POLITECNICO DI TORINO

Master Degree in Computer Engineering

Master Thesis

Deep Learning Solution for Analyzing Visual Imagery in Industrial Applications

Machine learning on low-power low-cost platforms: an application case study

Supervisors
Prof. Bartolomeo Montrucchio
Prof. Renato Ferrero

Candidate
Carmine D'Amico

Company supervisors
Istituto Superiore Mario Boella
Dr. Olivier Terzo, Dr. Alberto Scionti

December 2018

A mia madre, che mi osserva da lassù.
A chi mi supporta ogni giorno, credendo in me.

Summary

Machine learning is one of the hottest topics of the last years in the computer industry. The growing interest on methods for processing large amount of heterogeneous data and new cognitive systems is creating new challenges and opportunities. In the ICT domain, a major effort is spent on improving and applying machine learning, deep learning and in general artificial intelligence techniques. Applications can be seen in various fields, from civil to military, through industrial. This work of thesis is focused precisely on this last area of application and precisely on the image recognition problem, which is addressed using deep learning (DL) models based on a state-of-the-art Convolutional Neural Network. Image recognition is used in the industrial area for the quality control of the products, for tracking, counting and measuring objects, etc.

Although high performance devices are often required to perform image recognition operations, one of the most popular market trends is to try to use devices that require a lower amount of electric power to work. Starting from this statement, the goal of this thesis was to try to use the Parallella board, that is a modern low-power parallel general-purpose device, to run a deep learning model based on Darknet, an open source neural network framework. The limitations of the device used, in terms of both performances and available resources, have represented the main challenges of this research and also the starting point for all the solutions found. Different approaches have indeed been followed and investigated to optimize the evaluation times for a single image: from basic solutions for making the most of the board's multicore architecture to ad- hoc solutions developed to bypass the main bottlenecks of the device (like the poor amount of memory available), up to the use of optimized methods to speed up the convolution operation.

Finally, the different approaches used have been tested and evaluated, allowing to express some considerations on the use of low-power devices for machine learning applications and in particular on the direction that the scientific research could take in order to improve such use.

Acknowledgements

This thesis was carried out at *Istituto Superiore Mario Boella*, in the *Advanced Computing and Electromagnetics (ACE)* laboratory. I would like to offer my most heartfelt thanks to Dr. Olivier Terzo, head of the research area, and to Dr. Alberto Scionti, that was my supervisor for this work. It was thanks to them and their support if I had the chance to do this thesis.
I would also like to thank all the researchers of this laboratory for making the working environment as welcoming as possible, allowing me to work serenely and feel like a real colleague.

Contents

List of Tables VIII

List of Figures IX

1 Introduction 1

1.1 Thesis Motivations . . . 1

1.1.1 Industrial Applications . . . 1

1.1.2 Heterogeneous Computing . . . 2

1.2 Limitations . . . 2

1.3 Thesis Structure . . . 3

2 State Of The Art 4

2.1 Machine Learning . . . 4

2.1.1 Classification Algorithms . . . 5

2.1.2 What Is A Neural Network? . . . 6

2.1.3 Convolutional Neural Network . . . 11

2.2 Darknet . . . 14

2.2.1 Implementation Details . . . 15

2.2.2 Convolutional Layer . . . 16

3 Hardware Platform 17

3.1 Parallella . . . 17

3.1.1 Why The Parallella? . . . 18

3.1.2 Hardware Architecture . . . 19

3.1.3 Epiphany Coprocessor . . . 20

3.1.4 Application Development . . . 24

3.1.5 Problems Faced . . . 26

4 Solution Design 28
4.1 General Idea . . . 28
4.1.1 Zynq as master, Epiphany as slave . . . 28
4.1.2 Focus On The Shared Memory . . . 29
4.1.3 Implemented Layers . . . 33
4.1.4 Synchronization . . . 38
4.1.5 Training The Network . . . 40
4.2 The Models Used . . . 40
4.2.1 Tiny Darknet . . . 40
4.2.2 MNIST Custom Model . . . 41
4.3 Basic Solution . . . 42
4.4 Extended Memory Solution . . . 44
4.4.1 Implementation . . . 45
4.5 Optimized Convolution Solution . . . 48
4.5.1 Memory-efficient Convolution (MEC) . . . 49
4.5.2 Implementation . . . 50
4.6 Summary . . . 51

5 Experimental Evaluation 52
5.1 Performance Analysis . . . 53
5.1.1 Methodologies Used . . . 54
5.1.2 Results for Tiny Darknet . . . 56
5.1.3 Results for the MNIST Custom Model . . . 58
5.1.4 Analysis . . . 61
5.2 Power Consumption Analysis . . . 65
5.2.1 Methodologies Used . . . 65
5.2.2 Results for Tiny Darknet . . . 69
5.2.3 Results for the MNIST Custom Model . . . 70
5.2.4 Analysis . . . 72

6 Conclusions 75
6.1 Future Works . . . 75

Bibliography 77

List of Tables

3.1 Coordinates to identify the Epiphany cores 22
4.1 Tiny Darknet 41
4.2 Network trained on the MNIST dataset 42
4.3 Epiphany Architecture - Memory Performance 45
5.1 Tiny Darknet - Basic Solution - Layer Execution Times 56
5.2 Tiny Darknet - Basic Solution - Total Execution Time 57
5.3 Tiny Darknet - Extended Memory Solution - Layer Execution Times 57
5.4 Tiny Darknet - Extended Memory Solution - Total Execution Time . 57
5.5 MNIST Custom Model - Basic Solution - Layer Execution Times . . . 58
5.6 MNIST Custom Model - Basic Solution - Total Execution Time . . . 58
5.7 MNIST Custom Model - Extended Memory Solution - Layer Execution Times 59
5.8 MNIST Custom Model - Extended Memory Solution - Total Execution Time 59
5.9 MNIST Custom Model - MEC Solution - Layer Execution Times . . . 60
5.10 MNIST Custom Model - MEC Solution - Total Execution Time . . . 60
5.11 Base Energy Cost 66
5.12 Base Energy Cost For Remote Loads And Stores On The eMesh . . . 66
5.13 Tiny Darknet - Basic Solution - Energy Consumption 69
5.14 Tiny Darknet - Basic Solution - Power Consumption 69
5.15 Tiny Darknet - Basic Solution - Parallella Power Consumption 69
5.16 Tiny Darknet - Extended Memory Solution - Energy Consumption . . 69
5.17 Tiny Darknet - Extended Memory Solution - Power Consumption . . 70
5.18 Tiny Darknet - Extended Memory Solution - Parallella Power Consumption 70
5.19 MNIST Custom Model - Basic Solution - Epiphany Energy Consumption 70
5.20 MNIST Custom Model - Basic Solution - Epiphany Power Consumption 70

5.21 MNIST Custom Model - Basic Solution - Parallella Power Consumption 70
5.22 MNIST Custom Model - Extended Memory Solution - Epiphany Energy Consumption 71
5.23 MNIST Custom Model - Extended Memory Solution - Epiphany Power Consumption 71
5.24 MNIST Custom Model - Extended Memory Solution - Parallella Power Consumption 71
5.25 MNIST Custom Model - MEC Solution - Epiphany Energy Consumption 71
5.26 MNIST Custom Model - MEC Solution - Epiphany Power Consumption 71
5.27 MNIST Custom Model - MEC Solution - Parallella Power Consumption 72

List of Figures

2.1 Graphical representation of a perceptron 6
2.2 Example of an input layer for a 3×3 image 8
2.3 Example of an output layer with 4 neurons 9
2.4 Example of an hidden layer with 7 neurons 10
2.5 Local receptive field for an hidden neuron 12
2.6 Example of a CNN 14
3.1 The Parallella board 20
3.2 Representation of the two-dimensional array of eNodes 21
3.3 Graphic representation of an Epiphany node 22
4.1 Conceptual division of the shared memory 30
4.2 Representation of the workflow followed 32
4.3 Allocation of input channels to eCores 35
4.4 Allocation of input submatrices to eCores 35
5.1 Tiny Darknet - Basic Solution - Kernel Execution Times 56
5.2 Tiny Darknet - Extended Memory Solution - Kernel Execution Time 58
5.3 MNIST Custom Model - Basic Solution - Kernel Execution Times . . 59
5.4 MNIST Custom Model - Extended Memory Solution - Kernel Execution Time 60
5.5 MNIST Custom Model - MEC Solution - Kernel Execution Times . . 61
5.6 Tiny Darknet - Total Execution Times Comparison 62
5.7 MNIST Custom Model - Total Execution Times Comparison 62
5.8 Tiny Darknet - Epiphany Power Consumption Comparison 72
5.9 MNIST Custom Model - Epiphany Power Consumption Comparison . 73

Chapter 1

Introduction

Among all the ICT trends, in recent years, machine learning (ML) has had the greatest impact, revolutionizing various fields, from civil to military, through industrial. Scientific community started to look at deep neural networks (DNNs) and other ML techniques as a convenient way to crunch the enormous amount of data generated by experiments. This work of thesis is focused precisely on the industrial area of application and precisely on the image recognition problem, which is addressed using deep learning (DL) models based on a state-of-the-art Convolutional Neural Network. Image recognition is used in the industrial domain for the production quality control of the products, for tracking, counting and measuring objects, etc.

The work carried out is an application case study for machine learning techniques, regarding in this thesis the recognition of images, on a low-power low-cost platform. In particular the targeted device was the Parallella single board computer.

1.1 Thesis Motivations

The work summarized in this thesis was carried out at the "Istituto Superiore Mario Boella (ISMB)" at the laboratory of Advanced Computing and Electromagnetics. This has determined some characteristics of this study and some of the decisions made. In particular, the term *industrial applications* in the title of this thesis is attributable to the possible application contexts, as well as to the choice of the development device.

1.1.1 Industrial Applications

During the research activity, a job opportunity arrived at the ISMB; it concerned the design of a low energy consumption solution, to be used in an industrial context, for the production and processing of hazelnuts. This opportunity has helped to shape and above all to direct this work. In particular, in fact, what was required was a solution capable of being used in the production chain, in order to identify and recognize the defective hazelnuts. These can be recognized through some features they possessed, such as atypical shape and/or color. This led to the need to use

and apply machine learning techniques for this purpose, using image recognition as a mean of achieving the desired final result. As it can be seen in the next chapter, in which an excursus about some of the most used machine learning techniques is presented, *Convolutional Neural Networks* represent as today the state of the art for this purpose. They have been the ones used during this research.

In general, machine learning is taking more and more ground in industrial contexts and in particular in manufacturing. As reported by several articles released in recent years [1][2], it is revolutionizing the way of working in some cases, helping companies to increase their efficiency, reducing the number of committed and increasing the accuracy of work. Some examples of applications are shown below:

- It is used to monitor, record, and analyze everything in manufacturing, to try to proactively identify possible problems in order to solve them promptly. Data from a variety of different sensors is used for this purpose, so as to reach an amount that can be used to train machine learning models.

- It is used to make robots smarter, integrating deep learning techniques in them to improve both their work and their integration with the work done by humans.

- It is used to improve quality control, in particular within assembly lines, where the weaknesses of the machines are identified in order to be minimized.

1.1.2 Heterogeneous Computing

The choice of the device to be used to obtain a solution capable of executing with low energy consumption and good performances has been influenced by another activity under way at the ISMB. This research activity concerns in particular a study on the use of heterogeneous architectures. This thesis has represented an excellent opportunity to be integrated into this framework, providing an application case study for machine learning techniques on a low-cost low-power platform. The platform chosen for this work was the *Parallella* board, conceived and developed by Adapteva. A comprehensive analysis of this device will be provided during the third chapter of this thesis.

1.2 Limitations

Despite one of the main reasons behind this thesis was the request from a customer, this work is more like a theoretical evaluation on the use of a platform with specific characteristics, such as the Parallella, for ML applications in the case of image recognition. At the end of this work, therefore, a real working prototype was not produced, rather an analysis based on the performance achieved, in terms of required execution times, and on energy consumption was provided.

1.3 Thesis Structure

This thesis is structured as follows:

- Chapter 2 gives an overview of the state of the art of the main machine learning techniques, starting from the perceptron up to the convolutional neural networks. It also introduces the Darknet framework, used as a starting point for this work to develop the solutions found.
- Chapter 3 describes and analyzes the Parallella board, the computing system used for this thesis, focusing in particular on the Epiphany architecture and its characteristics.
- Chapter 4 describes the approach used in this work, analyzing the basic principles that distinguish it. All the optimizations made to increase the performance of the solutions found are introduced, emphasizing on the differences between them and on what limitations of the hardware they have effect.
- Chapter 5 presents the obtained results in the course of the work, both from the point of view of performance and energy consumption. The same results are also analyzed and discussed.
- Chapter 6 concludes the thesis, providing a summary of the entire work done, with a look at possible future developments.

Chapter 2

State Of The Art

Machine Learning is one of the hottest topics of these years. Its application ranges in different areas: from the medical to the economic field, from the purely academic to the industrial scope.

This chapter introduces the topic, analyzing the concepts of *neural network* and *convolutional neural network*, together with their founding elements and their main characteristics. A particular focus is placed on image recognition, as this is the main application of machine learning that is dealt within this thesis. Finally this chapter presents and describes *Darknet*, an open source neural network framework which is used in the continuation of the practical part of this work.

2.1 Machine Learning

As suggested in the paper "A brief introduction into Machine Learning" [3] by G. Ratsch, *artificial intelligence*, i.e., the scientific field which aims to mimic intelligent abilities of humans by machines, could be considered the parent of *machine learning*, which is instead focused on how to make these machines able to "learn". As explained by the author, in this context the term *learn* is referred to an *inductive inference*, where the machine learns by observing a series of examples regarding a "statistical phenomenon".

Machine learning can be classified into two categories:

- *unsupervised learning*: features (or patterns) are typically found by searching regularities or by detecting anomalies in the available data;
- *supervised learning*: each considered example is associated to a *label*. A machine tries to extrapolate information from such examples in order to *predict* the labels for other cases, not contained in the examples.

In this chapter the focus is mainly on supervised learning, in particular on that class of problems where the labels are discrete, also called *classification problems*. Classification algorithms have the goal of distinguishing different input data, based solely

on the patterns recognized in them. In general they are used to find a functional mapping between the input data X and a class label Y:

$$Y = f(X)$$

In the scientific literature there are several examples of classification algorithms. In the following subsection the most important ones are described, lingering later on *neural networks* and especially on *convolutional neural networks*, that nowadays represent the state of the art in the image recognition field.

2.1.1 Classification Algorithms

In the following, five among the most important and the most used classification algorithms are described:

- *k-Nearest Neighbor Classification*: Such method was described in the paper "Nearest neighbor pattern classification" [4] by T. Cover and P. Hart. It finds the k points of the training set that are closer to the considered input; then, a label is assigned based on the label which is more present among the identified points;

- *Linear Discriminant Analysis*: This method was described by R.A. Fisher in the paper "The use of multiple measurements in taxonomic problems" [5]. An hyperplane is computed within the input space to minimize the variance between data of the same class (label), maximizing on the other hand the distance between different classes;

- *Decision Trees*: One of the most important and famous implementation of such method is the one provided by R. Quinlan et al. in the book "C4.5: Programs for Machine Learning" [6]. In this method a tree is built by partitioning the input data space recursively, with the aim to create the purest nodes possible, i.e. nodes that contain only points of the same class. The classification of a new input point is computed by visiting the tree previously built from the top to the bottom;

- *Support Vector Machines (SVM)*: This classification method was introduced for the first time in the paper "Support-Vector Networks" [7]. Its operating mechanism is given by the identification of an hyperplane, which is defined starting from the labeled training data. From these data the classification algorithm creates an optimal hyperplane which will categorize the new arriving data;

- *Neural Networks*: This is probably the most important and used method to resolve a classification problem. Such method takes inspiration from the mammalian's brain, trying to re-create it. Given the great importance of this type of algorithms, they are analyzed in depth in the next section.

2.1.2 What Is A Neural Network?

From Perceptron To Sigmoid Neuron

One of the first models used to try to represent human decision-making was the *perceptron*, which was introduced at first by Frank Rosenblatt in his paper "The perceptron: a probabilistic model for information storage and organization in the brain" [9]. Such model is very simple, it produces indeed a single binary output based on the input vector received. The output is computed using a vector of *weights*, where each element is associated to a single input component, indicating the importance of that input with respect to the output. Each weight is a real number. The mathematical formula that expresses how to calculate whether the output is 0 or 1 can be written as follows:

$$output = \begin{cases} 0 & \text{if } \sum_i x_i w_i \leq threshold \\ 1 & \text{if } \sum_i x_i w_i > threshold \end{cases}$$

where the weighted sum $\sum_i x_i w_i$ is compared to a certain *threshold* to determine the output. The threshold value is a real number belonging to the perceptron itself.

Such model can be simplified just moving the threshold to the other side of the inequality, renaming it as *bias*; moreover $\sum_i x_i w_i + bias$ could be rewritten as a dot product $\bar{x_i} \cdot \bar{w_i} + bias$. In this way the previous mathematical model becomes:

$$output = \begin{cases} 0 & \text{if } \bar{x_i} \cdot \bar{w_i} + bias \leq 0 \\ 1 & \text{if } \bar{x_i} \cdot \bar{w_i} + bias > 0 \end{cases}$$

Intuitively a perceptron can be imagined as an element that "takes decisions weighting up the inputs". A possible graphical representation of a perceptron could be the one in Fig. 2.1.

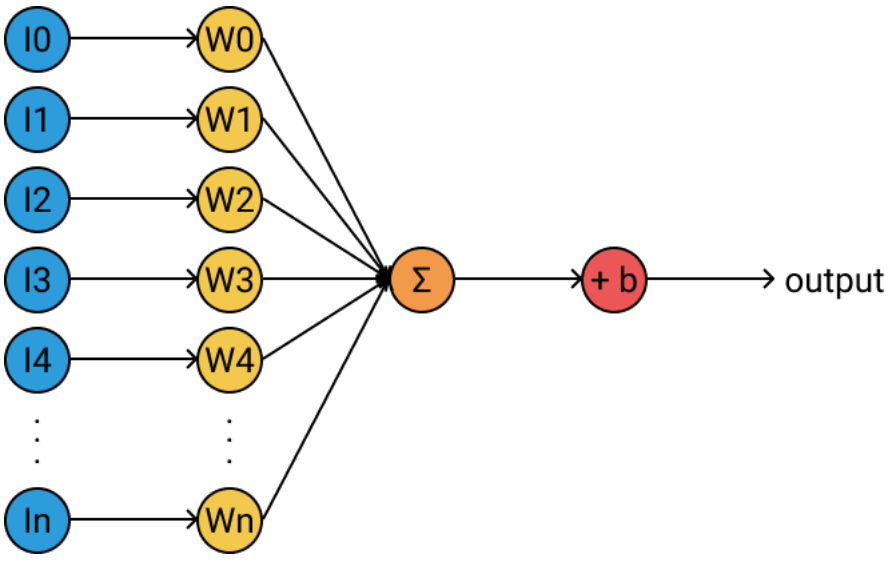

Figure 2.1. Graphical representation of a perceptron

Nowadays it is more common to use different models rather than the one based on the concept of perceptron. Among these, the most used one is definitely the *sigmoid neuron*.

As it can be imagined every modification made to the parameters used in a perceptron, whether it is performed out on the weights or on the bias, involves a change in the used model that can lead to different classification results. In a network made of perceptrons every change brings indeed to a complete "flip" of the output result from 0 to 1 or vice versa. On the contrary a very useful property for a practice neuron would be to make a small variation in the network's parameters coinciding with a small variation in the output itself. For instance, it is sufficient to think about a network that is classifying a 0 as a 1, the first thing to do should be to modify some weights and/or biases in order to fix such classification. With perceptrons what would happen is that, on the one hand, such particular case would be solved, but on the other hand all the other cases (or anyway a large part of them) could be compromised. This would make really hard a network of perceptrons to "learn".

On the contrary sigmoid neurons overcome such problem in a very efficient way. They are very similar to perceptrons, accepting a vector of inputs and using as their own parameters a vector of weights and a bias. The real difference is in the mathematical model used, which, for a sigmoid neuron, is:

$$output = f(\bar{x} \cdot \bar{w} + bias)$$

where *f()* takes the name of *activation function*. In particular in a sigmoid neuron the activation function is a *sigmoid function* in the form:

$$\sigma(x) = \frac{1}{1 + e^{-z}}$$

where in our case z is equal to $\sum_i x_i w_i + bias$. Using this model the resulting outputs are no longer binary, but they can be any real number. In this way, thanks to the properties of the sigmoid function, a small change in the local parameters of a neuron would reflect in a small change in the resulting output. At this point another way to think about a perceptron would be to consider it as a particular case of a neuron where the activation function is a *step function*.

Since the output of a sigmoid function is no longer a binary result, but a real number, such value should be interpreted. In some cases, indeed, such result could be convenient, for example to represent the "intensity" of some feature; in other cases it could be better to have a result that represent a particular class of results. In these cases a good convention could be to establish some ranges of results, classifying an output based on the range in which it falls.

A Complete Network

Obviously a single neuron could not be enough to achieve a good classification. What is needed is a complete network of neurons, able through its own topology to classify a given input.

In the case of image recognition, the input is usually represented by an input image that can be abstract as a matrix of input. In this particular case the *input layer*, i.e., the leftmost layer, is composed by a neuron for each pixel having as value the intensity of that pixel. For example, considering a 3×3 greyscale image the input layer would be composed by 9 neurons. A graphical representation of such case can be seen in Fig. 2.2.

