write "pip3 Keras".

5.1.3 Hardware simulation

The code was developed in C language. If you are in a windows environment, it just to double click on the .exe file and execute the program. If you're under unix open the terminal and through the commands goes to the folder where the program is placed and launch it. After launching the program, it bring you step by step asking you the topology and the main characteristic of the system. It gives as output, the power consumption, the energy and the clock latency for completing the operation.

5.1.4 Pixel Clustering

To set the environment, put the program and the images you want to apply the algorithm on the same folder, just they're are named with "imageXXXX.jpg (where XXXX is a number that goes from 0001 to 9999).

After downloading the program, open the Python shell and goes through the folder where there is the program. Launch the program and after the conversion you should have the images converted called "imageXXXXopt.jpg". It also returns the average memory accesses reduction.

5.1.5 Result comparison

Put the program in two folders: the one where is present the images without Pixel Clustering and the one where is applied

Launch the program and it produces a .txt value where, according to user choice, writes the top 1/3/5 prediction score results. (in the folder containing image applied Pixel Clustering algorithm the file produce is called "opt.txt", while in the other folder is "norm.txt").

The part about the prediction was developed using Keras and its pretrained model.

5.1.6 Result comparison2

After producing the two txt files, put them in folder where is present "result_comparison2.py". It give the percentage of how the two systems produce the same prediction.

5.2 Memory access

In the x-axis are set all systems tested, represented with diferents colour, while in y-axis shows normalized values in percentage memory access off-chip drops compared to a traditional system. For example if a certain system has a y value equal 0.1 it means that compared to the same system without any smart facilities, it accesses in memory 10 times less. As depicted the figure 5.1 the CNN architecture with the lowest memory access is INRI set with g=4 and t=15 (red bar), reducing the accesses about 3x times.

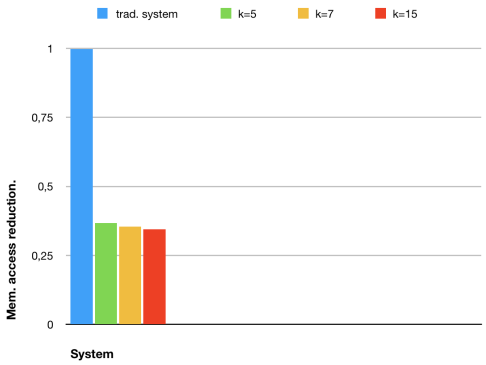


Figure 5.1

Figure 5.2 repeats the same experiments, but change g=7. In this case, it is possible to obtain even an higher memory access off-chip reduction , reaching a percentage about 75% (4x times lower)

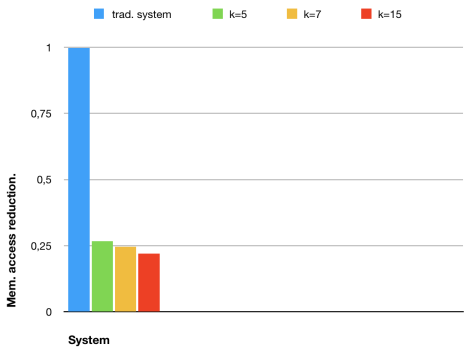


Figure 5.2

Evaluating INRI with EYERISS, a dataflow state of art for CNN, it achieves a better ratio memory access/operation, requiring less data transfer on-off chip for operate correctly. Figure 5.3 shows an average reduction of 30% using CNN Alexnet an INRI g=4,k=5.

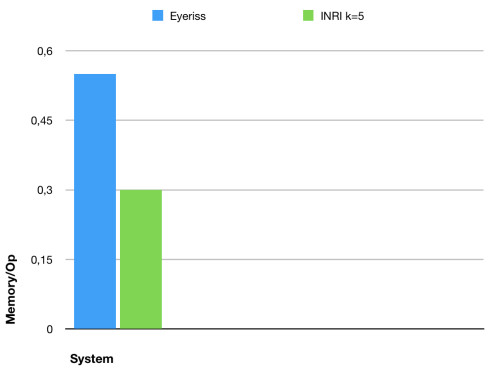
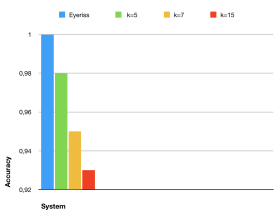


