Εθνικό Μετσόβιο Πολυτεχνείο
Σχολή Ηλεκτρολόγων Μηχανικών
και Μηχανικών Υπολογιστών

Τομέας Τεχνολογίας Πληροφορικής και Υπολογιστών
Εργαστήριο Μικροϋπολογιστών & Ψηφιακών Συστημάτων

Implementation of Convolutional Neural Networks on Embedded Architectures

Υλοποίηση Συνελικτικών Νευρωνικών Δικτύων σε Ενσωματωμένες Αρχιτεκτονικές

Διπλωματική Εργασία

του

ΑΘΑΝΑΣΙΟΥ Α. ΞΥΓΚΗ

Επιβλέπων: Δημήτριος Σούντρης
Αν. Καθηγητής Ε.Μ.Π.

Αθήνα, Σεπτέμβριος 2017

Εθνικό Μετσόβιο Πολυτεχνείο
Σχολή Ηλεκτρολόγων Μηχανικών και Μηχανικών Υπολογιστών
Τομέας Τεχνολογίας Πληροφορικής και Υπολογιστών
Εργαστήριο Μικροϋπολογιστών & Ψηφιακών Συστημάτων

Implementation of Convolutional Neural Networks on Embedded Architectures

*Υλοποίηση
Συνελικτικών Νευρωνικών Δικτύων
σε Ενσωματωμένες Αρχιτεκτονικές*

Διπλωματική Εργασία

του

ΑΘΑΝΑΣΙΟΥ Α. ΞΥΓΚΗ

Επιβλέπων: Δημήτριος Σούντρης
Αν. Καθηγητής Ε.Μ.Π.

Εγκρίθηκε από την τριμελή εξεταστική επιτροπή την 28η Σεπτεμβρίου 2017.

(Υπογραφή)	*(Υπογραφή)*	*(Υπογραφή)*
....	....	....
Δημήτριος Σούντρης Αν. Καθηγητής Ε.Μ.Π.	Κιαμάλ Πεκμεστζή Καθηγητής Ε.Μ.Π.	Γεώργιος Γκούμας Επ. Καθηγητής Ε.Μ.Π.

Αθήνα, Σεπτέμβριος 2017

Εθνικό Μετσόβιο Πολυτεχνείο
Σχολή Ηλεκτρολόγων Μηχανικών και Μηχανικών Υπολογιστών
Τομέας Τεχνολογίας Πληροφορικής και Υπολογιστών
Εργαστήριο Μικροϋπολογιστών & Ψηφιακών Συστημάτων

Copyright © – All rights reserved. Με την επιφύλαξη παντός δικαιώματος.
Αθανάσιος Ξύγκης, 2017.

Απαγορεύεται η αντιγραφή, αποθήκευση και διανομή της παρούσας εργασίας, εξ ολοκλήρου ή τμήματος αυτής, για εμπορικό σκοπό. Επιτρέπεται η ανατύπωση, αποθήκευση και διανομή για σκοπό μη κερδοσκοπικό, εκπαιδευτικής ή ερευνητικής φύσης, υπό την προϋπόθεση να αναφέρεται η πηγή προέλευσης και να διατηρείται το παρόν μήνυμα. Ερωτήματα που αφορούν τη χρήση της εργασίας για κερδοσκοπικό σκοπό πρέπει να απευθύνονται προς τον συγγραφέα.

Οι απόψεις και τα συμπεράσματα που περιέχονται σε αυτό το έγγραφο εκφράζουν τον συγγραφέα και δεν πρέπει να ερμηνευθεί ότι αντιπροσωπεύουν τις επίσημες θέσεις του Εθνικού Μετσόβιου Πολυτεχνείου.

ΔΗΛΩΣΗ ΜΗ ΛΟΓΟΚΛΟΠΗΣ ΚΑΙ ΑΝΑΛΗΨΗΣ ΠΡΟΣΩΠΙΚΗΣ ΕΥΘΥΝΗΣ

Με πλήρη επίγνωση των συνεπειών του νόμου περί πνευματικών δικαιωμάτων, δηλώνω ενυπογράφως ότι είμαι αποκλειστικός συγγραφέας της παρούσας Διπλωματικής Εργασίας, για την ολοκλήρωση της οποίας κάθε βοήθεια είναι πλήρως αναγνωρισμένη και αναφέρεται λεπτομερώς στην εργασία αυτή. Έχω αναφέρει πλήρως και με σαφείς αναφορές, όλες τις πηγές χρήσης δεδομένων, απόψεων, θέσεων και προτάσεων, ιδεών και λεκτικών αναφορών, είτε κατά κυριολεξία είτε βάσει επιστημονικής παράφρασης. Δηλώνω, συνεπώς, ότι αυτή η Διπλωματική Εργασία προετοιμάστηκε και ολοκληρώθηκε από εμένα προσωπικά και αποκλειστικά και ότι, αναλαμβάνω πλήρως όλες τις συνέπειες του νόμου στην περίπτωση κατά την οποία αποδειχθεί, διαχρονικά, ότι η εργασία αυτή ή τμήμα της δεν μου ανήκει διότι είναι προϊόν λογοκλοπής άλλης πνευματικής ιδιοκτησίας.

(Υπογραφή)

....
Α. Ξύγκης

28η Σεπτεμβρίου 2017

Περίληψη

Τα τελευταία χρόνια, στο επιστημονικό πεδίο της μηχανικής μάθησης, πραγματοποιούνται εκτεταμένες έρευνες γύρω από τα Τεχνητά Νευρωνικά Δίκτυα (ΤΝΔ). Τα ΤΝΔ αποτελούν υπολογιστικά μοντέλα, εμπνευσμένα από βιολογικούς οργανισμούς, τα οποία έχουν καταφέρει να ξεπεράσουν σε απόδοση τις προηγούμενες μορφές τεχνητής νοημοσύνης, για αρκετά από τα προβλήματα της μηχανικής μάθησης. Μια υποκατηγορία των ΤΝΔ είναι τα Συνελικτικά Νευρωνικά Δίκτυα (ΣΝΔ), που εμφανίζουν μεγάλη επιτυχία στην αποτελεσματική επίλυση προβλημάτων της όρασης υπολογιστών, όπως είναι η αναγνώριση πεζών, στερεοσκοπική όραση κ.α.

Για πολλά από τα σύγχρονα προβλήματα της όρασης υπολογιστών υπάρχει μεγάλη επιθυμία για εκτέλεση προτεινόμενων λύσεων σε ενσωματωμένες πλατφόρμες, που δίνουν έμφαση στην κατανάλωση ενέργειας, εφόσον όλα δείχνουν πως το Διαδίκτυο των Πραγμάτων θα κυριαρχήσει τα επόμενα χρόνια.

Στόχος της παρούσας διπλωματικής εργασίας είναι η ανάπτυξη ενός συστήματος εκτέλεσης ΣΝΔ, για τον ενσωματωμένο πολυεπεξεργαστή Myriad2. Πιο αναλυτικά, η Myriad2 έχει ως μεγαλύτερο πλεονέκτημα την χαμηλή κατανάλωση ενέργειας, ανά μονάδα υπολογισμού. Για να επιτύχει αυτά τα χαρακτηριστικά, η Myriad2 συνίσταται από 12 VLIW επεξεργαστές, χτισμένους γύρω από μια μικρή αλλά και γρήγορη μνήμη. Η υλοποίηση μιας υποδομής εκτέλεσης δεν είναι απλή υπόθεση, όταν κανείς έχει την απόδοση και την κατανάλωση ενέργειας ως κυρίαρχα κριτήρια. Από τη φύση τους, τα ΣΝΔ απαιτούν μεταφορά μεγάλης ποσότητας δεδομένων, με το μεγαλύτερο πρόβλημα να είναι η αποτελεσματική διατήρηση υψηλού ρυθμού ροής δεδομένων προς τους 12 επεξεργαστές. Κατά συνέπεια, απαιτείται προσεκτικός σχεδιασμός στην τοποθέτηση των δεδομένων και του πηγαίου κώδικα στις διάφορες ιεραρχίες μνήμης. Υψηλή απόδοση και χαμηλή κατανάλωση μπορεί να επιτευχθεί μόνον εάν ορισμένα τμήματα του συστήματος εκτέλεσης υλοποιηθούν σε συμβολική γλώσσα, καθιστώντας απαραίτητη την εμβάθυνση στις ιδιαιτερότητες του υλικού. Τέλος, το γεγονός ότι το σύστημα είναι πολυεπεξεργαστικό, αυξάνει την πολυπλοκότητα ακόμα περισσότερο.

Σε μεγάλο βαθμό, τα παραπάνω προβλήματα συναντώνται σε κάθε ενσωματωμένο επεξεργαστή, συνεπώς οι προτεινόμενες μεθοδολογίες και προσεγγίσεις που ακολουθούνται έχουν πεδίο εφαρμογής έξω από τη Myriad2.

Λέξεις Κλειδιά

Μηχανική μάθηση, Συνελικτικά νευρωνικά δίκτυα, Ενσωματωμένα συστήματα, Πολυεπεργαστικά συστήματα, Myriad 2

Abstract

During the recent years, the scientific field of machine learning has made an extended research effort towards the development of Artificial Neural Networks (ANN). ANNs are biologically inspired computational models, which have managed to exceed the performance of previous forms of artificial intelligence for a lot of machine learning tasks. A subcategory of ANNs are Convolutional Neural Networks (CNN), that show great success in the effective solution of computer vision problems, such as pedestrian detection, stereoscopic vision etc.

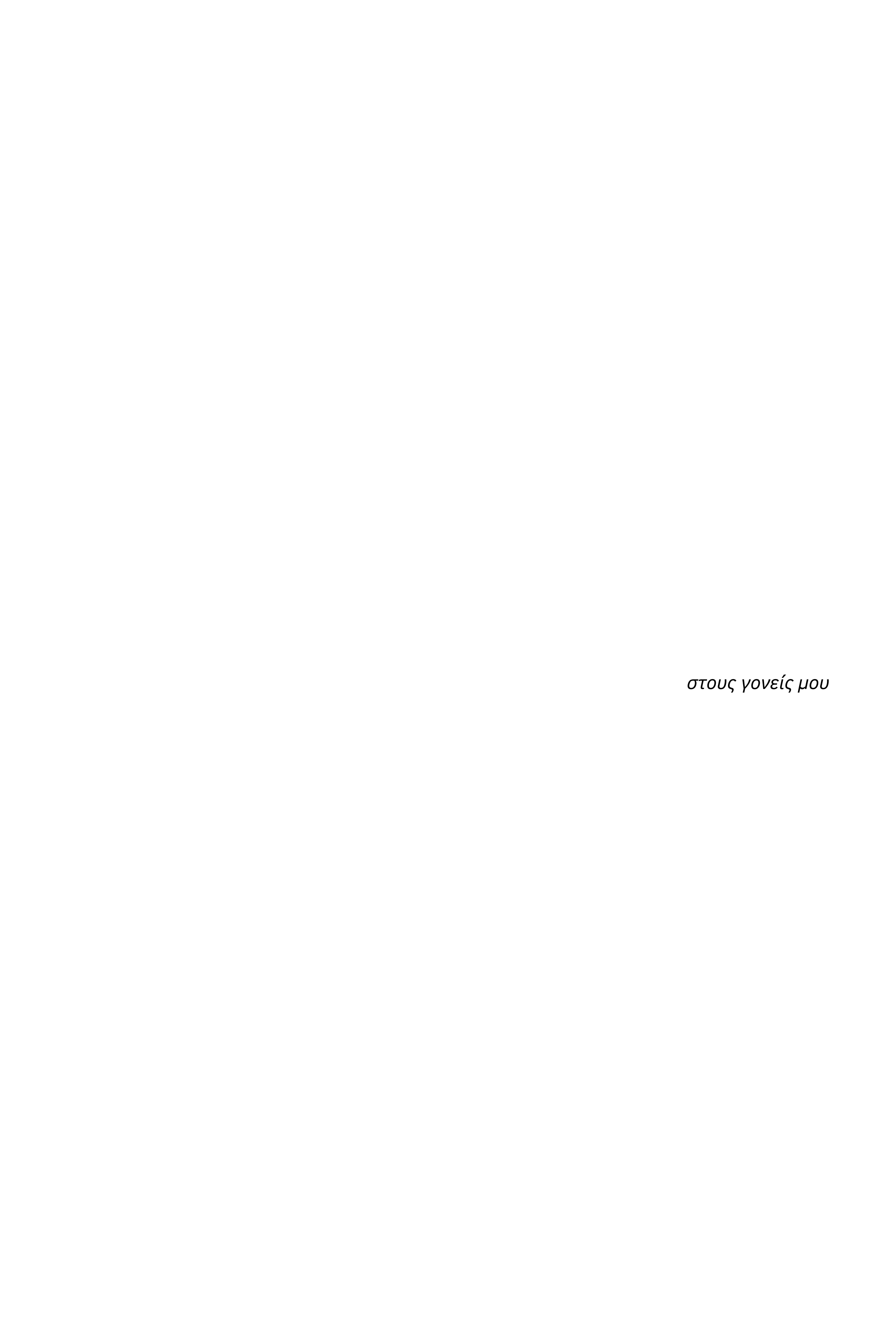
For many of the contemporary computer vision problems, there is a great desire to be able to come up with ways so that their solutions can be executed in embedded platforms. These platforms pay great attention to energy consumption, since the Internet of Things will most likely prevail in the coming years.

The goal of this thesis is to develop a CNN engine for the Myriad2 embedded processor. Specifically, the greatest advantage of this processor is the low energy consumption, per unit of computation. In order to achieve such characteristics, Myriad2 comprises of 12 VLIW processors, built around a small yet fast memory block. Implementing such an engine is not an easy task, especially when somebody is driven by performance and energy consumption as leading criteria. Naturally, CNNs require the transfer of large amount of information. This makes the task of keeping a high rate of data flow towards the 12 processors the major problem. Consequently, a careful data and source code placement design across the various memory hierarchies is required. High performance and low energy can be achieved only if some parts of the engine are implemented in assembly language, which requires delving into hardware subtleties. Finally, the fact that the system is multiprocessing increases the complexity even further.

To a great extent, the problems above arise in every embedded processor, making the suggested methodologies and approaches applicable outside the scope of Myriad2.

Keywords

Machine learning, Convolutional neural networks, Embedded systems, Multiprocessing systems, Myriad 2



στους γονείς μου

Ευχαριστίες

Θα ήθελα καταρχήν να ευχαριστήσω τον καθηγητή κ. Δημήτριο Σούντρη για την επίβλεψη αυτής της διπλωματικής εργασίας και για την ευκαιρία που μου έδωσε να την εκπονήσω στο εργαστήριο Μικροϋπολογιστών και Ψηφιακών Συστημάτων. Επίσης, ευχαριστώ ιδιαίτερα τον Δρ. Λάζαρο Παπαδόπουλο για την καθοδήγησή του και την εξαιρετική συνεργασία που είχαμε. Τέλος θα ήθελα να ευχαριστήσω τους γονείς μου για την ηθική συμπαράσταση που μου προσέφεραν όλα αυτά τα χρόνια.

Αθήνα, Σεπτέμβριος 2017

Αθανάσιος Ξύγκης

Table of Contents

Περίληψη **1**

Abstract **3**

Ευχαριστίες **7**

1 Υλοποίηση Συνελικτικών Νευρωνικών Δικτύων σε Ενσωματωμένες Αρχιτεκτονικές **19**

1.1 Λίγα λόγια για τη μηχανική μάθηση . . . 19

1.1.1 Τα τρία είδη μηχανικής μάθησης . . . 19

1.1.2 Ταξινόμηση στην επιτηρούμενη μάθηση: Προβλέποντας τις ετικέτες 20

1.1.3 Παλινδρόμηση στην επιτηρούμενη μάθηση: Προβλέποντας συνεχή αποτελέσματα . . . 21

1.2 Τεχνητά νευρωνικά δίκτυα . . . 22

1.2.1 Βιολογικό κίνητρο και συσχέτιση . . . 22

1.2.2 Αρχιτεκτονικές τεχνητών νευρωνικών δικτύων . . . 23

1.3 Συνελικτικά νευρωνικά δίκτυα . . . 25

1.3.1 Διάταξη των δεδομένων σε ένα ΣΝΔ . . . 25

1.3.2 Συνήθεις στρώσεις που χρησιμοποιούνται στα ΤΝΔ . . . 26

1.4 Caffe: Συνελικτική Αρχιτεκτονική για Γρήγορη Ενσωμάτωση Χαρακτηριστικών . . . 28

1.4.1 Εκπαίδευση ενός δικτύου . . . 28

1.5 Πολυεπεξεργαστικό SoC Myriad 2 . . . 29

1.5.1 Γενικά χαρακτηριστικά . . . 29

1.5.2 Ελεγκτής DMA της μνήμης CMX . . . 31

1.6 Βελτιστοποιήσεις που εφαρμόστηκαν . . . 32

1.7 Αξιολόγηση της υλοποίησης . . . 35

1.7.1 Τα νευρωνικά δίκτυα CIFAR10 Quick και nViso . . . 35

1.7.2 Μετρήσεις . . . 35

1.7.3 Αριθμητική ακρίβεια των υπολογισμών . . . 38

I Theory **41**

1 Background on artificial neural networks **43**

1.1 Machine learning in general 43
1.1.1 The three different types of machine learning 43
1.1.2 Classification in supervised learning: Predicting class labels 44
1.1.3 Regression in supervised learning: Predicting continuous outcomes 45
1.2 Mathematics of linear classification for images 46
1.2.1 Parameterized mapping from images to label scores 46
1.2.2 The linear classifier matrix-vector multiplication 47
1.2.3 Linear classifier: Images as high-dimensional points 48
1.2.4 The loss function 49
1.3 Artificial neural networks 50
1.3.1 Biological motivation and connections 50
1.3.2 Commonly used activation functions 51
1.3.3 ANNs architectures 52
1.3.4 Forward-step computation 54
1.4 Convolutional neural networks 55
1.4.1 Data arrangement in a CNN 55
1.4.2 Common layers used to build CNNs 56
1.4.3 Convolutional Layer 57
1.4.4 Pooling Layer 61
1.4.5 Fully-connected Layer 62

2 Introduction to Caffe and Myriad2 **63**
2.1 Caffe: Convolutional Architecture for Fast Feature Embedding 63
2.1.1 Layers 63
2.1.2 Training a network 64
2.1.3 Usage in the CNN implementation 64
2.2 Myriad 2 multiprocessor SoC 64
2.3 CMX DMA Controller 67
2.4 Myriad2 Development Kit 67
2.4.1 MDK Components 68

II Implementation **69**

3 Configuring and running a CNN architecture **71**
3.1 Description of a particular CNN 71
3.1.1 Pictorial representation of the CNN 71
3.1.2 Storage of the weights required by the CNN 73
3.1.3 Provided API 73
3.2 Detailed explanation of the API 74
3.2.1 API internals 74

4 Description of the source code peripherals 81
4.1 Memory Layout . . . 81
4.1.1 Why CMX is not enough . . . 81
4.1.2 Proposed memory map . . . 82
4.1.3 Creating the memory map in code . . . 85
4.2 Setting up Myriad2 SoC . . . 87
4.2.1 Setting up RTEMS . . . 89
4.2.2 Switching off power islands . . . 92
4.2.3 Setting up SHAVEs cache . . . 94
4.3 SHAVE code residing in CMX . . . 97
4.3.1 Memory allocator code . . . 98
4.3.2 Bootstrap code . . . 99
4.4 SHAVE code and data residing in DDR . . . 106
4.4.1 Trained parameters . . . 106
4.4.2 The Jump Table . . . 112
4.4.3 Operation of the `jumpTable` . . . 115

5 Optimization of CNN computational nodes 121
5.1 DMA CMX Driver . . . 121
5.2 Convolution . . . 123
5.2.1 Parallelization schema . . . 123
5.2.2 Convolution in assembly . . . 124
5.2.3 Optimization: Reduced number of routine calls . . . 126
5.2.4 Optimization: Reduced number of DMA transfers . . . 129
5.2.5 Optimization: DMA transfers are “hidden” in computation . . . 131
5.3 Pooling . . . 135
5.3.1 Parallelization schema . . . 135
5.3.2 Pooling in assembly . . . 136
5.3.3 Optimization: Reduced number of routine calls . . . 137
5.4 Fully connected . . . 140
5.4.1 Matrix-vector multiplication is I/O bounded . . . 140
5.4.2 Parallelization schema . . . 140
5.4.3 Optimized implementation . . . 141

III Epilogue 147

6 Evaluation of the Implementation 149
6.1 Specific CNNs used . . . 149
6.1.1 CIFAR10 Quick CNN . . . 149
6.1.2 nViso CNN . . . 149
6.2 Measurements . . . 151
6.2.1 Different input sizes . . . 151
6.2.2 Different number of SHAVEs . . . 152

6.2.3 Different frequency 158
6.3 Comparison with other implementations 159
6.4 Accuracy 161

7 Summary **165**
7.1 Conclusion 165
7.2 Future work 168

Bibliography **172**

Abbreviations **173**

List of Figures

1.1 The supervised learning approach 44

1.2 Binary classification in supervised learning 45

1.3 Linear regression in supervised learning 46

1.4 An example of mapping an image to class scores. For the sake of visualization, it is assumed that the image has only 4 pixels (4 monochrome pixels, color channels are not considered for in this example for brevity), and that there are 3 classes (red (cat), green (dog), blue (ship) class). (Clarification: in particular, the colors here simply indicate 3 classes and are not related to the RGB channels). The image pixels are stretched into a column and the matrix-vector multiplication gives the scores for each class. Note that this particular set of weights W is not good at all: the weights assign the cat image a very low cat score. In particular, this set of weights seems convinced that it is a dog. 48

1.5 Illustration of the bias trick. Doing a matrix multiplication and then adding a bias vector (left) is equivalent to adding a bias dimension with a constant of 1 to all input vectors and extending the weight matrix by 1 column - a bias column (right) 48