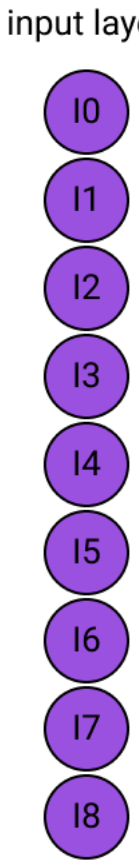

Figure 2.2. Example of an input layer for a 3×3 image

The *output layer*, instead, is the rightmost layer in the network. There could be different choices in the design of such layer. Considering four possible classification classes for the previous example, it could be decided to implement the output layer following different philosophies:

- There could be just one neuron in the output layer, indicating only whether a given input belongs or not to a single class;
- There could be four neurons in the output layer, i.e. a neuron for each different class. In this case the neuron with the highest value would represent the class assigned to a given input;
- There could be only two neurons, treating each of them as a binary value. They would be indeed enough to encode the result class, because $2^2 = 4$.

The choice about the design of such layer depends most of the times by an empirical observation of the described network designs: the one that performs better should be the right one. In Fig. 2.3 could be seen an example of the network previously described with four neurons in the output layer.

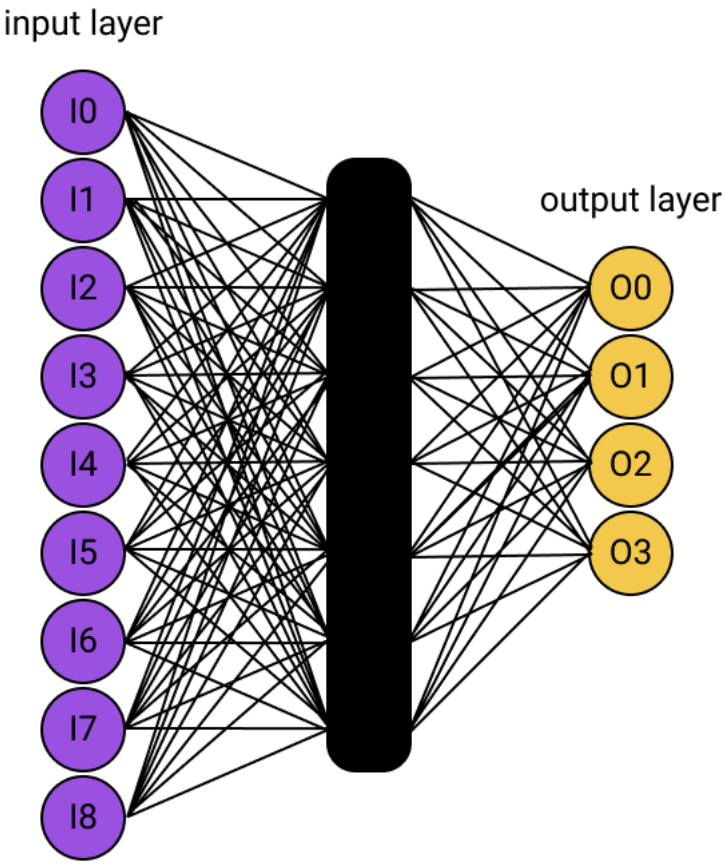

Figure 2.3. Example of an output layer with 4 neurons

The black box portrayed in the middle of Fig. 2.3 represents the one or more *hidden layers* present in a neural network architecture. These layers are the ones that contain neurons that are neither inputs nor outputs. As mentioned, a neural network could have just one hidden layers, as it can have more than one. There is not any exact rule that can be applied to the design of the topology of these layers. Researchers have developed, during the years, different heuristics that can be used in order to achieve a good trade off for the composition of such layers, in order to obtain the desired behavior from the neural network. The parameters that can be manipulated in the design phase regard the number of hidden layers as well as the number of neurons present in each hidden layer. In Fig. 2.4 is showed the network previously described as example with just one hidden layer composed by seven neurons.

From the same figure it can be seen how all the neurons in one layer are connected to all the neurons in the next layer, i.e., all the outputs from one layer are used as inputs for each neuron of the next layer. Such kind of nets, without loops in their topology, are also known as *feedforwarding networks*, as opposed to *recurrent neural*

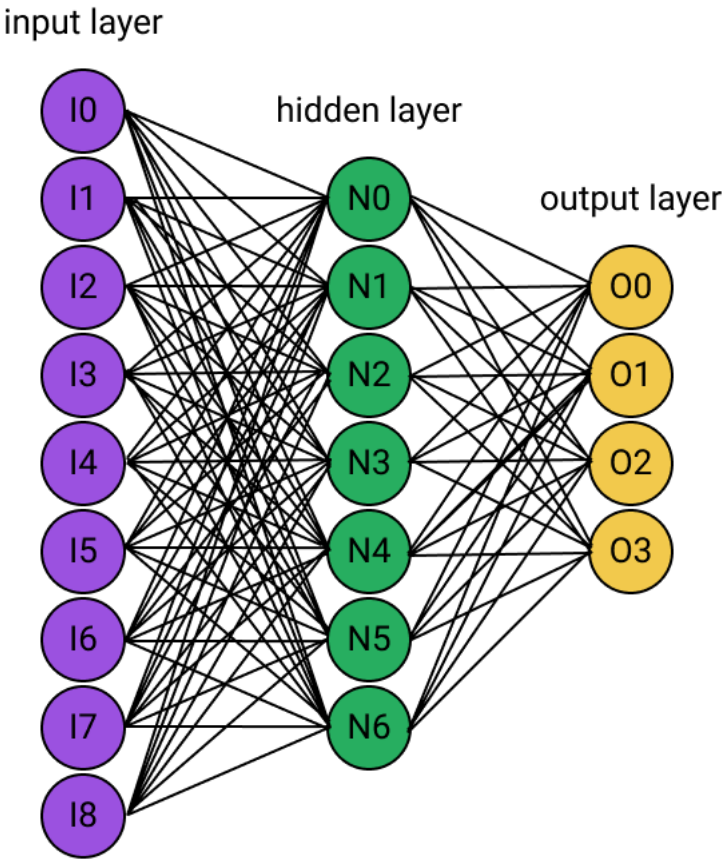

Figure 2.4. Example of an hidden layer with 7 neurons

networks where instead feedback loops are possible.

Training Neural Networks

Up to now, the parameters of the network have been discussed without specifying where they come from or how they are calculated. Weights and biases have been taken for granted, in reality they are computed through the so called process *training of the network*. The most used algorithm for such process is called *backpropagation* and it represents the state of the art training model for neural networks (and also for convolutional neural networks, as it will be seen).

The training of a network is not an argument of this work of thesis. The reason behind such statement lies in the fact that, for the practical experiences carried on during this work, already trained networks have been used. This was due to the poor performances of the board analyzed and used. The training of a network is, indeed, one of the most resource-intensive processes, taking a long time to be completed. *Darknet*, the open source neural network library used for this thesis that is described in the continuation of this chapter, provides some networks' models already trained, including *Tiny Darknet* that was argument of this thesis, together with the required parameters (weights, biases, etc...) to work with them. As stated during the introduction, the point on which we focused is on the evaluation process

and on how to optimize it, making it run on the board studied.

2.1.3 Convolutional Neural Network

As stated in the previous paragraph, *Convolutional Neural Networks* (or *CNNs*) are widely used to recognize and classify images. This is due to the spatial nature of an image, which does not fit perfectly with the architecture of a classical neural network.

The structure of a typical neural network, like the one previously described, involves the use of adjacent layers that are fully connected between each other. In this way a single neuron in a layer is connected to every neurons in the previous layer as well as with the ones in the next. This type of network treats all the pixels in the same way, whether they are near or far from each other: in this way the concept of spatial structure is lost within the input image.

Convolutional Neural Networks were introduced during the 1970s, but their modern use for image recognition and classification was described only in the 1998 with the paper "Gradient-based learning applied to document recognition" by Y. Lecun et al. [8]. The main focus of such publication was to highlight the superiority of carefully designed learning machines that operate directly on pixel images over hand-designed heuristics. In particular the case study analyzed in such paper regarded character recognition, using Convolutional Networks instead of traditional fully-connected multi-layer networks. The main advantages that have led to this choice are:

- The large number of parameters that would be required using a fully-connected network, considering the typical sizes of an image (several hundred pixels), and the system's memory needed to meet these requirements;
- The fact that traditional fully-connected networks have no built-in invariance with respect to translations or local distortion of the inputs. This variance can not be perfectly compensated by the preprocessing of the input image. On the contrary, as we will see later, with convolutional networks invariance is guaranteed by the nature of the network itself;
- As previously mentioned the convolutional networks, unlike the traditional ones, take into account the two-dimensional topology of an image, extracting local features of the input before recognizing spatial or temporal objects.

The three architectural pillars that allow Convolutional Networks to ensure such advantages over fully-connected multi-layer networks are: *local receptive fields*, *shared weights* and *pooling* or *sub-sampling*. Such concepts will be analyzed and described in the following paragraphs.

Local Receptive Fields

In a convolutional network each layer can be molded as if all the neurons that compose it form a two-dimensional area. In particular in the first layer each value/neuron

corresponds to the pixel intensity used as input. Such representation is opposite to what has been seen before with fully-connected networks, where the inputs could be represented as a vertical line of neurons. To connect such inputs to a layer of hidden neurons not all pixels will be connected to each single new hidden neuron, which will be instead connected only to small region of the initial area. Such region takes the name of *local receptive field*.

For example, for an input image with dimensions 12 $\times$ 12 and local receptive field of 3 $\times$ 3, an hidden neuron could have a connection that looks like the one in Fig. 2.5.

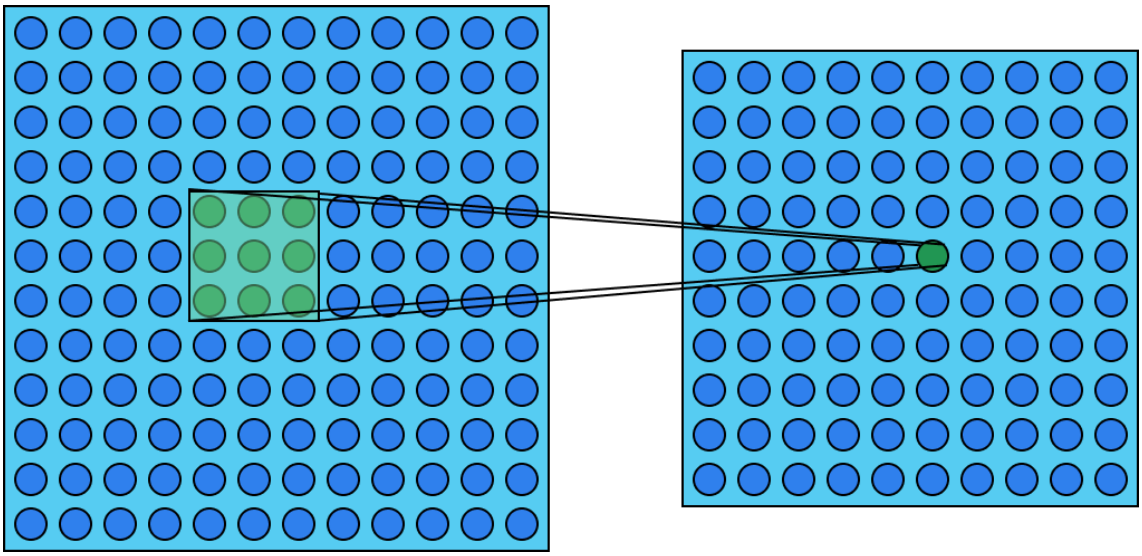

Figure 2.5. Local receptive field for an hidden neuron

A local receptive field can be seen as a window that slides over the entire input, forming a connection to the new hidden neuron at every step. Each connection "learns" both a weight and a bias. In this way to each local receptive field corresponds an hidden neuron in the new hidden layer. The parameter that indicates by how many pixels each window will slide is called *stride length*.

The number of neurons in the new layer depends on the number of possible moves that a window can perform on the input area. Such value can be computed by knowing the input size (W), the receptive field size (F), the amount of zero padding applied (Z) and the stride length (S). The formula for calculating the number of neurons in the new hidden layer is given by:

$$\frac{W - F + 2P}{S} + 1$$

Shared Weights

As mentioned, talking about the local receptive field, each new hidden neuron has a bias and a number of weights equal to the number of values present in the receptive field. These parameters are the same for each hidden neuron that belongs to the

same hidden layer. In this way for a local receptive field of $N \times N$ values there will be N^2 *shared weights* and a single shared bias that will be used to create the new hidden layer. The following equation represents such concept:

$$output[x][y] = \sigma(bias + \sum_{j=0}^{N} \sum_{j=0}^{N} input[x+i][y+j] * weight[i][j])$$

in which the value for the new hidden neuron (*output*) is given by the activation function $\sigma()$ applied to the sum of the shared bias for the product of the receptive field's weights with the input considered region. Such operations is also called *convolution*, hence the name of convolutional networks.

The meaning of such operation is that each neuron in the hidden layer is used to locate exactly the same feature, i.e., the same input pattern that will activate such neuron (for example a particular shape in the image). Such feature is searched along all the image, from this the use of shared weights. It is this peculiarity that makes the convolutional networks invariant with respect to translations of the input.

To detect more than a single feature, from the same input neurons must derive multiple hidden layers. Such hidden layers are called also *kernels* or *filters*. Each filter will have its shared bias and its shared weights, that will allow it to detect different image pattern.

One of the main advantages of shared weights is that they allow to reduce the number of parameters required by each layer. For each filter, indeed, there will be only a single bias and, considering a local receptive field with dimensions $N \times N$, only N^2 weights are needed.

Pooling

In addition to convolutional layers, CNNs contain also *pooling layers*. Such layers are often placed after a convolution layer and they are used to "condense" an hidden layer, simplifying it. In practice such kind of layer takes the output of a convolution operation and sampling it. For each filter deriving from a convolutional layer it summarizes small regions into a single hidden neuron. In this way the number of hidden neurons decreases, without losing any information. Conceptually when a pattern has been found, its exact position can be discarded, while the only important thing is its rough location relative to other features.

To summarize a small region different approaches can be taken, three of the most used pooling layers types follows:

- *Max Pooling Layer*: from a small considered region within the output of a convolutional layer, only the highest value is considered, discarding the other values present;
- *Average Pooling Layer*: from a small considered region within the output of a convolutional layer, the mean value of all the values present is calculated and taken;

- *L2 Pooling Layer*: from a small considered region within the output of a convolutional layer, the square root of the sum of the squares of the neurons' values is taken.

A Complete Network

After having described the basic blocks of a convolutional neural network, it is possible to give an example of the instantiation of a network of this type. In Fig. 2.6 is showed an example of a minimal CNN.

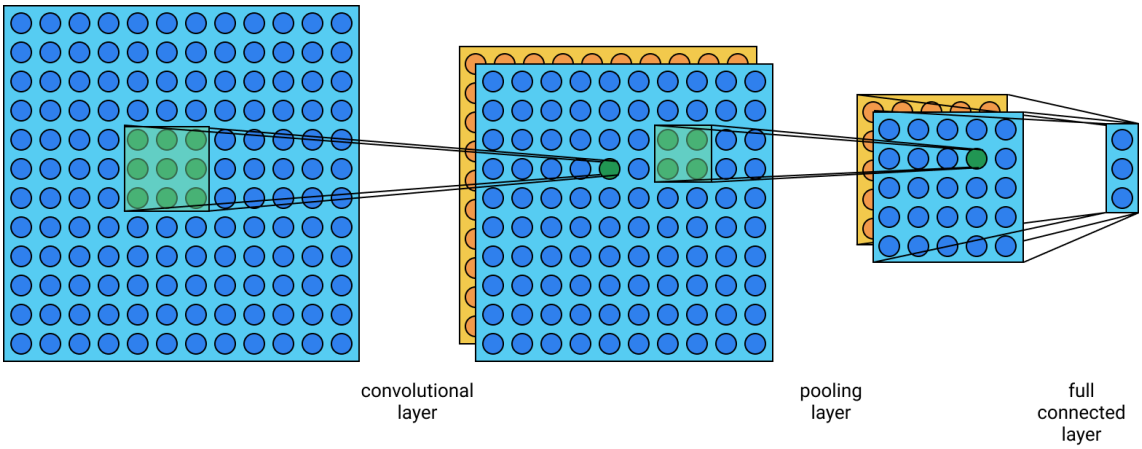

Figure 2.6. Example of a CNN

In the hypothesized example, the network starts with the input image of size 12×12, where the neurons represent the intensity of the input pixels. Then, there is a convolutional layer with a local receptive field of size 3×3. Without zero padding and with a stride length of 1, using the mathematical equation previously described it is possible to see how the output of this layer has size $2 \times 10 \times 10$, considering two filters. Such result goes through a pooling layer, that is applied to 2×2 regions, producing an output with $2 \times 2 \times 5$ hidden neurons. The final layer is a fully-connected layers (like those seen previously), which connects all the neurons from the previous layer to every neuron of the 3 output neurons.

Typical convolution networks have dozens of hidden layers, hence the name *deep learning*. Such definition has nothing to do with the precise number of layers used, it is rather used as opposed to the networks used until mid-2000s, which provided for the use of at most one or two hidden layers.

2.2 Darknet

With the incredible popularity achieved in these years by machine learning, many software libraries and frameworks were born. Among these we must certainly mention important machine learning frameworks like *TensorFlow* [10], *Caffe* [11] and *PyTorch* [12]. This work of thesis focuses on *Darknet* [13], an open source neural network framework which supports CPU and GPU computation.

Darknet was developed by Joseph Redmon and it came to the fore thanks to *YOLO* (*You Only Look Once*) [14][15]. As reported by its website, YOLO is a state-of-the-art, real-time object detection model, distinguished by the fact that it is both extremely fast and accurate. It is able to classify different regions of a single image passed as input. Differently from other image classification algorithms it does not apply its model multiple times on different regions to achieve multiple detections. Instead, it applies solely a single deep convolutional neural network, dividing the image into regions and predicting bounding boxes and probabilities for each region. The fact that it uses a single evaluation to detect multiple regions inside a single image, make it faster than other similar system, which runs many evaluations for the same purpose.

The reasons behind the decision to use Darknet for this work of thesis are multiple and they are summarized below:

- On the official site of Darknet many models already trained are available (including the already cited YOLO) such as SSD300, SSD500, and Tiny YOLO. For many of these models there are also different versions depending on the dataset used for the training. Being already trained models has been a positive factor for this work, that, as stated before, does not focus on the training process for convolutional neural networks, but only on the evaluation part;
- *Tiny YOLO*, the just mentioned model available, is one of the two network models used for this research. It is the smallest available variant, perfectly suitable to run fast also on devices with low performances like smartphones, or the Parallella board used for this thesis. Tiny YOLO is described and analyzed in the fourth chapter, talking about the networks used;
- Darknet has incredibly supporting community, which has developed many other pre-trained models, using the most diverse possible datasets. Among these there is also a custom network model trained using the *MNIST dataset* that was used for this thesis. The analysis of such model was done in the fourth chapter of this work;
- Darknet has only a single programming interface, written in C. While this could be a problem in many cases, for this work it is perfect. As it will be seen in the next chapter, indeed, the Parallella board can be programmed using the C language. This made it easier to adapt some methods already developed for Darknet on the used board.

2.2.1 Implementation Details

Darknet has represented the starting point for the practical solutions developed for this work of thesis. The porting of such framework was only partial, with only few original features really used. For example, Darknet's training methods have been ignored, as well as all those implementations of layers that were not required in the models used for this work of thesis. On the contrary, instead what has been ported is (for example) the library used for the pre-processing of the images and the parameters (weights, biases, etc.) produced. What was developed at the end is almost

a newly framework that just reads the model's parameters produced by Darknet, but then execute all the required layers *in its own way*. This approach is mainly due to the characteristics and limitations of the Parallella board, the following subsection shows the example of the convolutional layer and how the implementation on Darknet was impracticable for the porting developed.

2.2.2 Convolutional Layer

As it can be expected convolutional layers are the most important part of a convolutional neural networks. During the continuation of this work it will be seen how the convolution operation is the balance needle for the performances of an implemented model, that are indeed highly influenced by the methods and the optimizations used to execute the convolutional operations.

Earlier in this chapter the convolutional layers were described analyzing a method also known as direct convolution, i.e. an execution of the convolution operation made by performing the dot product between the input matrix and the sliding matrix of the weights. This is a really basic way to execute a convolution, not used and discouraged in the scientific literature, due to its poor performances caused by the irregular memory accesses involved in it. What is instead advised to do is to use one of the many optimized versions of such operation, that, performing different transformations on the input matrix, achieve to transform the dot product into a matrix multiplication. The benefit of this transformation is that the matrix multiplication is an operation widely studied in literature and which execution has been highly optimized, both on the CPUs and on the GPUs. As it will be described later, talking about the solution design, for this work of thesis the majority of the solutions are based on the use of the direct convolution method. The reasons behind this decision lie in the fact that the main negative point of most of these optimized versions of the convolution is the extremely high memory consumption that they need. Indeed, in order to transform the dot product into a matrix multiplication, huge support matrices are used. The main bottleneck of the Parallella board used for this research, as it will be highlighted many times in the continuation, is just the small amount of memory available. For this reason it was not possible to implement most of these optimizations, but it had to be used the direct convolution instead.

On the other hand, Darknet uses an optimized method for the convolutional layer, called *im2col*. Such method will be analyzed and described in the fourth chapter, when a solution with an optimized convolution is introduced and presented also for this work. As it will be seen, im2col arranges the data in a way that the memory accesses are regular for matrix multiplication, adding however in this way a lot of data redundancy. Such redundancy is counterbalanced by the performance benefit achieved with the matrix multiplication used instead of a dot product. With Darknet, in particular, matrix multiplication is executed using *GEMM (General Matrix to Matrix Multiplication)*, that is an optimized method for such operation. GEMM is one of the most used methods to perform multiplications between matrices, probably the most used [16]. It is part of the *Basic Linear Algebra Subprograms (BLAS)* [17], that is the *de facto* standard low-level routines for linear algebra libraries.