Figure 5.3

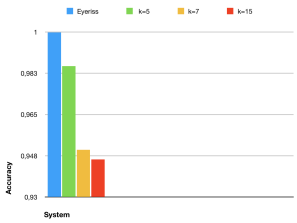
5.3 Accuracy

An important parameter for a CNN architecture is accuracy. A good prediction score is around 80% for top-1 chart and 90% for top-5 one. , For analizing it , it was used 10k randomly images from ImageNet. For every image was applied the Pixel Clustering and then it was compared with its same one but with applied the algorithm and see if they have the same top 1.Y-axis shows in percentage normalised by 10 how much the INRI top-1 output is equal to a system without Pixel Clustering. For instance if a certain system get a score equal '1', means that it has obtained, for every image, the same top-1 prediction with a system feed with no modified image. x-axis shows all architecture. So it wasn't analysed directly the prediction of the INRI, but rather, in percentage, how the outputs differ from a traditional system.

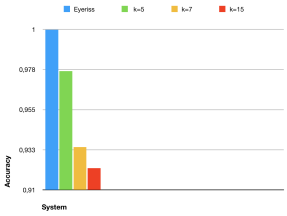
The setup used was ZFNet, AlexNet and VGG 18 implementing them on a system without any memory deduction technique and INRI with g=4,7 and t=5,7,15.



(a) ZfNet



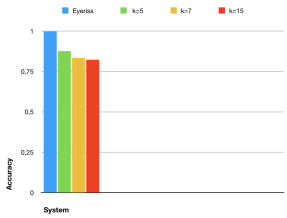
(b) AlexNet



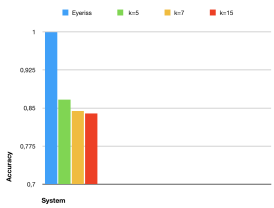
(c) Vgg19

Figure 5.4: CNN comparison with g=4

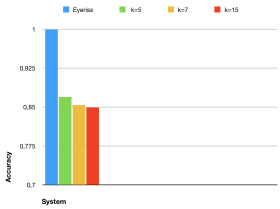
As we can notice in figure 5.4 , a good performance has reached when g=5 with an average loss around 5%, achieving its best performance has reached when g=5 and k=4. When g=7 (figure 5.5) the accuracy has lower performance, with an average around 15%.



(a) ZfNet



(b) AlexNet



(c) Vgg19

Figure 5.5: CNN comparison with g=7

The best trade off among memory access/ accuracy is setting g=4 and k=5. With this experiment was not analyse the prediction of iNRI , but rather how differs the outputs from a traditional architecture.

5.4 Prediction

CNNs are used most of the times for image and video recognition. For this reason the main purpose is realise a machine that achieves a good prediction.

In this section was analyse the INRI prediction score top-1. It was used a set of 30 images coming from 5 categories (cat, dog, cow, car, chair) and AlexNet, VGG-16 and ZfNet as CNN .

The results in figure 5.6 shows INRI achieved an average score of 70% in the three CNN, while a tradition system reached 76,6% .

5.5 Bottom-up: training and testing a CNN architecture with INRI

The tests done until now has demonstrated a significant memory reduction without affecting too much performance (above all using INRI with g=5 and t= 4 settings), next step was about built a CNN from the bottom, training the system with Pixel Clustering images input and after starting the predictions.

For this reason was developed a CNN able to recognise if the input received contains was a dog or a cat. The source code is here available

https://*github.com/nmorando/real*$_t$*b/blob/master/thesis/dogcat*

. For this experiment was trained two networks, one with Pixel Custering algorithm and the other one without any algorithm. Then for every network was send the same dataset with and without Pixel Clustering algorithm. For training the networks was used a dataset containing 1k image splitted halved for both.

(https://*github.com/nmorando/real*$_t$*b/blob/master/thesis/dogcat*.)

For the prediction set, was used 100 randomly image of cat or dog

(https://*github.com/nmorando/real*$_t$*b/blob/master/thesis/dogcat*)

. Both CNN architecture achieve a score about 90% of accuracy, without any significant different among them.

5.6 Area & power consumption

The introduction of a dispatcher and a Pixel Clustering bring improvements in terms of system performance and memory access, garantueeing reasonable containment in terms of area and power consumption.