1.6 Cartoon representation of the image space, where each image is a single point, and three classifiers are visualized. Using the example of the car classifier (in red), the red line shows all points in the space that get a score of zero for the car class. The red arrow shows the direction of increase, so all points to the right of the red line have positive (and linearly increasing) scores, and all points to the left have a negative (and linearly decreasing) scores. 49

1.7 A cartoon drawing of a biological neuron (left) and its mathematical model (right) 51

1.8 The sigmoid $\sigma(x)$ activation function (left) and the Rectified Linear Unit (ReLU) activation function (right) 52

1.9 **Left**: A 2-layer Neural Network with three inputs, one hidden layer of 4 neurons and an output layer with 2 neurons. **Right**: A 3-layer neural network with three inputs, two hidden layers of 4 neurons each and an output layer with one neuron. Notice that in both cases there are connections (synapses) between neurons across layers, but not within a layer. 53

1.10 Example topology of a CNN suitable for handwritten digit recognition . . 55

1.11 Cross-section of an input volume of size: $4 \times 4 \times 3$. It comprises of the 3 color channel matrices of the input image. 56
1.12 **Left**: A regular 3-layer Neural Network. **Right**: A CNN arranges its neurons in three dimensions (width, height, depth), as visualized in one of the layers. Every layer of a CNN transforms the 3D input volume to a 3D output volume of neuron activations. In this example, the red input layer holds the image, so its width and height would be the dimensions of the image, and the depth would be 3 (Red, Green, Blue channels). 56
1.13 Computing the output values of a discrete convolution. The shaded regions indicate the numbers involved at each step of the computation. The result is placed in the teal colored grid. 58
1.14 An example input volume in red (e.g. a 32×32 RGB image), and an example volume of neurons in the first convolutional layer. Each neuron in the convolutional layer is connected only to a local region in the input volume spatially, but to the full depth (i.e. all color channels). Note, there are multiple neurons (5 in this example) along the depth, all looking at the same region in the input, each one contributing to the generation of a separate output feature map. 59
1.15 **Left**: The neuron strided across the input in stride of 1, giving output of size 5. **Right**: The neuron strided across the input in stride of 2, giving output of size 3. 60
1.16 An example of a zero-padded 4×4 matrix that becomes a 6×6 matrix. 60
1.17 Convolving a 3×3 kernel over a 5×5 input padded with a 1×1 border of zeros using 2×2 strides. 61
1.18 **Left**: In this example, the input volume of size $[224 \times 224 \times 64]$ is pooled with filter size 2, stride into the output volume of size $[112 \times 112 \times 64]$. Notice that the volume depth is preserved. **Right**: The most common downsampling operation is max, giving rise to max pooling, here shown with a stride of 2. That is, each max is taken over 4 numbers (little 2x2 square). 62

2.1 An MNIST digit classification example of a Caffe network, where blue boxes represent layers and yellow octagons represent data blobs produced by or fed into the layers. 64
2.2 Overview of Myriad2 hardware 65
2.3 More detailed overview of Myriad2 hardware 66
2.4 CMX DMA engine of Myriad2 68

3.1 An example CNN architecture that will be built with the CNN implementation. 71

4.1 Proposed memory map that is suitable for CNNs 83

5.1 2D striding illustration. 122
5.2 2D striding transaction with different `DST_WIDTH`, `SRC_WIDTH`. 122

5.3 Parallelization of convolution in an RGB image . . . 123
5.4 Layout of input channel in memory . . . 127
5.5 Layout of input channel in memory with pointers placed appropriately . . 128
5.6 How the convolution routine “sees” the data in memory . . . 128
5.7 Parallelization of pooling in an 3D volume . . . 136
5.8 Layout of input channel in memory . . . 138
5.9 Layout of input channel in memory with pointers placed appropriately . . 138
5.10 How the pooling routine “sees” the data in memory . . . 139
5.11 Parallelization of fully connected nodes matrix-vector multiplication . . . 141

6.1 **Left**: CIFAR10 Quick CNN for terrain classification. **Right**: nViso CNN for emotion classification from facial expressions. . . . 150
6.2 Execution time of CIFAR10 Quick CNN for various input images. . . . 151
6.3 Energy consumption of CIFAR10 Quick CNN for various input images. . . 152
6.4 Time execution of convolution with respect to the number of SHAVEs. . . 153
6.5 Scalability convolution with respect to the number of SHAVEs . . . 153
6.6 Energy consumption of convolution with respect to the number of SHAVEs. 154
6.7 Time execution of pooling with respect to the number of SHAVEs. . . . 154
6.8 Scalability pooling with respect to the number of SHAVEs. . . . 155
6.9 Energy consumption of pooling with respect to the number of SHAVEs. . 156
6.10 Time execution of fully connected with respect to the number of SHAVEs. 156
6.11 Scalability fully connected with respect to the number of SHAVEs. . . . 157
6.12 Energy consumption of fully connected with respect to the number of SHAVEs. . . . 157
6.13 Time execution of the CIFAR10 Quick CNN with respect to the SoC frequency. 158
6.14 Energy consumption of the CIFAR10 Quick CNN with respect to the SoC frequency. . . . 159
6.15 Time execution of the CIFAR10 Quick CNN on different devices and/or implementations. . . . 160
6.16 Time execution of the nViso CNN on different devices and/or implementations. 160

7.1 Optimization 1: Increasing CMX available memory space for data. . . . 165
7.2 Optimization 2: Reducing the number of required DMA transfers. . . . 166
7.3 Optimization 3: Reducing DMA transfers overhead. . . . 166
7.4 Optimization 4: Reducing the number of convolution kernel calls. . . . 167

List of Tables

3.1 CNN implementation API 74

4.1 Memory areas of Myriad2 82
4.2 CMX memory overview 84

6.1 Comparison of accuracy of CIFAR10 Quick CNN layer "ip2" between Caffe and Myriad2 implementation. 162
6.2 Comparison of accuracy of nViso CNN layer "hidden_out" between Caffe and Myriad2 implementation. 163

Κεφάλαιο 1

Υλοποίηση Συνελικτικών Νευρωνικών Δικτύων σε Ενσωματωμένες Αρχιτεκτονικές

1.1 Λίγα λόγια για τη μηχανική μάθηση

Στη σύγχρονη εποχή στην οποία ζούμε, υπάρχει αφθονία διαθέσιμων δεδομένων, τα οποία βρίσκονται σε δομημένη ή αδόμητη μορφή. Κατά το δεύτερο μισό του 20ου αιώνα, τέθηκαν οι επιστημονικές βάσεις της μηχανικής μάθησης, ενός κλάδου του πεδίου της τεχνητής νοημοσύνης, που έχει ως στόχο την ανάπτυξη αυτοδίδακτων αλγορίθμων, οι οποίοι είναι σε θέση να αντλούν γνώση από τα διαθέσιμα δεδομένα. Ειδοποιός διαφορά της μηχανικής μάθησης από τις προηγούμενες προσεγγίσεις είναι το γεγονός ότι δεν απαιτείται ανθρώπινη παρέμβαση για τη μοντελοποίηση των κανόνων που περιγράφουν τα υπό μελέτη δεδομένα. Αντιθέτως, οι ίδιοι οι αλγόριθμοι είναι σε θέση να κατασκευάζουν και να βελτιώνουν τα μοντέλα περιγραφής των δεδομένων. Εκτός από το μεγάλο ενδιαφέρον της μηχανικής μάθησης από επιστημονική άποψη, οι εφαρμογές που προσφέρει στην καθημερινή ζωή είναι επίσης αξιοσημείωτες. Για παράδειγμα, η μηχανική μάθηση είναι ο λόγος που υπάρχουν αποτελεσματικές μηχανές αναζήτησης, αξιόπιστα λογισμικά αναγνώρισης ήχου και εικόνας, ευφυέστερα αυτόνομα ρομπότ κ.α. [1].

1.1.1 Τα τρία είδη μηχανικής μάθησης

Αυτή η υποενότητα θα παρουσιάσει συνοπτικά τα τρία είδη μηχανικής μάθησης: *Επιτηρούμενη μάθηση*, *Ενισχυτική μάθηση* και *Μη επιτηρούμενη μάθηση* [1].

- *Επιτηρούμενη μάθηση*: Ο κύριος στόχος στην επιτηρούμενη μάθηση είναι η εκμάθηση ενός μοντέλου - κάνοντας χρήση δεδομένων εκπαίδευσης με ετικέτες - που επιτρέπει την πραγματοποίηση προβλέψεων σε *νέα* δεδομένα.

- *Ενισχυτική μάθηση*: Έχει ως στόχο την ανάπτυξη ενός συστήματος (*πράκτορα*) που βελτιώνει την απόδοσή του καθώς αλληλεπιδρά με το περιβάλλον. Στην ενισχυτική μάθηση, η πληροφορία σχετικά με την τρέχουσα κατάσταση του περιβάλλοντος περιλαμβάνει επιπλέον και ένα *σήμα επιβράβευσης*, που μοντελοποιεί

πόσο καλή είναι η μια δράση του πράκτορα. Κατά συνέπεια, μέσα από την αλληλεπίδρασή του με το περιβάλλον, ο πράκτορας είναι σε θέση να μάθει μια σειρά από δράσεις που έχουν ως στόχο την μεγιστοποίηση της επιβράβευσής του.

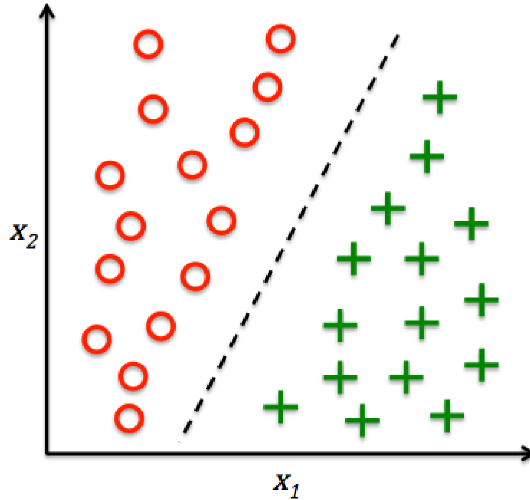
- *Μη επιτηρούμενη μάθηση*: Στην επιτηρούμενη μάθηση - κατά τη διάρκεια εκπαίδευσης του μοντέλου - η σωστή απάντηση είναι γνωστή εκ των προτέρων, ενώ στην ενισχυτική μάθηση ορίζεται ένα σήμα επιβράβευσης για τις ενέργειες του πράκτορα. Ωστόσο, στην μη επιτηρούμενη μάθηση, τα δεδομένα είτε δεν διαθέτουν ετικέτες, είτε έχουν *άγνωστη* δομή. Κάνοντας χρήση των τεχνικών αυτής της μορφής μάθησης, υπάρχει η δυνατότητα εξερεύνησης της δομής των δεδομένων, με στόχο την εξαγωγή χρήσιμης πληροφορίας, δίχως την παροχή γνώσης που απαιτείται στις δύο παραπάνω κατηγορίες μηχανικής μάθησης.

1.1.2 Ταξινόμηση στην επιτηρούμενη μάθηση: Προβλέποντας τις ετικέτες

Η ταξινόμηση είναι μια υποκατηγορία της επιτηρούμενης μάθησης, στην οποία στόχος είναι η πρόβλεψη της κλάσης/κατηγορίας ενός νέου αντικειμένου, βάσει παλαιότερων παρατηρήσεων [2]. Η ταξινόμηση ενός αντικειμένου σε μια κατηγορία γίνεται με την ανάθεση μιας ετικέτας σε αυτό, με τις ετικέτες να είναι διακριτές και μη διατεταγμένες τιμές.

Το μοντέλο πρόβλεψης το οποίο μαθαίνει ένας αλγόριθμος επιτηρούμενης μηχανικής μάθησης μπορεί να αναθέσει οποιαδήποτε ετικέτα, που εμφανίστηκε στο σύνολο δεδομένων κατά τη διάρκεια της εκπαίδευσης, σε ένα νέο μη ταξινομημένο αντικείμενο. Ένα τυπικό παράδειγμα είναι η αναγνώριση χειρόγραφων χαρακτήρων. Σε αυτή την περίπτωση, ένα σύνολο δεδομένων εκπαίδευσης που περιέχει χειρόγραφα παραδείγματα για κάθε γράμμα του αλφάβητου είναι ένα σημείο εκκίνησης. Έπειτα, εάν ο χρήστης εισάγει ένα χειρόγραφο χαρακτήρα, μέσω μιας συσκευής εισόδου, το μοντέλο πρόβλεψης θα είναι σε θέση να εκτιμήσει το σωστό γράμμα του αλφάβητου με κάποια ακρίβεια. Ωστόσο, ο αλγόριθμος δε θα είναι σε θέση να αναγνωρίσει επιτυχώς οποιοδήποτε από τα ψηφία μηδέν εώς εννέα, εάν αυτά δεν υπάρχουν στο σύνολο εκπαίδευσής του.

Το σχήμα 1.1 αποτυπώνει την ιδέα της *δυαδικής ταξινόμησης*, υποθέτοντας ότι έχουν δοθεί 30 δείγματα κατά το στάδιο της εκπαίδευσης: 15 δείγματα έχουν την ετικέτα της αρνητικής κατηγορίας (κύκλοι) και 15 δείγματα έχουν την ετικέτα της θετικής κατηγορίας (σύμβολο συν). Σε αυτό το σενάριο, το σύνολο δεδομένων είναι δισδιάστατο, το οποίο σημαίνει ότι κάθε δείγμα εμπεριέχει δύο τιμές: x_1 και x_2. Ένας αλγόριθμος (επιτηρούμενης) μηχανικής μάθησης μπορεί να χρησιμοποιηθεί για να μάθει ένα κανόνα - το σύνορο της απόφασης που αναπαρίσταται με τη μαύρη διακεκομμένη γραμμή - που διαχωρίζει τις δύο κατηγορίες και ταξινομεί τα νέα δεδομένα σε μία από τις κατηγορίες δεδομένων των τιμών x_1 και x_2 [1].

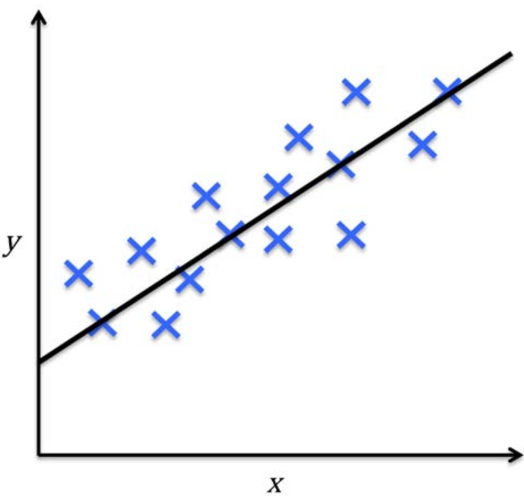


Σχήμα 1.1: *Δυαδική ταξινόμηση στην επιτηρούμενη μάθηση*

1.1.3 Παλινδρόμηση στην επιτηρούμενη μάθηση: Προβλέποντας συνεχή αποτελέσματα

Η προηγούμενη υποενότητα έδειξε πως ο στόχος της ταξινόμησης είναι η ανάθεση μη διατεταγμένων ετικετών σε αντικείμενα [2]. Ένα δεύτερο είδος της επιτηρούμενης μάθησης είναι η πρόβλεψη συνεχών αποτελεσμάτων, που είναι γνωστό στον κλάδο της στατιστικής ως *ανάλυση παλινδρόμησης* [2]. Στην ανάλυση παλινδρόμησης, υποθέτουμε ότι δίνονται ένα πλήθος από μεταβλητές *πρόβλεψης* και μια συνεχής μεταβλητή απόκρισης (*αποτέλεσμα*). Στόχος είναι η εύρεση μια σχέσης μεταξύ των μεταβλητών πρόβλεψης που επιτρέπει την πρόβλεψη ενός αποτελέσματος. Για παράδειγμα, ας υποθέσουμε ότι θέλουμε να προβλέψουμε τους βαθμούς των μαθητών στο διαγώνισμα των μαθηματικών. Εάν υπάρχει μια σχέση μεταξύ του χρόνου που αφιερώθηκε στη μελέτη για το διαγώνισμα και των βαθμών που έλαβαν οι μαθητές που το έγραψαν, θα μπορούσε να χρησιμοποιηθεί ως σύνολο δεδομένων εκπαίδευσης για την εκμάθηση ενός μοντέλου. Το μοντέλο, δεδομένου του χρόνου μελέτης που σκοπεύει να επενδύσει ένας μελλοντικός μαθητής, προβλέπει το βαθμό που θα λάβει στο συγκεκριμένο διαγώνισμα.

Το σχήμα 1.2 αποτυπώνει την ιδέα της *γραμμικής παλινδρόμησης*. Δεδομένων μιας μεταβλητής πρόβλεψης x και μιας μεταβλητής απόκρισης y, σχεδιάζεται μια ευθεία γραμμή που “ταιριάζει” στα δεδομένα και ελαχιστοποιεί την απόσταση - που συνήθως είναι η μέση τιμή του τετραγώνου της απόστασης - μεταξύ των δειγμάτων και της γραμμής. Έπειτα, αυτή η γραμμή χρησιμοποιείται για να προβλέψει τη μεταβλητή απόκρισης σε νέα δεδομένα [1].



Σχήμα 1.2: *Γραμμική παλινδρόμηση στην επιτηρούμενη μάθηση*

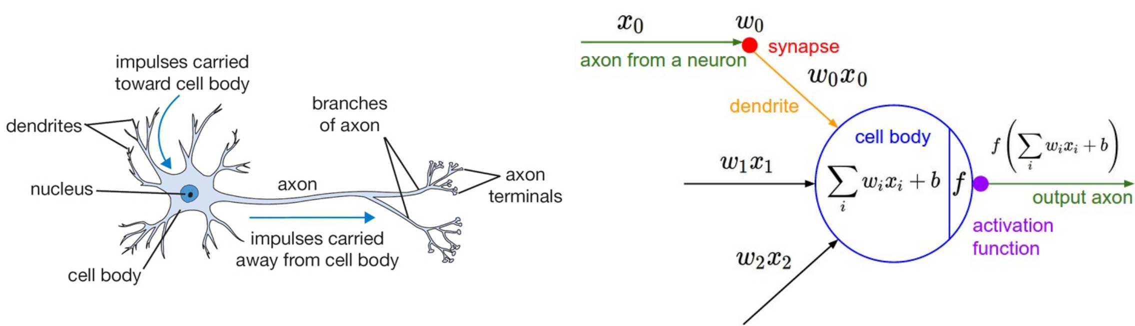
1.2 Τεχνητά νευρωνικά δίκτυα

Τα Τεχνητά Νευρωνικά Δίκτυα (ΤΝΔ) είναι δίκτυα εμπνευσμένα από βιολογικούς οργανισμούς, που συνδέονται με ένα συγκεκριμένο τρόπο, ανάλογα με τις απαιτήσεις της εκάστοτε εφαρμογής. Ένα από τα κυριότερα πλεονεκτήματα των τεχνητών νευρωνικών δικτύων είναι ότι απαιτούν ελάχιστη ή καμία προεπεξεργασία των δεδομένων εισόδου. Από την άλλη μεριά, οι παραδοσιακοί περίτεχνοι μηχανισμοί εξαγωγής χαρακτηριστικών (feature extractors) ρυθμίζονται χειροκίνητα για το συγκεκριμένο - υπό μελέτη - σύνολο δεδομένων. Τα μοντέλα τεχνητών νευρωνικών δικτύων έχουν τη δυνατότητα να μαθαίνουν και να γενικεύουν χρησιμοποιώντας παραδείγματα. Κατά συνέπεια, αυτή η ικανότητα προσαρμογής σε συγκεκριμένη εργασία αναγνώρισης, ακόμα και μετά το στάδιο σχεδιασμού τους, τα κάνει μοναδικά, συγκριτικά με άλλες τεχνικές της τεχνητής νοημοσύνης.

1.2.1 Βιολογικό κίνητρο και συσχέτιση

Η βασική μονάδα υπολογισμών του εγκεφάλου είναι ο νευρώνας [2]. Υπάρχουν περίπου 86 δισεκατομμύρια νευρώνες στο ανθρώπινο νευρικό σύστημα και συνδέονται μεταξύ τους με περίπου 10^{14} - 10^{15} νευρικές συνάψεις. Το σχήμα 1.3 δείχνει μια απλοποιημένη απεικόνιση ενός βιολογικού νευρώνα στα αριστερά και του συνήθους μαθηματικού μοντέλου στα δεξιά. Κάθε νευρώνας λαμβάνει σήματα εισόδου από τους δενδρίτες και παράγει σήματα εξόδου κατά τα μήκος του (ενός) άξονά του. Στο υπολογιστικό μοντέλο του νευρώνα, τα σήματα που ταξιδεύουν στον άξονα (π.χ. x_0) αλληλεπιδρούν πολλαπλασιαστικά (π.χ. $w_0 x_0$) με τους δενρίτες του άλλου νευρώνα, ανάλογα με τη δύναμη της σύναψης (π.χ. w_0). Η ιδέα είναι ότι η δύναμη των συνάψεων (τα βάρη w) είναι μεταβλητά, μπορούν να εκπαιδευτούν και ελέγχουν την επιρροή ενός νευρώνα σε ένα άλλο νευρώνα. Στο βασικό βιολογικό μοντέλο, οι δενδρίτες μεταφέρουν το σήμα το σώμα του κυττάρου, όπου πραγματοποιείται η άθροιση. Εάν το τελικά άθροισμα εί-

ναι πάνω από κάποιο κατώφλι, ο νευρώνας πυροδοτεί, στέλνοντας ένα σύντομο σήμα κατά μήκος του άξονά του. Στο υπολογιστικό μοντέλο, γίνεται η υπόθεση ότι οι ακριβείς χρόνοι του σύντομου σήματος δεν έχουν σημασία, και ότι μόνο η συχνότητα των πυροδοτήσεων μεταδίδει πληροφορία. Βάσει αυτής της ερμηνείας, ο ρυθμός πυροδότησης του νευρώνα μοντελοποιείται με μια συνάρτηση ενεργοποίησης *f*, που αναπαριστά τη συχνότητα με την οποία τα σύντομα σήματα ταξιδεύουν στον άξονα. Ιστορικά, μία συνήθης επιλογή για τη συνάρτηση ενεργοποίησης είναι η *σιγμοειδής* συνάρτηση σ, που λαμβάνει ως είσοδο ένα πραγματικό αριθμό (το σήμα μετά την άθροιση) και το περιορίζει το εύρος μεταξύ 0 και 1.