Chapter 3

Hardware Platform

Heterogeneous computing, i.e., the mix of different processing architectures on the same computing system, is one of the most popular market trends of the last years in the ICT domain. Its popularity is mainly due to the possibilities that it can offer, like the opportunity to create devices with high performance but low power consumption. Even many Internet of Things (IoT) devices have started to integrate acceleration units, in order to keep low their energy consumption.

The importance reached by Machine Learning (ML) and Deep Learning (DL) algorithms these years is overwhelming, as it has been highlighted during the introduction to this thesis, starting to influence the evolution of modern computer architectures. Among these, also heterogeneous devices have been targeted for ML/DL applications.

These concepts have been the driving force for this thesis and the idea of applying a DL algorithm on a low-power heterogeneous platform has represented the main challenge faced, accentuated by the decision to use a low-cost general purpose board for this purpose. In particular the target for this thesis has been the *Parallella* board, a cheap, credit-card-sized computer built on top of the many-core Epiphany coprocessor (Adapteva, 2011). This chapter introduces such hardware platform, describing and analyzing its main features.

3.1 Parallella

The Parallella board, as reported in the paper [18] was born in 2013 following a crowdfunding campaign on Kickstarted, and launched by Adapteva, an hardware company founded by Andreas Olofsson in 2008 with the aim of creating floating-point processors easy to program and with a high energy efficiency. The campaign, that reported as main goal the construction of a complete parallel computing eco-system around the Epiphany architecture, was a success and it was funded in less than 30 days. The first product announced was the Parallella board itself, introduced as a 99 USD open source parallel computing project, developed with the dual goal of democratizing access to high performance parallel computing and building a software eco-system around the Epiphany architecture.

3.1.1 Why The Parallella?

The decision to use the Parallella board for this work of thesis was depending on many factors. Among these the most important and perhaps most trivial is the fact that this platform was already available in the "Advanced Computing and Electromagnetics" laboratory at the Istituto Superiore Mario Boella, to be studied and analyzed for a project concerning heterogeneous computing architectures. For this reason it was decided to further deepen the use of this board, to understand up to what could be pushed into highly demanding areas of use such as machine learning and artificial intelligence.

Beyond that we must certainly mention the incredible attention recalled by this device and the incredible community that has been gradually creating and that is still active after five years from its launch. The fact of being a low-cost device with a low energy consumption has certainly been a fundamental factor in the great popularity achieved. We must also consider that it is a board with a parallel architecture that can bring a wide range of development possibilities, also thanks to Linux support. Another important and interesting opportunity offered by Parallella is to create a cluster of boards connected to each other, in order to obtain a high performance computing architecture with relatively low costs.

To make the comparison more precise, we taken into consideration (where possible) alternative acceleration platform with similar features in terms of solution cost, power consumption and manufacturing process.

Field Programmable Gate Array

Field programmable gate array (FPGAs) are reconfigurable devices, which offer large flexibility over more performing dedicated ASICs, still providing high performance in several application domains (e.g., signal processing, image manipulation, etc.). Flexibility is achieved by a full programmable fabric, where different resources (LUTs, flip-flops, DSPs, and integrated memory blocks) can be dynamically configured to perform any (complex) Boolean function. As such, internal fabric resources can be (even only partially) reorganized to provide the functionalities of a dedicated digital circuit. Application developers express the desired configuration (i.e., what is called the bitstream) using complex tool-chains that transform HDL (i.e., hardware description language) models (e.g., a Verilog or VHDL source code) into the final bitstream through several stages. Among the others, such transformation requires to generate the equivalent set of logic gates (i.e., logic synthesis), and to map it on the fabric resources (i.e., place & route). Unlike traditional high-level programming languages, HDL models require understanding the underlying hardware system, as well as how parallelization of the operations will take place in the synthesized circuit. To overcome such challenge, in recent years, tool-chains started to support high-level synthesis (HLS) capabilities. As such, C/C++ code can be automatically transformed into synthesisable HDL models.

Although the availability of FPGA devices targeting embedded systems (e.g., Xilinx Zynq, Xilinx Artix-7, Intel Max10) and HLS tool-chains, the design of modern complex applications as the case of a complete CNN, still requires larger effort

than of using standard compilers. Indeed, performance of synthesized circuit may be affected by the way high-level code is written, thus requiring long time to be optimized. Furthermore, complex applications may still demand for features only available on mid-range and high-end products, such as the case of hardened DSP blocks supporting IEEE-754 single precision arithmetic.

Graphics Processing Unit

Graphic processing units (GPUs) are nowadays the standard de-facto platform chosen for accelerating complex applications. The reason behind their success, out of the 2D-3D graphics rendering, is in their massive parallelism. Compared to a standard multi-core, a GPU is equipped with thousands of tiny processing elements. Each single processing element is less performing and complex (architecturally speaking) with regards to a traditional (both in-order and out-of-order) core; however, their aggregated computing power outclass any standard CPU architecture. Indeed GPU architectures has been, over the years, optimized for quickly crunching a huge number of floating point operations. Being initially devised as discrete components of a computer, GPUs became common in System-on-Chip designed for mobile and low-power applications.

Comparing the Parallella multi-core accelerator (i.e., the Epiphany-III) with a GPU requires for taking in to consideration, at least, a GPU based on the same manufacturing process. Also, the programming frameworks should be compared. Manufacturing technology evolved very quickly, and for the 2013 (i.e., the year of commercialization of the Epiphany-III) 28 nm technology was used to implement the Nvidia Tegra-4 chip. The most similar device family is the Nvidia Tegra-2, manufactured at 40 nm. This was a SoC equipped with a dual-core ARMv7 solution coupled with an embedded GPU. Such manufacturing processes are up to two generations ahead of that available for the Epiphany-III, and giving to Tegra-2 and Tegra-4 a big advantage in terms of power consumption (around 1 W). Despite their processing power, their architecture was initially restricted to graphics tasks; only with newer version of the platform was possible to exploit the more flexible general purpose programming framework, named *CUDA*.

Programming GPUs requires dedicated frameworks (i.e., compilers and developing libraries), spanning from OpenGL (for graphics) to OpenCL and CUDA for general purpose computations. Compared to the technology level provided by Epiphany-III, such frameworks were only available on discrete devices, leaving out all the embedded low cost systems.

3.1.2 Hardware Architecture

The Parallella board, that can be seen in Fig. 3.1, presents a Xilinx Zynq 7010/7020 System-on-Chip (SoC) with two 667 MHz ARM Cortex-A9 processor cores (that are fully supported by Linux), 1 GB of RAM, GBit-Ethernet, USB and HDMI interfaces and a MicroSD card-reader (that can be used for booting the system). Furthermore, one of the fundamental components of this system is the 16-core 32 GFLOPS

Epiphany E16G301 accelerator, which uses a custom Adapteva's e-Link interface, implemented inside the FPGA logic of the Zynq processor, to allow communication and data exchange between the ARM cores and the Epiphany ones.

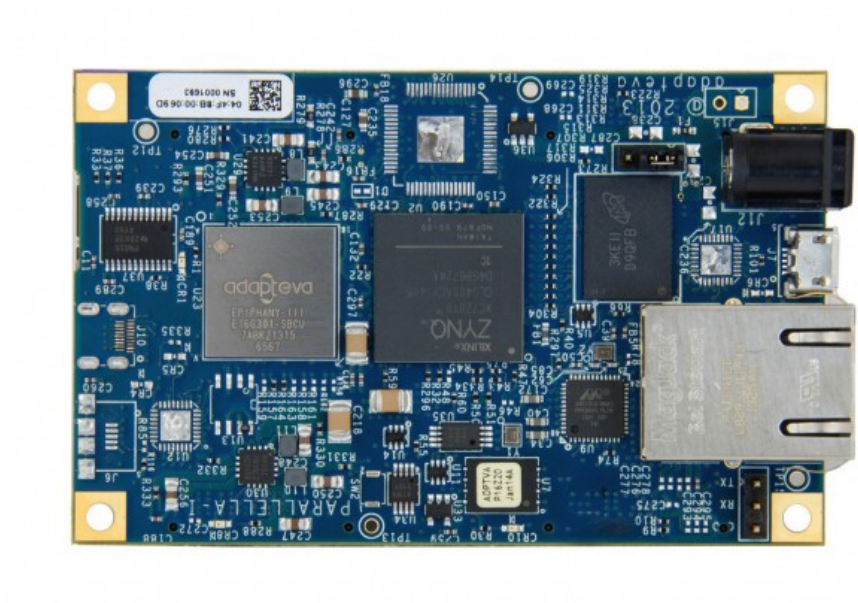

Figure 3.1. The Parallella board

The dual-core ARM Cortex A9 is the main processor (or host) of this system consuming up to 5 W, it has 32 kB L1 cache per core and 512 kB shared L2 cache, running Linux OS as operating system. In particular in the course of this work Ubuntu 15.04 was used as an operating system. The Epiphany chip is instead used as co-processor, running without any operating system with a flat, unprotected memory map. It consumes up to 2 W in addition. In the following subsections are described the Parallella's main components, in particular the ones that have represented a focal point for the thesis, starting from the Epiphany chip which can be considered the main focus for this work.

3.1.3 Epiphany Coprocessor

The Epiphany architecture consists of a two dimensional matrix of 16 mesh nodes (*eNode*). Each node is subdivided in turn in a 32-bit floating-point RISC CPU, also called *eCore*, a local scratchpad memory, a direct memory access (DMA) engine, two event timers and a network interface. The "eCore" nomenclature will be used several times in the course of this work to refer precisely to the cores present in the Epiphany coprocessor, opposing to the use of the word "core" that will instead often be used to refer to the two ARM cores on the Zynq. The eNodes are connected to the Network on Chip (NoC) through their network interface. Such NoC is called *eMesh* and it allows the eNodes to be interconnected to each other. A graphical representation of such architecture can be seen in Fig. 3.2, with with a greater detail regarding the architecture of an eCore in Fig. 3.3. These figures are largely inspired by those present in the presentation paper [18] of the Parallella board itself.

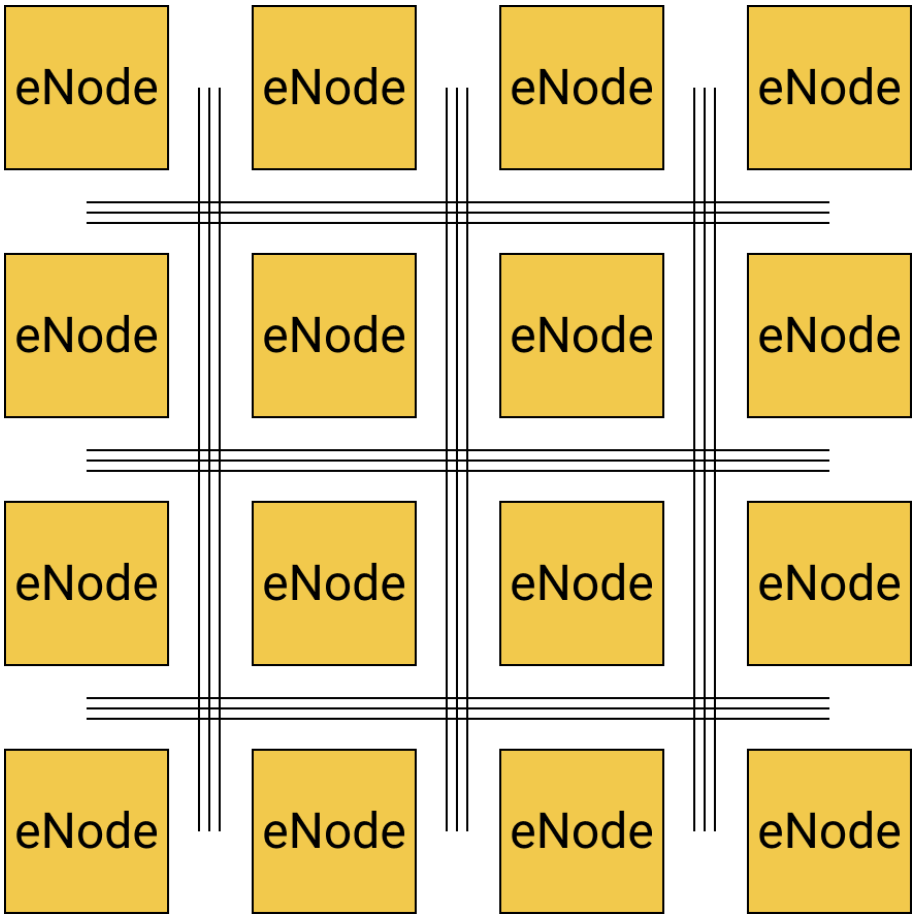

Figure 3.2. Representation of the two-dimensional array of eNodes

Each eCore is a 32-bit superscalar RISC processor, equipped with a 9-port 64-word register file, an integer arithmetic logic unit (ALU) and a floating-point unit (FPU). The instruction set architecture (ISA) is focused mainly on floating-point operations and C-programmability. All the eCores have also a program sequencer in order to support typical program flows (jumps, branches, etc.) and an interrupt controller.

Memory is not hierarchical in the Epiphany architecture, not including Level 1 (L1) or Level 2 (L2) caches. On the contrary it was opted for a solution that could maximize the amount of local storage and memory bandwidth. For this reason a banked scratchpad memory (SRAM or Static RAM) was used, with support to simultaneous instruction fetching, data fetching, and multicore communication. The Epiphany memory architecture is based on a flat distributed shared memory, with a partitioned global address space of 512 kB. Each eCore has 32 kB available subdivided in four banks of 8 kB, that can be used for code, stack and data. Furthermore, each eCore can access to the memory of the other eCores present on the Epiphany, using the eMesh. To each eCore two coordinates, represented by two tags, are assigned to distinguish it on the eMesh according to its position on the Epiphany's 2D matrix of eNodes. These coordinates make up the identifier itself of the eCore, which can be used in concatenation with the desired memory location to access the memory of another Epiphany core. The coordinates for each eCore are showed in

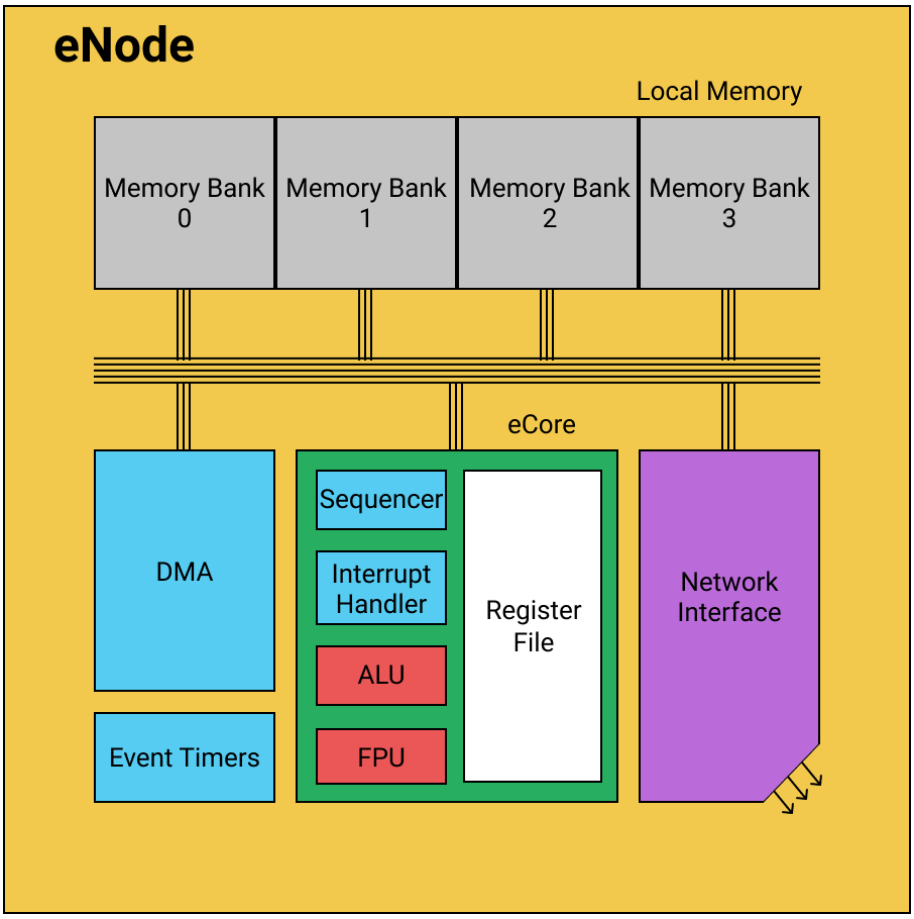

Figure 3.3. Graphic representation of an Epiphany node

Table 3.1.

		TAG			
		80	**90**	**A0**	**B0**
TAG	**80**	(0, 0)	(0, 1)	(0, 2)	(0, 3)
	84	(1, 0)	(1, 1)	(1, 2)	(1, 3)
	88	(2, 0)	(2, 1)	(2, 2)	(2, 3)
	89	(3, 0)	(3, 1)	(3, 2)	(3, 3)

Table 3.1. Coordinates to identify the Epiphany cores

The Epiphany eCores have also access to 32 MB of the host main memory. This memory is often referred as shared memory, external memory or DRAM. Such memory space is located between the addresses `0x8e000000` and `0x8fffffff`, and it is divided into two parts of 16 MB: the first part is used for storing C's newlib; the second one can be used by both the Epiphany and the ARM's cores to store data. This memory represent often the appointed communication channel between the host and the Epiphany cores. As it will be seen later, for the implementations done in this work such area was used for transferring the input images and the network's

parameters from the main program running on the ARM cores to the kernel executed by the eCores. ARM cores could also access directly to the internal memory of the Epiphany cores, but this approach is widely discouraged due to its poor performances. It is important to highlight that also read/write times between the eCores and the shared memory are much slower compared to the inter-communication times between the eCores.

All the memory sections described until now can be accessed from the Epiphany cores dereferencing a pointer to an address, by hardcoding it as follows:

```
int *C = (int*)0x8f000000;
float *D = (float*)0x8f800000;
```

Another possibility is to use *section labels* to indicate to the compiler where a variable should be stored. Both the two 16 MB areas of the shared memory have indeed a label, the first part is labelled as *shared_ dram*, while the second one as *heap_ dram*. Such these memory access method is reported below:

```
int A SECTION("shared_dram");
float B SECTION("heap_dram");
```

As said before, each eNode has two 32-bit event timers available. These timers can be used for monitoring events that happen inside the node, sampling real-time events. They have been widely used for this work, as it will be showed in the analysis of the achieved results. As described in the "Epiphany Architecture Reference" [19] the events that can be monitored are several and the one to be monitored can be configured using the *CONFIG* register. The monitorable events are reported below:

- *Clk*: The number of clock-cycles between two events is reported.
- *Idle*: The number of clock-cycle spent in IDLE.
- *IALU valid instructions*: The number of operations executed by the integer ALU unit. It includes the local loads and stores, that are also executed by the ALU.
- *FPU valid instructions*: The number of FPU instructions issued.
- *Dual issues instructions*: The number of cycles with two instructions issued simultaneously.
- *E1 stalls*: The number of pipeline stalls due to load/store register dependencies.
- *RA stalls*: The number of register dependency pipeline stalls.
- *Fetch contention stalls*: The number of stall cycles caused by memory-bank contention in the processor node.
- *Ext fetch stalls*: The number of instructions executed from external memory instead of local memory.

- *Ext data stalls*: The number of stalls clock-cycle due to a load instruction accessing external memory and stalling the pipeline.
- *Mesh traffic*: The number of wait or access events on the local cMesh network node.

3.1.4 Application Development

To take advantage of the Epiphany architecture two different approaches can be used:

1. Using the *Epiphany Software Development Kit (eSDK)*;
2. Using one of the higher-level frameworks available.

The eSDK allows a developer to manage the use of some specific low-level resources, like registers operations, interrupts handling, timers, mutexes, barriers and DMA functions. It supports the C programming language with mathematics functions, offering also partial support for C++. The available compiler is a modified version of the GNU/GCC compiler. The eSDK makes available two different libraries to be used: the *Epiphany Host Library (eHAL)* and the *Epiphany Utility Library (eLib)*. The former can be used to manage the Epiphany chip inside from the host side, while the latter provides hardware abstraction in the eCores. For this work of thesis it was decided to use the eSDK and not other higher-level frameworks, which however will be described below for completeness, because its compatibility with the C programming language has accelerated the porting of various functions of the Darknet framework.

As said, also higher-level frameworks are available to be used for programming the Epiphany architecture. In particular *COPRTHR* and *APL* are worthy of note. COPRTHR is used for many different heterogeneous platforms, like CPUs, GPUs and also the Parallella itself. It was developed by Brown Deer Technology and allows the support for OpenCL, STandarD Compute Layer (STDCL) and bare metal coprocessor threads. Array manipulation language (APL) was developed instead by Lab-Tools Ltd. It is also possible to use the Erlang functional programming language to manage the Epiphany chip.

Epiphany Host Library

Epiphany Host Library (eHAL) is a library included in the Epiphany Software Development Kit that makes possible to control and manage the Epiphany cores from the host (the two ARM cores). It allows to load programs, start and reset the eCores. It can also be used to access the internal memory of the Epiphany cores and the shared memory, both for reading and writing.