The main area is occupied by the PE, followed by the INRI dispatcher and then to the Pixel Clustering Module.

5.6.1 Area

The overall area occupied by the system (local buffer excluded) is equal to . A single PE size is $7744um^2$, but in the system 121 PEs are presents, occupying $937024um^2$. The whole area occupied is equal to $939473um^2$. The insertion of a dispatcher and a pixel clustering , occupy in percentage, only the 3,30% total area.

Table 5.1: Area results

	total area um^2	total area %
PE	7744	96,7
INRI DISPATCHER	1539,14	2,78
Pixel Clustering Module	455,46	0,52

5.6.2 Energy & Power consumption

Concerning about the power supply, as depicted in table 5.2, PIxel castering module consumes 2,48 mW, while INRI DISPATCHER 51,2 mW and PE 32 mW.

The module off-chip memory considered was Micron DDR DRAM. During operation mode, it consumes 81 mW. More informations are available on this link $https://www.micron.com/parts/dram/ddr4-sdram/mt40a512m16jy-062e?pc=6BD1DB2E-A721-4109-8CE3-420A8A22E528$.

Table 5.2: power consumption

	power consumption mW
PE	32
INRI DISPATCHER	51,2
Pixel Clustering Module	2,48

Table 5.3 shows INRI power consumption and average energy consumed to complele a CNN layer considering off-chip memory. The average energy necessary to complete a layer is equal to 4465 J (The measure was obtained doing an average between all layers of all CNN architectures), A traditional system, without any stratagem to reduce memory access consume 6577 J. In comparison it was able to achieve an energy reduction equal about 35%.

Table 5.3: power & energy consumption

	power consumption mW	energy consumption J
Memory	81	11902,00
INRI DISPATCHER	51,2	1539,15
Pixel Clustering Module	2,48	2222,00

Figures 5.6 shows the average energy reduction concerning memory accesses. A traditional system consumes about 30517 J, while INRI system 11902 J a reduce around 0,4%. The tests consider the partial sum, ofmaps, ifmaps, weights.

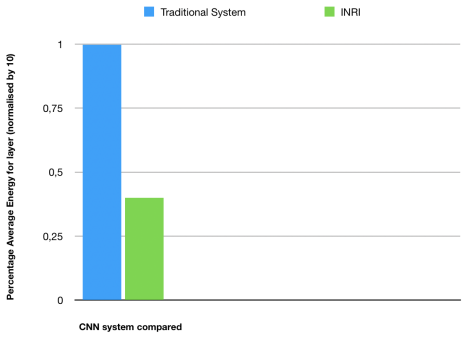


Figure 5.6

Chapter 6

Conclusion

Compared to a traditional system, this works demonstrates that implementing a systolic architecture and pixel clustering algorithm, is possible to reduce 3 times memory off-chip reducing up to 70% power consumption. This can be useful above all in that system that needs to implement vision recognition, without having high performance system.

Besides was also demonstrate that is possible to train. the network achieving interesting result. A further step enharced of the system could be a better memory management concerning the pixel clustering algorithm.

The system implement a SRAM with 32 bit word size cell, but only 19 (16 bit contains data,3 bit for algorithm) of them are really use.

An interesting implementation could be reduce the bit data to 8 (different studies demonstrate that up to 8 bit, the accuracy of CNN is still accettable),testing the performance and so implementing a sram with 16 bit word cell.

Chapter 7

FIles

In https://*github.com/nmorando/real*$_t$*b/blob/master/thesis/pe*
are available all files described in the previous chapters, where is possible to download an make every modification in order to make improvements.