Σχήμα 1.3: *Απλοποιημένη απεικόνιση του βιολογικού νευρώνα (αριστερά) και του μαθηματικού του μοντέλου (δεξιά)*

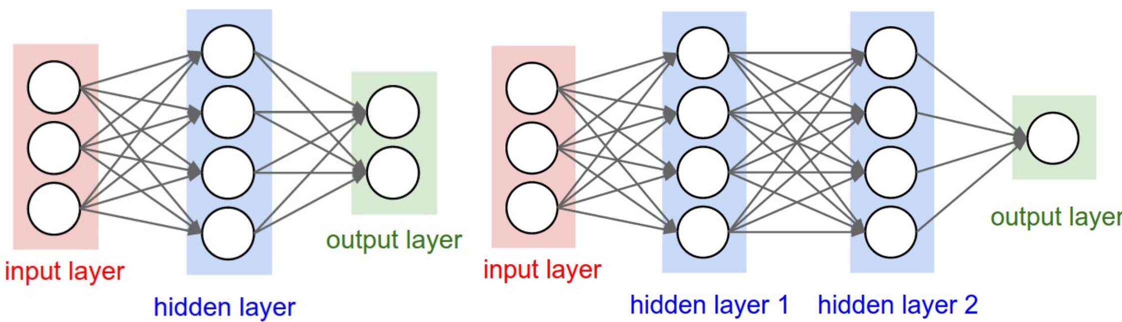
Με άλλα λόγια, κάθε νευρώνας πραγματοποιεί ένα εσωτερικό γινόμενο της εισόδου και των βαρών, προσθέτει την τιμή πόλωσης και εφαρμόζει μια μη γραμμική συνάρτηση (γνωστή και ως συνάρτηση ενεργοποίησης). Σε αυτή την περίπτωση, η συνάρτηση ενεργοποίησης είναι η σιγμοειδής συνάρτηση $\sigma(x) = 1/(1 + e^{-x})$.

Πρέπει να τονιστεί ότι το μοντέλο του νευρώνα που παρουσιάστηκε είναι πολύ χονδροειδές. Για παράδειγμα, υπάρχουν διαφορετικά είδη νευρώνων, καθένα με διαφορετικές ιδιότητες. Οι δενδρίτες στους βιολογικούς νευρώνες μπορούν να πραγματοποιήσουν πολύπλοκους μη γραμμικούς υπολογισμούς. Οι νευρωνικές συνάψεις δεν είναι απλώς ένα βάρος, άλλα πολύπλοκα μη γραμμικά δυναμικά συστήματα. Εξαιτίας όλων αυτών των απλουστεύσεων, τα ΤΝΔ είναι απλώς εμπνευσμένα από του πραγματικούς νευρώνες και δεν προσπαθούν να τους προσομοιώσουν.

1.2.2 Αρχιτεκτονικές τεχνητών νευρωνικών δικτύων

Τα τεχνητά νευρωνικά δίκτυα μπορούν να αναπαρασταθούν ως γράφοι [3]. Συγκεκριμένα, μοντελοποιούνται ως μια συλλογή από νευρώνες που συνδέονται μεταξύ τους, σχηματίζοντας ένα ακυκλικό γράφο. Με άλλα λόγια, οι έξοδοι κάποιων νευρώνων μπορεί να είναι είσοδοι σε κάποιους άλλους νευρώνες. Οι κύκλοι δεν επιτρέπονται, καθώς αυτό θα υπονοούσε ένα ατέρμονο βρόχο κατά την τροφοδότηση δεδομένων στο δίκτυο. Αντί για άμορφες διατάξεις νευρώνων, τα μοντέλα των ΤΝΔ οργανώνονται συνήθως σε διακεκριμένες στρώσεις νευρώνων. Στα κανονικά νευρωνικά δίκτυα, η πιο συνηθισμένος τύπος στρώσης είναι η *πλήρως συνδεδεμένη* στρώση, στην οποία οι

νευρώνες μεταξύ δύο γειτονικών στρώσεων είναι πλήρως συνδεδεμένοι, αλλά οι νευρώνες που ανήκουν στην ίδια στρώση δεν έχουν μεταξύ τους συνδέσεις. Στο σχήμα 1.4 απεικονίζονται δύο τοπολογίες ΤΝΔ που χρησιμοποιούν πλήρως συνδεδεμένες στρώσεις.

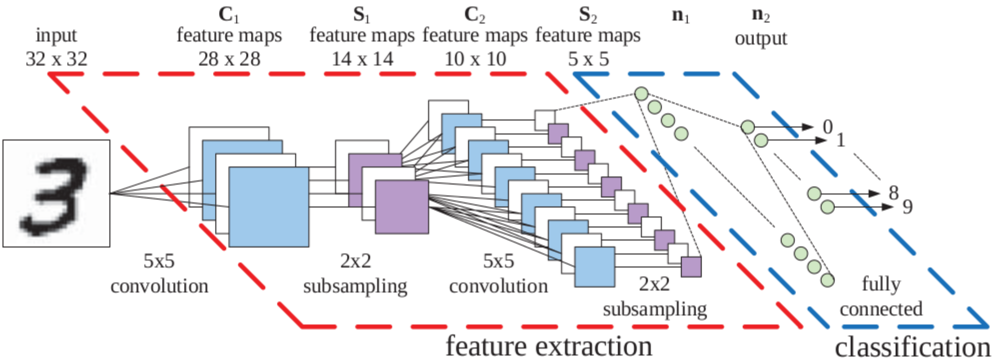


Σχήμα 1.4: ***Αριστερά****: Ένα ΤΝΔ δύο στρώσεων με 3 εισόδους, μια κρυφή στρώση 4 νευρώνων και την στρώση εξόδου με 2 νευρώνες.* ***Δεξιά****: Ένα ΤΝΔ τριών στρώσεων με 3 εισόδους, 2 κρυφές στρώσεις με 4 νευρώνες η καθεμία και τη στρώση εξόδου με 1 νευρώνα.*

Τα μοντέρνα συνελικτικά νευρωνικά δύκτια που παρουσιάζονται στη συνέχεια περιέχουν πλήθος παραμέτρων της τάξης των 100 εκατομμυρίων και αποτελούνται από 10-20 στρώσεις. Ωστόσο, το πλήθος των ουσιαστικών συνδέσεων είναι σημαντικά μεγαλύτερο, εξαιτίας του γεγονότος ότι πολλές από τις παραμέτρους μοιράζονται μεταξύ των συνδέσεων.

1.3 Συνελικτικά νευρωνικά δίκτυα

Ένα Συνελικτικό Νευρωνικό Δίκτυο (ΣΝΔ) αποτελεί μιας ειδική τοπολογία τεχνητού νευρωνικού δικτύου, η οποία είναι εμπνευσμένη από τον οπτικό εγκεφαλικό φλοιό των ζώων. Οι παράμετροι αυτών των τοπολογιών ρυθμίστηκαν κατάλληλα από τον Yann LeCun στις αρχές του 1990 [4], ώστε να μπορέσουν τα λύσουν προβλήματα στην όραση υπολογιστών. Επί της ουσίας, ένα ΣΝΔ είναι ένα μοντέλου ΤΝΔ που έχει σχεδιαστεί αποκλειστικά για την αναγνώριση δισδιάστατων αντικειμένων, παρουσιάζοντας υψηλό βαθμό αναλλοίωτης συμπεριφοράς κατά την μετάθεση, κλιμάκωση, στρέβλωση και άλλες παραμορφώσεις της εισόδου.



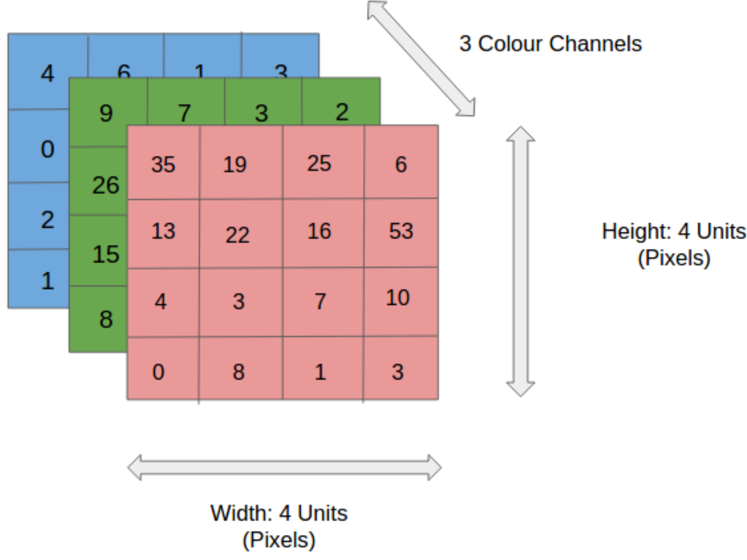
Σχήμα 1.5: *Παράδειγμα τοπολογίας ένος ΣΝΔ κατάλληλου για αναγνώριση χειρόγραφων ψηφίων*

Σε ένα ΣΝΔ κάθε νευρώνας λαμβάνει κάποιες εισόδους, πραγματοποιεί μια μαθηματική πράξη και προαιρετικά τη συνοδεύει από μια γραμμικότητα. Όλο το δίκτυο έχει ως στόχο να ταξινομήσει μια εικόνα σε κάποια κατηγορία βάσει των εικονοστοιχείων της. Όλη η γνώση που εφαρμόστηκε στα ΤΝΔ εξακολουθεί να ισχύει και να εφαρμόζεται στα ΣΝΔ. Ειδοποιός διαφορά των αρχιτεκτονικών ΣΝΔ είναι το γεγονός ότι κάνουν τη ρητή υπόθεση ότι η είσοδος είναι μια εικόνα, γεγονός που επιτρέπει την κωδικοποίηση ορισμένων ιδιοτήτων στην αρχιτεκτονική τους.

1.3.1 Διάταξη των δεδομένων σε ένα ΣΝΔ

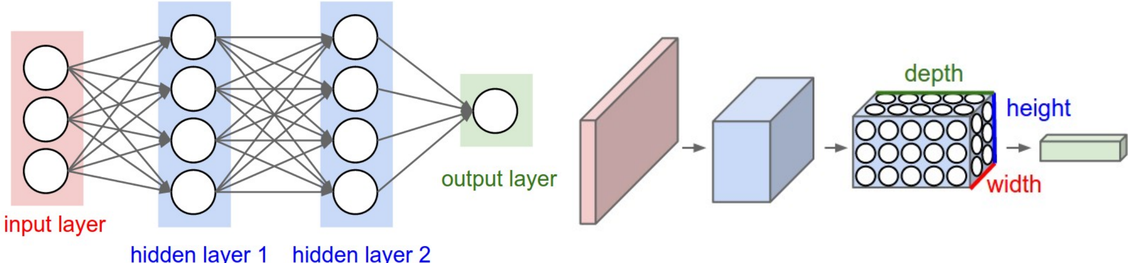
Τα ΣΝΔ εκμεταλλεύονται το γεγονός ότι η είσοδος αποτελείται από εικόνες και περιορίζουν την αρχιτεκτονική τους με τρόπο που να έχει περισσότερο νόημα [5]. Συγκεκριμένα, σε αντίθεση με ένα κανονικό ΤΝΔ, οι στρώσεις ενός ΣΝΔ οργανώνονται σε 3 διαστάσεις: πλάτος, ύψος, βάθος. Για παράδειγμα, 32×32 RGB εικόνες εισόδου αναπαριστούν ένα όγκο εισόδου που έχει διαστάσεις $32 \times 32 \times 3$ (πλάτος, ύψος, βάθος αντίστοιχα). Οι νευρώνες σε μία στρώση είναι συνδεδεμένοι μόνο με μια μικρή περιοχή της προηγούμενης στρώσης, σε αντίθεση με όλους τους νευρώνες της προηγούμενης στρώσης που συναντάται στις πλήρως συνδεδεμένες στρώσεις. Η περιορισμένη, τοπική συνδεσιμότητα των ΣΝΔ θα αποσαφηνιστεί σύντομα. Ένας 3Δ όγκος εισόδου απεικονίζεται στο σχήμα 1.6 για μια 4×4 RGB εικόνα. Σημειώστε ότι το εύρος των τιμών στον 3Δ όγκο εισόδου μπορεί να ρυθμιστεί, ώστε να βοηθήσει στην εκπαίδευση

του δικτύου. Η διαδικασία μετασχηματισμού των δεδομένων εισόδου - προτού αυτά τροφοδοτηθούν στο ΣΝΔ - ώστε να έχουν την επιθυμητή μορφή, λέγεται *προεπεξεργασία*.



Σχήμα 1.6: *Διατομή ενός όγκου δεδομένων εισόδου* $4 \times 4 \times 3$. *Αποτελείται από 3 μήτρες, που αντιστοιχούν στα 3 κανάλια της εικόνας.*

Κατά συνέπεια, ένα ΣΝΔ αποτελείται από στρώσεις, με την καθεμία να έχει μια απλή διεπαφή. Μετασχηματίζει ένα 3Δ όγκο εισόδου σε ένα 3Δ όγκο εξόδου, χρησιμοποιώντας συναρτήσεις που μπορεί να έχουν ή όχι παραμέτρους. Μια απεικόνιση παρουσιάζεται στο σχήμα 1.7.



Σχήμα 1.7: ***Αριστερά**: Ένα κανονικό ΤΝΔ 3 στρώσεων. **Δεξιά**: Ένα ΣΝΔ οργανώνει τους νευρώνες του σε τρεις διαστάσεις, όπως παρουσιάζεται σε μία από τις στρώσεις. Κάθε στρώση του ΣΝΔ μετασχηματίζει τον 3Δ όγκο εισόδου σε ένα 3Δ όγκο εξόδου από νευρώνες ενεργοποίησης. Σε αυτό το παράδειγμα, η κόκκινη στρώση εισόδου εμπεριέχει την εικόνα, συνεπώς το πλάτος και το ύψος του όγκου πρέπει να αντιστοιχεί στις διαστάσεις της εικόνας. Το βάθος του όγκου θα είναι 3, υποθέτοντας ότι η είσοδος είναι μια RGB εικόνα.*

1.3.2 Συνήθεις στρώσεις που χρησιμοποιούνται στα ΤΝΔ

Υπάρχουν τρία είδη στρώσεων που χρησιμοποιούνται για τη δόμηση αρχιτεκτονικών ΣΝΔ: *Συνελικτικές στρώσεις*, *Pooling Layer*, and *Πλήρως συνδεδεμένες στρώσεις*

[3]. Η τελευταία είναι ίδια με αυτή που συναντά κανείς στα κανονικά ΤΝΔ. Αυτές οι τρεις στρώσεις στοιβάζονται σε μια αλληλουχία, ώστε να παράξουν ενδιαφέρουσες αρχιτεκτονικές. Υπάρχει επίσης και η *Στρώση Εισόδου*, που δεν είναι τίποτε περισσότερο από τον ταυτοτικό μετασχηματισμό, δηλαδή η έξοδός της είναι η ίδια με την είσοδό της. Αυτές οι στρώσεις αναλύονται παρακάτω:

- *Στρώση Εισόδου*: Εμπεριέχει τα δεδομένα εισόδου, που μπορεί είναι οι τιμές των εικονοστοιχείων της εικόνας εισόδου ή το αποτέλεσμα της προεπεξεργασίας αυτών. Το βάθος της στρώσης εισόδου πρέπει να ίδιο με το πλήθος των καναλιών της εικόνας εισόδου.

- *Συνελικτική Στρώση*: Θα υπολογίσει την έξοδο των νευρώνων που είναι συνδεδεμένοι με τοπικές περιοχές της εισόδου. Κάθε υπολογισμός γίνεται σε ένα μικρό παράθυρο στην πρόσοψη (που ορίζει το πλάτος και το ύψος) του όγκου εισόδου, αλλά σε όλο το βάθος του όγκου εισόδου. Το βάθος του όγκου εισόδου εξαρτάται από το πλήθος των φίλτρων που παρέχονται στην στρώση ως μια επιπλέον παράμετρος.

- *Στρώση Pooling*: Θα πραγματοποιήσει μια υποδειγματοληψία ή/και εξομάλυνση κατά μήκος του βάθους του όγκου εισόδου.

- *Πλήρως συνδεδεμένη Στρώση*: Όπως είναι ήδη γνωστό από το ΤΝΔ, κάθε νευρώνας σε αυτή τη στρώση συνδέεται με όλα τα στοιχεία του όγκου εισόδου της στρώσης.

- *Στρώση ReLU*: Εφαρμόζει μια συνάρτηση ενεργοποίηση στοιχείο προς στοιχείο, την $f(x) = max(0, x)$ που εμφανίζει κατώφλι στο μηδέν. Αυτή η στρώση χρησιμοποιείται με σκοπό την υποβοήθηση της εκπαίδευσης του δικτύου και παράγει στην έξοδο ένα όγκο που έχει διαστάσεις ίδιες με αυτές του όγκου εισόδου.

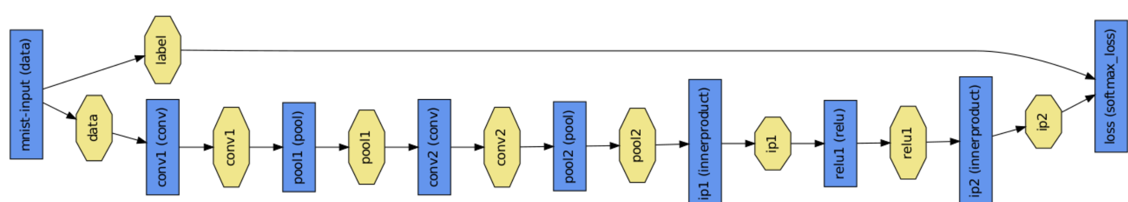
Κατ' αυτό τον τρόπο, τα ΣΝΔ μετασχηματίζουν την εικόνα εισόδου καθώς αυτή διασχοίζει τις στρώσεις του. Έτσι μετατρέπει την είσοδο από εικονοστοιχεία σε ετικέτες που αντιπροσωπεύουν την κατηγορία στην οποία εκτιμάται ότι ανήκουν. Επισημαίνεται ότι κάποιες στρώσεις περιέχουν/απαιτούν παραμέτρους, ενώ άλλες όχι. Συγκεκριμένα, η συνελικτική και πλήρως συνδεδεμένη στρώση μετασχηματίζει την είσοδο απαιτώντας επιπλέον τις τιμές των βαρών (και της πόλωσης) διασυνδέσεων των νευρώνων. Από την άλλη μεριά, η ReLU και Pooling στρώση εφαρμόζει μια γνωστή μη παραμετρική συνάρτηση. Οι παράμετροι των συνελικτικών και πλήρως συνδεδεμένων στρώσεων εκπαιδεύονται αξιοποιώντας τη μαθηματική τεχνική βελτιστοποίησης της κατάβασης με τη μέθοδο της κλίσης.

1.4 Caffe: Συνελικτική Αρχιτεκτονική για Γρήγορη Ενσωμάτωση Χαρακτηριστικών

Το Caffe είναι ένα συγκροτημένο και τροποποιήσιμο πλαίσιο λογισμικού, που παρέχει στους χρήστες του μια σειρά από αλγορίθμους μηχανικής μάθησης, καθώς επίσης και μια συλλογή από μοντέλα αναφοράς. Αυτό το λογισμικό υποστηρίζει την εκπαίδευση και εκτέλεση πληθώρας συνελικτικών δικτύων γενικού σκοπού, δίνοντας έμφαση στην αποδοτικότητα και την ταχύτητα. Το Caffe συντηρείται και αναπτύσσεται από το Berkeley Vision and Learning Center (BVLC) και αποτελεί κεντρικό εργαλείο σε ερευνητικά πρότζεκτ ή βιομηχανικές εφαρμογές μεγάλης κλίμακας στους τομείς της όρασης υπολογιστών, της επεξεργασίας φυσικής γλώσσας και των πολυμέσων [6].

1.4.1 Εκπαίδευση ενός δικτύου

Το Caffe εκπαιδεύει τα μοντέλα χρησιμοποιώντας τον αλγόριθμο γρήγορης ή κανονικής στοχαστικής κατάβασης με τη μέθοδο της κλίσης. Το σχήμα 1.8 παρουσιάζει ένα τυπικό παράδειγμα δικτύου (για την ταξινόμηση ψηφίων από το σύνολο δεδομένων MNIST) κατά τη φάση της εκπαίδευσης [6]: μια στρώση δεδομένων λαμβάνει τις εικόνες και τις ετικέτες τους από το σκληρό δίσκο, τροφοδοτεί τα δεδομένα διαμέσου πολλαπλών στρώσεων, όπως είναι η συνέλιξη, και τροφοδοτεί το τελικό αποτέλεσμα πρόβλεψης σε μια στρώση κατηγοριοποίησης. Αυτή η στρώση υπολογίζει το σφάλμα κατηγοριοποίησης και τις κλίσεις που εκπαιδεύουν όλο το δίκτυο, με σκοπό τη βελτιστοποίηση των παραμέτρων του. Το συγκεκριμένο παράδειγμα μπορεί να βρεθεί στον πηγαίο κώδικα του Caffe, στη διαδρομή `examples/lenet/lenet_train.prototxt`. Η επεξεργασία των δεδομένων γίνεται σε μικρές ομάδες που τροφοδοτούνται στο δίκτυο σειριακά. Κρίσιμο στη εκπαίδευση είναι η ρύθμιση του ρυθμού εξασθένησης της εκμάθησης, η ορμή και τα στιγμιότυπα. Τα τελευταία επιτρέπουν τη παύση και συνέχιση της εκπαίδευσης του δικτύου.