A typical workflow to allocate space in the shared memory and read and write from the same is shown in the following code snippet. The same read and write functions below can also be used to manage the internal memory of eCore.

```
e_mem_t mbuf;

e_alloc(&mbuf, 0x01000000, MEM_SIZE_16MB);

e_write(&mbuf, 0, 0, MEM_SIZE_1MB, &input, sizeof(input));

...

e_read(&mbuf, 0, 0, MEM_SIZE_1MB, &output, sizeof(output));
```

The subsequent lines of code instead show the typical workflow to follow to access the Epiphany architecture, receiving information on it, and open the connection to the desired eCore indicating them through their coordinates (in the example shown are used only the first eight eCore) . The program to be executed is subsequently loaded and launched, after its execution the chip is reset and the connection to it is closed.

```
e_platform_t platform;
e_epiphany_t dev;

e_init(NULL);
e_reset_system();
e_get_platform_info(&platform);

e_open(&dev, 0, 0, 2, 4);

e_reset_group(&dev);

e_load_group("some_task.elf", &dev, 0, 0, 2, 4, E_FALSE);

e_start_group(&dev);

...

e_close(&dev);

e_free(&mbuf);

e_finalize();
```

Epiphany Utility Library

Epiphany Utility Library (eLib) is another library provided by the eSDK, that can be used instead from inside the Epiphany cores to obtain access to the underlying hardware. It allows, among other things, to: read and write from the internal memory of all the eCores and from the shared memory; access the event timers; use the DMA engine; use mutex and barrier functions for synchronization purposes;

attach and detach interrupt-handlers. The read and write functions calls are the same as the ones previously seen in the eHal library, while examples of the other functionalities will be provided in the next chapters.

3.1.5 Problems Faced

During the work for this thesis some problems were faced developing for the Parallella board. Most of these problems are related to the use of the Epiphany architecture and to some bugs present in the compiler. These have slowed down the progress of the works; furthermore, most of them do not have a plausible explanation.

The debugging of the solutions developed was carried out by analyzing the partial results produced. This was done by simply printing out the intermediate matrices produced by both the convolutional neural network running on the Parallella board and the same network running on a standard personal computer using the Darknet framework. In this way it was possible to identify the differences between the developed version and the real framework, obviously taking the results produced by the latter as correct. In particular, after the execution of each layer, what was done was to print the matrices representing the various output channels for both, comparing these through the *diff* command on the terminal.

The next subsubsections describe some of the bugs found, trying to give an explanation when possible.

(activation_value = 0.1; output[i][j] *= activation_value) != (output[i][j] *= 0.1)

Although the title of this subsection may seem strange, it is probably the title that best describes the bug found in this case.

As seen in the previous chapter, during the convolution operation a phase is foreseen in which the result of this operation is passed through an activation function. During Darknet's porting, some anomalies were noted in this context. In particular, in the original framework when the so-called Leaky function is used as activation function (for example in Tiny Darknet): it multiplies by 0.1 the scalar value if this is less than zero. This value is saved in a variable, then just be multiplied. This practice is actually very common in software development, saving a frequently used value in a variable that can be reused, rather than using an immediate value. In the work carried out, we tried to stick to this practice, but found considerable problems: observing the results obtained from the implementation made with those produced by the original framework, we noticed strongly conflicting results. Empirically it was noted that by modifying the activation function, so as to multiply the partial output by an immediate value, rather than by a variable, the correct results were obtained. For this reason it was decided to proceed in that way.

Although it may seem a smallness, this problem has represented a strong slowdown of the work, being this bug difficult to identify. The cause of this bug could be found in the fact that the value 0.1, used in Tiny Darknet, does not have a precise floating point representation, but there is a loss of information to represent

it when it is saved in memory. On the contrary this would probably not happen if it were used as a value of 0.5, 0.25, 0.25, etc ... which can instead be effectively represented in floating point notation. Not saving this value in a variable, but using it as an immediate value does not cause problems for a different representation used probably.

Malloc Within The Shared Memory

During the development of the various solutions found, among other things, an attempt was made to dynamically allocate the matrices required for input and output directly within the shared memory.

This would have greatly facilitated the development, allowing to write a single function for each type of layer needed, allocating the necessary memory according to the size of the input and output matrices and the number of their channels. As will also be described in the next chapter talking about the actual implementations made, this was not possible due to a bug in the Epiphany architecture. This bug crashes the running application if a dynamic allocation is made in the shared memory, using the *malloc* function to which an address referring to that memory location is passed.

Chapter 4

Solution Design

In this chapter the practical work carried out within this thesis is described, analyzing in particular the approach used and the implementations made. In the next chapter, instead, the achieved results will be exposed and analyzed.

The chapter opens with the description of the general idea followed to achieve a working porting of Darknet framework on the Parallella board. It explains the decisions that have been made and the causes that lead to them. Then a description of the two convolutional neural networks used during the work is provided, exploring their features and the layers required. Finally a basic solution is provided and described. From such implementation derive the other solutions analyzed in the continuation of the chapter, with optimized and ad-hoc implementations made to enhance performance.

4.1 General Idea

4.1.1 Zynq as master, Epiphany as slave

Describing the board used and how to develop a program for it in the previous chapter, it has emerged how its programming model is very similar to the one used for GPU programming, as for NVIDIA's CUDA [20] for example. Such programming model, indeed, expects the use of *kernels*, that are pieces of code that are executed by parallel units, the Epiphany eCores in our case. These kernels, in particular their management and scheduling, are handled, as seen, by the Zynq architecture on the Parallella board.

Since one of the main objectives of this work is to test the performance of the Epiphany coprocessor, one of the first decisions made was to relegate the ARM cores only to the role of coordinators, i.e., they only have to schedule all the operations, to synchronize the Epiphany cores between them and to prepare and load all the necessary parameters in the shared memory. Most of these operations will be described in detail in the following subsections. With such division of the work, the eCores are in charge of all the operations closely related to the convolutional neural networks, i.e., they have to implement all the requested layers of the network. They

have, in a certain way, the role of *slaves* in a classical master/slave paradigm, while in this case the Zynq architecture plays the role of *master*.

Another implementation decision taken was to use not just one kernel, but multiple ones. Such choice was taken for two reasons:

- Using multiple kernels there is a better separation of the concerns. In this way, indeed, each network's layer or series of consecutive related layers could have its own kernel, making the code cleaner and more understandable;
- The second reason is more related to the architecture itself of the Epiphany cores. This, as explained in the previous chapter, have a little amount of internal memory and just an even smaller part dedicated to the program code and to the global and local variables. Such restriction makes impossible to write all the layers of the network in just one kernel. Therefore the division of the layers in multiple kernels.

As it would be better explained in the next chapter, talking about results and performance, the temporal overhead due to the context switching between one kernel to the other is negligible compared to the total execution time.

4.1.2 Focus On The Shared Memory

After having described the programming model for the Parallella board in the previous chapter and how the tasks were divided between the Epiphany and the ARM cores in the last subsection, it is now time to delve into some implementation details.

One of the most important problem faced during the work was how to actually manage data for an architecture like the one used, which has a poor amount of memory resources. The biggest limitation encountered was in particular the internal memory. As described previously, each Epiphany core has indeed only 32 kB of internal memory and a part of it is dedicated to program code, so only an even smaller amount can be actually used to store local variables. Talking about convolutional neural networks, one of their advantages is the lower number of parameters required compared to the classic neural networks. Despite this, there are still too many parameters and they take up too much space to be saved inside an eCore memory.In addition to them we must also have in memory, for each layer, the input and the output matrices. All these values, both the network parameters (weights, biases, etc.) and the input and output values, are stored as float on 4 B, so theoretically each Epiphany core could store just eight thousand of these values. In practice, for what was said just before, the number of parameters that can be memorized is even smaller, because a part of the internal memory is dedicated to the program code and to the global variables.

From these premises, for the final solution the attention was focused on the use of the shared memory as communication channel between the ARM and Epiphany cores and as main storage method. The idea was to subdivide the usable 16 MB of shared memory in three parts:

- A part dedicated to the inputs, where all the input values of a layer were saved;
- A part dedicated to the outputs, where each eCore could store the results of its computations;
- A part dedicated to the storage of the required parameters, where are memorized all of those parameters required by a convolutional layer (weights, biases, etc...).

Such division can be seen in Fig. 4.1.

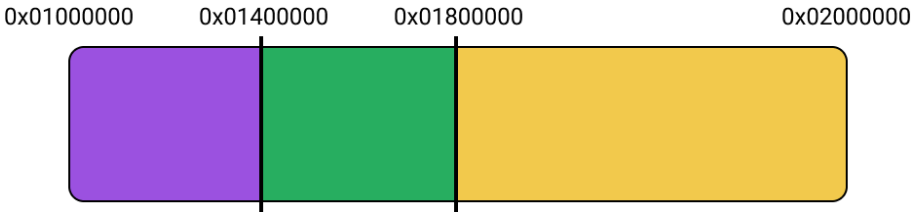

Figure 4.1. Conceptual division of the shared memory

At each new layer of the network, the parts dedicated to input and output exchange each other, because, as it can be imagined, the computational results of one layer become the input values of the following layer, as well as the the area previously dedicated to inputs becomes overwritable for the subsequent outputs.

The imagined flow of execution prevides different steps of execution to achieve the recognition of an input image:

1. At first the ARM cores load into the shared memory the parameters required by the convolutional neural network. If possible all the parameters are loaded into their dedicated area, in that way such operation could be executed just one time at the start of the program. If this is not possible, because of the size occupied by the parameters, such loading happens multiple times at each evaluation, between the execution of a kernel and the next one;
2. When an image is ready to be analyzed it is preprocessed by the ARM cores and then loaded into the input area in the shared memory. The preprocessing phase is equal to the one used in the Darknet library. During it, the initial image is divided into three channels of color (red, green and blue), then each channel is resized to the dimensions expected by the network. The data loaded into the memory are the matrices correspondent to each channel;
3. When both the network's parameters and the input image are loaded into the shared memory, the first kernel is loaded and executed by the eCores. For each layer implemented by the kernel in execution, the eCores load the corresponding required values using the DMA engine. Then, when the computation ends, they load the generated outputs in the dedicated area of the shared memory. For each layer, as explained, the memory locations for the inputs and the outputs exchange each other, while the parameters' area always remain in the same memory segment;

4. When a kernel ends the next one is loaded and launched by the ARM cores, until the last one is reached. At the end the ARM cores read the final output from the shared memory, printing it out.

Such intensive use of the shared memory represent a bottleneck for the solution, due to lower performance in reading and writing compared to those achievable using the internal memory of each individual eCore. This reason is one of the starting points for the research of the subsequent optimizations described in the continuation of this chapter.

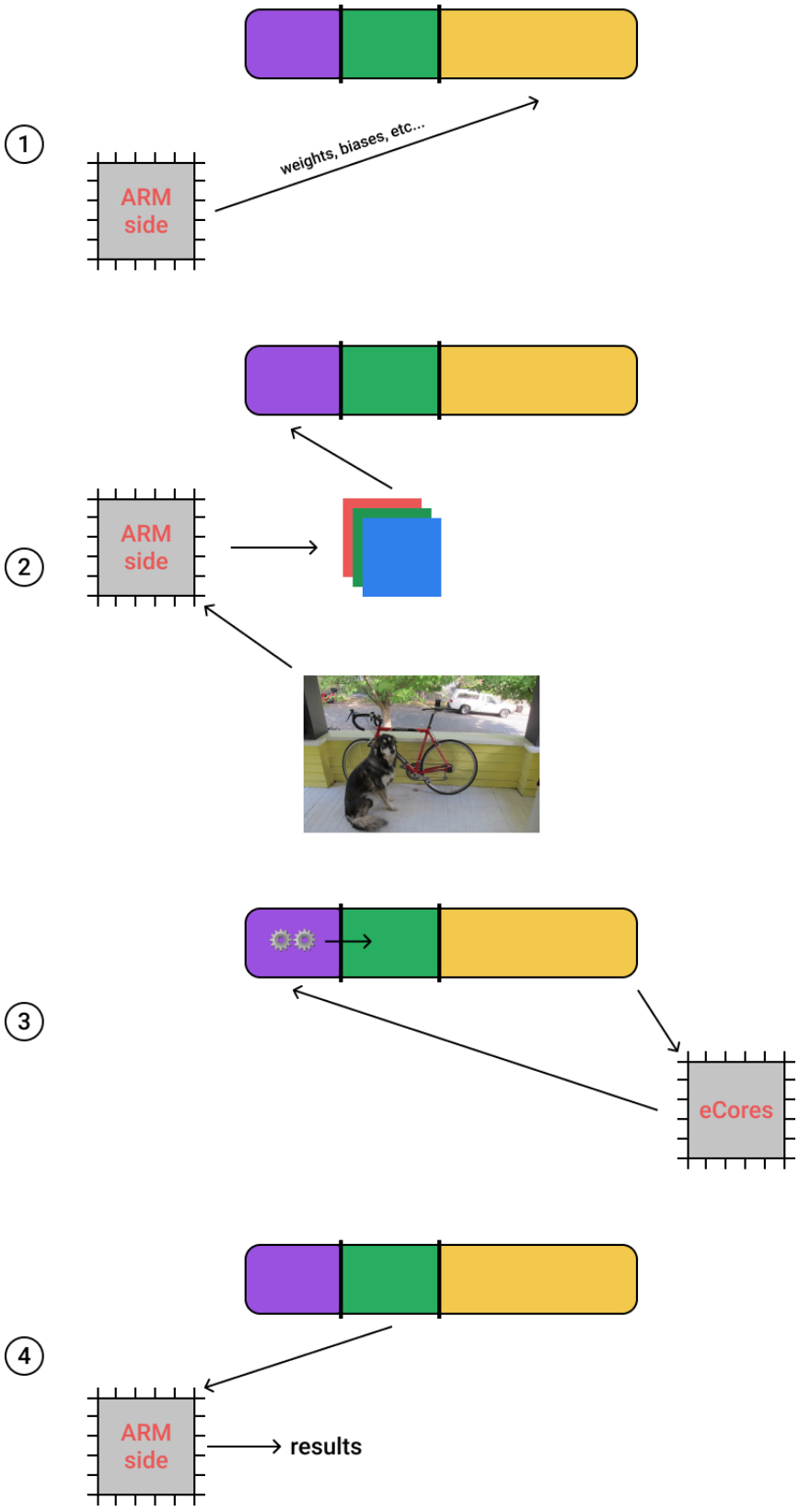

Figure 4.2. Representation of the workflow followed

4.1.3 Implemented Layers

As said in the first subsection, the actual layers of the convolutional networks are executed by the eCores. They are in charge of loading the inputs, the required parameters and of executing the needed computations. As it will be seen in the continuation, the two analyzed networks present almost the same mix of layers, that is: *convolutional layers*, *max pooling layers*, *average pooling layers*, *full connected layers* and *softmax layers*.

Despite these kinds of layers are used multiple times in each network, their implementation is not the same but change everytime, i.e. for each call to each layer there is a different implementation. The reason behind such decision can be found, again, in the little amount of memory present in each eCore. Such limitation has led to ad-hoc solutions. In fact, depending on the size of input matrices of the called layers, there is a different used approach: if the input matrix is too large to be stored in the internal memory, then all computations take place directly in the shared memory; otherwise it is copied through the DMA engine inside the eCore memory, carrying out all the necessary operations directly inside this memory.

Talking about the input and output matrices of each layer, it is also important to highlight the approach used for such objects. As mentioned before, on which memory the computations take place strictly depends on the size of the matrices under consideration. A little premise is important here to reiterate how inputs and outputs are structured in the case of a convolutional network. They are, as explained in the second chapter, composed of a certain number of channels, where to each channel corresponds a matrix with some specified size. Depending on the layer type, between inputs and outputs may change either the number of channels present or the size of the matrices used. Going to analyze concretely in detail the implementation approach used, we can identify mainly two cases, which are the same described above and which strictly depend on the size of the layer's matrices:

- On one hand there are those layers were input and output matrices have small sizes, so they can be handled by the eCores internal memory. In these cases the matrices are directly copied from the shared memory, using the DMA engine, into local variables. Then, after all the computations, they are stored again inside the shared memory, always using the DMA engine. With these layers, the applied policy lies in dividing the number of channels of the layers by the number of used cores. For example, considering a layer with 32 channels and a program which uses 8 of the available eCores, each core would have 4 channels to compute. Such method can be seen in Fig. 4.3.

- On the other hand in most cases the layers present input and output matrices of big sizes, that cannot be handled directly inside an eCore's internal memory. In such cases a different policy has been applied. In particular, it was decided to subdivide each channel's matrix in a set of submatrices with sizes congruous to the amount of memory present in each core. Each submatrix is managed by a different eCore, copying it as described above using the DMA engine. The number of submatrices was decided so that was a multiple of the number of the eCores used. In that way each Epiphany core would work on each channel

of the network, although only on small part of this. Let's consider for example a layer of 32 channels with matrices having dimensions sizes 224 $\times$ 224, using 16 eCores. In such case each core would work on every single channel, but computing only a submatrix with sizes 14 $\times$ 14. An example of this approach is observable in Fig. 4.4.

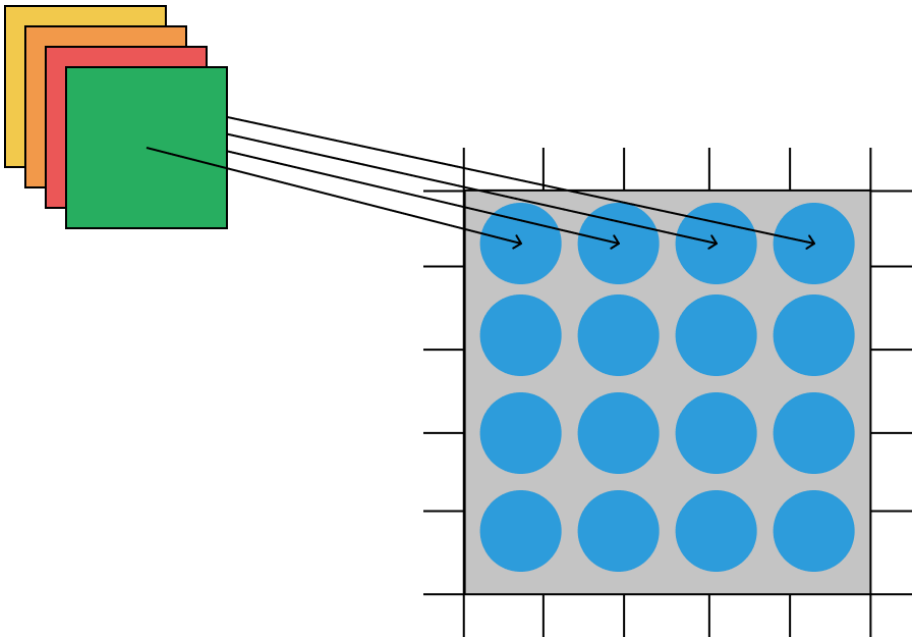

Figure 4.3. Allocation of input channels to eCores

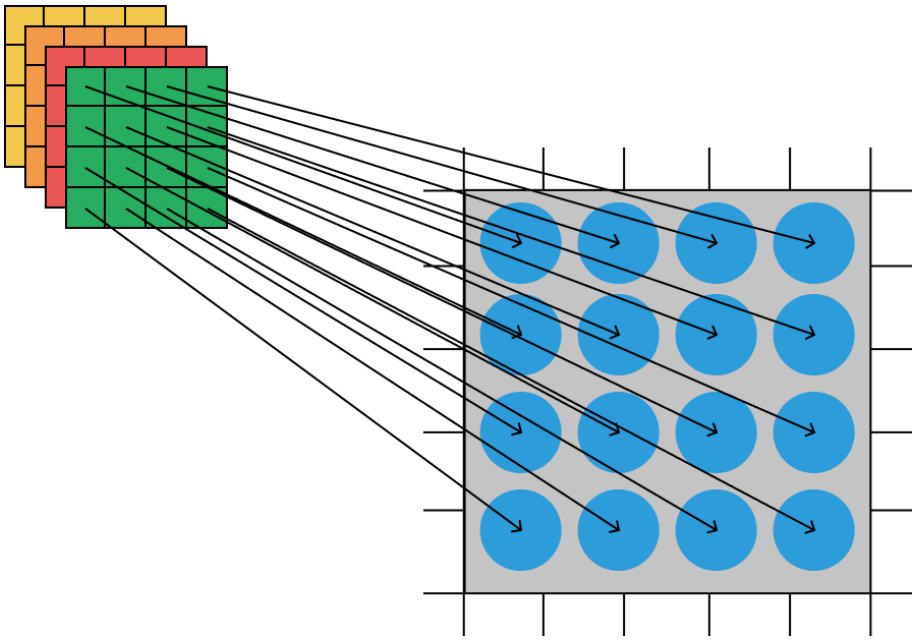

Figure 4.4. Allocation of input submatrices to eCores

The following subsubsections describe the general implementations of the layers used in the analyzed convolutional networks. In the continuation of the chapter additional implementation details will be provided for each solution found.

Convolutional Layer

Convolutional layers are perhaps the most important layers implemented. The reason of such statement lies in the fact that they are the real balance needle between the solutions found. They are indeed the layers with the most evident differences between each implementation and the ones that, as it will be seen, allow performance speedup.

The general idea followed in order to implement such layers was to use a direct convolution, i.e., the classic one described in the second chapter. As explained describing the Darknet library, faster algorithms require (most of the times) a huge amount of memory, so they were initially discarded. Nevertheless, as it will be seen in the continuation of this chapter, a solution of this kind was tested using the *Memory Efficient Convolution (MEC)* algorithm. Regarding the direct convolutional algorithm used, it was very simple. It consists indeed in four nested *for* loops: the most external one that iterates through the number of filters, the second one through the number of channels and the last two that loop between the two dimensions of the matrices.