Bibliography

[1] PREETI RANJAN PANDA ,*On-Chip vs. Off-Chip Memory: The Data Partitioning Problem in Embedded Processor-Based Systemsi*, 2016

[2] Tim Stack ,*Data Center Internet of Things (IoT) Data Continues to Explode Exponentially. Who Is Using That Data and How?*, 2018

[3] *https://medium.com/@sidereal/cnns-architectures-lenet-alexnet-vgg-googlenet-resnet-and-more-666091488df5*,

[4] *https://machinelearningmastery.com/supervised-and-unsupervised-machine-learning-algorithms/*,

[5] *https://www.datascience.com/blog/supervised-and-unsupervised-machine-learning-algorithms /*,

[6] *https://medium.com/@kanchansarkar/relu-not-a-differentiable-function-why-used-in-gradient-based-optimization-7fef3a4cecec*,

[7] Karen Simonyan, Andrew Zisserman*Very Deep Convolutional Networks for Large-Scale Image Recognition*, 2015

[8] *https://www.extremetech.com/extreme/188776-how-l1-and-l2-cpu-caches-work-and-why-theyre-an-essential-part-of-modern-chips*, 2015

[9] Yu-Hsin Chen,Tushar Krishna, Joel Emer, Vivienne Sze,*NEyeriss: An Energy-Efficient Reconfigurable Accelerator for Deep Convolutional Neural Networksi*, 2016

[10] Kevin Kai-Wei Chang, Donghyuk Lee, Zeshan Chishti ,*Improving DRAM Performance by Parallelizing Refreshes with Accessesi*, 2016

[11] Liangzhen Lai,Naveen Suda,Vikas Chandra,*Deep Convolutional Neural Network Inference with Floating-point Weights and Fixed-point Activationsi*, 2017

[12] Liangzhen Lai,Naveen Suda,Vikas Chandra,*Deep Convolutional Neural Network Inference with Floating-point Weights and Fixed-point Activationsi*, 2017

[13] Shahariar Rabby, *https://medium.com/@shahariarrabby/lenet-5-alexnet-vgg-16-from-deeplearning-ai-2a4fa5f26344* , 2017

[14] Yann LeCun,Lèon Bottou,Yoshua Bengio, Patrick Haffner, *Gradient-Based learning Applied to Document Recognition* , 1998

[15] *https://pythonmachinelearning.pro/introduction-to-convolutional-neural-networks-for-vision-tasks* , 1998

[16] *https://stats.stackexchange.com/questions/269893/2d-convolution-with-depth* , 2017

[17] *http://cs231n.github.io/convolutional-networks/*

[18] *https://people.inf.ethz.ch/omutlu/pub/VAMPIRE-DRAM-power-characterization-and-modeling_sigmetrics18-talk.pdf*

[19] *https://pdfs.semanticscholar.org/64db/333bb1b830f937b47d786921af4a6c2b3233.pdf*

[20] Adi Deshpande,*https://adeshpande3.github.io/A-Beginner%27s-Guide-To-Understanding-Convolutional-Neural-Networks-Part-2/*, 2010

[21] R. Hameed, W. Qadeer, M. Wachs, O. Azizi, A. Solomatnikov, B. C. Lee, S. Richardson, C. Kozyrakis, and M. Horowitz, *Understanding Sources of Inefficiency in General-purpose Chips*, in ISCA, 2010

[22] H.T. Kung, S.W.Song,*A Systolic 2-D Convolution Chip*, 1981

[23] Shihao Wang, Dajiang Zhou, Xushen Han, Takeshi Yoshimura,*Chain-NN: An Energy-Efficient 1D Chain Architecture for Accelerating Deep Convolutional Neural Networks*, 2016

[24] S. Arthur,*Some Studies in Machine Learning Using the Game of Checkers* , IBM Journal of Research and Development, 1959

[25] Alex Krizhevsky, Ilya Sutskever, Geoffrey E. Hinton,*ImageNet Classification with Deep Convolutional Neural Networks* , 2012

[26] Matthew D. Zeiler, Rob Fergus,*Visualizing and Understanding Convolutional Networks* , 2012

[27] Karen Simonyan, Andrew Zisserman,*Very Deep Convolutional Networks for Large-Scale Image Recognition* , 2012

[28] http://www.image-net.org/challenges/LSVRC/2013/results.php#cls, 2012

[29] https://towardsdatascience.com/applied-deep-learning-part-1-artificial-neural-networks-d7834f67a4f6, 2017

[30] https://medium.com/coinmonks/paper-review-of-alexnet-caffenet-winner-in-ilsvrc-2012-image-classification-b93598314160

[31] Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev

Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, Alexander C. Berg, Li Fei-Fei*Large Scale Visual Recognition Challenge* , 2015