Σχήμα 1.8: *Παράδειγμα κατηγοριοποίησης ψηφίων του συνόλου δεδομένων MNIST, όπου τα μπλε ορθογώνια αναπαριστούν στρώσεις και τα κίτρινα ορθογώνια αναπαριστούν τα δεδομένα που παράγονται και τροφοδοτούνται σε αυτές.*

1.5 Πολυεπεξεργαστικό SoC Myriad 2

1.5.1 Γενικά χαρακτηριστικά

Για την υλοποίηση του συστήματος εκτέλεσης συνελικτικών δικτύων επιλέχθηκε η ενσωματωμένη πλατφόρμα Myriad2 [7]. Το συγκεκριμένο SoC αναπτύσσεται από τη Movidius Ltd, η οποία πρόσφατα έγινε μέλος της ομάδας Perceptual Computing Group της Intel, με στόχο την επίσπευση της δημιουργίας ευφυών συσκευών σε εφαρμογές όρασης υπολογιστών. Η Myriad2 καταφέρνει να προσφέρει υψηλή απόδοση σε εφαρμογές της όρασης υπολογιστών, κάτω από εξαιτερικά περιοριστικές συνθήκες κατανάλωσης ισχύος. Γι' αυτό το λόγο, αποτελεί τον πρώτο επεξεργαστή όρασης υπολογιστών - Vision Processing Unit (VPU) - που στοχεύει στην επιτάχυνση ενσωματωμένων εφαρμογών. Τα κυριότερα χαρακτηριστικά της Myriad2 είναι:

- *Σχεδιασμός πολύ χαμηλής ισχύος*: Κάνοντάς τη κατάλληλη για χρήση σε φορητές συσκευές, που η αυτονομία της μπαταριάς είναι κυρίαρχη παράμετρος.

- *Επεξεργαστής υψηλής απόδοσης*: Δίνοντας τη δυνατότητα εκτέλεσης των υπολογιστικά απαιτητικών σύγχρονων εφαρμογών της όρασης υπολογιστών.

- *Ευέλικτη αρχιτεκτονική*: Παρέχοντας πρόσβαση στις λεπτομέρειες της αρχιτεκτονικής, οι προγραμματιστές είναι σε θέση να βελτιστοποιήσουν τις εφαρμογές τους ακόμα περισσότερο.

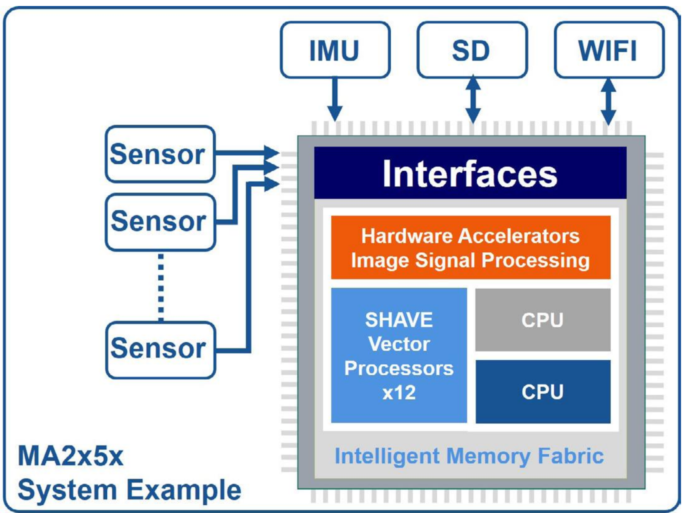
- *Μικρές φυσικές διαστάσεις*: Ώστε να είναι εφικτή η ενσωμάτωση της ψηφίδας σε οποιαδήποτε φορητή συσκευή.

Μια περιγραφή υψηλού επιπέδου του υλικού της Myriad2 δίνεται στο σχήμα 1.9. Από αυτό, είναι φανερό πως το συγκεκριμένο SoC διαθέτει 14 επεξεργαστές. Οι δύο επεξεργαστές στα δεξιά του σχήματος είναι θεμελιωδώς διαφορετικοί από τους 12 επεξεργαστές στα αριστερά. Πιο αναλυτικά, οι επεξεργαστές με το όνομα "CPU" υλοποιούν την 32-bit SPARC αρχιτεκτονική, που ανήκει στην οικογένεια επεξεργαστών RISC.

Μια λεπτομερέστερη περιγραφή ακολουθεί στη συνέχεια [8]:

- *Leon OS*: Είναι ο ένας από τους επεξεργαστές SPARC. Ανηκεί στο υποσύστημα CPU sub-system (CSS) που έχει σχεδιαστεί ώστε να είναι η κύρια μονάδα επικοινωνίας και ελέγχου με τον εξωτερικό κόσμο, όντας εφοδιασμένο με τα ακόλουθα περιφερειακά συστήματα επικοινωνίας: I2C, I2S, SPI, UART, GPIO, ETH και USB3.0. Η μονάδα ελέγχου του CSS είναι ο επεξεργαστής Leon OS (LOS), που διαθέτει αρκετά μεγάλες κρυφές μνήμες L1 (32 KB) και L2 (256 KB), επιτρέποντας τη δυνατότητα εκτέλεσης ενός μοντέρνου λειτουργικού συστήματος πραγματικού χρόνου (RTOS).

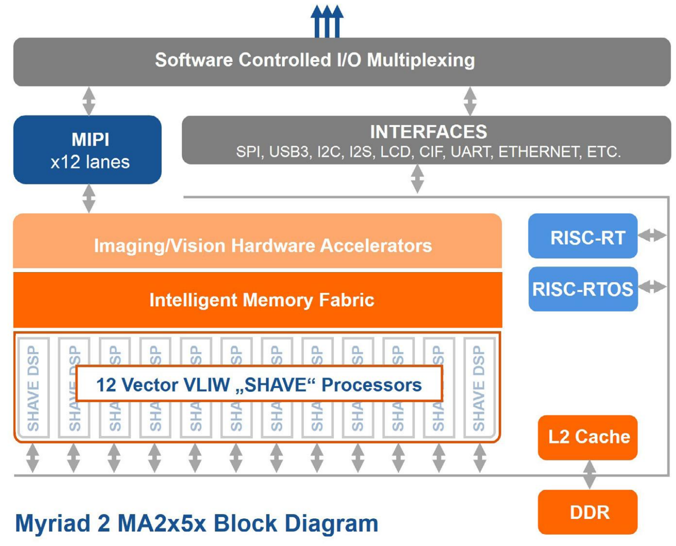
- *Leon RT*: Αποτελεί το δεύτερο από τους επεξεργαστές SPARC. Ανήκει στο υποσύστημα Media sub-system (MSS), μια δομική μονάδα που επιτρέπει την διασύνδεση με συσκευές εικόνας, όπως αισθητήρες εικόνας, οθόνες LCD, ελεγκτές HDMI



Σχήμα 1.9: *Επισκόπηση του υλικού της Myriad2*

κ.λπ.. Ταυτόχρονα, το MSS είναι υπεύθυνο για το έλεγχο φίλτρων (όπως π.χ. το DeBayer) υλοποιημένων στο υλικό, που διατείθενται από την Myriad2.

- *SIPP*: Πρόκειται για ένα ιδιοταγή μηχανισμό υλικού/λογισμικού που χρησιμοποιείται απο τη Myriad2, με σκοπό την αποδοτική δρομολόγηση εργασιών ψηφιακής επεξεργασίας εικόνας. Αυτός ο μηχανισμός είναι βασισμένος σε επεξεργασία μορφής σωλήνωσης και χρησιμοποιεί τα φίλτρα υλικού που παρέχονται από την Myriad2, ώστε να επιτύχει την ταχύτερη δυνατή εκτέλεση. Το υποσύστημα SIPP απεικονίζεται στο σχήμα 1.9 με πορτοκαλί χρωμα.



Σχήμα 1.10: *Λεπτομερέστερη επισκόπηση του υλικού της Myriad2*

- *Microprocessor Array (UPA)*: Είναι η αρχιτεκτονική μονάδα της Myriad2 που συγκροτείται από τους 12 Very Long Instruction Word (VLIW) διανυσματικούς επεξεργαστές SHAVE (βλ. σχήμα 1.10), τη CMX μνήμη SRAM μεγέθους 2 MB και με-

ρικές ακόμα μονάδες, εκ των οποίων οι πιο σημαντικές είναι: Η εξειδικευμένη DMA engine και η 256 KB L2 κρυφή μνήμη που είναι κοινή για τους SHAVE επεξεργαστές. Ο στόχος του UPA είναι να υποστηρίξει την ανάπτυξη εξειδικευμένων αλγορίθμων, που απαιτούνται από πολλές εφαρμογές όρασης υπολογιστών και μηχανικής μάθησης, καθώς επίσης και άλλων υπολογιστικά απαιτητικών αλγορίθμων. Καθένας από τους VLIW επεξεργαστές είναι σε θέση να ελέγχει πολλαπλές δομικές μονάδες, οι οποίες διαθέτουν δυνατότητες SIMD, για μεγαλύτερο παραλληλισμό και διεκπεραιωτική ικανότητα, τόσο σε επίπεδο δομικής μονάδας, όσο και σε επίπεδο επεξεργαστή. Καθεμία από τις μονάδες του επεξεργαστή μπορεί να εκτελείται ταυτόχρονα, στον ίδιο κύκλο ρολογιού. Οι SHAVE υποστηρίζουν εντολές SIMD για διάφορους τύπους, όπως: 8 bits ακεραίους, 16 bits ακεραίους, 32 bits ακεραίους, 16 bits αριθμούς κινητής υποδιαστολής, 32 bits αριθμούς κινητής υποδιαστολής.

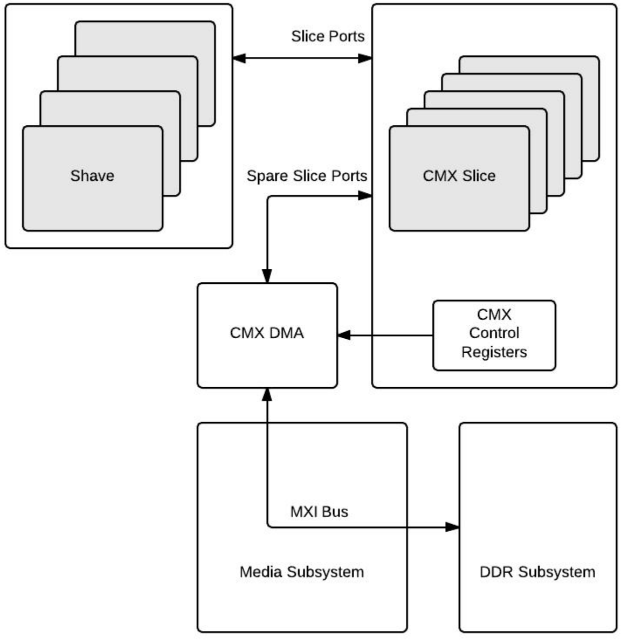
- *CMX*: Πρόκειται σύντμηση του "Connection Matrix", το οποίου δικαιολογείται από το γεγονός ότι η CMX αποτελείται από αρκετές μικρότερες μονάδες SRAM, με συνολικό μέγεθος τα 2 MB. Κάθε επεξεργαστής SHAVE έχει ξεχωριστές θύρες για πρόσβαση σε μία συγκεκριμένη φέτα των 128KB της μνήμης CMX. Συνεπώς, τα 12x128 KB = 1536 KB χρησιμοποιούνται με τον καλύτερο δυνατό τρόπο από τους πυρήνες SHAVE, ενώ τα υπόλοιπα 512 KB της μνήμης CMX memory χρησιμοποιούνται από άλλες μονάδες. Συνίσταται η χρήση των παραπάνω 512 KB από τα φίλτρα υλικού που είναι ενσωματωμένα στο SIPP ή από κρίσιμα κομμάτια κώδικα που πρέπει να εκτελούνται όσο το δυνατό γρηγορότερα, και συνεπώς δεν μπορούν να τοποθετηθούν στη μνήμη DDR.

- *DDR*: Είναι η μεγαλύτερη μονάδα πτητικής μνήμης που διαθέτει η Myriad2, με το μέγεθός της να είναι 128MB ή 512MB, αναλόγως με την έκδοση αναθεώρησης του SoC. Η κυριότερη διαφορά μεταξύ της Myriad2 και άλλων επεξεργαστών είναι η θέση της DDR. Στη Myriad2, η DDR βρίσκεται εντός του SoC, ωστόσο η μνήμη είναι τοποθετημένη εκτός της ψηφίδας, που σημαίνει ότι οι 14 επεξεργαστές χρησιμοποιούν τον ίδιο ελεγκτή για να την προσπελάσουν.

1.5.2 Ελεγκτής DMA της μνήμης CMX

Αυτός ο ελεγκτής βρίσκεται ανάμεσα του διαύλου MXI των 128-bit και της μνήμης CMX [9]. Παρέχει μεταφορές δεδομένων υψηλού εύρους ζώνης μεταξύ της CMX και της DDR, προς οποιαδήποτε κατεύθυνση. Επιπλέον, υποστηρίζει μεταφορές δεδομένων από DDR σε DDR και από CMX σε CMX. Το σχήμα 1.11 παρουσιάζει ένα υψηλού επιπέδου διάγραμμα της DMA engine.

Η DMA engine μοντελοποιεί την μεταφορά δεδομένων μέσω δοσοληψιών. Μπορούν να συνυπάρχουν έως και τέσσερις συνδεδεμένες λίστες από δοσοληψίες ταυτόχρονα, το οποίο σημαίνει ότι η ικανότητα εξυπηρέτησης δοσοληψιών από την DMA engine δεν είναι απεριόριστη. Ο προγραμματιστής μπορεί εύκολα να ξεπεράσει τα φυσικά όρια της DMA engine, αν την χρησιμοποιεί δίχως μέτρο.



Σχήμα 1.11: *CMX DMA engine of Myriad2*

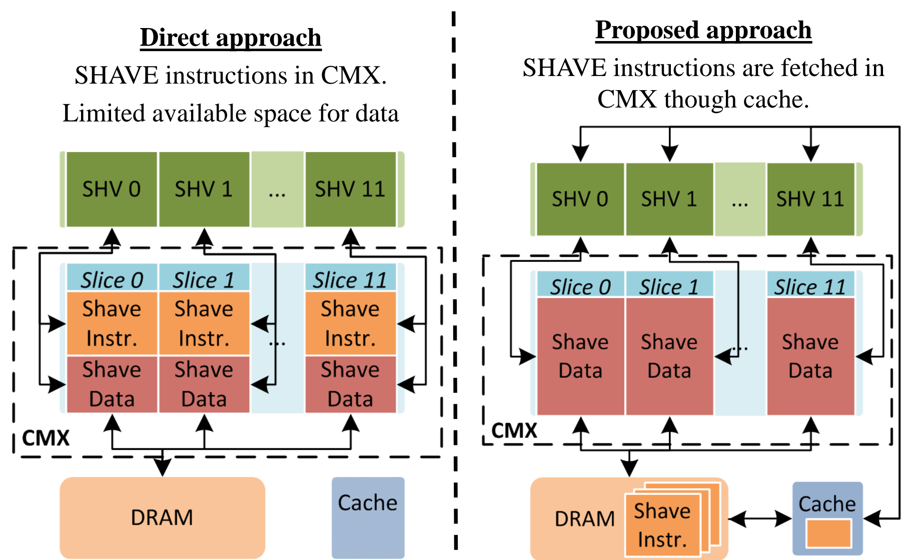
1.6 Βελτιστοποιήσεις που εφαρμόστηκαν

Η ανάπτυξη του πηγαίου κώδικα για το σύστημα εκτέλεσης των συνελικτικών νευρωνικών δικτύων πραγματοποιήθηκε έχοντας τη Myriad2 κατά νου. Ωστόσο, αντί για πλήρη εξειδίκευση στο συγκεκριμένο επεξεργαστή, έγιναν προσπάθειες ώστε το αποτέλεσμα να είναι ανεξάρτητο από το παραπάνω υλικό. Αυτό οδήγησε σε μια πιο γενική μεθοδολογία, τα βήματα της οποίας μπορούν να εφαρμοστούν και σε άλλες παρόμοιες ενσωματωμένες αρχιτεκτονικές, με στόχο την αύξηση της απόδοσης και τη μείωση της κατανάλωσης ενέργειας.

Ένα από τα προβλήματα που εμφανίστηκαν στα αρχικά στάδια της ανάπτυξης ήταν η αποδοτική αξιοποίηση της μνήμης CMX. Είναι φανερό, πως η μνήμη CMX μπορεί να προσφέρει μεγάλη βελτίωση στη απόδοση, εάν κανείς καταφέρει να την εκμεταλλευτεί με σωστό τρόπο. Η προτεινόμενη προσέγγιση απεικονίζεται στο σχήμα 1.12, με αποτέλεσμα την αύξηση του διαθέσιμου χώρου της CMX - που διατίθεται στα δεδομένα - κατά περίπου 50%.

Συνοπτικά, η σημαντική διαφορά εντοπίζεται στο θέση τοποθέτησης του κώδικα που εκτελούν οι επεξεργαστές SHAVE. Μετακινώντας τον κώδικα από τη CMX στη DDR και κάνοντας χρήση του υποσυστήματος της κρυφής μνήμης έχει μεγάλο κέρδος. Αυτό συμβαίνει διότι - κατά γενικό κανόνα - η εκτέλεση του κώδικα από τον επεξεργαστή γίνεται σειριακά, το οποίο ωφελεί την κρυφή μνήμη. Επιπλέον, η κρυφή μνήμη είναι αρκετά μεγάλη ώστε να χωράει σημαντικά τμήματα του κώδικα που εκτελούν οι SHAVE για κάποιο χρονικό διάστημα. Τα παραπάνω, μαζί με το γεγονός ότι ελευθερώνεται μεγάλο μέρος της CMX, κάνουν αυτή τη βελτιστοποίηση εξαιρετικά αποτελεσματική.

Εξαιτίας της μεγάλης έντασης μετακίνησης των δεδομένων μεταξύ της μνήμης DDR



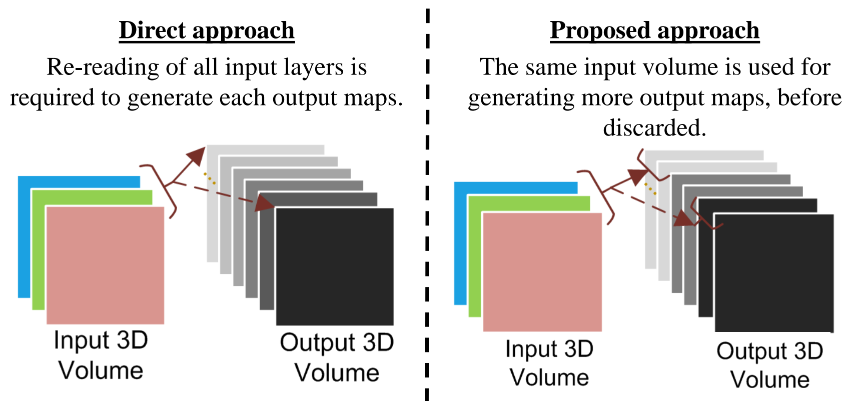
Σχήμα 1.12: *Βελτιστοποίηση 1: Αύξηση του χώρου της μνήμης CMX που διατείθεται στα δεδομένα.*

και CMX, είναι απαραίτητη μια στρατηγική πρόσβασης των δεδομένων που να ελαχιστοποιεί τις μετακινήσεις αυτών. Η βασική ιδέα είναι να πραγματοποιείται η μέγιστη δυνατή εκμετάλλευση ενός τμήματος των δεδομένων που μεταφέρεται από τη DDR στη CMX, προτού αυτό αντικατασταθεί από ένα άλλο τμήμα δεδομένων. Για να επιτευχθεί κάτι τέτοιο, απαιτείται η αναδιοργάνωση των εμφωλευμένων επαναληπτικών βρόχων που πραγματοποιούν τους υπολογισμούς, το οποίο παρουσιάζεται στο σχήμα 1.13.

Σύμφωνα με το σχήμα, η διαφορά έγκειται στο γεγονός ότι με τη φόρτωση των δεδομένων εισόδου μια φορά, παράγονται πολλαπλά κανάλια των δεδομένων εξόδου. Ρυθμίζοντας προσεκτικά την αναλογία των δεδομένων εισόδου και των δεδομένων εξόδου που μπορούν να συνυπάρχουν στη μνήμη CMX την ίδια χρονική στιγμή, είναι δυνατή η επίτευξη σημαντικής επιτάχυνσης στην εκτέλεση. Συγκεκριμένα, η προτεινόμενη προσέγγιση έχει ως αποτέλεσμα μείωση του χρόνου εκτέλεσης κατά περίπου 30%.