A peculiar trait of the treated convolutional layers is that in the majority of the cases they required zero padding of dimension one. The addition of such further pixel is made when possible directly when the input matrix is copied in the eCore's internal memory, otherwise, if the computations happen directly within the shared memory, another function responsible for this task is called. In this last case such function simply moves the input matrices between the two locations of shared memory responsible for storing inputs and outputs, adding during the copy the required values for the zero padding.

Max Pooling Layer

Max pooling layers are present in all the analyzed networks, almost always appearing following the execution of one or more convolutional layers. Their implementations is a really simple one. It consists of three nested *for* loops, one that iterates through the number of channels and two that loop between the two dimensions of the matrices. In all the analyzed cases such layers appear with size and stride both equal to two, that means that each matrix is subdivided into submatrices of just four values. During the loops such submatrices are analyzed and the highest value between the four is the one that is saved into the output matrix.

Average Pooling Layer

The average pooling layers follow the same implementation of the max pooling ones, having almost the same characteristics. In that case, instead of choosing the highest

value among the four present in the analyzed submatrices, the average value between these values is calculated and it is saved in the output matrix.

Full Connected Layer

The full connected layers follow an implementation very similar to the convolutional layers. They consist in two nested *for* loops, one which iterates through the input neurons and one which loops through the number of output neurons instead.

Softmax Layer

The softmax layers, differently from the other implemented layers, are executed by the ARM cores. Such decision derives from the fact that they operates always at the end of the evaluation, working with a small number of neurons. For this reason it would have been too much overhead to parallelize this small amount of computations between the eCores.

Such layers simply implement the softmax operation, which takes as input a vector, that has as values the classification classes of the network, and gives as output the probability for a given class to be the predicted one. The pseudocode for such function is as follows:

```
  float *softmax_layer(float *input, int number_of_classes)
{
  int i;
  static float output[number_of_classes];
  float sum = 0;
  float largest = input[0];

  for (i = 0; i < number_of_classes; i++)
  {
    if (input[i] > largest)
      largest = input[i];
  }
  for (i = 0; i < number_of_classes; i++)
  {
    float e = exp(input[i] - largest);
    sum += e;
    output[i] = e;
  }
  for (i = 0; i < number_of_classes; i++)
  {
    output[i] /= sum;
  }
  return B;
}
```

As it can be seen, it simply consists of three *for* loops through the number of classes of the network: in the first the largest value is selected among the ones present in the input arrays; in the second cycle each element of the output array is computed with the equation $output[i] = e^{input[i]-largest}$; finally in the last loop each value of the output array is divided by the sum of all the elements computed in the second *for*.

4.1.4 Synchronization

Working on the Epiphany coprocessor without an intervening operating system and in cooperation with two ARM cores, synchronization was one of the main initial problems faced.

The general idea followed was to work with two level of synchronization, one within the eCores themselves and the other between the main program in execution on the ARM cores and the whole coprocessor. For these two kind of synchronization were used two different implementation approaches, using the different set of tools made available by the Parallella's libraries:

- To synchronize the eCores between themselves after the execution of each network layer, it was used a classic *barrier* mechanism. A barrier system is a typical synchronization method, widely used especially in the Linux environment. It allows in this case to prevent any core to go further with its execution until all the cores have finished their computation on that layer. Parallella's "e_lib.h" library provides by default the barrier objects *e_barrier_t* that can be used by the eCores for synchronization purpose. The following code snippet shows an example of a typical use of such method:

```
volatile e_barrier_t barriers[number_of_cores];
volatile e_barrier_t *tgt_bars[number_of_cores];

e_barrier_init(barriers, tgt_bars);

layer_1();

e_barrier(barriers, tgt_bars);

...

layer_n();

e_barrier(barriers, tgt_bars);

...
```

  As shown by this example, an array of barrier objects is declared by passing, as size, the number of running eCores that must be synchronized; then, such array is initialized using the system call *e_barrier_init*. After these preliminary steps, the algorithm used prevides the synchronization of all the cores in order

to wait everyone to finish the current layer before starting the next one. This need arises from the fact that, before executing each layer, it is necessary to guarantee the previous computations are finished.

- When all the eCores have finished their own execution, the ARM cores must be informed in order to start the next set of operations or to read the produced results. The approach used in this case is very simple and follow the guidelines and the advises provided by the Parallella community. Each eCore store a local variable used as flag and initialized to zero, before finishing their execution they turn the value of such variable to one. On the ARM side, the main program, after having launched the execution of the eCores, continuously loops, checking the memory locations (flags) of each eCore. When all the flags are placed equal to one it means that they have all finished their execution and the ARM's main program can continue its work. The following code snippet shows an example of use of this method, both on the Epiphany side and on the ARM side. On the eCore side there will be something like:

```
unsigned *flag;

flag = (unsigned *)0x00007000;

...

// Execution of all the required operations

...

*flag = 0x00000001;

return 0;
```

On the ARM side instead:

```
while (1)
{
  all_done = 0;
  for (i = 0; i < number_of_cores; i++)
  {
    e_read(&dev, i, j, 0x00007000, &done[i], sizeof(int));
    all_done += done[i];
  }
  if (all_done == number_of_cores)
  {
    break;
  }
}
```

4.1.5 Training The Network

As explained in the second chapter, talking about backpropagation, the training of the neural networks is not an argument of this thesis. The reason can be found in the poor performance provided by the Parallella board, as well as by other embedded systems of the same type. Such devices, indeed, are not suitable for this kind of operations, that are very exorbitant in terms of requested resources.

So, a very followed trend is to train the requested network (*learning phase*) on high performance systems and then to use the "learned parameters" on the embedded devices just for the *inference phase*. In such a way the less performing devices will be in charge of just the evaluation of the input, that is the less expensive in terms of both execution time and resources.

In our case the training of the networks was carried out on desktop computers, using the standard library provided by Darknet, then the results of such operation were used on the Parallella board. In this way two important results were achieved:

- The time needed for training was almost solved, using computer systems able to complete such operations in a relative short time;
- The parameters used by our implementation are completely compatible with parameters used by the Darknet library.

4.2 The Models Used

During this work of thesis two models of convolutional neural networks were implemented (using the Darknet library to train them): *Tiny Darknet* and a custom network trained on the *MNIST dataset*. In the following subsections there is an analysis of these two networks.

4.2.1 Tiny Darknet

Tiny Darknet [21] is a small convolutional neural network model proposed by the author of Darknet himself. Such model, as described by its presentation, was inspired by *SqueezeNet* [22], which is a deep neural network with a small architecture that requires a smaller amount of memory to be stored without sacrificing its accuracy.

Tiny Darknet was born with the same goal, indeed its model occupies only 4.0 MB maintaining however a good accuracy (58.7% of accuracy for the Top-1 predicted classes and 81.7% instead for the Top-5 predicted classes). The model provided with the Darknet library was trained with 1000 classes of prediction. Among all the proposed Darknet network, the choice for this work fell on Tiny Darknet because its small model was the one that fits better for the Parallella board. As reiterated over and over again in fact, the biggest bottleneck for such board is indeed the memory, for this reason a network's model which requires a smaller amount of memory to be saved seemed the right choice.

Tiny Darknet accepts, as input, images with sizes of 224 × 224 pixels, with three channels of color. This network consists of 22 layers, 16 of which are convolutional layers, 4 are max pooling layers, one is an average pooling layer and the last one is a softmax layer. Having such a high number of layers, the number of operations required for an evaluation is in turn very high and consequently also the time required. All the convolutional layers use a zero-padding of one pixel and a variant of the *ReLu* activation function, called *Leaky ReLu*, which can be described by the following equation:

$$output = \begin{cases} 0.1 * input & \text{if } input \leq 0 \\ input & \text{if } input > 0 \end{cases}$$

Table 4.1 shows in detail the Tiny Darknet's architecture.

Layers	**Filters**	**Filter Size**	**Stride**	**Input**	**Output**
conv	16	3 x 3	1	224 x 224 x 3	224 x 224 x 16
max		2 x 2	2	224 x 224 x 16	112 x 112 x 16
conv	32	3 x 3	1	112 x 112 x 16	112 x 112 x 32
max		2 x 2	2	112 x 112 x 32	56 x 56 x 32
conv	16	1 x 1	1	56 x 56 x 32	56 x 56 x 16
conv	128	3 x 3	1	56 x 56 x16	56 x 56 x 128
conv	16	1 x 1	1	56 x 56 x 128	56 x 56 x 16
conv	128	3 x 3	1	56 x 56 x 16	56 x 56 x 128
max		2 x 2	2	56 x 56 x 128	28 x 28 x 128
conv	32	1 x 1	1	28 x 28 x 128	28 x 28 x 32
conv	256	3 x 3	1	28 x 28 x 32	28 x 28 x 256
conv	32	1 x 1	1	28 x 28 x 256	28 x 28 x 32
conv	256	3 x 3	1	28 x 28 x 32	28 x 28 x 256
max		2 x 2	2	28 x 28 x 256	14 x 14 x 256
conv	64	1 x 1	1	14 x 14 x 256	14 x 14 x 64
conv	512	3 x 3	1	14 x 14 x 64	14 x 14 x 512
conv	64	1 x 1	1	14 x 14 x 512	14 x 14 x 64
conv	512	3 x 3	1	14 x 14 x 64	14 x 14 x 512
conv	128	1 x 1	1	14 x 14 x 512	14 x 14 x 128
conv	1000	1 x 1	1	14 x 14 x 128	14 x 14 x 1000
avg				14 x 14 x 1000	1000
softmax					1000

Table 4.1. Tiny Darknet

4.2.2 MNIST Custom Model

MNIST dataset [23] is one of the most well known machine learning database in literature. It is used in particular to recognize handwritten digits. It has a training set of 60000 examples and a test set of 10000 examples.

For this work of thesis a custom network was trained using the Darknet library along with this dataset. In particular it was used an open source repository on Github, *darknet_mnist* [24], to perform such training. The architecture of the network, that can be seen in detail in table 4.2, is very "slim": it presents indeed just 2 convolutional networks, 2 max pooling layers, 2 full connected layers and 1 softmax layers. Both the convolutional layers use ReLu as the activation function, while the two full connected layers use a *linear* activation function which has the form $input = output$. Moreover it uses small images for the evaluation, which can be easily stored in the eCores' internal memory. The negative aspect of this network is that, presenting two full connected layers, it needs a huge amount of parameters to work. For this reason, the loading of them into the shared memory can not take place in one shot, but must be repeated several times by alternating the execution of the various kernels used. Nevertheless, being a small network and therefore requiring a low number of operations, it allows to appreciate the performance of the Parallella board in a context much closer to a real case for it, i.e. a context in which just a feature of an image must be recognized.

Layers	**Filters**	**Filter Size**	**Stride**	**Input**	**Output**
conv	32	5 x 5	1	28 x 28 x 3	28 x 28 x 32
max		2 x 2	2	28 x 28 x 32	14 x 14 x 32
conv	64	5 x 5	1	14 x 14 x32	14 x 14 x 64
max		2 x 2	2	14 x 14 x 64	7 x 7 x 64
full connected				7 x 7 x 64	1024
full connected				1024	10
softmax					10

Table 4.2. Network trained on the MNIST dataset

4.3 Basic Solution

The first attempts to implement working convolutional neural networks on the Parallella board just follow a basic and simple philosophy: *just make them working*. They do not have any optimizations, they just follow the general ideas previously introduced in this thesis. This section describes such first basic and most importantly working solution implemented.

This solution implements both the two networks previously analyzed. It uses all the 16 eCores available on the board, following the split of the tasks and the shared memory usage described above. The approach to the convolution operations is really basic, using a simple direct convolution algorithm where all the computations happen in 4 nested *for* loops along the channels, the number of filters and the two dimensions of the input and output matrices. All the problems related to the memory bottleneck can be seen here, indeed in most of the layers the operations are made completely in the shared memory with a tremendous overhead in terms of execution time.

In such solution the main program running on the ARM side loads all the parameters in one time in the shared memory, then it sets the coprocessor to use all the 16 eCores. On the eCores all the computations happen in the shared memory: each convolutional layer has two local variables for the input and the output matrices that reference two memory locations in the external memory, and all the operations use such variables. Only the convolutional parameters are stored in the internal memory.

As seen, all the layers used by the two networks uses a zero-padding of one pixel. While the first operation of zero-padding is executed directly by the ARM cores (loading the input image), all the subsequent ones are executed by the eCores. By never copying the input and output matrices to internal memory, an additional function has been created to make this operation happening. Such function copies the input matrices to the second memory location dedicated to the matrices, but during the copy it adds the additional zero pixel copying the input value in a memory reference shifted by one. To help to understand such mechanism the following code snippet is provided, where is taken as example an input matrix with dimensions 112 x 112 x 16:

```
void zero_padding()
{
  float (*input)[112][112] = (void *) 0x8f000000;
  float (*output)[114][114] = (void *) 0x8F400000;

  int i, j, k;

  ...

  for (k = 0; k < 16; k++)
  {
    for (i = 0; i < 112; i++)
    {
      for (j = 0; j < 112; j++)
      {
        output[k][i + 1][j + 1] = input[k][i][j];
      }
    }
  }
}
```

The function simply makes a copy of the input matrix in a bigger output matrix, shifting the indices by one. It can also be seen how the matrices are referenced in the shared memory. In order to be sure that the location of the output matrix contains only zeros for a perfect zero-padding, a *memset* operation is performed on that memory location before calling the method just described.

4.4 Extended Memory Solution

The basic solution just described follows perfectly the philosophy with which it was created: it *just works*. It showed how even a low power board like the Parallella could run a deep convolutional neural network for image recognition. However at this point the question to wonder become: how can it become usable, i.e., how can performance improve in order to be used in real application contexts?

As it will be seen in the next chapter, the performance achieved by the Parallella using this solution are not close to a possible use in real contexts, nor using a network like Tiny Darknet nor using a more suitable smaller network like the custom one trained using the MNIST dataset. The biggest bottleneck was the memory and in particular how to store the network's parameters and the input and output matrices in order to perform all the necessary computations in a performing way. Thus, any improvement should only start from solving this problem.

With the basic solution previously described, all the operations on the input and output matrices take place in the shared memory. This involves enormous latencies, due to the times of reading and writing from the eCores to the external memory and vice versa (exposed in the third chapter analyzing the board) even using the DMA engine. Obviously, the best and ideal solution would to execute all the operations in the internal memory of each eCore. Unfortunately this is not possible given the small amount of available internal memory. With that in mind, the only possible solution was to try to "*extend*" such internal memory.

The solution imagined and then implemented goes precisely in this direction and starts from a very simple question: is it necessary to use all the available Epiphany cores? The initial answer should be trivially yes, more cores mean almost always more power and so better performance. A more careful analysis can, however, lead to slightly different answers. In particular, it is important to keep in mind that the main problem with the basic solution previously analyzed was how to store the huge amount of data derived from the use of a convolutional neural network in order to speedup performance. It was not a problem on how to increase the number of working parallel units. Resuming what was said about the split of work among the eCores, one can understand how the real question was not how to further subdivide the input and output matrices in order to reduce the number of computations to perform, but it was mostly how to reach a subdivision that allows the execution of the elaborations in the internal memory. From this assumption it is possible to resume the initial question, i.e. whether it is necessary or not to use all the eCores present on the Epiphany coprocessor, giving a more complete answer that could be: no, it is not mandatory to use all the available eCores, if this is translatable into some increase in general performance.

Summarizing what has been said so far and keeping in mind the possibilities of communication between the eCores themselves, as offered by the Epiphany coprocessor, the hypothesis formulated was: using only half of the eCores available for the real computations and the remaining other half only for their "storage capabilities", we could greatly counter the bottleneck due to the lack of memory. Explaining better such concept, it could be more performing to use part of the available Epiphany cores only for their internal memory, using this as an *extension* for the internal

memory of the other eCores that actually perform the operations. In such a way the cores in action would have a larger memory area to work with, with the possibility to perform their operations directly in such space. Obviously, as it can be seen in 4.3, the read/write performance between an eCore and another one are not the same as the one achieved by using directly the internal memory, but they are much better compared to the use of the shared memory.

From	To	Method	Write Speed (Mb/s)	Write Speed (Clock cycles)
eCore	Internal Memory	memcpy	504.09	9299
Internal Memory	eCore	memcpy	115.65	40531
eCore	External Memory	memcpy	142.99	32782
External Memory	eCore	memcpy	4.19	1119132
eCore	Internal Memory	DMA	1949.88	2404
Internal Memory	eCore	DMA	480.82	9749
eCore	External Memory	DMA	493.21	9504
External Memory	eCore	DMA	154.52	30336

Table 4.3. Epiphany Architecture - Memory Performance

4.4.1 Implementation

Obviously, to test such hypothesis, a new approach to the implementation of the networks has been taken. Although the part concerning the main program executed by the ARM side has remained almost unchanged, the implementation regarding the kernels executed by the Epiphany cores has been revolutionized.

The basic idea was to create two groups of eCores:

- The first group is the one in charge of performing the operations, i.e., the *active group*. Their tasks are the same as the ones used in the basic solution described in the previous subsections. What really changes is how they perform such tasks, like for example how the convolutional layer is implemented having more memory space available;
- The second group, instead, could be called *silent or sleeping group*. The eCores of this group are not involved in the execution of the operations required by the convolutional neural networks, they do not even have any kernels to execute. They are only used for their memory, in such a way that the active cores can access to their memory segment, storing local variables and executing the required computations directly in such location.

The planned division consist of assigning the same number of eCores to each group. In this way to each active eCore correspond a silent one (with a 1 to 1 mapping).

With such solution, it was as if every Epiphany core in action had doubled its amount of internal memory storage, from 32 kB to 64 kB. In reality, as explained above regarding the use of internal memory by the cores, only a part of the internal memory proper to each core can therefore be used, so the actual amount of available memory is about 48 kB. It is good to note that the silent eCores, not having any kernel to execute, have all their 32 kB available, because they do not have any part of their memory dedicate to store the program code, the global variables, etc.

The implemented solution prevides that to each active eCore is assigned a silent one. In practice it was decided to use as active group the first two rows of cores of the coprocessor and the last two as silent group, having 8 active eCores and 8 silent. In particular, to each core of a given row it was assigned the correspondent core from the same position but belonging to the other group. For instance, to the first core of the second row of the active Epiphany cores was assigned the first core of the second row of the silent group. Absolute addresses are used to implement such assignment. At the start of each kernel there is a *switch* construct on the indices of the cores that is executing it. The following piece of code shows such implementation in practice.

```
void *A, *B,;

switch (4 * e_group_config.core_row + e_group_config.core_col)
{
  case 0:
    A = (void *)0x88800000;
    B = (void *)0x88804000;
    break;

  case 1:
    A = (void *)0x88900000;
    B = (void *)0x88904000;
    break;

  case 2:
    A = (void *)0x88A00000;
    B = (void *)0x88A04000;
    break;

  case 3:
    A = (void *)0x88B00000;
    B = (void *)0x88B04000;
    break;

  case 4:
    A = (void *)0x8C800000;
    B = (void *)0x8C804000;
    break;

  case 5:
    A = (void *)0x8C900000;
```

```
      B = (void *)0x8C904000;
      break;

    case 6:
      A = (void *)0x8CA00000;
      B = (void *)0x8CA04000;
      break;

    case 7:
      A = (void *)0x8CB00000;
      B = (void *)0x8CB04000;
      break;

    default:
      break;
  }
```

As it can be seen, in the *switch* is computed the absolute index of the eCore in execution and depending on it A and B takes the absolute address of the internal memory of the correspondent assigned silent Epiphany core. The memory segment of such silent eCore is divided in two equal parts of 16 kB, i.e. A and B in the piece of code below that are used to reference the input or the output matrix depending on the case.

Convolution Layer

Having more performing memory available has meant changing the way in which layer, such as the convolutional one could be executed. The decision made was to continue to follow the split in submatrices previously described, with the big difference that these could be now stored in the extended memory.

The implementation made for this second solution prevides the use, as before, of a direct convolutional algorithm, still not optimized. Differently from before a new step was added, which consists in the copy, using the DMA engine, of the input matrix or submatrix from the shared memory to the extended memory location. In this way all the computations can happen directly inside the silent eCore, saving the results also in it. Such results are, at the end, transferred to the shared memory using again the DMA engine. In this way the bottleneck of executing all the operations in the shared memory can be avoid, having, as it will be seen, a consistent speedup of the performance.

Another big advantage reached is that such solution can avoid using a further function to implement the zero padding almost always required for a convolutional. As seen, with the basic solution, a new function was created in order to add the padding. Such function simply copied the input to a new memory location in the shared memory adding the required zeroes. Although functional, this method was highly ineffective, because it required many operations to happen in the shared memory and also the use of the *memset* function directly in such location of memory.

With this second solution can be avoided. Being that the input matrix or submatrix is directly stored inside the extended memory locations, the padding can be added during such operation, using the same method as before, i.e., shifting the indices from the input to the new location by one. The following code snippet shows such operation, in which each row of the input matrix is copied to the extended memory from the shared memory, using the DMA engine:

```
for (i = 0; i < 56; i++)
{
  e_dma_copy(&input[i + 1][1], &s_input[v][off_i + i][off_j], 56
     * sizeof(float));
}
```

Max Pooling Layer

The new implementation for the max pooling layer is very similar to the one made before for the basic solution. In the case of the second solution the main difference resides in the fact that, as done for the convolutional layer, a new step is required. In such step the input matrix or submatrix is copied from the shared memory to the extended memory, in order to make all the computations happen in such memory location. At the end, the new output matrix or submatrix is copied directly in the shared memory again.

4.5 Optimized Convolution Solution

Having extended the internal memory of an Epiphany core, at the price of having to use only half of the available cores, the next obvious question posed for this work of thesis was: *how can we go further in order to increase performance?*

A possible approach could be to continue to work on the memory, trying to further reduce the bottleneck deriving from it. It is true, indeed, that with silent eCores the available high performing memory was increased, but it is also true that despite this an eCore could not still always work with an entire input matrix, but it has to use in most cases a submatrix that could fit in memory (internal or external).