Appendix A

image_converter.py

```
import cv2
    import numpy as np
    from PIL import Image
    import os
    # read image into matrix.
    directory = os.fsencode("/home/nicolomorando/Desktop/testset")
    counter_p=1
    fc = 0
    fco = 0
    for file in os.listdir(directory):
     filename = os.fsdecode(file)
     if filename.endswith(".jpg") and filename.find("opt") == -1:
     if(counter_p<10) :
     m = cv2.imread("image_000" + str(counter_p) + ".jpg" )
                 if(counter_p<100) and (counter_p>=10) :
     m = cv2.imread("image_00" + str(counter_p) + ".jpg" )
     if(counter_p<1000) and (counter_p>=100):
m = cv2.imread("image_0" + str(counter_p) + ".jpg" )
    # get image properties.
    h,w,bpp = np.shape(m)
    counter=0
    tmp=0
    counter_noOpt=0
    counter_op=0
     # BLUE = 0, GREEN = 1, RED = 2.
```

```
for i in range(0,3):
        for px in range(0,h):
        for py in range(0,w):
                    if(counter==0):
                         tmp=m[px][py][i]
                        counter_op=counter_op+1
                            if((m[px][py][i]>tmp-5) and (m[px][py][i]<tmp+5)):
                                m[px][py][i]=tmp
                                counter=counter+1
                            else:
                                counter=0
                            if counter==3:
                                counter=0
                                counter_noOpt=counter_noOpt+1
cv2.imwrite('img_R-G.jpg',m)
print(counter_op)
print(counter_noOpt)
```

Appendix B

comparison_result.py

```
from keras.preprocessing.image import load_img
   from keras.preprocessing.image import img_to_array
   from keras.applications.vgg19 import preprocess_input
   from keras.applications.vgg19 import decode_predictions
   from keras.applications.vgg19 import VGG19
   import cv2
   from time import sleep
   # load the model
   import os
   # read image into matrix.
   files = open("opt.txt","w")
   directory = os.fsencode("/home/nicolomorando/Desktop/opt")
   model = VGG19() # AlexNet() ZfNet()
   counter_p = 1
   # load an image from file
   for file in os.listdir(directory):
         filename = os.fsdecode(file)
         if(counter_p<10) :
               image = load_img("image_000" + str(counter_p) +"opt"+
".jpg", target_size=(224, 224))
         if(counter_p<100) and (counter_p>=10) :
               image = load_img("image_00" + str(counter_p) + "opt"+
".jpg", target_size=(224, 224))
         if(counter_p<1000) and (counter_p>=100):
```

```
            image = load_img("image_0" + str(counter_p) + "opt"+
".jpg", target_size=(224, 224))
        # convert the image pixels to a numpy array
        image = img_to_array(image)
        # reshape data for the model
        image = image.reshape((1, image.shape[0], image.shape[1], im-
age.shape[2]))
        # prepare the image for the VGG model
        image = preprocess_input(image)
        # predict the probability across all output classes
        yhat = model.predict(image)
        # convert the probabilities to class labels
        label = decode_predictions(yhat)
        # retrieve the most likely result, e.g. highest probability
        label = label[0][0]
        # print the classification
              files.write(label[1]+"
n")
```

Appendix C

comparison_result.py pt2

```
c = 0
    ce = 0
    with open("norm.txt") as f1, open("opt.txt") as f2:
            for x, y in zip(f1, f2):
                    x = x.strip()
                    y = y.strip()
                    c+= 1
                    if x==y :
                            ce+= 1
    print(ce/c)
```