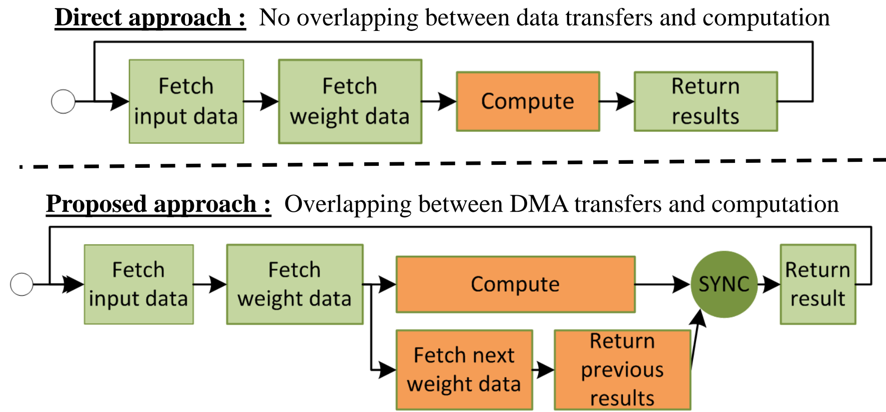
Προκειμένου να μειωθεί ο χρόνος εκτέλεσης ακόμα περισσότερο, οι μεταφορές της DMA engine επικαλύπτονται χρονικά με την πραγματοποίηση υπολογισμών. Αποτέλεσμα της επικάλυψης είναι η ελαχιστοποίηση του κόστους μεταφοράς, καθώς κάθε επεξεργαστής SHAVE δεν χρειάζεται να περιμένει τα διαθέσιμα δεδομένα. Σημειώνεται πως αυτή η βελτιστοποίηση απαιτεί την ύπαρξη διπλών προσωρινών χώρων δεδομένων (double buffering). Επιπλέον, είναι αποτελεσματική όταν κανείς πραγματοποιεί υπολογισμούς των οποίων η πολυπλοκότητα είναι αρκετά μεγάλη, ώστε να διαρκούν περισσότερο από τη μεταφορά των δεδομένων.

Θα έλεγε κανείς πως η παραπάνω τεχνική δεν είναι κάτι καινούριο, ωστόσο είναι πάρα πολύ σημαντική. Για παράδειγμα, οι επεξεργαστές γραφικών της nVidia κατασκευάζονται με διπλές DMA engines, ώστε να υποστηρίξουν ακόμα πιο αποδοτικά το double buffering. Η προτεινόμενη προσέγγιση απεικονίζεται στο σχήμα 1.14 και προ-



Σχήμα 1.13: *Βελτιστοποίηση 2: Μείωση του αριθμού των απαιτούμενων μεταφορών DMA.*

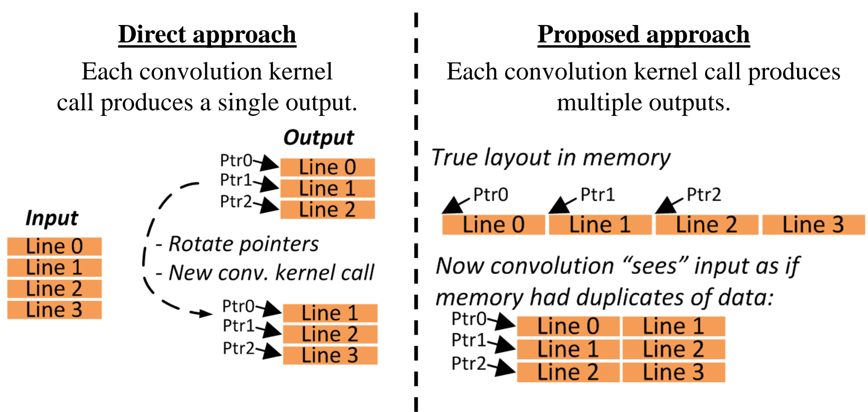
σφέρει αύξηση της απόδοσης των υπολογιστικά απαιτητικών πράξεων κατά περίπου 20%.



Σχήμα 1.14: *Βελτιστοποίηση 3: Μείωση του κόστους των μεταφορών DMA.*

Τέλος, εξαιτίας του αρχιτεκτονικού σχεδιασμού της Myriad2, η επίλυση των διακλαδώσεων (σε ορολογία συμβολικής γλώσσας) έχει σχετικά μεγάλο κόστος, καθώς ο συγκεκριμένος επεξεργαστής δε διαθέτει μηχανισμό πρόβλεψης διακλαδώσεων. Γι' αυτό το λόγο, οι μετρήσεις που πραγματοποιήθηκαν έδειξαν ότι οι βρόχοι με πολύ μικρό σώμα και οι εκτεταμένες κλήσεις σε συναρτήσεις έχουν μη αμελητέο κόστος. Έτσι, έγιναν προσπάθειες να μειωθούν οι κλήσεις σε συναρτήσεις εκτελούν τους υπολογισμούς. Το σχήμα 1.15 εξηγεί σχηματικά πως αυτό επιτυγχάνεται για τη συνέλιξη.

Συγκεκριμένα, η ιδέα βασίζεται στο γενονός ότι υπάρχει η δυνατότητα να υπολογιστεί η συνέλιξη σε ένα ορθογώνιο χωρίο, χωρίς να απαιτείται διατήρηση του δισδιάστατου σχήματος αυτού. Με άλλα λόγια, τοποθετώντας τους δείκτες όπως φαίνεται στα δεξιά του σχήματος, η συνέλιξη μπορεί να πραγματοποιηθεί με μία μόνο κλήση στον υπολογιστικό πυρήνα. Το όφελος μιας τέτοιας μετατροπής είναι περίπου 6% μείωση του χρόνου εκτέλεσης.



Σχήμα 1.15: *Βελτιστοποίηση 4: Μείωση του αριθμού των κλήσεων για τον υπολογιστικό πυρήνα της συνέλιξης.*

1.7 Αξιολόγηση της υλοποίησης

Αυτή η ενότητα παρουσιάζει δυο συνελικτικά νευρωνικά δίκτυα μαζί με τις μετρήσεις που ελήφθησαν από την εκτέλεσή τους στη Myriad2.

1.7.1 Τα νευρωνικά δίκτυα CIFAR10 Quick και nViso

Το πρώτο δίκτυο που χρησιμοποιήθηκε για την αξιολόγηση της υλοποίησης είναι το CIFAR10 Quick, που παρουσιάζεται στο σχήμα 1.16. Η συγκεκριμένη αρχιτεκτονική εκπαιδεύτηκε στην ταξινόμηση της υφής του εδάφους από δορυφορικές φωτογραφίες. Πιο αναλυτικά, χρησιμοποιήθηκε το σύνολο δεδομένων [10], το οποίο αποτελείται από εικόνες μεγέθους 28×28 που ανήκουν σε μία από τις ακόλουθες 6 κατηγορίες: άγονη γη, δένδρα, χαμηλή βλάστηση, δρόμοι, κτίρια, υδάτινοι όγκοι.

Το δεύτερο δίκτυο που χρησιμοποιήθηκε, προσφέρθηκε από την εταιρία nViso, που εξειδικεύεται σε τεχνολογίες μέτρησης του συναισθήματος. Εξαιτίας περιορισμών στην άδεια χρήσης του δικτύου, καμία άλλη πληροφορία σχετικά με το σύνολο δεδομένων με το οποίο έγινε η εκπαίδευση του δικτύου ή άλλες τεχνικές λεπτομέρειες δεν είναι διαθέσιμη. Το συγκεκριμένο συνελικτικό νευρωνικό δίκτυο χρησιμοποιείται για ταξινόμηση του συναισθήματος στις ακόλουθες κατηγορίες, με βάση τις εκφράσεις του προσώπου: θυμός, απέχθεια, φόβος, χαρά, ουδέτερο συναίσθημα, θλίψη, έκπληξη. Το σχήμα 1.16 απεικονίζει την αρχιτεκτονική του.

1.7.2 Μετρήσεις

Ακολουθεί σύγκριση του χρόνου εκτέλεσης των δύο προαναφερθέντων συνελικτικών νευρωνικών δικτύων, χρησιμοποιώντας διαφορετικές συσκευές ή/και υλοποιήσεις. Η παρούσα υλοποίηση θα συγκριθεί με το Fathom NCS, το οποίο αποτελείται από το ίδιο υλικό (Myriad2), αλλά συνοδεύεται από πακέτο λογισμικού εκτέλεσης νευρωνικών δικτύων κλειστού κώδικα. Επίσης, τα δίκτυα θα εκτελεστούν και με το Caffe, σε x86 Intel επεξεργαστές, ώστε να υπάρξει μια αίσθηση για τη συμπεριφορά του χρόνου

Σχήμα 1.16: ***Αριστερά***: *CIFAR10 Quick CNN για ταξινόμηση εδάφους.* ***Δεξιά***: *nViso CNN για ταξινόμηση συναισθήματος από τις εκφράσεις του προσώπου.*

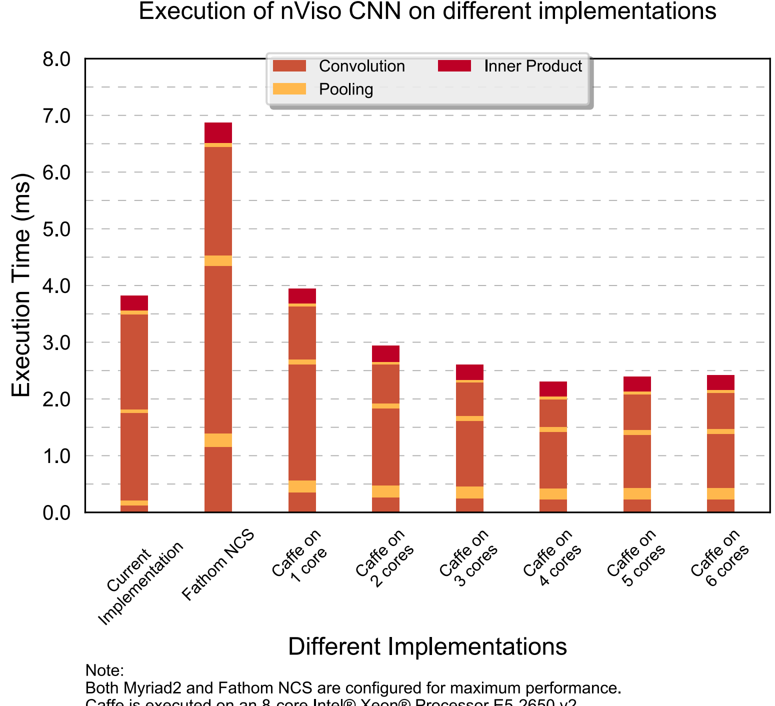
εκτέλεσης σε διαφορετικούς επεξεργαστές. Τα σχήματα 1.17 και 1.18 απεικονίζουν τη σύγκριση.

Υπενθυμίζεται πως η Myriad2 που χρησιμοποιήθηκε για την εκτέλεση της παρούσας υλοποίησης είναι το μοντέλο MA2150, που διαθέτει 128MB μνήμη DDR, στα 533MHz.

Execution of CIFAR10 CNN on different implementations

Convolution
Inner Product
Pooling
Execution Time (ms)
4.5
4.0
3.5
3.0
2.5
2.0
1.5
1.0
0.5
0.0
Current Implementation
Fathom NCS
Caffe on 1 core
Caffe on 2 cores
Caffe on 3 cores
Caffe on 4 cores
Caffe on 5 cores
Caffe on 6 cores
Different Implementations
Note:
Both Myriad2 and Fathom NCS are configured for maximum performance.
Caffe is executed on an 8-core Intel® Xeon® Processor E5-2650 v2.

Σχήμα 1.17: *.Χρόνος εκτέλεσης του CIFAR10 Quick CNN σε διαφορετικούς επεξεργαστές ή/και υλοποιήσεις.*



Σχήμα 1.18: *Χρόνος εκτέλεσης του nViso CNN σε διαφορετικούς επεξεργαστές ή/και υλοποιήσεις.*

Από την άλλη, το Caffe εκτελείται στον επεξεργαστή Intel Xeon E5-2650 v2 CPU, που διαθέτει 20MB από ιεραρχίες κρυφής μνήμης, έχει 8 πυρήνες, 16 νήματα και 2.60 GHz

βασική συχνότητα επεξεργαστή. Σε γενικές γραμμές, ο E5-2650 είναι ένας πολύ ισχυρός επεξεργαστής που προορίζεται για πολυπρογραμματιστικά περιβάλλοντα.

Από τις γραφικές παραστάσεις είναι ορατό πως το Caffe δεν αποδίδει πάρα πολύ καλύτερα συγκριτικά με τη Myriad2. Εάν το έλλειμμα απόδοσης της Myriad2 δεν είναι τόσο σημαντικό, τότε η Myriad2 αποτελεί καλύτερη επιλογή, εξαιτίας της χαμηλής κατανάλωσης ενέργειας που διαθέτει.

1.7.3 Αριθμητική ακρίβεια των υπολογισμών

Προκειμένου η εκτέλεση των νευρωνικών δικτύων στη Myriad2 να γίνει ταχύτερη, χρησιμοποιούνται 16-bit αριθμοί κινητής υποδιαστολής. Ωστόσο, η εκπαίδευση των δικτύων με το Caffe γίνεται με 32-bit αριθμούς κινητής υποδιαστολής. Κατά συνέπεια, το ερώτημα που εγείρεται είναι εάν μια τόσο μεγάλη μείωση στην αριθμητική ακρίβεια των υπολογισμών, θα έχει επίπτωση στην ακρίβεια πρόβλεψης της ταξινόμησης του συνελικτικού νευρωνικού δικτύου. Τονίζεται, πως το ενδιαφέρον επικεντρώνεται στο μέγεθος του σφάλματος μεταξύ των αριθμών που παράγει η Myriad2 και των αριθμών που παράγει το Caffe, κατά τη διαδικασία της πρόβλεψης.

Εκτελώντας τα δύο δίκτυα που περιγράφονται παραπάνω, χρησιμοποιώντας ως είσοδο μερικές τυχαίες εικόνες, ελήφθησαν αποτελέσματα που δείχνουν ότι η εκτέλεση με αριθμούς κινητής υποδιαστολής των 16 bits δεν είναι καταστροφική. Ενδεικτικά, παρουσιάζονται ορισμένα αποτελεσμάτα από το nViso CNN στον πίνακα 1.1. Σε αυτό τον πίνακα, με το γράμμα "C" συμβολίζονται τα αποτελέσματα που δίνει το Caffe, με γράμμα "M" συμβολίζονται τα αποτελέσματα που δίνει η υλοποίηση στη Myriad2, ενώ με το δίγραμμα "RE" συμβολίζεται το σχετικό σφάλμα των τιμών που παράγει η Myriad2, αναφορικά με τις τιμές που παράγει το Caffe.

Συμπερασματικά, ο πίνακας 1.1 επιβεβαιώνει πως η εκπαίδευση σε 32-bit και η εκτέλεση του βήματος πρόβλεψης στα 16-bit είναι μια αποδεκτή πρακτική. Επιπλέον, διαφαίνεται και η ακόλουθη ενδιαφέρουσα ιδιότητα. Αν και τα απόλυτα μεγέθη στην αριθμητική των 16-bit μπορεί να διαφέρουν, συγκριτικά με την αριθμητική των 32-bit, η σειρά τους διατηρείται ίδια με αυτή των αποτελεσμάτων του Caffe. Κατά συνέπεια, ο τρόπος με τον οποίο το Caffe αναθέτει τις πιθανότητες μια είσοδος να ανήκει σε κάποια κατηγορία, είναι ίδια με αυτή που παράγεται από την εκτέλεση στη Myriad2.

Είσοδος / Κατηγορία		Θυμός	Απέχθεια	Φόβος	Χαρά	Ουδέτερο	Λύπη	Έκπληξη
	C	-17.850915	-2.357175	-6.676564	52.664138	-9.152770	-5.112073	-9.091076
	M	-17.843750	-2.351562	-6.683594	52.625000	-9.156250	-5.121094	-9.093750
	RE	0.040%	0.238%	0.105%	0.074%	0.038%	0.176%	0.029%
	C	50.073471	4.734987	-17.832698	-26.339084	-4.803456	4.931145	-13.976916
	M	50.09375	4.710938	-17.828125	-26.359375	-4.796875	4.941406	-13.976562
	RE	0.040%	0.508%	0.026%	0.077%	0.137%	0.208%	0.003%
	C	-20.158855	-33.927001	44.842117	-0.139439	-21.007415	-15.929889	64.673278
	M	-20.15625	-33.90625	44.84375	-0.145996	-21.015625	-15.921875	64.6875
	RE	0.013%	0.061%	0.004%	4.702%	0.039%	0.050%	0.022%
	C	28.647573	31.595224	-13.235262	-11.848876	-21.539316	-0.38034	-17.441757
	M	28.640625	31.609375	-13.25	-11.851562	-21.546875	-0.372559	-17.453125
	RE	0.024%	0.045%	0.111%	0.023%	0.035%	2.046%	0.065%
	C	-20.353111	-31.784439	48.527591	-2.38341	-20.314197	-17.113338	61.004989
	M	-20.34375	-31.75	48.5	-2.388672	-20.296875	-17.109375	60.96875
	RE	0.046%	0.108%	0.057%	0.221%	0.085%	0.023%	0.059%

Πίνακας 1.1: *Σύγκριση της αριθμητικής ακρίβειας της στρώσης "hidden_out" του nViso CNN layer, εκτελώντας το σε Caffe και στη Myriad2.*

Part I

Theory

Chapter

Background on artificial neural networks

This chapter begins with an overview of artificial neural networks and continues with the description of convolutional neural networks. By the end of this chapter the reader should have a basic understanding of neural networks, which is required for subsequent chapters.

1.1 Machine learning in general

In this age of modern technology, there is one resource that exists in abundance: a large amount of structured and unstructured data. In the second half of the twentieth century, machine learning evolved as a subfield of artificial intelligence that involved the development of self-learning algorithms to gain knowledge from that data in order to make predictions. Instead of requiring humans to manually derive rules and build models from analyzing large amounts of data, machine learning offers a more efficient alternative for capturing the knowledge in data to gradually improve the performance of predictive models, and make data-driven decisions. Not only is machine learning becoming increasingly important in computer science research but it also plays an ever greater role in the everyday life. Thanks to machine learning, robust e-mail spam filters, convenient text and voice recognition software, reliable web search engines, challenging chess players, and, hopefully soon, safe and efficient self-driving cars are a reality [1].

1.1.1 The three different types of machine learning

This subsection takes a look at the three types of machine learning: *supervised learning*, *unsupervised learning* and *reinforcement learning* [1].

- *Supervised Learning*: The main goal in supervised learning is to learn a model from labeled *training* data that allows to make predictions about *unseen* or future data. Here, the term supervised refers to a set of samples where the desired output signals (labels) are already known.

- *Reinforcement Learning*: The goal is to develop a system (*agent*) that improves its performance based on interactions with the environment. Since the informa-

tion about the current state of the environment typically also includes a so-called *reward* signal, reinforcement learning can be thought of as a field related to supervised learning. However, in reinforcement learning this feedback is not the correct ground truth label or value, but a measure of how well the action was measured by a reward function. Through the interaction with the environment, an agent can then use reinforcement learning to learn a series of actions that maximizes this reward via an exploratory trial-and-error approach or deliberative planning.

- *Unsupervised Learning*: In supervised learning, the right answer is known beforehand when the training of the model is performed, and in reinforcement learning, a measure of reward for particular actions by the agent is defined. In unsupervised learning, however, the data are unlabeled or have *unknown structure*. Using unsupervised learning techniques, it is possible to explore the structure of the data, in order to extract meaningful information without the guidance of a known outcome variable or reward function.

The focus of subsequent sections will be solely on supervised learning, whose essence is shown in figure 1.1. Considering the example of e-mail spam filtering software: The approach is to train a model using a supervised machine learning algorithm on a corpus of labeled e-mail, –mail that are correctly marked as spam or not-spam - to predict whether a new e-mail belongs to either of the two categories. A supervised learning task with discrete class labels, such as in the previous e-mail spam-filtering example, is also called a *classification* task. Another subcategory of supervised learning is *regression*, where the outcome signal is a continuous value.

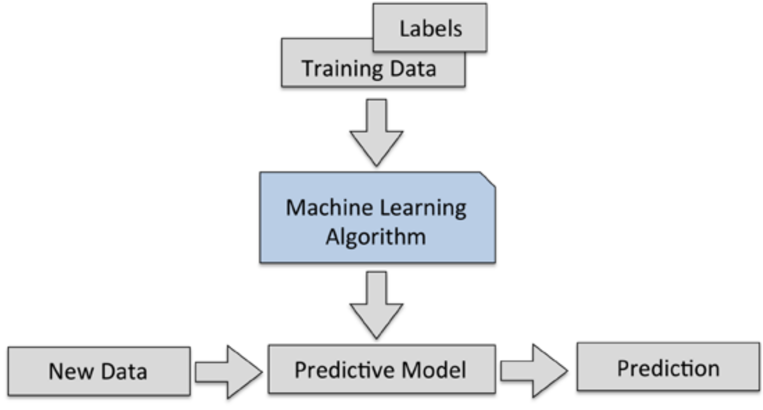


Figure 1.1: *The supervised learning approach*

1.1.2 Classification in supervised learning: Predicting class labels

Classification is a subcategory of supervised learning where the goal is to predict the categorical class labels of new instances based on past observations [2]. Those class labels are discrete, unordered values that can be understood as the group memberships of the instances. The previously mentioned example of e-mail spam detection represents

a typical case of a *binary classification* task, where the machine learning algorithm learns a set of rules in order to distinguish between two possible classes: spam and non-spam e-mail. However, the set of class labels does not have to be of a binary nature.

The predictive model learned by a supervised learning algorithm can assign any class label that was presented in the training dataset to a new, unlabeled instance. A typical example of a multi-class classification task is handwritten character recognition. Here, a training dataset that consists of multiple handwritten examples of each letter in the alphabet would be the starting point. Now, if a user provides a new handwritten character via an input device, the predictive model will be able to predict the correct letter in the alphabet with certain accuracy. However, the machine learning system would be unable to correctly recognize any of the digits zero to nine, for example, if they were not part of the training dataset.