Another possible approach, the one with which we decided to continue, prevides to optimize directly the convolutional layer itself, that is the one used more often and also the one *heavier* in terms of computations. Going in this direction it was decided to try change the previous implementations in order to bring them closer to the approach used in the Darknet library. As explained in the second chapter, Darknet does not use a direct convolutional algorithm to implement the convolutional layer, it uses a more optimized version known as *im2col* method.

im2col [25] is a very well known method in the scientific literature, used to speedup the convolutional operations. It allows to achieve better performance transforming the dot product between the filter window and the local receptive field to a matrix multiplication. This is done by expanding all the possible moving windows,

i.e., all the regions sampled by the sliding movement of the filter windows, in memory. To expand these windows two new "supporting matrices" are created, one for the input image and the other for the weights. The latter is simply a row matrix made by stretching all the weights on a single row. The first, instead, also called *lowered matrix*, is a huge matrix having as width the square of the size of a single filter multiplied by the number of channels of the input image and the height that can be calculated as follows:

$$X = ((i_h \ + \ 2 \ * \ p \ - \ f_s) \ / \ s) \ + \ 1$$

$$Y = ((i_w \ + \ 2 \ * \ p \ - \ f_s) \ / \ s) \ + \ 1$$

$$height = X * Y$$

where i_h and i_w are the sizes of the input image, p is the amount of zero padding to be used in the convolution and s is the stride to use. The matrix multiplication between these two matrices gives the result of the convolutional operation, that can be reshaped back to the classical image matrix by applying the reverse operation *col2im*. The negative aspect of this method derives from the sizes of the support matrix used for the input image. Such matrix, as it can be imagined, has huge sizes that cause an high memory consumption. This reason, despite the better performance achievable, made such method impossible to be applied on the Parallella board, due to the small amount of memory available.

A method very similar to the one just described is the *memory-efficient convolutional*, better known as *MEC*. Such method, that will be described in the next subsection, allows to improve the performance of the convolutional layer, but using a support matrix with small sizes that reduces significantly the memory consumption compared to the im2col algorithm. MEC was used for this work to implement the custom network trained with the MNIST dataset and described below. Being that such network uses already by itself smaller matrices compared to Tiny Darknet, it would be ideal for testing the MEC algorithm.

4.5.1 Memory-efficient Convolution (MEC)

Memory-efficient convolutional or MEC [26] is a convolutional algorithm with the aim of improving performance, reducing however the memory consumption compared to algorithms like the im2col, previously described. Its principle is the same as the im2col's one, i.e., to transform the dot product between the filter window and the local receptive field to a matrix multiplication. The difference lies in how the support matrix is created, indeed MEC uses a different way for lowering the input matrix, which allows to create a much more compact matrix, reducing the memory overhead. The lower impact on memory was the reason behind the decision to use such algorithm.

The lowered matrix in the MEC algorithm is created starting from dividing the input matrix in submatrices with dimensions $i_h \ \times \ f_s$, where i_h is the height of the input matrix and f_s is the size of the convolutional filter. Each submatrix is created from the start of the input matrix to the end of it, sliding it by s, which is the size of the convolutional stride. Each submatrix formed in this way is then

copied into one row of the lowered matrix. Once this operation is completed, MEC further subdivides the newly created lowered matrix in other partitions, of sizes $o_w \times f_s \times f_s$, where o_w is the output width and d_s is the size of the filter. Each of these submatrices, created from the start of the lowered matrix shifting by $s \times f_s$, is then multiplied with the support matrix of the weights, created in the same already seen in the im2col algorithm.

4.5.2 Implementation

The MEC algorithm was implemented only for the custom network trained with the MNIST dataset. Such decision derives from the sizes of the lowered matrix, which, even if smaller than the one that would be built with the im2col algorithm, occupies a big amount of memory. For this reason it has been preferred to implement it only for a network where the input matrices are considerably smaller than those used in Tiny Darknet. Compared to previous implementations, what is really changed is the addition of a new method for the creation of the lowered matrix and completely, as it can be imagined, new method for the convolutional layer. The starting point for this new solution was the previous solution based on the extended memory. Also in that case, like in the two before, each eCore does not work with the entire input matrix (lowered matrix in this case), but only on a submatrix of it.

mec_ shaped() is the new method created for the creation of the lowered matrix required by the MEC algorithm. It uses four nested *for* loops that cycle on the number of filters, the number of channels and on the dimensions of the input matrix. What this method does is simply to copy from one location of the shared memory to the other assigned for storage of input and output matrices. Such copy is done in order to directly create the lowered matrix in the shared memory.

The new convolutional function, instead, starts from the lowered matrix in the shared memory and copies into the extended memory one of the partition that must multiplied by the support matrix of the weights. This latter matrix is created directly in the internal memory when copying the weights from the shared memory. The algorithm used for the matrix multiplication is simply based on three *for* loops and can be seen in the following code snippet:

```
float input[V][I], weights[I][J];

...

for (v = 0; v < V; v++)
  {
    for (j = 0; j < J; j++)
    {
      output[v][j] = 0;

      for (i = 0; i < I; i++)
      {
        output[v][j] += input[v][i] * weights[i][j];
      }
```

```
    }
  }
```

The results of such multiplication are then saved in the shared memory.

4.6 Summary

In this chapter all the designed solution for this work of thesis were described and analyzed. The chapter started from the general ideas used during the whole work, showing the convolutional neural networks that were decided to use on the Parallella board and describing how these were actually implemented.

Although the initial purpose was to try to implement the entire Darknet library, at the end it was decided to use the original Darknet library only for the training of the networks and for parameters obtained in this way. The main reason behind such decision, stated many times during the chapters, was substantially the bottleneck caused by the poor amount of memory present on the board. It is just this bottleneck to have generated most of the decisions around this work, searching for new ways to get around it, improving consequently the performance.

In conclusion, three different solutions were found and implemented, in order to make the Parallella board capable of running convolutional neural networks:

- A first basic solution which exploits the use of the shared memory, both as communication channel, between the ARM and the Epiphany cores, and as memory location where all the computations happen.
- A more optimized solution, where it was decided to avoid the use of half of the Epiphany cores present on the coprocessor in order to make them silent cores. These cores are used only for their memory space, that is exploited by the active cores to store the required variables (input and output matrix/submatrix mainly) and to perform the needed computations.
- A third solution, which starts from the results and the new methods implemented in the second solution to achieve an optimized version of the convolutional layer based on the use of the MEC algorithm.

In the next chapter the results achieved using these three solutions will be analyzed.

Chapter 5

Experimental Evaluation

This chapter offers a view of the results achieved with the solutions and described in the previous chapter. The analysis is divided into two parts: a first one focused on the obtained performance, i.e., on the times required to run the models object of this work, using the implementations made ad-hoc for the Parallella board; a second one, where the focus is instead on the power energy consumption of the board during the execution of such convolutional neural networks.

These two aspects represent the focal points of this thesis, as highlighted also in the introduction to this work. They are indeed an indication of how it is possible to use the Parallella board in a real application context, both in terms of execution times and power energy consumed to perform the recognition of an image. Through the results achieved it was possible to draw the conclusions on the hypothesis at the basis of this thesis, that are exposed in the next and conclusive chapter.

General Methodology

In the course of this chapter, referring to both the measurement of performances and power energy consumption, the descriptions of the specific methodologies used for these two measurements will be provided. It will be explained how the presented data were calculated, in particular the method and the tools used to obtain them.

During the two types of measurement, however, some similarities can be found in the approaches used. In particular, it was decided to carry out at least 10x measurements for each type of result, reporting, as shown below, for each result, the average value and its standard deviation. Furthermore, the results presented are initially presented in tabular form, referring to the data obtained for each network layer in each solution; then, there is a summary histogram for each of the solution found that indicates the execution time for all the kernels executed by the Epiphany architecture. Finally, for each model used there is a conclusive histogram which compares the total times of execution.

Another important point to underline is how the measurements were divided for each layer. As mentioned in the previous chapter, above all for the convolution operations, each different solution follows its own approach: the extended memory solution has a unique function for each type of layer, the basic solution requires an

additional function to perform zero padding for convolution, as well as the solution based on the MEC algorithm requires a method called *MEC Shape* to "reorder" the input matrix. In the measurements made, all the auxiliary functions were considered as being part of the same layer; for instance for the basic solution a single measurement was performed that considers both the real convolution function and the zero padding function. In the same way we proceeded in the solution based on the MEC algorithm.

As for the used name, the following legend was used:

- The first solution found is called *basic solution*;
- The second solution, based on the use of only half of the available cores (using the other ones only for their memory), is called *extended memory solution*;
- The third one, based on the use of the MEC algorithm, is simply called *MEC solution*.

5.1 Performance Analysis

The analysis of the achieved performance provides an answer to the question: how fast are the implementations found? Although this may seem a trivial question, in reality it concerns one of the fundamental aspects for a possible realistic use of the work carried out, i.e. the usability of the solutions found.

In a real context, especially in a possible industrial application, the time required to perform an operation is fundamental. Although there were no specific constraints on the required computation times, the direction followed during all the work was to try to implement the most efficient solutions possible, where the efficiency in this case was seen in terms of execution times. It can be said that the leitmotif "*it is enough that it works*" was followed only at the beginning of this thesis, once the criticalities arising from the limits of the board used were identified (limits widely quoted several times in the previous chapters, like the amount of available memory in primis) and above all, once these difficulties have been overcome, the focus has been placed on some possible optimizations aimed at improving the execution times.

As expressed in the previous chapter, talking about the approach used for the design of the implemented solutions, three different types of solutions were found: a first one, that is really basic, which tries to overcome the limits of the Parallella board in the simplest way possible, making the most on the shared memory and using all the sixteen available eCores, without implementing any kind of ad-hoc optimization; a second and more optimized solution, that, as described, tries to bypass the memory limits by using only eight cores as active and the other eight as silent ones, used only for accessing to their memory; a third and final solution that is an attempt to use a different algorithm for the convolution operation, the MEC algorithm, in order to try to speedup the performances following an approach widely used by most of the neural networks frameworks.

All the results achieved using these three solutions are reported in the following subsections. As said in the previous chapter the first two solutions described were

used for running both Tiny Darknet and the custom model based on the MNIST dataset, while the third solution was used only for the latter model. For this reason the results reported in the remainder of this chapter are divided into five parts, one for each solution and model used, moreover the results achieved for the same model are compared between the three solutions found.

5.1.1 Methodologies Used

Before reporting all the obtained results, the methodologies used to measure these performance will be described in this subsection. What we have tried to do is to obtain different levels of granularity in the results found. To better express this concept we can say that three types of results have been found with regard to the execution times: one more coarse and referred purely to the total time required by the entire program to be executed, which goes from when the image is received in input to when the final result is returned; one referring to the entire Epiphany coprocessor, which reports the execution times of every single kernel developed for it and executed by all the involved eCores; finally, a finer one that takes into consideration every single operation performed by each of the Epiphany cores.

As it can be imagined, different approaches were taken for obtaining these different kinds of measures. Regarding the first two types of results, the method used is rather simple and based solely on the use of the *clock_t* struct and of the *clock()* function, both provided by the C programming language and used inside the main program executed by the two ARM cores. The *clock()* [27] function returns the CPU time used so far as a *clock_t* struct. To compute the time elapsed for the execution of some operations it is sufficient to call such function just before and after the operations that one wants to measure and then to compute the difference of the two values returned. To get the number of seconds used, it is necessary to divide the result by *CLOCKS_PER_SEC* and to multiply by 1000. To measure the total time of execution of the program, the *clock()* function was just called at the start of the program, before even loading the network parameters into the shared memory, and after the result of the classification is returned. It was decided to provide such kind of measure because it is also the one provided by running Darknet. In this way it could be possible to compare the results achieved using the Parallella board with the ones that could be possible to obtain running the framework on other devices. To measure instead the time required by the Epiphany coprocessor to run each developed kernel for such architecture, the same technique was used. In this case the *clock()* function was called before and after the execution of each kernel, in particular during the synchronization between the eCores and the ARM cores, after each Epiphany core finished its execution.

As for the results with finer granularity, i.e. those concerning the execution times of each individual layer by each eCore involved, a different method was used. As said during the description of the Parallella board, each core of the Epiphany architecture has two 32-bit event timers that can be used to sample different real-time events within the system, including also a general-purpose clock-cycle counter, which can be used to measure time and to profile function execution time. These two registers are called *CTIMER0* and *CTIMER1* and they contain the current

value of the event being monitored. Both these two registers count down from an high value to zero, in particular the high value used during the measures taken was *E_CTIMER_MAX* that is the highest value possible. To use such methodology, inside each kernel the following schema was followed:

```
e_ctimer_set(E_CTIMER_0, E_CTIMER_MAX);
start_ticks = e_ctimer_start(E_CTIMER_0, E_CTIMER_CLK);

...

stop_ticks = e_ctimer_get(E_CTIMER_0);
e_ctimer_stop(E_CTIMER_0);

elapsed_ticks = start_ticks - stop_ticks;
```

As it can be seen, at first, the register used (in this case *CTIMER0*) is set to the highest value. Then, the timer is started, passing as parameter the type of event to measure. In this case *E_CTIMER_CLK*, in order to measure the number of elapsing clock ticks. At the end the timer is stopped and the measured value is retrieved. To compute the number of clock ticks that are elapsed it is necessary to calculate the difference between the initial and the final value. As said, in this way the value returned is the number of elapsed clock ticks. From such value the time of execution in milliseconds can be obtained by dividing it by the clock frequency of the cores involved and then by multiplying the result returned by 1000. In our case the clock frequency of each eCore, as also said describing the Epiphany platform, is set to 600 MHz, so the formula used, is as follows:

$$execution\ time\ (ms) = \frac{clock\ ticks}{600000000} * 1000$$

In the following subsections the results are reported as just described, in tabular form and histograms. These were divided into two macro subsections related to the two models used, Tiny Darknet and the custom model based on the MNIST dataset. In turn, these subsections are subsequently subdivided for each of the solutions found for them.

At the end of these subsections there is a further subsection in which an analysis of the results achieved is carried out, discussing the main strengths and weaknesses of the solutions found. This analysis will be used as a starting point, together with the next one concerning power consumption, for the final conclusions on this thesis work, as reported in the following chapter.

5.1.2 Results for Tiny Darknet

1. Basic Solution

Kernel	Layer	Time (ms)	Standard Deviation
1st	conv	6838.527	5.615
	max	87.559	0.151
	conv	7102.861	1.307
	max	43.622	0.0277
2nd	conv	1312.574	1.0513
	conv	7158.27881	0
	conv	1838.681	1.700
	conv	7158.279	0
	max	44.271	0.0192
3rd	conv	1434.349	1.627
	conv	7158.279	0
	conv	1846.357	1.520
	conv	7158.279	0
	max	23.349	0.0184
4th	conv	1458.694	1.583
	conv	6613.702	3.737
	conv	1882.711	1.634
	conv	6689.387	4.329
5th	conv	2931.679	2.352
	conv	4445.396	5.197

Table 5.1. Tiny Darknet - Basic Solution - Layer Execution Times

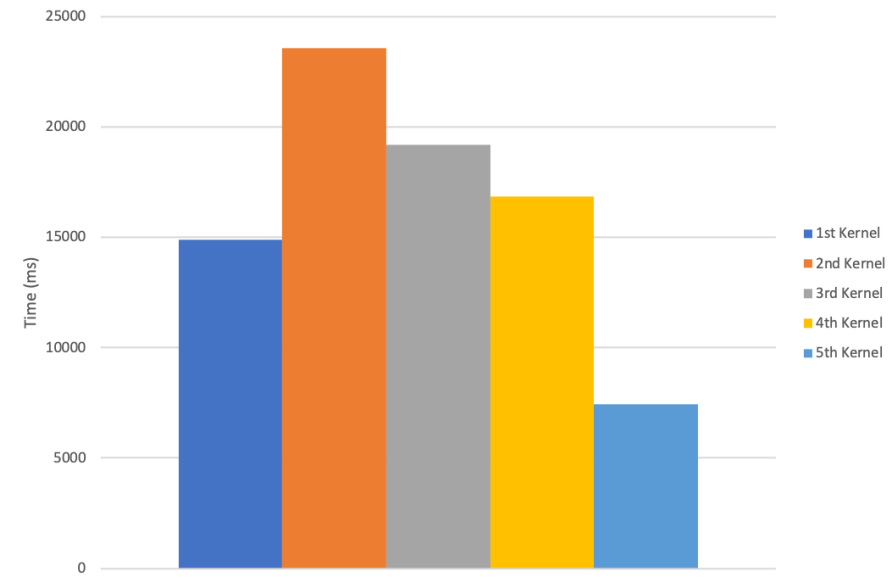

Figure 5.1. Tiny Darknet - Basic Solution - Kernel Execution Times

Total Time Of Execution (ms)	Standard Deviation
84317.837	48.263

Table 5.2. Tiny Darknet - Basic Solution - Total Execution Time

2. Extended Memory Solution

Kernel	Layer	Time (ms)	Standard Deviation
	conv	865.948	0.668
	max	32.307	0.0763
	conv	1438.061	0.919
1st	max	18.936	0.015
	conv	68.759	0.264
	conv	1400.959	0.619
	conv	251.0149	0.815
	conv	1401.120	0.752
2nd	max	19.006	0.0144
	conv	224.222	0.154
	conv	5537.610	0.803
	conv	375.825	0.309
	conv	6131.086	0.924
3rd	max	9.933	0.025
	conv	262.639	0.234
	conv	3816.325	1.241
	conv	430.017	0.219
4th	conv	3949.849	1.425
	conv	1117.222	0.444
5th	conv	1102.213	0.436

Table 5.3. Tiny Darknet - Extended Memory Solution - Layer Execution Times

Total Time Of Execution (ms)	Standard Deviation
31577.799	5.735

Table 5.4. Tiny Darknet - Extended Memory Solution - Total Execution Time

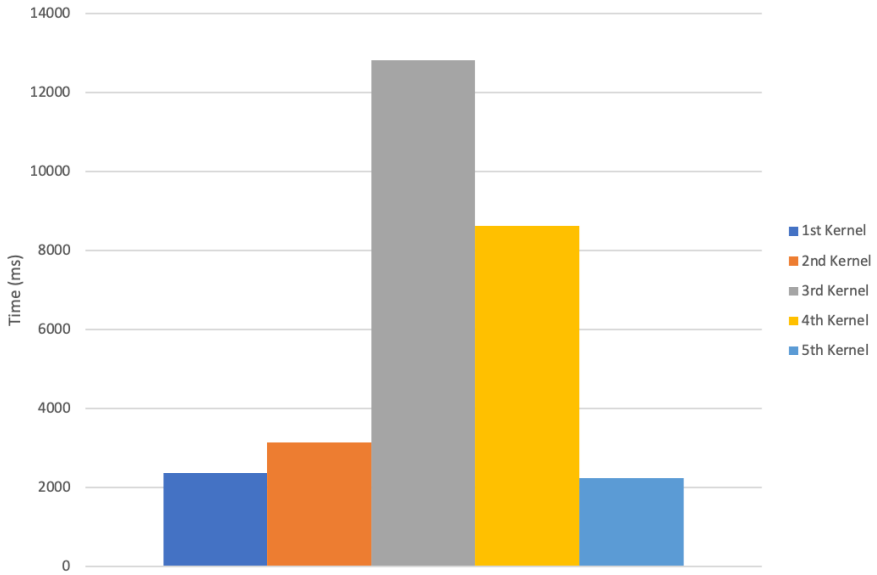

Figure 5.2. Tiny Darknet - Extended Memory Solution - Kernel Execution Time

5.1.3 Results for the MNIST Custom Model

1. Basic Solution

Kernel	Layer	Time (ms)	Standard Deviation
1st	conv	109.199	0.115
	max	2.568	0.003
2nd	conv	563.001	0.269
	max	1.319	0.002
3rd	full connected	199.808	1.688
4th	full connected	0.599	0.001

Table 5.5. MNIST Custom Model - Basic Solution - Layer Execution Times

Total Time Of Execution (ms)	Standard Deviation
3177.344	3.3921

Table 5.6. MNIST Custom Model - Basic Solution - Total Execution Time

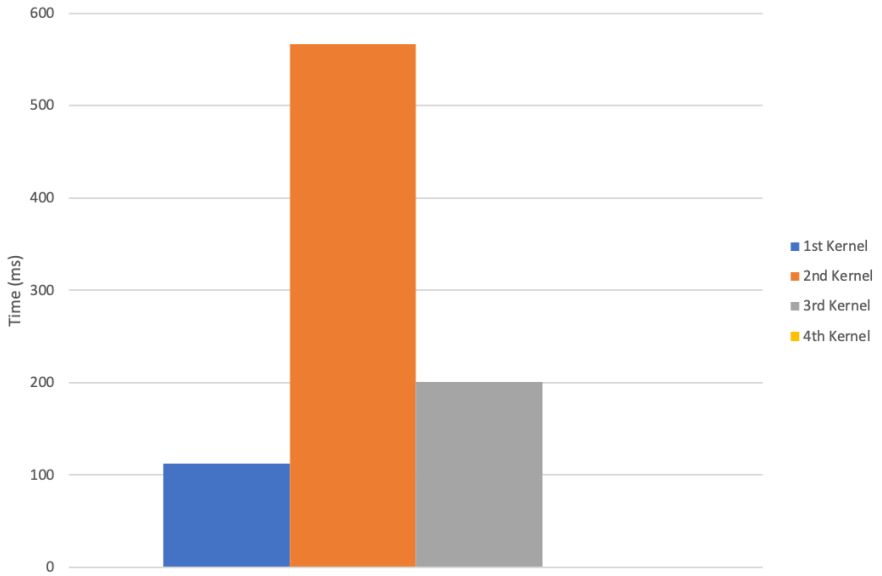

Figure 5.3. MNIST Custom Model - Basic Solution - Kernel Execution Times

2. Extended Memory Solution

Kernel	Layer	Time (ms)	Standard Deviation
1st	conv	48.049	0.0131
	max	1.273	0.004
2nd	conv	241.633	0.029
	max	0.806	0.002
3rd	full connected	180.926	0.485
4th	full connected	3.471	0.019