Appendix D

performance_extractor

```
#include <stdio.h>
   #include <stdlib.h>
   FILE *f;
   int n_ram_access=0;
   float energy_ram_access = 0;
   float energy_conv =0 ;
   int clk_ram=0;
   int clk_tot=0;
   float energy_tot=0;
   float pe_energy=0.00000135;
   float ram_energy=0.228;
   int stride = 0 ;
   int padding = 0;
   int image_size = 0;
   int kernel_image_depht = 0 ;
   int kernel_size = 0;
   int kernel_pack = 0;
   int pe = 0;
   int average_mem_access = 0;
   int local_buffer_size = 0 ;
   int total_memory_access= 0;
   float total_number_element=0;
   int quante_conv_posso_fare = 0;
   int is_1dd = 0;
   float how_many_time_i_split=0;
```

```
int main(int argc, const char * argv[])
        f = fopen("info.txt","r");
        if ( f != NULL )

             char line [ 128 ]; /* or other suitable maximum line size */
            while ( fgets ( line, sizeof line, f) != NULL ) /* read a line */
             if(f != NULL)

                  fputs ( stride, atoi(fgets ( line, sizeof line, f) ) );
                  fputs ( padding, atoi(fgets ( line, sizeof line, f) ) );
                  fputs ( image_size, atoi(fgets ( line, sizeof line, f) ) );
                 fputs ( kernel_image_depht, atoi(fgets ( line, sizeof line,
f) ) );
                  fputs ( kernel_size, atoi(fgets ( line, sizeof line, f) ) );
                  fputs ( kernel_pack, atoi(fgets ( line, sizeof line, f) ) );
                   fputs ( average_mem_access, atoi(fgets ( line, sizeof
line, f) ) );
                  fputs ( local_buffer_size, atoi(fgets ( line, sizeof line, f)
) );
                  fputs ( is_1dd, atoi(fgets ( line, sizeof line, f) ) );
                  fputs ( pe, atoi(fgets ( line, sizeof line, f) ) );

        fclose ( f );
             total_number_element = (float)(average_mem_access*image_size*image_
             if (total_number_element>local_buffer_size)

                  how_many_time_i_split=(float)(total_number_element/local_buffe

             if((int)(how_many_time_i_split)==0)
                  n_ram_access= total_number_element + kernel_image_depht*kerne
             else
                  n_ram_access= total_number_element*kernel_pack+kernel_image_
             quante_conv_posso_fare=pe/(kernel_size*kernel_size);
             if(is_1dd==0)

                  clk_tot = (padding*2+image_size-kernel_size)*(kernel_size/quante_
```

```
            if(is_1dd==1)

                clk_tot = (padding*2+image_size-kernel_size+kernel_size*kernel_siz

        energy_ram_access = n_ram_access+ram_energy;
        energy_conv = clk_tot*pe_energy;
        energy_tot= energy_ram_access+energy_conv;
return 0;
```

Appendix E

Processing_Element.vhd

```
library ieee;
   use ieee.std_logic_1164.all;
   use ieee.std_logic_arith.all;
   use ieee.std_logic_unsigned.all;
   entity PE is
   Port (
         clk : in std_logic;
        rst: in std_logic;
        a : in std_logic_vector (7 downto 0);
        b : in std_logic_vector (15 downto 0 );
        y : out std_logic_vector (15 downto 0);
        a1: out std_logic_vector(7 downto 0)
   );
   end PE;
   architecture BEH of pe is
        signal reg1 : std_logic_vector(7 downto 0);
   begin
   process(clk,rst)
   begin
        if(rst='0') then
             reg1<=a;
             y<="0000000000000000";
             a1<=reg1;
        else
             if(rising_edge(clk)) then
```

```
                y<=a*reg1+b;
                reg1<=a;
                a1<=reg1;
            end if;
        end if;
    end process;
    end beh;
```

Appendix F

Pixel_Clustering.vhd

```
library ieee;
   use ieee.std_logic_1164.all;
   use ieee.numeric_std.ALL;
   useieee.std_logic_arith.all;
   useieee.std_logic_unsigned.all;
   entity pixel_clustering is
   Port (
   clk : in std_logic;
   rst: in std_logic;
   a : in std_logic_vector (15 downto 0);
   b : out std_logic_vector (15 downto 0 );
   th,c: in std_logic_vector( 2 downto 0);
   write_en : out std_logic
   );
   end pixel_clustering;
   architecture BEH of pixel_clustering is
   signal th1,counter,counter1 : std_logic_vector(2 downto 0) := "000";
   signal sel : std_logic_vector(15 downto 0);
   begin
   process(clk,rst)
   begin
         if(rst='0') then
               th1<=th;
               counter<=c;
               else
```

```
                if(rising_edge(clk)) then
                    if((counter1>counter) or (( sel<a+th1 ) and (sel>a-
th1))) then
                        counter1<=counter1+1;
                        write_en<='0';
                    else
                        b<=a;
                       sel<=a;
                        counter1<="000";
                        write_en<='1';
                    end if;
            end if;
        end if;
    end process;
    end beh;
```