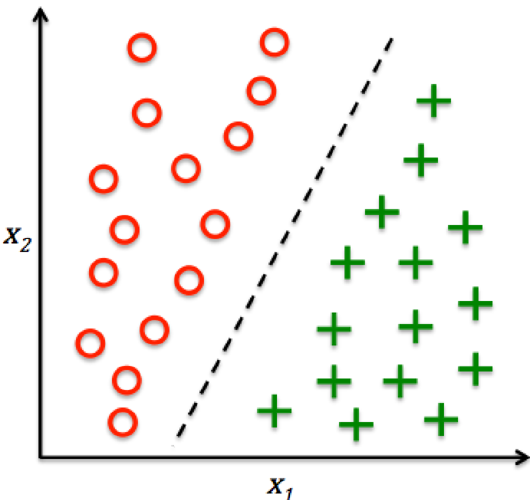


Figure 1.2: *Binary classification in supervised learning*

Figure 1.2 illustrates the concept of a *binary classification* task given 30 training samples: 15 training samples are labeled as negative class (circles) and 15 training samples are labeled as positive class (plus signs). In this scenario, the dataset is two-dimensional, which means that each sample has two values associated with it: x_1 and x_2. Now, a supervised machine learning algorithm can be used to learn a rule - the decision boundary represented as a black dashed line - that can separate those two classes and classify new data into each of those two categories given its x_1 and x_2 values [1].

1.1.3 Regression in supervised learning: Predicting continuous outcomes

The previous section showed that the task of classification is to assign categorical, unordered labels to instances [2]. A second type of supervised learning is the prediction of continuous outcomes, which is also called *regression analysis* [2]. In regression analysis, a number of *predictor* (explanatory) variables and a continuous response variable (*outcome*) is given, and the goal is to find a relationship between those variables that allows the prediction of an outcome. For example, assume that there is interest in

predicting the Math SAT scores of students. If there is a relationship between the time spent studying for the test and the final scores, it could be used as training data to learn a model that given the study time, predicts the test scores of future students who are planning to take this test.

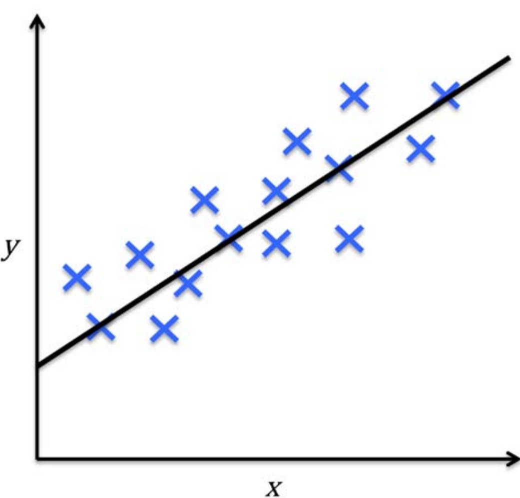


Figure 1.3: *Linear regression in supervised learning*

Figure 1.3 illustrates the concept of *linear regression*. Given a predictor variable x and a response variable y, a straight line is fitted to this data that minimizes the distance - most commonly the average squared distance - between the sample points and the fitted line. Then, the intercept and slope learned from this data is used to predict the outcome variable of new data [1].

1.2 Mathematics of linear classification for images

This section will delve into the mathematics of the linear classification problem. The formulation presented here will be useful in the next section, which will extend the notions discussed here. In particular, the problem of linear classification will be applied to images. Eventually, the approach described will naturally extend to entire Artifical Neural Networks and Convolutional Neural Networks, which will be presented in the following sections. The methodology will have two major components: a *score* function that maps the raw data to class scores, and a *loss* function that quantifies the agreement between the predicted scores and the ground truth labels [3].

1.2.1 Parameterized mapping from images to label scores

The first component of this approach is to define the score function that maps the pixel values of an image to confidence scores for each class [3]. The approach will be developed making references to examples for better understanding. Assume a training dataset of images $x_i \in R^D$, each associated with a label y_i. Here, $i = 1, \dots, N$ and $y_i \in \{0, \dots, K-1\}$. That is, there are N examples (each with dimensionality D) and K distinct

categories (or classes). For example, if the dataset contains 32×32 RGB images, then $D = 32 \times 32 \times 3 = 3072$ pixels. The score function $f : R^D \to R^K$ maps the raw image pixels to class scores. The linear classifier is arguably the simplest possible score function. It is expressed as:

$$f(x_i, W, b) = Wx_i + b$$

In the equation above, it is assumed that the image x_i has all of its pixels flattened out to a single column vector of shape $[D \times 1]$. The matrix W (of size $[K \times D]$), and the vector b (of size $[K \times 1]$) are the *parameters* of the function. The parameters in W are often called the *weights*, and b is called the *bias* vector because it influences the output scores, but without interacting with the actual data x_i. However, ofter people use the terms weights and parameters interchangeably.

There are a few things to note:

- First, note that the single matrix multiplication Wx_i is effectively evaluating K separate classifiers in parallel (one for each class), where each classifier is a row of W.

- Notice also that the input data (x_i, y_i) is given and it is fixed, but there is control over the setting of the parameters W, b. The goal is to set these in such way that the computed scores match the ground truth labels across the whole training set. Intuitively it is expected that the correct class have a score that is higher than the scores of incorrect classes.

1.2.2 The linear classifier matrix-vector multiplication

Notice that a linear classifier computes the score of a class as a weighted sum of all of its pixel values across all of its color channels [3]. Depending on precisely what values are set for these weights, the function has the capacity to like or dislike (depending on the sign of each weight) certain colors at certain positions in the image. For instance, you can imagine that the "ship" class might be more likely if there is a lot of blue on the sides of an image, which could likely correspond to water. You might expect that the "ship" classifier would then have a lot of positive weights across its blue channel weights, i.e. the presence of blue increases score of "ship" class. Similarly, the classifier would have negative weights in the red/green channels, i.e. the presence of red/green decreases the score of this class. A more pictorial explanation of this is presented in figure 1.4.

The reader also needs to be aware of one common trick regarding the bias term introduced it the score function of the linear classifier. This trick is widely used across Machine Learning and will be silently inferred in subsequent sections mentioning Artificial Neural Networks. Recall that the score function was defined as:

$$f(x_i, W, b) = Wx_i + b$$

It is a little cumbersome to keep track of two sets of parameters (the biases b and weights

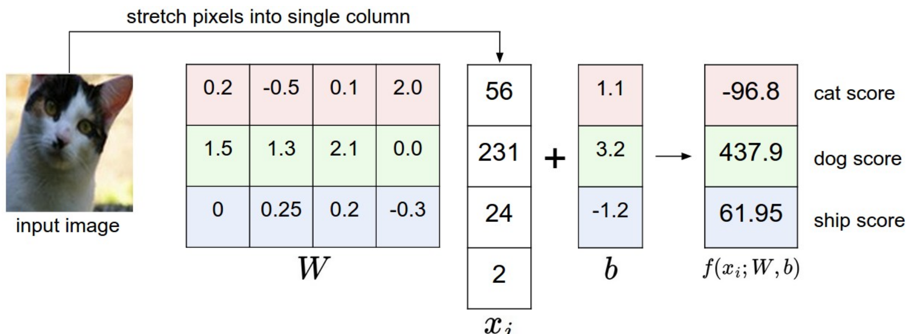


Figure 1.4: *An example of mapping an image to class scores. For the sake of visualization, it is assumed that the image has only 4 pixels (4 monochrome pixels, color channels are not considered for in this example for brevity), and that there are 3 classes (red (cat), green (dog), blue (ship) class). (Clarification: in particular, the colors here simply indicate 3 classes and are not related to the RGB channels). The image pixels are stretched into a column and the matrix-vector multiplication gives the scores for each class. Note that this particular set of weights W is not good at all: the weights assign the cat image a very low cat score. In particular, this set of weights seems convinced that it is a dog.*

W) separately. A commonly used trick is to combine the two sets of parameters into a single matrix that holds both of them by extending the vector x_i with one additional dimension that always holds the constant 1 - a default bias dimension. With the extra dimension, the new score function will simplify to a single matrix multiplication:

$$f(x_i, \hat{W}, b) = \hat{W}x_i$$

This trick is presented in a concise manner in figure 1.5. For notational simplicity, W instead of $\hat{W}$ will be used, even when bias term is present.

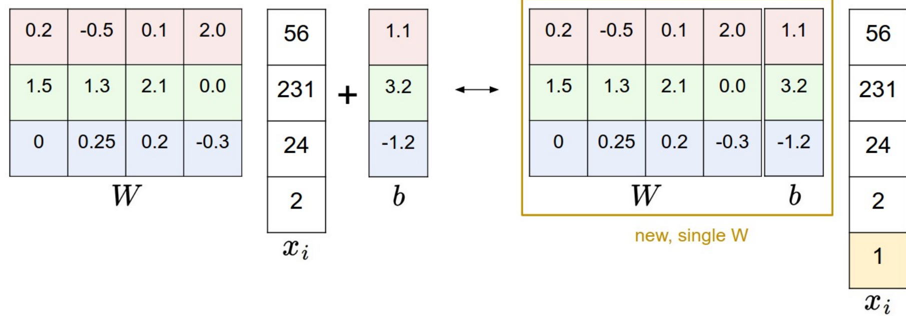


Figure 1.5: *Illustration of the bias trick. Doing a matrix multiplication and then adding a bias vector (left) is equivalent to adding a bias dimension with a constant of 1 to all input vectors and extending the weight matrix by 1 column - a bias column (right)*

1.2.3 Linear classifier: Images as high-dimensional points

Since the images are stretched into high-dimensional column vectors, each one can be interpreted as a single point in a D-dimensional space [3]. Analogously, the entire dataset is a (labeled) set of points. Having defined the score of each class as a weighted

sum of all image pixels, each class score is a linear function over this space. This *D*-dimensional space cannot be visualized, but if it was comprised of only two dimensions, then it would look like something presented in figure 1.6.

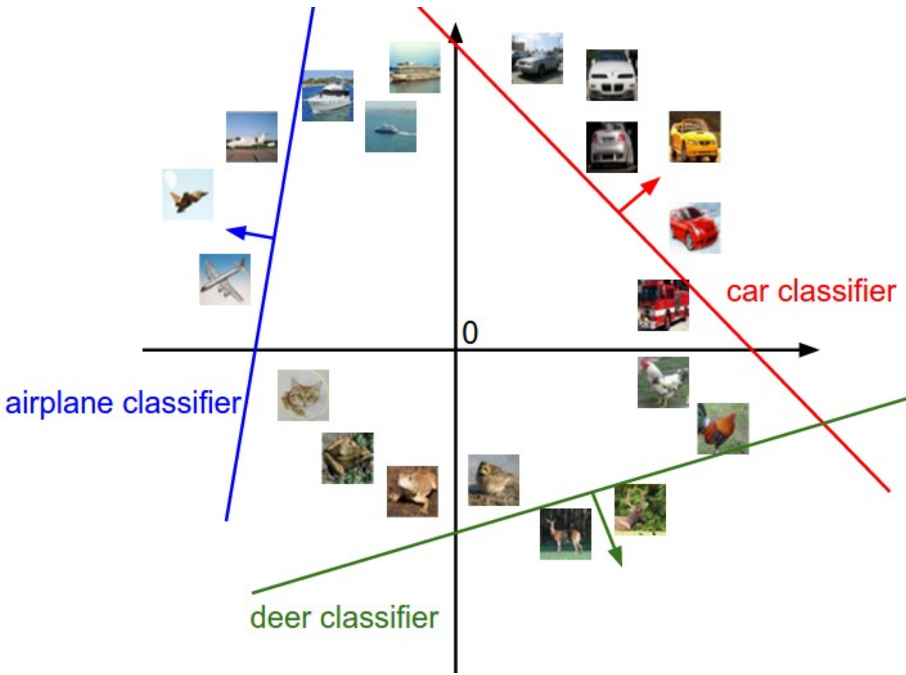


Figure 1.6: *Cartoon representation of the image space, where each image is a single point, and three classifiers are visualized. Using the example of the car classifier (in red), the red line shows all points in the space that get a score of zero for the car class. The red arrow shows the direction of increase, so all points to the right of the red line have positive (and linearly increasing) scores, and all points to the left have a negative (and linearly decreasing) scores.*

1.2.4 The loss function

The score function from the pixel values to class scores, that was parameterized by a set of weights W, is essential for predicting which image belongs to what class [3], [11]. Moreover, in the context of supervised learning, there is no control over the data (x_i, y_i), but the weights are free to change. These weights should be set so that the predicted class scores are consistent with the ground truth labels in the training dataset.

For example, going back to the example image of a cat and its scores for the classes "cat", "dog" and "ship" shown in figure 1.4 it is obvious that the particular set of weights in that example was not very good at all, since the cat is recognized to be a dog instead. A measure of unhappiness is given by the *loss function*, sometimes also referred to as the *cost function* or the *objective*. Intuitively, the loss will be high if the algorithm is doing a poor job of classifying the training data, and it will be low if it is doing well.

One popular choice of loss function is using the *softmax function*, which gives a slightly more intuitive output in terms of class probabilities. In this case, the function mapping $f(x_i, W, b) = Wx_i$ stays unchanged, but now these scores are fed into the softmax function:

$$P(Y = y_i \mid X = x_i, W) = \frac{\exp(f_{y_i})}{\sum_j \exp(f_{y_i})}$$

where the notation f_j means the j-th element of the vector of class scores $f = Wx$, for some input vector x. To put it another way, f_j is actually a shorthand for the inner product of line j in weight matrix W with some input vector x, i.e. $f_j = {w_j}^T x$. For a specific input vector x_i, then: $f_{y_i} = w_{y_i}^T x_i$.

In other words, a given input x_i with a given set of parameters W, has a probability of belonging to the class y_i that can be calculated by the formula above. Notice that for mathematical convenience the different classes "cat", "dog", etc. have been converted to the numbers 0, 1, etc.. That is, $y_i \in \{0, 1, \ldots, K-1\}$. Since there is no control over the data (x_i, y_i), the loss function is actually dependent upon the parameters W. As a result, the goal of training is to find suitable values of the parameters W in order to minimize - in some meaningful way - the loss function. The search for suitable values is called *optimization*. For very simple problems, such as this, the optimization can be performed analytically and can be expressed in closed form. However, for more complicated problems, which will be presented in the next section, the optimization is performed heuristically using the numerical method of gradient descent and its extensions.

1.3 Artificial neural networks

Artificial Neural Networks (ANNs) are biologically inspired networks interconnected in a specific manner as per the application requirement. One of the many advantages of artificial neural networks is that they require minimum or no preprocessing of input data. On the other hand, the traditional elaborate feature extractors are hand tuned for particular sets of data. Artificial neural network models have the ability to learn and generalize by using examples. This ability to adapt to the recognition task even after design time, makes them unique compared to other artificial intelligence approaches.

1.3.1 Biological motivation and connections

The basic computational unit of the brain is a neuron [2]. Approximately 86 billion neurons can be found in the human nervous system and they are connected with approximately 10^{14} - 10^{15} synapses [3]. Figure 1.7 shows a cartoon drawing of a biological neuron on the left and a common mathematical model on the right. Each neuron receives input signals from its dendrites and produces output signals along its (single) axon. The axon eventually branches out and connects via synapses to dendrites of other neurons. In the computational model of a neuron, the signals that travel along the axons (e.g. x_0) interact multiplicatively (e.g. $w_0 x_0$) with the dendrites of the other neuron based on the synaptic strength at that synapse (e.g. w_0). The idea is that the synaptic strengths (the weights w) are learnable and control the strength of influence (and its direction: excitory (positive weight) or inhibitory (negative weight)) of one neuron on another. In the basic model, the dendrites carry the signal to the cell body where they all get summed. If the

final sum is above a certain threshold, the neuron can fire, sending a spike along its axon. In the computational model, it is assumed that the precise timings of the spikes do not matter, and that only the frequency of the firing communicates information. Based on this rate code interpretation, the firing rate of the neuron is modeled with an activation function *f*, which represents the frequency of the spikes along the axon. Historically, a common choice of activation function is the *sigmoid* function σ, since it takes a real-valued input (the signal strength after the sum) and squashes it to range between 0 and 1. Details of these activation functions will be discussed later in this section.

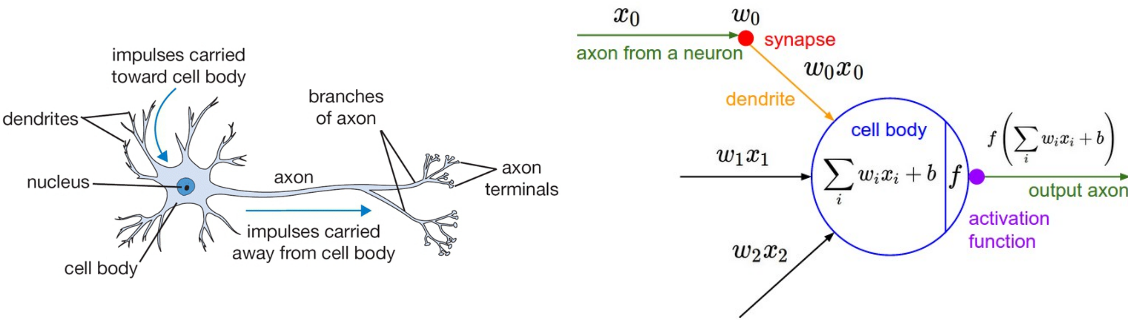


Figure 1.7: *A cartoon drawing of a biological neuron (left) and its mathematical model (right)*

In other words, each neuron performs a dot product with the input and its weights, adds the bias and applies the non-linearity (or activation function), in this case the sigmoid $\sigma(x) = 1/(1 + e^{-x})$.

It is important to stress that this model of a biological neuron is very coarse. For example, there are many different types of neurons, each with different properties. The dendrites in biological neurons perform complex nonlinear computations. The synapses are not just a single weight, they are a complex non-linear dynamical system. The exact timing of the output spikes in many systems in known to be important, suggesting that the rate code approximation may not hold. Due to all these and many other simplifications, ANNs are only inspired by real neurons, they do not try to simulate them.

1.3.2 Commonly used activation functions

Every activation function takes a single number and performs a certain fixed mathematical operation on it [3]. There are several activation functions encountered in practice and two of the most common are presented below.

- *Sigmoid*: This non-linearity has the mathematical form $\sigma(x) = 1/(1 + e^{-x})$ and is shown on the left of figure 1.8. As pointed out previously, it takes a real-valued number and “squashes” it into range between 0 and 1. In particular, large negative numbers become 0 and large positive numbers become 1. The sigmoid function has seen frequent use historically since it has a nice interpretation as the firing rate of a neuron: from not firing at all (0) to fully-saturated firing at an assumed

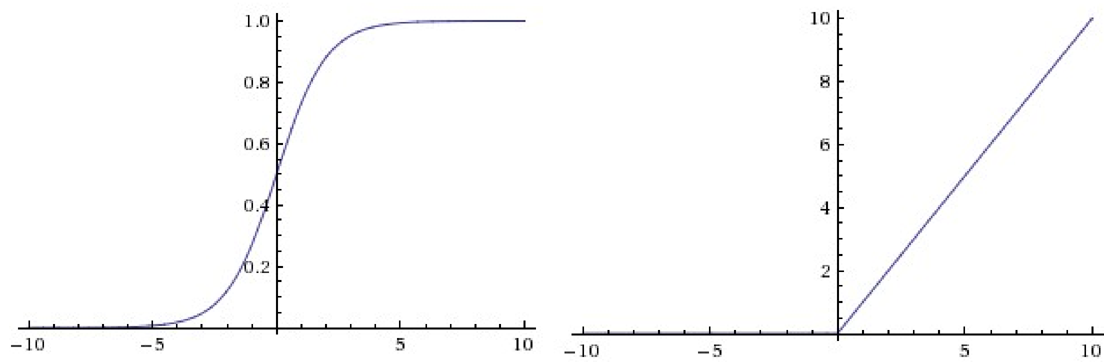


Figure 1.8: *The sigmoid $\sigma(x)$ activation function (left) and the Rectified Linear Unit (ReLU) activation function (right)*

maximum frequency (1). In practice, the sigmoid non-linearity has recently fallen out of favor and it is rarely ever used. It has two major drawbacks:

1. *Sigmoids saturate and kill gradients*: A very undesirable property of the sigmoid neuron is that when the activation of the neuron saturates at either tail of 0 or 1, the gradient at these regions is almost zero. This makes the training of the ANN a very difficult process.
2. *Sigmoid outputs are not zero-centered*: This has implications on the dynamics during gradient descent, the optimization algorithm that is used during the training, posing another difficulty during the training of the ANN.

- *ReLU*: The Rectified Linear Unit has become very popular in the last few years. It computes the function $f(x) = \max(0, x)$. In other words, the activation is simply thresholded at zero. The main advantages of ReLUs are:

  1. It was found to greatly accelerate the convergence of stochastic gradient descent compared to the sigmoid function. It is argued that this is due to its linear, non-saturating form.
  2. Compared to sigmoid neurons that involve expensive operations (i.e. exponentials, and divisions), the ReLU can be implemented very easily with assembly instructions.

In conclusion, ReLU activation functions are preferred in practice. However, care must be taken, because ReLUs are not problem-free. The ReLU units can irreversibly die during training. As much as 40% of the network can become "dead" (i.e. neurons that never activate across the entire training dataset) if the learning rate is set too high. With a proper setting of the learning rate this is less frequently an issue.

1.3.3 ANNs architectures

Artifical neural networks can be presented as neurons in graphs [3]. In particular, they are modeled as collections of neurons that are connected in an acyclic graph. Put

differently, the outputs of some neurons can become inputs to other neurons. Cycles are not allowed since that would imply an infinite loop in the forward pass of a network. Instead of amorphous blobs of connected neurons, ANN models are often organized into distinct layers of neurons. For regular neural networks, the most common layer type is the *fully-connected* layer in which neurons between two adjacent layers are fully pairwise connected, but neurons within a single layer share no connections. In figure 1.9 two example ANN topologies that use a stack of fully-connected layers are shown.