Table 5.7. MNIST Custom Model - Extended Memory Solution - Layer Execution Times

Total Time Of Execution (ms)	Standard Deviation
2756.158	3.579

Table 5.8. MNIST Custom Model - Extended Memory Solution - Total Execution Time

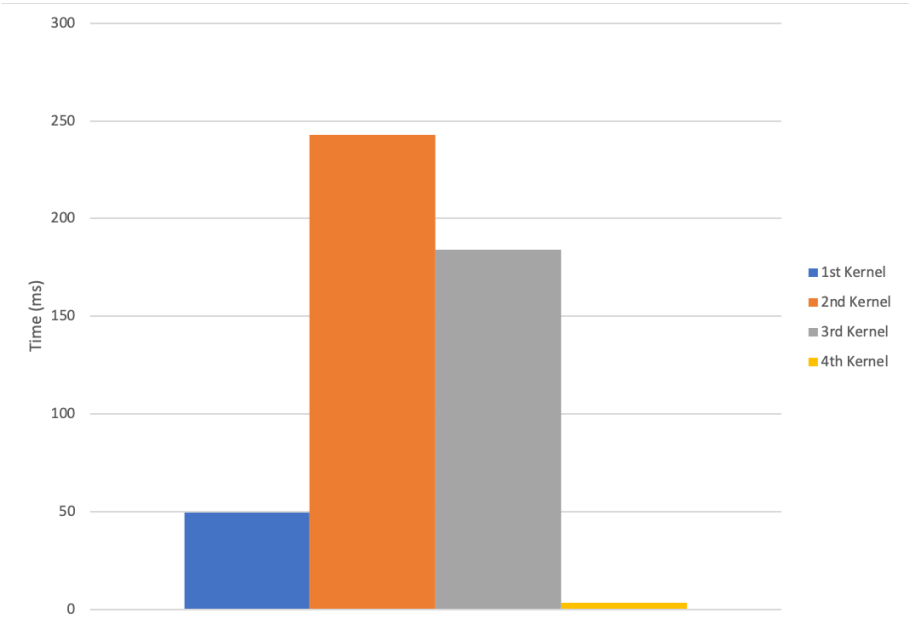

Figure 5.4. MNIST Custom Model - Extended Memory Solution - Kernel Execution Time

3. MEC Solution

Kernel	Layer	Time (ms)	Standard Deviation
1st	conv	52.596	0.163
	max	1.633	0.006
2nd	conv	2122.981	5.518
	max	0.807	0.002
3rd	full connected	180.970	0.539
4th	full connected	3.473	0.020

Table 5.9. MNIST Custom Model - MEC Solution - Layer Execution Times

Total Time Of Execution (ms)	Standard Deviation
4649.5824	5.3959

Table 5.10. MNIST Custom Model - MEC Solution - Total Execution Time

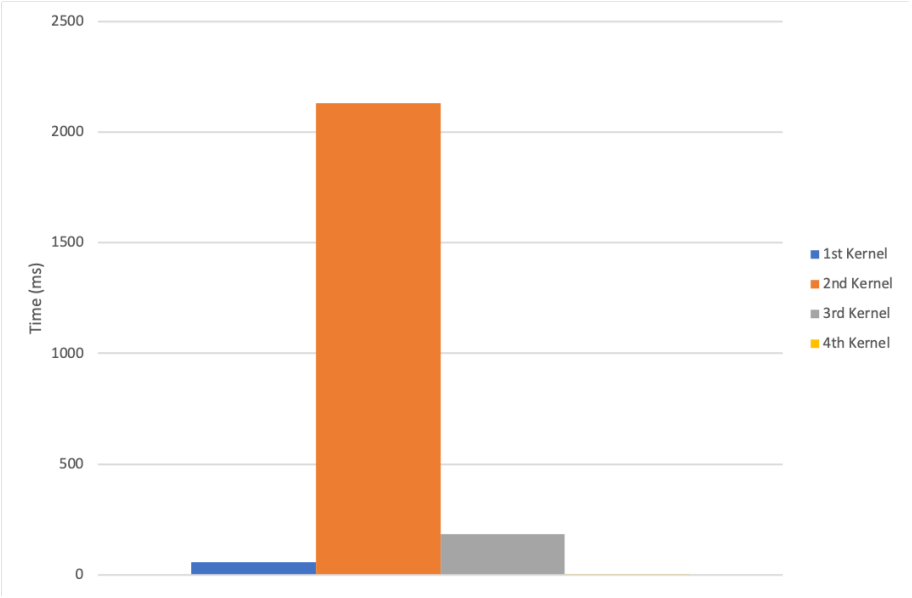

Figure 5.5. MNIST Custom Model - MEC Solution - Kernel Execution Times

5.1.4 Analysis

In this subsection there is an analysis of the results just presented, commenting on them and exposing the strengths and weaknesses of the solutions found. We will also try to highlight the possible causes of these results trying to understand why some approaches used are preferable to others. From this analysis and from those that will follow in the course of the chapter, about the power energy consumption of the board, we trigger the subsequent final considerations of the next chapter.

First of all, summary charts are shown in Fig. 5.6 and in Fig. 5.7. These charts report, for both the two network models used, a comparison between the three solutions found (two in the case of Tiny Darknet) regarding the total execution times.

From these two graphs the first evidence is how the execution times of the model based on the MNIST dataset are much lower than those required for Tiny Darknet. This observation has been described as trivial because it can be easily understood by the simple fact that the Tiny Darknet network model has an higher number of layers compared to the MNIST based network, in particular the number of convolutional layers (the most exorbitant in terms of both resources and necessary operations) turns out to be much higher. A first logical deduction deriving from this consideration is how a board with the characteristics of the Parallella is better suited for smaller networks, having a smaller number of layers to perform. This is not surprising, indeed it has already been foreseen in the introduction to this thesis, discussing the possible use cases that this work wanted to address. For those cases, as mentioned, it is expected to recognize a limited number of features present in an image (among those already mentioned we remember the recognition of the color of an object or its shape), which can usually be done through the use networks with a low number of layers, as happens in the network used to recognize handwritten numbers. On the contrary, Tiny Darknet represented only an experiment for this work, in an

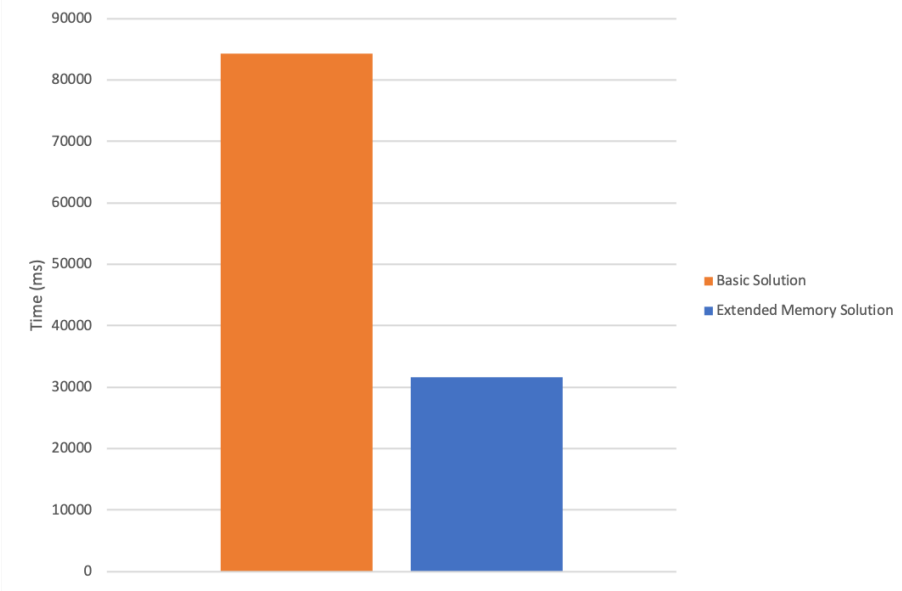

Figure 5.6. Tiny Darknet - Total Execution Times Comparison

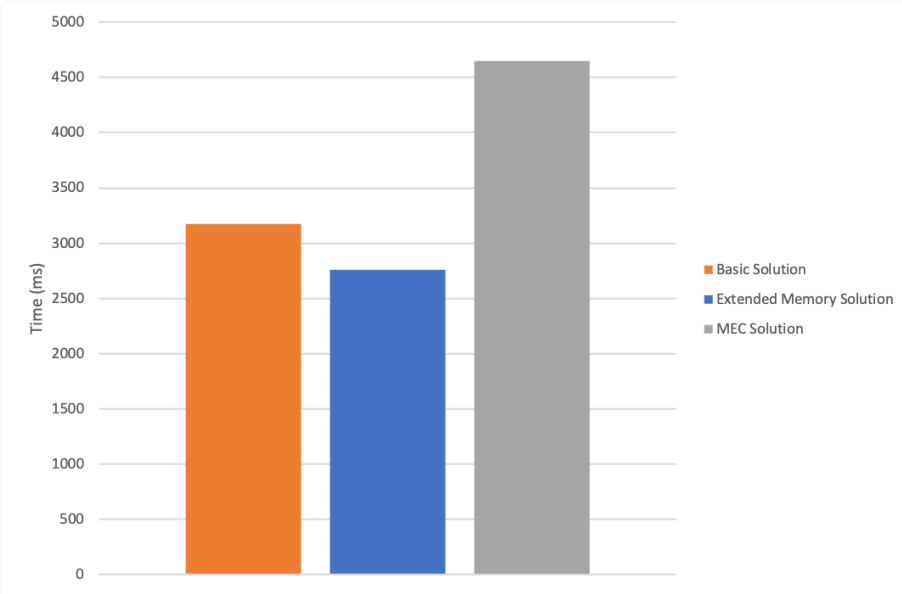

Figure 5.7. MNIST Custom Model - Total Execution Times Comparison

attempt to understand when we could go further with the resources made available by the Parallella board and in particular by the Epiphany architecture.

Comparing the two charts, a further consideration is how the difference in terms of time between the execution of the base solution and that with extended memory is much wider with Tiny Darknet than with the MNIST model. In the first case it amounts to about 52 000 ms, while in the second it is only about 400 ms. What is the reason for this sharp difference in terms of distance between the two solutions depending on the model used? Surely an explanation can be sought in how the models are made. Reconnecting to the last chapter, it is good to remember two

fundamental characteristics both about the networks themselves, and about the implemented solutions:

- The first thing to re-highlight is the difference in terms of the size of the input and output matrices between the two models used. In the case of Tiny Darknet, in fact, the dimensions of the intermediate matrices are significantly greater than those required by the MNIST network. Indeed, in the first case the input matrix is set to 224 x 224 x 3, while in the second case is set to 28 x 28 x 3;

- The second thing to remember is these matrices themselves are managed by eCores. In fact, if there is a difference in the management of large matrices between the base and the extended memory (in the first case the shared memory is used for storage, while in the second the memory of the dormant cores is used), in the case of matrices of small dimensions these are saved directly in the internal memory of the Epiphany core that is performing the computations.

Starting from these two underlined points, it is understandable that in the case of the network based on the MNIST dataset the Epiphany cores statistically find themselves managing small matrices more often, carrying out the necessary operations on them directly in their internal memory. Even if this were not possible, that is, if the required matrices were not small enough to be stored within an eCore, the basis solution would still perform fewer operations in the shared memory with the MNIST network than with Tiny Darknet. This entails a thinning in terms of execution times between the two solutions found.

This concept can be further explored. In fact, we can consider the two implementations of these solutions with regard to Tiny Darknet, observing both execution times of each layer for both, in particular convolution layers that are generally the heavier ones to execute, than the times required to perform every kernel. From these we can see how in general the main differences between the two in terms of performance are in the first layers/kernels executed. Obviously, it is not possible to state that the execution times are similar for the last operations performed, since these are almost longer than twice as long in the basic solution. However, what can be said is that the enormous initial difference is due to the conformation of the network, which initially provides large matrices but a small number of channels for these. In these conditions the difference between the two solutions is clear, since in the basic solution the eCores are to manage these matrices completely in external memory, performing a large number of operations on it. In the extended memory solution, on the other hand, the active cores are able to perform few transfers from the shared memory to that of the dormant cores, precisely because the number of channels is low, thus managing to perform the operations in a short time. On the contrary, when the number of channels increases, the number of transfers increases proportionally and therefore the performances are lowered. It can therefore be generally stated that the extended memory solution is clearly preferable to the basic one, as is clearly noticeable by observing the execution times. However, this solution also has its bending moments, which generally occur when the number of transfers from the external memory to that of the dormant cores increases.

This analysis obviously can not disregard the results achieved with the MEC solution. This solution, as mentioned, has been implemented only for the convolutional network based on the MNIST dataset. The results achieved by it in terms of the total time necessary for its execution, as clearly visible, are worse than those achieved with the other two solutions presented in this work. But how is it possible that a solution that contemplates an optimized version of the convolution operation is worse than solutions based on a direct convolution, although the latter is notoriously slower than the first? As explained in the course of this work, the optimized convolution algorithms, as well as im2col used in Darknet, were initially discarded due to the fact that the performance guaranteed by them are obtained at the expense of greater memory consumption. Therefore, working with a board having a small number of resources in this sense, it was preferred to try to improve performance by following other directions. Nevertheless, after the implementation of the extended memory solution it was decided to proceed with a solution based on the MEC algorithm, based on the same implementation principles (dormant eCores used as memory extension) and using the MNIST network, which already requires a smaller amount of memory (in general for the smaller dimensions of the matrices involved). Observing the data obtained on the performance, especially of the individual layers, a strong deterioration in performance can be observed especially in the second convolutional layer compared to the other two solutions. This can be attributed in particular to the *mec_shaped()* method used for the creation of the lowered matrix required by the MEC algorithm. The main problem of this function is not so much the fact that it happens completely in shared memory, because even the method to get the zero padding in the basic solution has the same behavior while still obtaining better results, but more than anything else that the operations involved in it provide access to non-contiguous memory areas (such as zero padding), due to the very nature of the lowered matrix. The same happens in the very heart of the algorithm, when this last matrix is further sectioned to be multiplied by the weight matrix. The resulting submatrix used by the Epiphany cores, in fact, are constituted in the same way, through non-consecutive accesses in the SDRAM. From this it can be affirmed, or better to confirm, the fact that such algorithms for performing on devices with adequate amount of memory, are not very efficient on devices such as the Parallella and in particular on architectures like as the Epiphany one.

By observing the execution times of the kernels performed and comparing them with the total execution times, it is possible to notice a certain discrepancy between them. This difference is attributable to the execution times required by the two ARM cores. Although the difference between these results is almost insignificant for Tiny Darknet, it becomes substantial in the case of the MNIST network. The total execution time becomes almost double, indeed compared to that given by the sum of the times required to run the kernels. This difference is attributable in particular to the fact that the network parameters in the case of Tiny Darknet are loaded at once, at the beginning of the program (basically, therefore, they could be loaded into shared memory only once and then used for recognizing more images), while in the second convolutional network they are loaded in three steps and are interspersed with the execution of the various kernels. This derives from the space itself occupied by these parameters, which makes it impossible to load them into shared memory,

thus making necessary to divide such loading in several times.

Finally, a general consideration must be given about the execution times of the various layers, reported in the previous tables for each solution. From these it can be seen how in each measurement the standard deviation is generally very low (the maximum value assumed by it is equal to about 5 ms). This is a sign of a good *stability* of the solutions, at least as far as performance and execution times are concerned. In general, the reliability of Epiphany architecture can be emphasized.

5.2 Power Consumption Analysis

Results achieved using a low-power device like the Parallella board can not but take into account the data about power consumption. As stated during the introduction to this work, power efficiency is becoming more and more often a focal point for modern implementations. Obviously such concept must also be applied to convolutional neural networks, that despite being often evaluated on the basis of their mere performances, as done for this work in the previous section, must take into account their energy yield, especially in application contexts such as the industrial one.

As for the performances analysis also in this case the measurements made refer to the three types of solutions found and to both the models used for the convolutional neural networks.

5.2.1 Methodologies Used

In order to measure the amount of power consumed by the Parallella board, during the execution of the solutions found, mainly two approaches were used: one relative to the total consumption achieved by the entire Parallella board and one in which it was attempted to measure the consumption of Epiphany architecture alone.

To measure the total consumption of the whole board, this was fed with a DC power supply keeping the constant voltage at 5.3 V and measuring each time the average electric current used. In this way power consumption can be traced back to the formula $power(W) = current(A) * voltage(V)$.

In the second case, however, it was decided to try to measure the energy consumption of only Epiphany architecture. Since there are no pins present on the board in order to be used for this purpose, it was decided to proceed with an indirect measurement. The idea followed was inspired by the scientific paper "Instruction level energy model for the Adapteva Epiphany multi-core processor" [28]. Such publication introduces a measurement-based instruction-level energy characterization for the Epiphany processor, that is the one presents on the Parallella board. Such characterization is used to create an energy model for the most important Epiphany instructions such as floating-point operations, integer operations, branches, etc. This model allows to estimate the power consumption of a developed software starting from the number of operations executed for each kind of instruction. What is done in practice is to evaluate each type of instruction from an energy point of view, assigning to each one a base energy cost. Starting from this mapping (instruction

to base energy cost) and from the number of operations performed it is possible to go back to the total energy consumption. The evaluation performed in the paper uses ad-hoc written programs for the eCore in order to know precisely how many operations are executed for any kind of instruction. In particular, as reported by the authors, these microbenchmarks were built using C with in-line assembly insertions to minimize the influence of compiler optimizations on the generated code. Using this method it was possible to assign a base energy cost to most of the instruction groups that can be used. These base energy costs are reported in 5.11 for the basic instructions, while in 5.12 are reported the base energy costs for remote loads and stores between two eCores (based on their Hamming distance).

Parameter	**Energy (pJ)**
Integer Operations	17.93
Floating Point Operations	29.39
Branch	154.22
Local Store	47.99
Local Load	39.82
Pipeline Stalls	53.65
Shared Memory Stores	581.72
Shared Memory Loads	2054.67
NOP	17.07
Idle Cycle	23.59

Table 5.11. Base Energy Cost

Hamming Distance	**Load Energy (pJ)**	**Store Energy (pJ)**
1	339.28	112.51
2	379.61	117.96
3	419.48	123.34
4	461.65	128.47
5	499.30	134.21
6	541.89	139.22

Table 5.12. Base Energy Cost For Remote Loads And Stores On The eMesh

The nature of the developed code for this work of thesis, however, has made the use of this method impracticable. In particular, the high complexity, at the code level, of the implemented solutions made it impossible to calculate the number of operations performed by type of instruction. The high number of cycles performed together with the fact of not using assembly code but relying on the optimizations performed by the compiler have emphasized this problem.

Not being able to have such a fine particle size about the number of instructions executed by type of operation, it was decided to proceed with an approximation of this count. As it will be seen later, many of the types of instructions executed can not be counted precisely because they are aggregated, as many operations can not be easily counted by exploiting the event timers present in each eNode. What has

been done is to use the "worst case" considering for those instructions aggregated the base energy cost relative to the most wasteful instruction. Following the same reasoning it was decided to overbook all those instructions otherwise not counted using the event timers.

Obviously this did not allow a precise calculation concerning the energy consumption of the Epiphany architecture. But what has been possible, however, was obtained by estimating the worst case, in any case capable of comparing the solutions found also from an energy point of view (although this is always an approximation) as well as that of mere performances.

Three approaches were used in particular to estimate these numbers:

- The first approach consisted of using the two event-timers present on each eCore to count the number of occurrences related to the events caused by the types of operations reported in 5.11. Specifically:
  - *E_CTIMER_IALU_INST* was to used as parameter for the timers in order to count the integer operations and the local loads and stores. These three kinds of operations are all executed by the ALU of the Epiphany coprocessor, so they are aggregated in the count performed by the event timer. In this case as base energy cost the one related to a *local store* was used.
  - *E_CTIMER_FPU_INST* was used as parameter to count the floating point operations performed by each eCore.
  - *E_CTIMER_E1_STALLS* was the parameter passed to the event timers in order to count the number of pipeline stalls.
  - *E_CTIMER_EXT_LOAD_STALLS* was used to calculate the number of external memory accesses to execute load instructions. In this case the number returned by the event timer indicates the number of stalls due to load operations in external memory. Instead, to estimate the data of our interest, the value returned by the timer has been divided by the average number of stalls due to an external data load, which amounts to about 10 as reported in the Epiphany Architecture Reference [19, Tab. 22].
  - *E_CTIMER_IDLE* to count the number of clock cycle spent in idle.
- The second approach used involved the counting of branch operations and NOPs. Since there are no parameters to use with the event timers available to be able to count this type of instructions, it was to use a static analysis technique based on the assembly code produced by the compiler. To obtain the assembly code produced, in particular, a tool [29] supplied by Adapteva was used: this receives the code written for the Epiphany architecture as input, returning the assembly code as output. Such tool emulates the modified version of the *gcc* compiler developed for the Parallella. From the assembly code it was possible, writing just a little parser in Python, to calculate the number of assembly instructions regarding the branch operations, as well as the NOP instructions executed.

Obviously such number does not take into account the executed loops, since these are computed only at execution time and not at compilation time. To overcome this problem, it was decided to count the number of cycles performed for each function, multiplying this number by the number of branch instructions counted by the parser. Also in this case the result obtained was calculated in the worst case, taking the maximum number of times in which the cycles could actually be executed.

- Finally, the third approach followed concerns the count of write/read operations between two eCores. For this purpose it was decided to use the *mesh timer*, as described in an article [30] written by Nick Oppen (a computer scientist who has dedicated part of his studies on the Parallella). As stated by the author of this post, such timer works exactly as the one described in the first method, the only difference is in the fact one has to specify which mesh event must be counted. To do this, the desired value must be set in the *E_MESH_CONFIG* register (the initial value of this register will be restored once the measurement is finished). In the case in question, this register has been set up so as to be able to count any kind of access to the eMesh, both for reading and writing. An example of the lines of code used to configure this register are shown below:

```
#define E_CTIMER_MESH_1 (0xe)
#define E_MESHEVENT_ANYWAIT1 0x00000200

...

int mesh_reg = e_reg_read(E_REG_MESHCFG);
int mesh_reg_timer = mesh_reg & 0xfffff0ff;
mesh_reg_timer = mesh_reg_timer | E_MESHEVENT_ANYWAIT1;
e_reg_write(E_REG_MESHCFG, mesh_reg_timer);

e_ctimer_set(E_CTIMER_1, E_CTIMER_MAX);
start_ticks = e_ctimer_start(E_CTIMER_1, E_CTIMER_MESH_1);

...

stop_ticks = e_ctimer_get(E_CTIMER_1);
e_ctimer_stop(E_CTIMER_1);

elapsed_ticks = start_ticks - stop_ticks;
```

  As seen in 5.12, the energy required to perform this type of operation depends on the Hamming distance between the two involved eCores. In the case of this work, a distance of 3 was taken as the average value.

Using these three methods it was possible to estimate the number of operations performed by type of instruction. From this estimate we proceeded with the calculation of the power consumed for each layer performed by each of the proposed solutions.

5.2.2 Results for Tiny Darknet

1. Basic Solution

Operation Type	Avg Total Number Of Operations	Base Energy (pJ)	Total Energy (pJ)
ALU Operation (Integer Operations, Local Loads and Stores)	28824604254	47.99	1383292758125.47
FPU Operation	1223578047	29.39	35960958801
Pipeline Stall	32542662920	53.65	1745913865652.63
External Memory Access	29709672950	2054.77	61046544687944.10
Idle Cycle	0	23.59	0
Branch Operation	65214910656.00	154.22	10057443521368.30
NOP	27	17.07	460.89
cMesh Operation	9328558303	419.48	3913143636942.44
		Total Energy (pJ)	78182299429294.80
		Total Energy (J)	78.182

Table 5.13. Tiny Darknet - Basic Solution - Energy Consumption

Energy Consumption (J)	Execution Time (s)	Power Consumption (W)
78.182	84.318	0.927

Table 5.14. Tiny Darknet - Basic Solution - Power Consumption

Voltage (V)	Current (A)	Power Consumption (W)
5.3	0.97	5.12

Table 5.15. Tiny Darknet - Basic Solution - Parallella Power Consumption

2. Extended Memory Solution

Operation Type	Avg Total Number Of Operations	Base Energy (pJ)	Total Energy (pJ)
ALU Operation (Integer Operations, Local Loads and Stores)	19107873405	47.99	916986844681.96
FPU Operation	1005657407	29.39	29556271192
Pipeline Stall	11072965559	53.65	594064602224.26
External Memory Access	888659711.5	2054.77	1825991315296.12
Idle Cycle	0	23.59	0
Branch Operation	17806218	154.22	9754058186737.92
NOP	21	17.07	358.47
cMesh Operation	2782368566	419.48	1167147966065.68
		Total Energy (pJ)	14287805186556.40
		Total Energy (J)	14.288

Table 5.16. Tiny Darknet - Extended Memory Solution - Energy Consumption

Energy Consumption (J)	Execution Time (s)	Power Consumption (W)
14.288	31.578	0.452

Table 5.17. Tiny Darknet - Extended Memory Solution - Power Consumption

Voltage (V)	Current (A)	Power Consumption (W)
5.3	0.93	4.95

Table 5.18. Tiny Darknet - Extended Memory Solution - Parallella Power Consumption

5.2.3 Results for the MNIST Custom Model

1. Basic Solution

Operation Type	Avg Total Number Of Operations	Base Energy (pJ)	Total Energy (pJ)
ALU Operation (Integer Operations, Local Loads and Stores)	852938788	47.99	40932532436
FPU Operation	30791963	29.39	904975792.6
Pipeline Stall	325138283	53.65	17443668883
External Memory Access	691576058	2054.77	1421029736984.33
Idle Cycle	0	23.59	0
Branch Operation	1476920448	154.22	227770671490.56
NOP	7	17.07	119.49
cMesh Operation	61124555	419.48	25640528331
		Total Energy (pJ)	1513722114036.98
		Total Energy (J)	1.514

Table 5.19. MNIST Custom Model - Basic Solution - Epiphany Energy Consumption

Energy Consumption (J)	Execution Time (s)	Power Consumption (W)
1.514	3.177	0.476

Table 5.20. MNIST Custom Model - Basic Solution - Epiphany Power Consumption

Voltage (V)	Current (A)	Power Consumption (W)
5.3	0.93	4.95

Table 5.21. MNIST Custom Model - Basic Solution - Parallella Power Consumption

2. Extended Memory Solution

Operation Type	Avg Total Number Of Operations	Base Energy (pJ)	Total Energy (pJ)
ALU Operation (Integer Operations, Local Loads and Stores)	920768714.5	47.99	44187690609
FPU Operation	30804240	29.39	905336614
Pipeline Stall	479348413.7	53.65	25717042395
External Memory Access	25872053.89	2054.77	53161120171.56
Idle Cycle	0	23.59	0
Branch Operation	2459860992	154.22	379359762186.24
NOP	12	17.07	170.7
cMesh Operation	69045680	419.48	28963281846
		Total Energy (pJ)	532294233992.50
		Total Energy (J)	0.532

Table 5.22. MNIST Custom Model - Extended Memory Solution - Epiphany Energy Consumption

Energy Consumption (J)	Execution Time (s)	Power Consumption (W)
0.532	2.756	0.193

Table 5.23. MNIST Custom Model - Extended Memory Solution - Epiphany Power Consumption

Voltage (V)	Current (A)	Power Consumption (W)
5.3	0.90	4.75

Table 5.24. MNIST Custom Model - Extended Memory Solution - Parallella Power Consumption

3. MEC Solution

Operation Type	Avg Total Number Of Operations	Base Energy (pJ)	Total Energy (pJ)
ALU Operation (Integer Operations, Local Loads and Stores)	4572171262	47.99	219418498863.38
FPU Operation	30791963	29.39	904975792.6
Pipeline Stall	2847752407	53.65	152781916646.28
External Memory Access	157144781.3	2054.77	322896382189.61
Idle Cycle	0	23.59	0
Branch Operation	2578384384	154.22	397638439700.48
NOP	12	17.07	2014.84
cMesh Operation	444111461	419.48	186295875660.28
		Total Energy (pJ)	1279936090867.47
		Total Energy (J)	1.280

Table 5.25. MNIST Custom Model - MEC Solution - Epiphany Energy Consumption

Energy Consumption (J)	Execution Time (s)	Power Consumption (W)
1.380	4.750	0.275

Table 5.26. MNIST Custom Model - MEC Solution - Epiphany Power Consumption

Voltage (V)	Current (A)	Power Consumption (W)
5.3	0.93	4.91

Table 5.27. MNIST Custom Model - MEC Solution - Parallella Power Consumption

5.2.4 Analysis

The results about the power consumption just shown in the previous graphs will be analyzed in this subsection. As previously done for the performance results, also in this case are presented the Fig. 5.8 and Fig. 5.9 charts that summarize the power consumption by comparing the approaches used for both the adopted models, Tiny Darknet and the MNIST custom model.

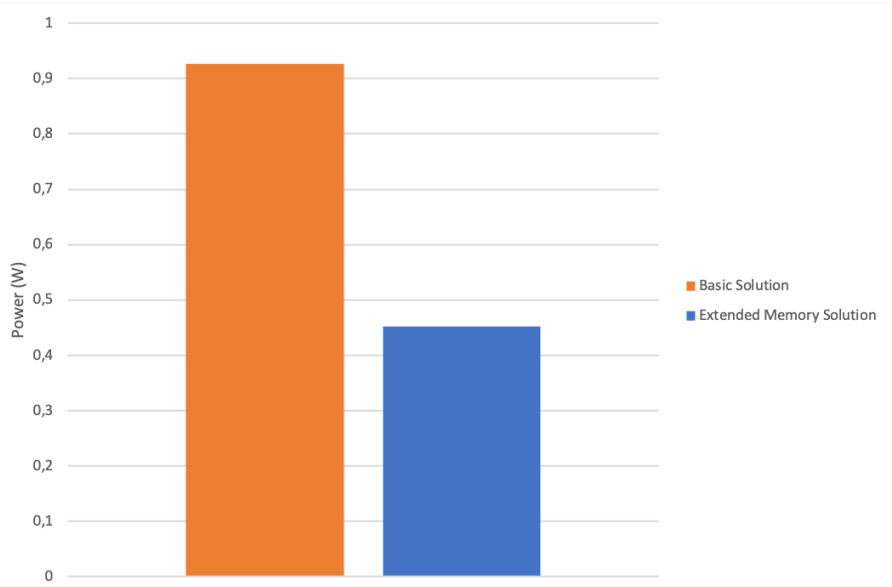

Figure 5.8. Tiny Darknet - Epiphany Power Consumption Comparison

Although it is right to remember that the method used for calculating the power consumption of the Epiphany architecture is the result of an approximation as previously mentioned, some considerations can be derived from it.

In general we can see, as widely predictable, how energy consumption is relatively low, approaching the consumption of 1 W only in the case of the Basic Solution with Tiny Darknet. In other cases, rather low consumption is found, which do not exceed 0.5 W and even in the best case (with the Extended Memory Solution and the MNIST network) fall below about 0.2 W, as can be expected from an architecture such as Epiphany which, as also reported in its datasheet, has a maximum peak consumption of 2 W.

A first interesting thing to note is how to the optimized solutions in terms of performance, in this case the Extended Memory Solutions in particular, is associated also a lower power consumption. This should not be a surprise considering the basic energy consumption by type of instruction shown in 5.11, where it can be seen how the energy consumption is related to instructions that access the external

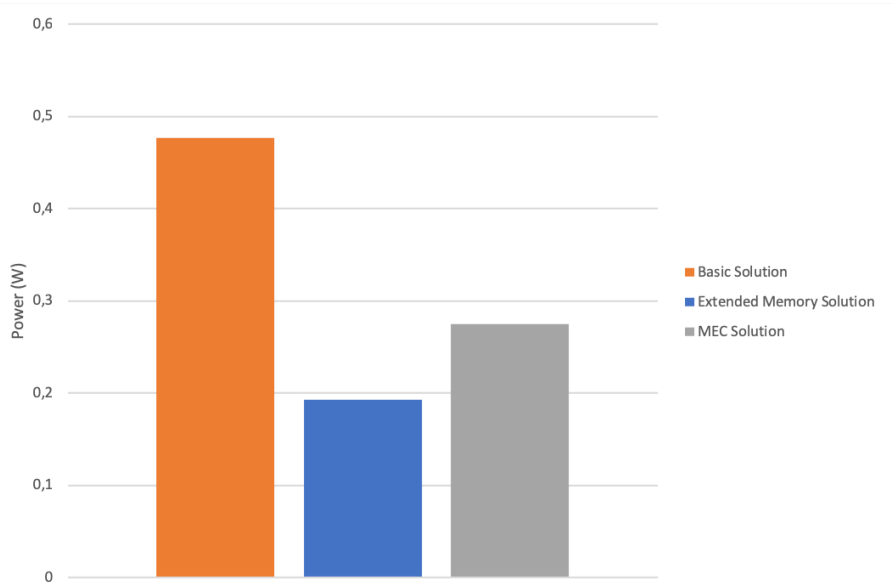

Figure 5.9. MNIST Custom Model - Epiphany Power Consumption Comparison

memory. In this sense, the Basic Solution is obviously strongly disadvantaged, since its functional logic is completely based on continuous access in shared memory.

A particular fact must be done on the MEC Solution. Although this has not presented exciting performances, as seen previously in this chapter, its consumption is still contained, especially compared to the Basic Solution. The reason for this can always be found in the fact that it performs a smaller number of external memory accesses, although these are more random accesses than the Basic Solution. The latter, in fact, performs a large number of contiguous accesses in external memory that from the point of view of performance make it stand out compared to the MEC Solution, but as regards power consumption make it less preferable.

In addition to the initial premise already made, it is right to spend a few more words about the energy model used for Epiphany architecture. As mentioned above, explaining the methodology used, it could not be perfectly applied to this thesis work, due to the complexity of the code developed in comparison with the one used in the paper taken as reference [28]. This is also demonstrated by the empirical measurements made on the power consumption of the whole board. If the instruction level energy model used was correct, it would mean, from the measurements made, that the consumption by the Zynq architecture would be almost at its maximum peak of 5 W. This is hardly credible, given that most of the processing is done on the Epiphany architecture. Obviously besides the application developed and in execution there are other active processes due to the operating system installed on the Zynq side, but these would not justify such a high peak consumption anyway. Are the measurements reported and so far commented totally unfounded? Not entirely, in fact, the differences in consumption found in these are very similar to those found by measuring the total consumption of the whole Parallel. What can be hypothesized is that, despite the fact that they were carried out considering the worst case, it turns out to be too low a valuation compared to the real one. This is easily

addressable to the fact that different types of instructions, among those reported in the paper introductive to this methodology, could not be correctly counted because or aggregated to others or simply because the tools available did not allow it. The model used nevertheless allows to have a basic idea on how certain approaches are preferable to others with regard to power consumption, based on the type of instructions executed. They support the thesis already supported by analyzing the performances, so the direction to follow in the development of applications for this board should go more towards reducing the number of external memory accesses to be efficient, at the cost of having to use a lower eCore number compared to those present.

Finally, a consideration on the total consumption of the board measured empirically. As already mentioned, these are very similar to each other on the whole, with differences in the order of one hundred milliwatts. These differences can be assumed to be due to the Epiphany architecture only, since the operations performed by the ARM cores are the same for all the solutions found (only the number of parameters loaded depends on the model considered).

Chapter 6

Conclusions

The main idea behind this thesis was to perform an evaluation on the use of machine learning techniques on a low-power low-cost platform like the Parallella board, exploiting as much as possible the resources provided by it.

Different approaches have been followed and investigated to optimize the evaluation times for a single image recognition (inference phase), with the aim of making the solution found as usable as possible in a real context: from a basic approach that tries to make the most of the board's multicore architecture; to an ad-hoc implementation developed to bypass the main bottlenecks of the device (like the poor amount of memory available) trying to exploit the many cores present in a smart way; finally trying to further improve the performances by intervening directly on the algorithm used for the convolution operation.

The implementations obtained through these approaches have been tested and evaluated, allowing, thanks to the results achieved, to make some consideration on their strengths and weaknesses. In general, the results obtained were encouraging, especially those related to the use of small networks, that proved to be the most suitable for the hardware used. Nevertheless, they are very far from exploitable results in industrial contexts, where the speed of execution is a fundamental requirement and the possibility of recognizing images in real time even almost. This does not mean, however, that future developments, which lead to optimizations in these directions, can not make the results found in this thesis, obviously readjusted, usable even in the contested hypotheses at the beginning of this work. In the next section some of the possible future developments to follow are reported in order to further carry out this study, improving the solutions found.

6.1 Future Works

This study was born with the aim of carrying out a feasibility assessment on the use of a heterogeneous low-power architecture such as the Parallella board for machine learning applications, in particular in the field of image recognition through the use of convolutional neural networks. The encouraging results obtained leave several doors open for future research.

A first way forward could be that of optimizing energy consumption. The main focus of this thesis, repeatedly reiterated, was that of obtaining a working solution that could be executed as quickly as possible. Starting from the energy model presented in the previous chapter, however, future developments could be addressed with a view to reducing the number of more energy-intensive operations. In particular, the event timers present on each eNode could be exploited, the same ones previously used for the only count of the number of operations per type of instruction, for this purpose.

As far as performance improvement is concerned, on the other hand, it is possible to explore optimizations inherent in the field of machine learning and in particular the design of convolutional network models. As explained in the course of this work, one of the main problems encountered was that of the amount of memory present on the device, especially the amount of internal memory present in each eCore. For this purpose, rather than looking for workarounds concerning the mere development of software as done in this study, just think of the solution based on the virtual extension of memory, we could intervene on the dimensions of the model itself. Several scientific studies have been presented in this sense, in particular the most encouraging are the representation of network parameters. They have shown that while reducing the accuracy of these parameters, the accuracy of the network does not diminish. In the used models the parameters are floating point numbers on 4 B, you could, for example, try to use the same but with parameters represented on 2 B, thus reducing their size by half. Assuming such an approach, it would be possible, for example, to load all the parameters of the MNIST-based network at once, greatly reducing the time required to execute it. Recalling the results obtained with this network, it is predictable that this approach would allow to have an image recognition with times close to near real-time.

Finally, another possible direction to follow could be the one concerning the construction of a Parallella cluster. This type of solution has already been followed several times by users of this board as well as scientific researchers, allowing to build an architecture with the same characteristics and type of development of a single Parallella board, but with performance increased proportionally to the number of used devices.

Bibliography

[1] How Machine Learning (ML) Is Transforming Manufacturing, https://towardsdatascience.com/how-machine-learning-ml-is-transforming-manufacturing-dfaaa30e87e4

[2] Machine Learning in Manufacturing - Present and Future Use-Cases, https://www.techemergence.com/machine-learning-in-manufacturing/

[3] G. Ratsch, "A brief introduction into Machine Learning"

[4] T. Cover and P. Hart, "Nearest neighbor pattern classification", IEEE Transactions on Information Theory, Vol. 13, No. 1, Jan. 1967, pp. 21-27 DOI 10.1109/TIT.1967.1053964

[5] R. A. Fisher, "The use of multiple measurements in taxonomic problems", Annals of Eugenics, Vol. 7, 1936, pp. 179-188

[6] R. Quinlan, "C4.5: Programs for Machine Learning", Morgan Kaufmann Publishers Inc., 1993, ISBN: 1-55860-238-0

[7] C. Cortes, V. Vapnik, "Support-Vector Networks", Mach. Learn., Vol. 20, No. 3, Sep. 1995, pp. 273-297 DOI 10.1023/A:1022627411411

[8] Y. Lecun, L.Bottou, Y.Bengio, P.Haffner, "Gradient-based learning applied to document recognition", Proceedings of the IEEE, Vol. 86, No. 11, Nov. 1998, pp. 2278-2324 DOI 10.1109/5.726791

[9] F. Rosenblatt, "The perceptron: a probabilistic model for information storage and organization in the brain", Psychological review, Vol. 65, No. 6, 1958, pp. 386

[10] TensorFlow, https://www.tensorflow.org/

[11] Caffe, http://caffe.berkeleyvision.org/

[12] PyTorch, https://pytorch.org/

[13] J. Redmon, "Darknet: Open Source Neural Networks in C", http://pjreddie.com/darknet/, 2013 - 2016

[14] J. Redmon, A. Farhadi, "YOLOv3: An Incremental Improvement", arXiv, 2018

[15] YOLO: Real-Time Object Detection, https://pjreddie.com/darknet/yolo/

[16] Why GEMM is at the heart of deep learning, https://petewarden.com/2015/04/20/why-gemm-is-at-the-heart-of-deep-learning/

[17] Basic Linear Algebra Subprograms, https://en.wikipedia.org/wiki/Basic_Linear_Algebra_Subprograms

[18] Kickstarting High-performance Energy-efficient Manycore Architectures with Epiphany, A. Olofsson, T. Nordstrom, Z. Ul-Abdin, CoRR, 2014, http://arxiv.org/abs/1412.5538

[19] Epiphany Architecture Reference, http://www.adapteva.com/docs/epiphany_arch_ref.pdf

[20] CUDA Toolkit Documentation, https://docs.nvidia.com/cuda
[21] Tiny Darknet, https://pjreddie.com/darknet/tiny-darknet/
[22] F. N. Iandola, M. W. Moskewicz, K. Ashraf, S. Han, W. J. Dally, K. Keutzer, "SqueezeNet: AlexNet-level accuracy with 50x fewer parameters and <1MB model size", CoRR, 2016, http://arxiv.org/abs/1602.07360
[23] The MNIST Database Of Handwritten Digits, http://yann.lecun.com/exdb/mnist/
[24] darknet_mnist, https://github.com/ashitani/darknet_mnist
[25] Making Faster, https://leonardoaraujosantos.gitbooks.io/artificial-inteligence/content/making_faster.html
[26] M. Cho, D. Brand, "MEC: Memory-efficient Convolution for Deep Neural Network", CoRR, 2017, http://arxiv.org/abs/1706.06873
[27] CLOCK, http://man7.org/linux/man-pages/man3/clock.3.html
[28] G. Ortiz, L. Svensson, E. Alveflo, P. Larsson-Edefors, "Instruction Level Energy Model for the Adapteva Epiphany Multi-core Processor", Proceedings of the Computing Frontiers Conference, 2017, pp. 380-384, DOI 10.1145/3075564.3078892
[29] Parallella Interctive Compiler, http://gcc.parallella.org/
[30] Benchmarking Broadcast Strategies, https://nicksparallellaideas.blogspot.com/2016/10/benchmarking-broadcast-strategies.html