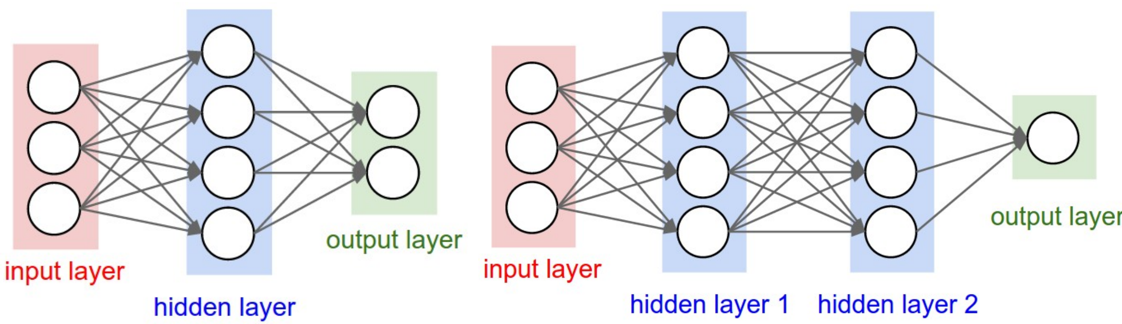


Figure 1.9: ***Left****: A 2-layer Neural Network with three inputs, one hidden layer of 4 neurons and an output layer with 2 neurons.* ***Right****: A 3-layer neural network with three inputs, two hidden layers of 4 neurons each and an output layer with one neuron. Notice that in both cases there are connections (synapses) between neurons across layers, but not within a layer.*

A couple of terminology is worth mentioning:

- *Naming conventions*: An N-layer neural network, does not count the input layer. Therefore, a single-layer neural network describes a network with no hidden layers (input directly mapped to output). ANNs are also called Multi-Layer Perceptrons (MLP).
- *Output layer*: Unlike all layers in an ANN, the output layer neurons most commonly do not have an activation function (or can be thought of as having a linear identity activation function). This is because the last output layer is usually taken to represent the class scores (e.g. in classification), which are arbitrary real-valued numbers, or some kind of real-valued target (e.g. in regression).
- *Sizing neural networks*: The two metrics that are commonly used to measure the size of ANNs are the number of neurons, or more commonly the number of parameters. Working with the two example networks in figure 1.9:
  1. The first network (left) has $4 + 2 = 6$ neurons (not counting the inputs), $(3x4) + (4x2) = 20$ weights and $4 + 2 = 6$ biases, for a total of 26 learnable parameters.
  2. The second network (right) has $4+4+1 = 9$ neurons, $(3x4)+(4x4)+(4x1) = 12+16+4 = 32$ weights and $4+4+1 = 9$ biases, for a total of 41 learnable parameters.

Modern Convolutional Neural Networks contain an order of 100 million parameters and are usually made up of approximately 10-20 layers (hence the term *Deep Learning*). However, the number of effective connections is significantly greater due to parameter sharing. More on Convolutional Neural Networks in the next section.

1.3.4 Forward-step computation

Forward-step computation is the computation involved in predicting the output result using already known network parameters [3], [12]. The processing required for learning these parameters is called *training* and is usually done through *backward-step computation*. Based upon the explanations regarding the linear classifier and the right image of figure 1.7, it is visible that there is a close relationship between the matrix-vector multiplication formulation in the linear classifier problem and the mathematical model of a neuron. As a matter of fact, these are two representations of the same thing, meaning that ANNs are comprised of several multiplications.

ANNs are essentialy repeated matrix multiplications interwoven with activation functions. One of the primary reasons that ANNs are organized into layers is that this structure makes it very simple and efficient to evaluate neural networks using matrix-vector operations. Working with the example three-layer neural network in the figure 1.9 the input would be a $[3 \times 1]$ vector. All connection strengths for a layer can be stored in a single matrix. For example, the weights W_1 of the first hidden layer would be of size $[4 \times 3]$, and the biases for all units would be in the vector b_1, of size $[4 \times 1]$. Here, every single neuron has its weights in a row of W_1, so the matrix vector multiplication $W_1 x$ evaluates the activations of all neurons in that layer. Similarly, W_2 would be a $[4 \times 4]$ matrix that stores the connections of the second hidden layer, and W_3 a $[1 \times 4]$ matrix for the last (output) layer. The full forward pass of this 3-layer neural network is then simply three matrix multiplications, interwoven with the application of the sigmoid activation function. For better understanding, the pseudocode algorithm 1.1 is given.

Source code 1.1: *Mathematical representation of 3-layer ANN from fig. 1.9*

Input: Input vector x
Input: Weight parameters W
Output: Value of output neuron o
Let f (*be the element-wise sigmoid activation function*)
Let $h1 = f(W_1 x + b_1)$ (*that calculates the activations of the first hidden layer (4x1)*)
Let $h2 = f(W_2 h_1 + b_2)$ (*that calculates the activations of the second hidden layer (4x1)*)
Let $o = W_3 h_2 + b_3$ (*that calculates the value of the output neuron (1x1)*)

In the pseudocode, W_1, W_2, W_3, b_1, b_2, b_3 are the learnable parameters of the network. Notice also that the final network layer usually does not have an activation function.

1.4 Convolutional neural networks

A Convolutional Neural Network (CNN) is a special type of artificial neural network topology, that is inspired by the animal visual cortex and tuned for computer vision tasks by Yann LeCun in early 1990s [4]. It is a multi-layer perceptron, which is an artificial neural network model, specifically designed to recognize two-dimensional shapes. This type of network shows a high degree of invariance to translation, scaling, skewing, and other forms of distortion.

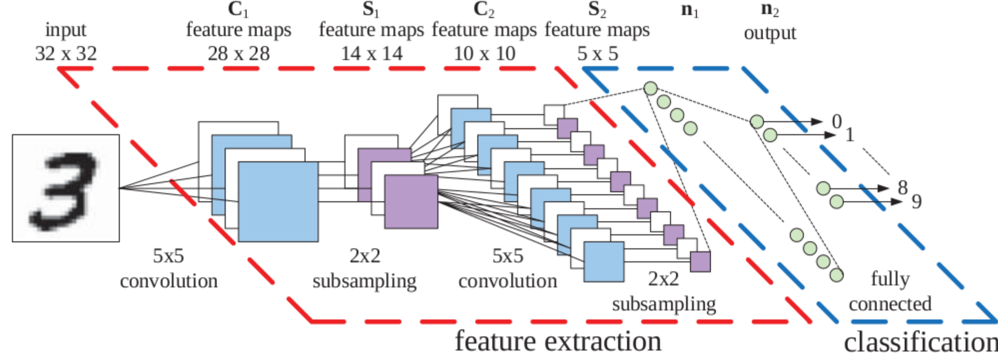


Figure 1.10: *Example topology of a CNN suitable for handwritten digit recognition*

In a CNN each neuron receives some inputs, performs a mathematical operation and optionally follows it with a non-linearity. The whole network still expresses a single score function: from the raw image pixels on one end to class scores at the other. Also, CNNs still have a loss function (e.g. softmax) on the last (fully-connected) layer and all the knowledge developed for regular ANNs still applies. The main difference of CNN architectures is that they make the explicit assumption that the inputs are images, which allows the encoding of certain properties into the architecture. The position invariance of the features makes it possible to reuse most of the results of the feature extractor, this makes a CNN very computationally efficient for object detection tasks.

1.4.1 Data arrangement in a CNN

CNNs take advantage of the fact that the input consists of images and they constrain the architecture in a more sensible way [5]. In particular, unlike a regular ANN, the layers of a CNN have neurons arranged in 3 dimensions: width, height, depth. Note that the word depth here refers to the third dimension of an activation volume, not to the depth of a regular ANN, which can refer to the total number of layers in the network. For example, 32×32 RGB input images represent an input volume of activations, that has dimensions $32 \times 32 \times 3$ (width, height, depth respectively). The neurons in a layer will only be connected to a small region of the layer before it, instead of all of the neurons in a fully-connected manner. The constrained, local connectivity of CNNs will be clarified shortly. An input 3D volume is shown in figure 1.11 for a 4×4 RGB image. It is noted that the range of values in the 3D volume can be adjusted to assist the training of the

network. The process of manipulating the input data, prior to feeding them to the CNN, in order to bring them to the desired form is called *preprocessing*.

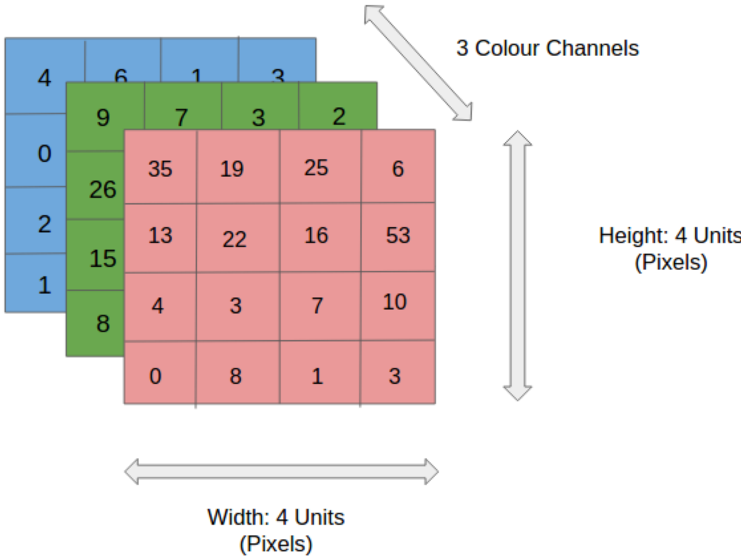


Figure 1.11: *Cross-section of an input volume of size:* $4 \times 4 \times 3$. *It comprises of the 3 color channel matrices of the input image.*

As a result, a CNN is made up of layers, each one having a simple interface. It transforms an input 3D volume to an output 3D volume using some function that may or may not have parameters. A visualization is shown in figure 1.12.

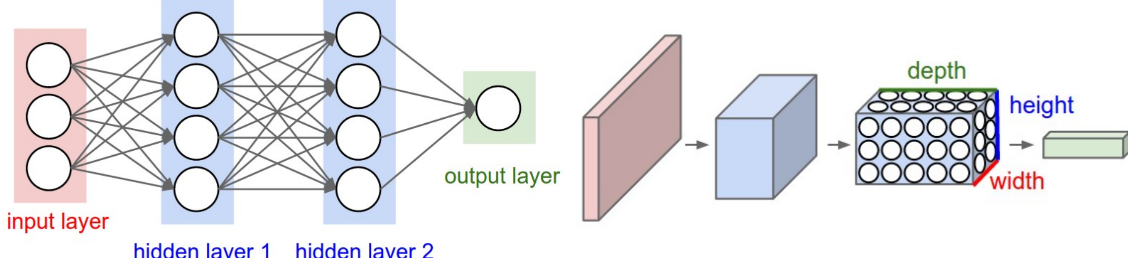


Figure 1.12: ***Left***: *A regular 3-layer Neural Network.* ***Right***: *A CNN arranges its neurons in three dimensions (width, height, depth), as visualized in one of the layers. Every layer of a CNN transforms the 3D input volume to a 3D output volume of neuron activations. In this example, the red input layer holds the image, so its width and height would be the dimensions of the image, and the depth would be 3 (Red, Green, Blue channels).*

1.4.2 Common layers used to build CNNs

There are three main types of layers to build CNN architectures: *Convolutional Layer*, *Pooling Layer*, and *Fully-Connected Layer* [3]. The latter is exactly as seen in regular ANNs. These three layers are stack interchangeably to produce interesting architectures. There is also the *Input Layer*, which is nothing more than the identity transform, i.e. its output is the same as its input. These layer are analyzed briefly below:

- *Input Layer*: Holds the raw pixel values of the input image. The depth of the Input Layer volume matches the number of channels of the input image. Also, the spatial dimensions of the input volume match the dimensions of the input image.

- *Convolutional Layer*: Will compute the output of neurons that are connected to local regions in the input. Each computation is a spatial (width, height) convolution between their weights and a small region they are connected to in the input volume. The depth of the output volume depends on the numbers of filters that is given to the layer as an extra parameter.

- *Pooling Layer*: Will perform a downsampling operation along the spatial dimensions (width, height).

- *Fully Connected (FC) Layer*: As with ordinary ANNs and as the name implies, each neuron in this layer will be connected to all the elements in the input volume of the layer.

- *ReLU Layer*: Will apply an elementwise activation function, namely $f(x) = max(0, x)$ thresholding at zero. This layer is used for helping the network training and leaves the size of the input volume unchanged.

In this way, CNNs transform the original image layer by layer from the original pixel values to the final class scores. Note that some layers contain parameters and other do not. In particular, the convolutional/fully-connected layers perform transformations that are a function of not only the activations in the input volume, but also of the parameters (the weights and biases of the neurons). On the other hand, the ReLU/pooling layers will implement a fixed function. The parameters in the convolutional/fully-connected layers are trained with gradient descent.

1.4.3 Convolutional Layer

The convolutional layer is the core building block of a CNN and lifts most of the computational burden [13]. Before proceeding any further, it is better to explain how convolution is performed, though a simple example.

Suppose the input image, which is commonly called *input feature map* is single channel, i.e. its 3D volume depth is 1. For the sake of this example, suppose the input image is 5×5 and the *kernel* of the convolution is 3×3. Their values are shown below.

$$Input = \begin{bmatrix} 3 & 3 & 2 & 1 & 0 \\ 0 & 0 & 1 & 3 & 1 \\ 3 & 1 & 2 & 2 & 3 \\ 2 & 0 & 0 & 2 & 2 \\ 2 & 0 & 0 & 0 & 1 \end{bmatrix} \qquad Kernel = \begin{bmatrix} 0 & 1 & 2 \\ 2 & 2 & 0 \\ 0 & 1 & 2 \end{bmatrix}$$

A discrete convolution is a linear transformation that is sparse (only a few input units contribute to a given output unit) and reuses parameters (the same weights are applied

to multiple locations in the input). Figure 1.13 provides an example of a discrete convolution. The light blue grid is the input feature map. To keep the drawing simple, a single input feature map is represented, but it is not uncommon to have multiple feature maps stacked one onto another. The kernel (shaded area) slides across the input feature map. At each location, the product between each element of the kernel and the input element it overlaps is computed and the results are summed up to obtain the output in the current location. The procedure can be repeated using different kernels to form as many *output feature maps* as desired.

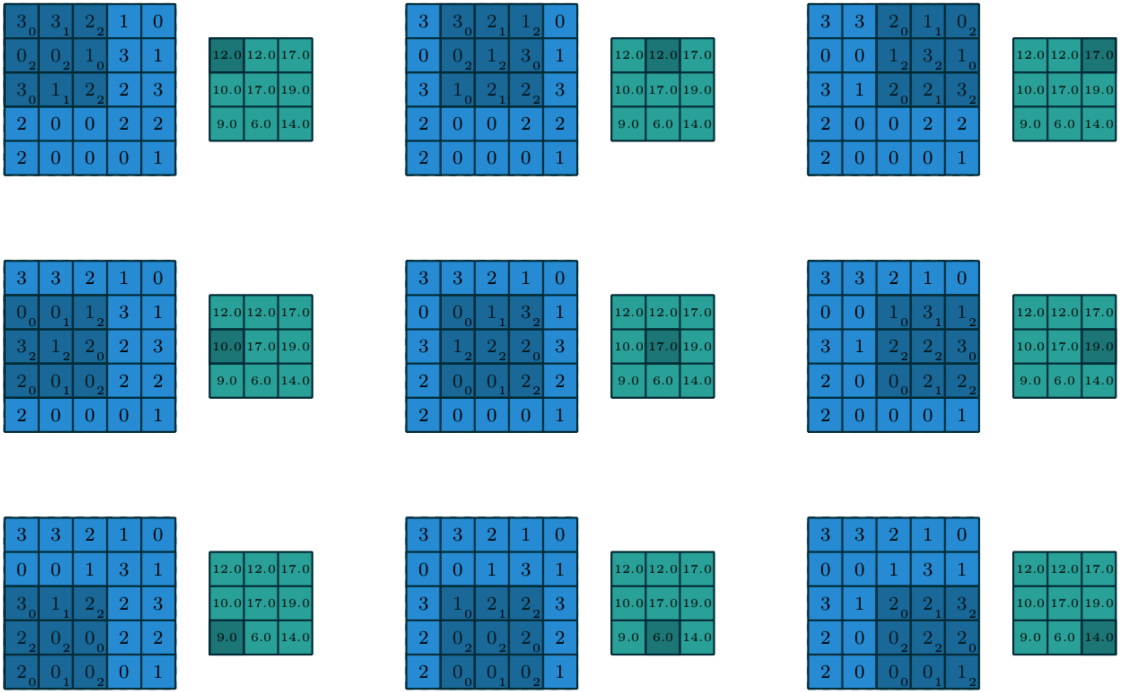


Figure 1.13: *Computing the output values of a discrete convolution. The shaded regions indicate the numbers involved at each step of the computation. The result is placed in the teal colored grid.*

Notice that the convolution can be also performed though matrix multiplication. This is clearly shown below:

$$
Output = \begin{bmatrix} 3 & 3 & 2 & 0 & 0 & 1 & 3 & 1 & 2 \\ 3 & 2 & 1 & 0 & 1 & 3 & 1 & 2 & 3 \\ 2 & 1 & 0 & 1 & 3 & 1 & 2 & 2 & 3 \\ 0 & 0 & 1 & 3 & 1 & 2 & 2 & 0 & 0 \\ 0 & 1 & 3 & 1 & 2 & 2 & 0 & 0 & 2 \\ 1 & 3 & 1 & 2 & 2 & 3 & 0 & 2 & 2 \\ 3 & 1 & 2 & 2 & 0 & 0 & 2 & 0 & 0 \\ 1 & 2 & 2 & 0 & 0 & 2 & 0 & 0 & 0 \\ 2 & 2 & 3 & 0 & 2 & 2 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 0 \\ 1 \\ 2 \\ 2 \\ 2 \\ 0 \\ 0 \\ 1 \\ 2 \end{bmatrix}
$$

With this representation, the convolution can be related to the model of interconnected

neurons presented for ANNs.

Extending the previous example for images of multiple input channels is straightforward. The parameters of this layer consist of a set of learnable *filters*. Every filter is spatially (i.e. along width and height) small, but extends through the full depth of the input volume. For example, a typical filter on a first layer of a CNN might have size 5x5x3, i.e. 5 pixels width and height stacked into 3 layers. The number three is because it is assumed that the input image has three color channels. During the forward pass, each filter is slided (more precisely, convolved) across the width and height of the input volume and dot products between the entries of the filter and the input is computed at any position. As the sliding of the filter over the width and height of the input volume occurs, it will produce a 2-dimensional activation map that gives the responses of that filter at every spatial position. There will be an entire set of filters in each convolutional layer (e.g. 12 filters), and each of them will produce a separate 2-dimensional activation map. These activation maps are stacked along the depth dimension and produce the output volume.

This paragraph will discuss the important matter of local connectivity [3]. When dealing with high-dimensional inputs such as images, it is impractical to connect neurons to all neurons in the previous volume. Instead, each neuron is connected only to a local region of the input volume. The spatial extent of this connectivity is a hyperparameter called the *receptive field* of the neuron. The extent of the connectivity along the depth axis is always equal to the depth of the input volume. It is important to emphasize again this asymmetry in how the spatial dimensions (width and height) and the depth dimension are treated: The connections are local in space (along width and height), but always full along the entire depth of the input volume. Notice , in conjunction with figure 1.14 that the neurons still compute a dot product of their weights with the input followed by a non-linearity, but their connectivity is now restricted to be local (spatially).

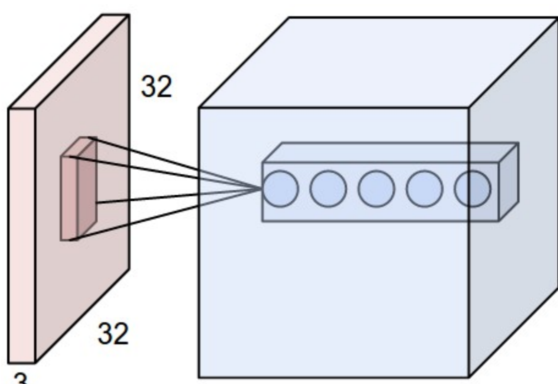


Figure 1.14: *An example input volume in red (e.g. a* 32×32 *RGB image), and an example volume of neurons in the first convolutional layer. Each neuron in the convolutional layer is connected only to a local region in the input volume spatially, but to the full depth (i.e. all color channels). Note, there are multiple neurons (5 in this example) along the depth, all looking at the same region in the input, each one contributing to the generation of a separate output feature map.*

Spatial arrangement

Previous paragraphs explained the connectivity of each neuron in the convolutional layer to the input volume, but did not discuss how many neurons exist in the output volume or how they are arranged [3]. Three *hyperparameters* control the size of the output volume, namely the depth, stride and the zero-padding.

- *Depth* of the output volume: This hyperparameter corresponds to the number of filters used, each learning to look for something different in the input. For example, if the first convolutional layer takes as input the raw image, then different neurons along the depth dimension may activate in presence of various oriented edges or blobs of color.

- *Stride*: Is the amount of sliding of the filter before performing a dot product. When the stride is 1 the filters move one pixel at a time. When the stride is 2, then the filters jump 2 pixels at a time as they get slided around. This will produce smaller output volumes spatially. In the example of figure 1.15 there is only one spatial dimension (x-axis), one neuron with a receptive field size of 3 and the input size is 7. The neuron weights are $[1, 0, -1]$ and its bias is zero. These weights are shared across all yellow neurons (see parameter sharing below).

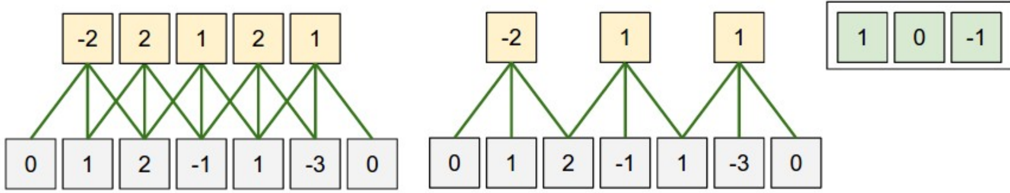


Figure 1.15: ***Left***: *The neuron strided across the input in stride of 1, giving output of size 5.* ***Right***: *The neuron strided across the input in stride of 2, giving output of size 3.*

- *Zero-padding*: is the amount of zeros around placed around the border of the input [5]. The size of the zero-padding is also a hyperparameter. Figure 1.16 shows an image (in red) with a zero-padding of one pixel. The nice feature of zero padding is

0	0	0	0	0	0
0	35	19	25	6	0
0	13	22	16	53	0
0	4	3	7	10	0
0	9	8	1	3	0
0	0	0	0	0	0

Figure 1.16: *An example of a zero-padded* 4×4 *matrix that becomes a* 6×6 *matrix.*

that it allows the control of the spatial size of the output volumes. Most commonly,

it is used to preserve the spatial size of the input volume so the input and output width and height are the same.

Figure 1.17 shows a visualization of convolution with zero-padding and non-unit stride [13].

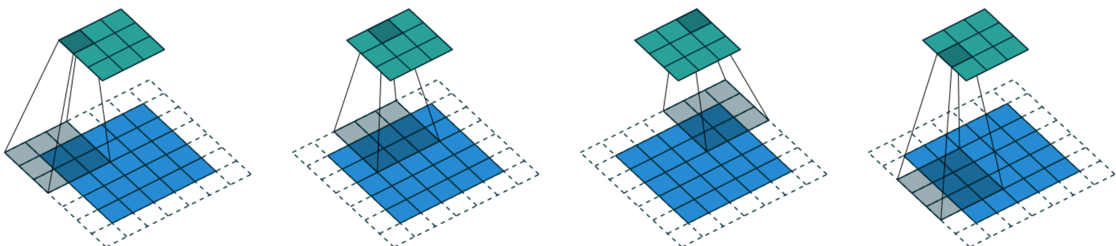


Figure 1.17: *Convolving a* 3×3 *kernel over a* 5×5 *input padded with a* 1×1 *border of zeros using* 2×2 *strides.*

Parameter Sharing

Is used in convolutional layers to control the number of parameters. It turns out that the number of parameters can be drammatically reduced by making one reasonable assumption: If one feature is useful to compute at some spatial position (x_1, y_1), then it should also be useful to compute at a different position (x_2, y_2). In other words, denoting a single 2-dimensional slice of depth as a *depth slice* (e.g. an input volume of size $[55 \times 55 \times 96]$ has 96 depth slices, each of size $[55 \times 55]$), then the neurons in each depth slice are constrained to use the same weights and bias. Notice that if all neurons in a single depth slice are using the same weight vector, then the forward pass of the convolutional layer be computed as followes: each depth slice is equal to a convolution of the neuron's weights with the input volume Put differently, the specific parameter sharing scheme converts a fully-connected layer into a convolutional one. This is why it is common to refer to the sets of weights as a filter (or a kernel), which is convolved with the input.

1.4.4 Pooling Layer

In a neural network, pooling layers provide invariance to small translations of the input [3], [5]. It is common to periodically insert a pooling layer in-between successive convolutional layers in a CNN architecture. Its function is to progressively reduce the spatial size of the representation, thus reducing the amount of parameters and computation in the network. The pooling layer operates independently on every depth slice of the input and resizes it spatially. The most common kind of pooling is *max* pooling, which consists in splitting the input in (usually non-overlapping) patches and outputting the maximum value of each patch. Other kinds of pooling exist, e.g., *mean* or *average* pooling, which all share the same idea of aggregating the input locally by applying a non-linearity to the content of some patches. In practice a frequent form is a pooling layer with filters of size 2×2 applied with a stride of 2, which downsamples every depth slice in the input by 2 along both width and height, discarding 75% of the activations.

Every max operation would in this case be taking a maximum over 4 numbers (little 2x2 region in some depth slice). The depth dimension remains unchanged.

The pooling layer operates in a similar manner as the convolutional layers. The main differences are:

- The pooling layer does need weights to operate. In fact it is just a fixed function applied to the input volume.
- In contrast with convolution, the pooling layer produces an output feature map for every depth slice of the input.

Figure 1.18 illustrates the operation of a pooling layer.

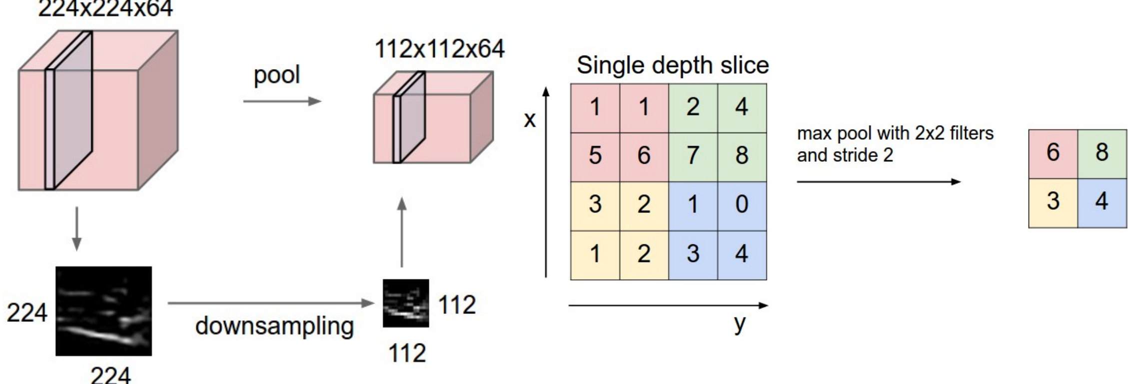


Figure 1.18: ***Left***: *In this example, the input volume of size* $[224 \times 224 \times 64]$ *is pooled with filter size 2, stride into the output volume of size* $[112 \times 112 \times 64]$. *Notice that the volume depth is preserved.* ***Right***: *The most common downsampling operation is max, giving rise to max pooling, here shown with a stride of 2. That is, each max is taken over 4 numbers (little 2x2 square).*

1.4.5 Fully-connected Layer

Neurons in a fully connected layer have full connections to all activations in the previous layer, as seen in regular ANNs. Their activations can hence be computed with a matrix multiplication followed by a bias offset. See section 1.3 explaining the ANNs for more information.

Chapter 2

Introduction to Caffe and Myriad2

This chapter is going to introduce basic terminology about the software and the hardware used that make up the CNN implementation. The reader is strongly suggested to pay attention to this chapter, since subsequent chapters will make references to notions defined here.

2.1 Caffe: Convolutional Architecture for Fast Feature Embedding

Caffe provides multimedia scientists and practitioners with a clean and modifiable framework for state-of-the-art deep learning algorithms and a collection of reference models. The framework is a BSD-licensed C++ library with Python and MATLAB bindings for training and deploying general-purpose convolutional neural networks and other deep models efficiently on commodity architectures. Caffe fits industry and internet-scale media needs by CUDA GPU computation, processing over 40 million images a day on a single K40 or Titan GPU ($\approx$ 2.5 ms per image). By separating model representation from actual implementation, Caffe allows experimentation and seamless switching among platforms for ease of development and deployment from prototyping machines to cloud environments. Caffe is maintained and developed by the Berkeley Vision and Learning Center (BVLC) with the help of an active community of contributors on GitHub. It powers ongoing research projects, large-scale industrial applications, and startup prototypes in vision, speech, and multimedia [6].

2.1.1 Layers

Caffe stores and communicates data in 4-dimensional arrays called *blobs*. Blobs provide a unified memory interface, holding batches of images (or other data), parameters, or parameter updates.

A Caffe layer is the essence of a neural network layer: it takes one or more blobs as input, and yields one or more blobs as output. Layers have two key responsibilities for the operation of the network as a whole: a *forward pass* that takes the inputs and produces the outputs, and a *backward pass* that takes the gradient with respect to the

output, and computes the gradients with respect to the parameters and to the inputs, which are in turn back-propagated to earlier layers. Caffe provides a complete set of layer types including: convolution, pooling, inner products, nonlinearities like rectified linear and logistic, local response normalization, elementwise operations, and losses like softmax and hinge. These are all the types needed for state-of-the-art visual tasks. Coding custom layers requires minimal effort due to the compositional construction of networks.

2.1.2 Training a network

Caffe trains models by the fast and standard stochastic gradient descent algorithm. Figure 2.1 shows a typical example of a Caffe network (for MNIST digit classification) during training [6]: a data layer fetches the images and labels from disk, passes it through multiple layers such as convolution, pooling and rectified linear transforms, and feeds the final prediction into a classification loss layer that produces the loss and gradients which train the whole network. This example is found in the Caffe source code at `examples/lenet/lenet_train.prototxt`. Data are processed in mini-batches that pass through the network sequentially. Vital to training are learning rate decay schedules, momentum, and snapshots for stopping and resuming, all of which are implemented and documented.

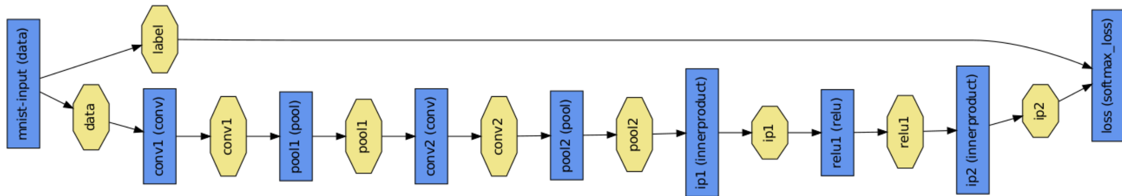


Figure 2.1: *An MNIST digit classification example of a Caffe network, where blue boxes represent layers and yellow octagons represent data blobs produced by or fed into the layers.*

2.1.3 Usage in the CNN implementation

In the context of the CNN implementation in Myriad2, the training of the network is performed by Caffe in an x86 machine. Afterwards, the blobs containing the trained parameters are copied and then placed inside the DDR of Myriad2. As a result, the implementation has all the required parameters for performing the forward pass, although this time the execution is performed on the specialized hardware of Myriad2.

2.2 Myriad 2 multiprocessor SoC

The target platform of implementation is the Myriad2 System-on-Chip (SoC) processing unit [7]. It is developed by Movidius Ltd, that recently joined Intel's Perceptual Computing Group to accelerate adoption of visually intelligent devices. Myriad2 delivers

high-performance machine vision and visual awareness in severely power-constrained environments. For that reason, it is the world's first Vision Processing Unit (VPU) that specifically targets embedded applications. The main characteristics of Myriad2 are:

- *An ultra-low power design*: For mobile and connected devices where battery life is critical, Myriad2 provides a way to combine advanced vision applications in a low power profile. This enables new vision applications in small form factors that could not exist before.

- *A high-performance processor*: Bringing vision technologies in connected devices closer to the capabilities of human vision is what Myriad2 is all about. It enables advanced vision applications that are impossible with conventional processors.

- *A programmable architecture*: The flexibility for developers to implement differentiated and proprietary applications is fundamental to Myriad2. The provided optimized software libraries give device manufacturers the ability to differentiate, not duplicate, at the core level.

- *A small-area footprint*: To conserve space inside mobile, wearable, and embedded devices, Myriad 2 was designed with a very small footprint that can easily be integrated into existing products.

A high level view of the hardware is shown in figure 2.2. From there, it is seen that Myriad2 SoC contains fourteen different processors. The two processors on the right are fundamentally different from the twelve vector processors on the left. In fact, the processors named "CPU" are of 32-bit SPARC architecture, which belongs to the RISC family of processors.

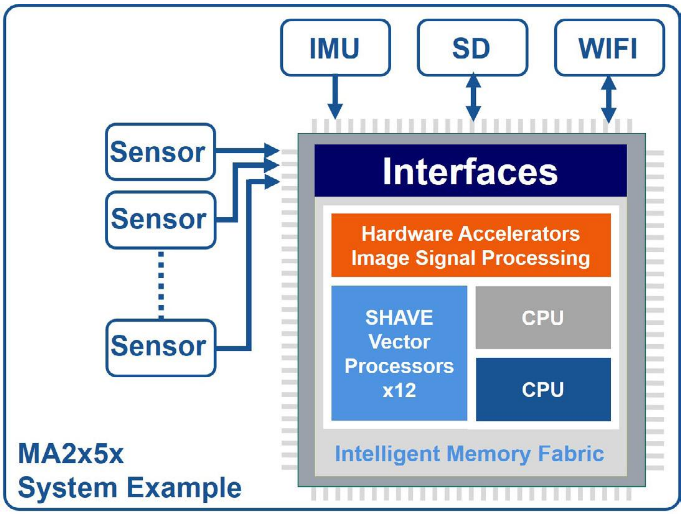


Figure 2.2: *Overview of Myriad2 hardware*

A more detailed view follows below [8]:

- *Leon OS*: Is one of the SPARC CPUs. It belongs to the CPU sub-system (CSS) that has been designed to be the main communication and control unit with the outside world via the external communication peripherals: I2C blocks, I2S blocks, SPI

blocks, UART, GPIO, ETH and USB3.0. The control unit of this block is the Leon OS (LOS) RISC processor, but in this block the Leon owns much bigger L1 (32 KB) and L2 (256 KB) caches, which allows to put a modern RTOS on it. This block also offers an AHB DMA engine for more optimal data transfer via the external peripherals. Beside handling the external interfaces and communication, Leon OS could also control SHAVE processors imaging algorithms.

- *Leon RT*: Is the second of the SPARC CPUs. It belongs to the Media sub-system (MSS), an architectural unit designed for allowing external connections with imaging devices (camera sensors, LCDs, HDMI controllers etc.) as well as allowing use of the Hardware (HW) filters available in Myriad2. As such it is comprised by the MIPI, LCD, CIF interfaces, the SIPP HW filters and well as the AMC block which enables connections between these and CMX (SRAM) memory. Coordinating frame input and controlling the pipelines set in place usually require some effort. As such the Myriad2 platform offers the Leon RT RISC as part of the MSS. Leon RT (LRT) is a RISC processor with a fair amount of L2 cache memory (32 KB). Leon RT is only one arbiter away from any Interface or HW filter register settings so it can efficiently change any required parameters of the MSS blocks with the minimum amount of delay due to bus arbitration.

- *SIPP*: Is a proprietary software/hardware mechanism used by the Myriad2 processor to achieve highly optimized scheduling of Image Signal Processing (ISP) pipeline functionality. This mechanism is responsible for utilizing the HW filters provided by Myriad2 to achieve the best performance possible. This component is the orange block shown in fig. 2.2.

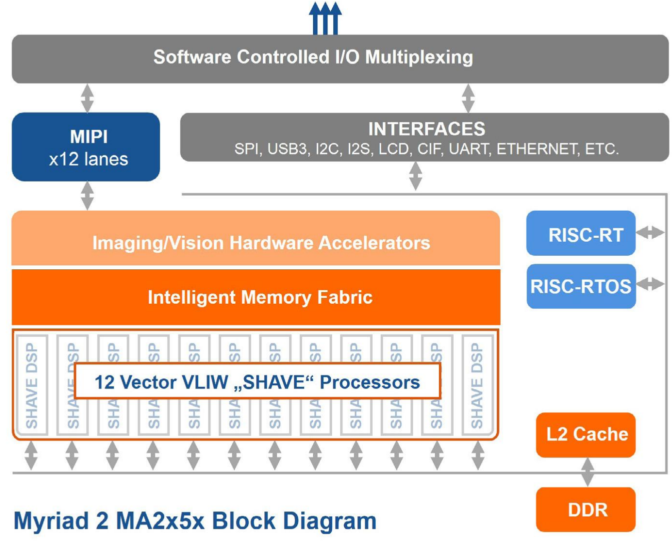


Figure 2.3: *More detailed overview of Myriad2 hardware*

- *Microprocessor Array (UPA)*: Is the unit in Myriad 2 holding the 12 Very Long Instruction Word (VLIW) SHAVE vector processors (see fig. 2.3), the 2 MB CMX SRAM

memory and a few other blocks from which the most important are: the specialized DMA engine and the 256 KB L2 cache memory available to the SHAVE cores. This unit's main purpose is to provide support for customized code required by many computer vision and machine learning applications, as well as any other general computation intensive algorithms. Each VLIW processor controls multiple functional units which have SIMD capability for high parallelism and throughput at a functional unit and processor level. Each of these units can be launched in parallel in a single instruction. SHAVEs support SIMD instructions on multiple types, including but not limited to: 8 bits integers, 16 bits integer, 32 bits integer, 16 bits float, 32 bits float.

- *CMX*: Stands for Connection Matrix, which belies the fact it is comprised of several smaller SRAM blocks reaching a total of 2 MB. Each SHAVE processor has preferential ports into a 128 KB slice of the CMX memory. As such, 12x128 KB = 1536 KB are preferentially used by SHAVE cores but the remaining 512 KB of CMX memory are generally usable by any other units. The recommended usage for these 512 KB is for HW SIPP filters usage or Leon OS timing critical code which would otherwise not be able to be kept in DDR.

- *DDR*: Is the largest volatile available memory unit of Myriad2 and has a size of 128MB or 512 MB, depending on the revision. The main difference between this and other platforms is that Myriad2 comes with DDR inside the SoC. However it memory is off-chip, meaning that the 14 processors use a single DDR controller to access it.

2.3 CMX DMA Controller

The CMX DMA resides between the 128-bit MXI bus and CMX memory [9]. It provides high bandwidth data transfers between CMX and DDR in either direction. It also supports data transfers from DDR back to DDR or from CMX to CMX, allowing data to be relocated within the same physical location. Figure 2.4 shows a high level description of the DMA engine.

The unit of work in the DMA engine is expressed though transaction tasks. Up to four linked lists of transactions are maintained in system memory, thus the DMA capability of serving transactions is not unlimited and can be easily flooded with requests if the programmer makes unregulated use of it.

2.4 Myriad2 Development Kit

The Myriad2 Development Kit (MDK) comprises common code which includes both drivers and components, and some example applications [8]. Also, the MDK provides an extensive build system - based on the GNU Makefile - that offers the means to build an

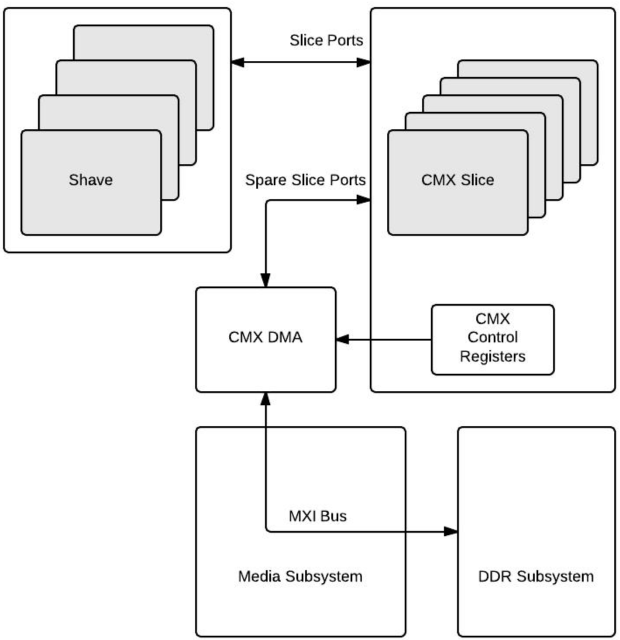


Figure 2.4: *CMX DMA engine of Myriad2*

application, the means to configure it and some functional targets, such as `make`, `make run` and `make start_server`.

2.4.1 MDK Components

This subsection provides a brief description of the reusable components included in the MDK [8]. Components are located under the `mdk/common/components` directory and selectively included in projects through the Makefile. A detailed description of each components can be found to the header file comments within each component. For the purposes of CNN implementation, an essential component is the `KernelLib/MvCV`, i.e. the Movidius Computer Vision kernel library. This library contains optimized assembly SHAVE routines for performing convolution, pooling and other related operations that are important to the implementation of the CNN.

Part II

Implementation

Chapter 3

Configuring and running a CNN architecture

This chapter describes all the steps required to make the CNN implementation ready to process an input image. It is suitable for programmers that need to use the existing implementation, without the overwhelming amount of details explaining the inner workings of the how everything is designed and written.

3.1 Description of a particular CNN

For better understanding, a specific CNN architecture is presented. The rest of the chapter will try to build the provided network in a step-by-step manner.

3.1.1 Pictorial representation of the CNN

Figure 3.1 shows the CNN.

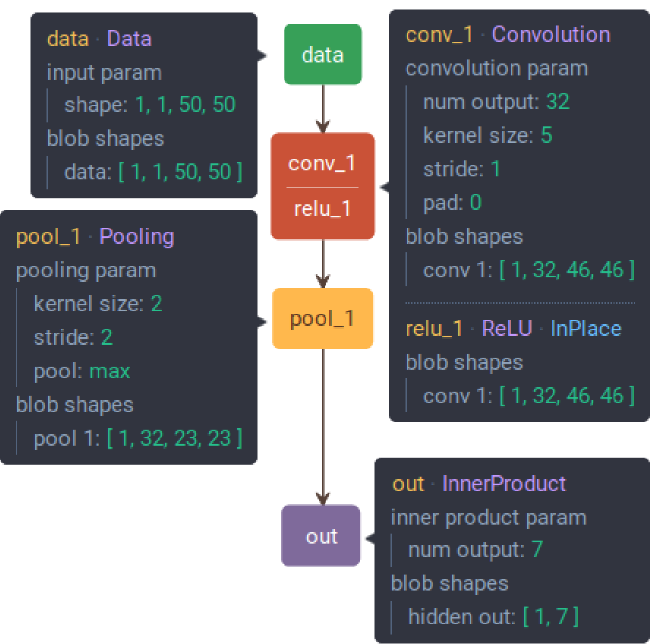


Figure 3.1: *An example CNN architecture that will be built with the CNN implementation.*

Note that the CNN presented in figure 3.1 consists of three computational nodes and uses square bracket syntax to declare the shape of the data moving between the nodes. The format of this syntax is $[A, B, C, D]$, where:

- A defines the batch size, i.e. the number of images that will be processed by the network in one forward pass. Although Caffe supports batch processing, the current CNN implementation does not. Thus, this value always needs to be 1.
- B define the number of channels in an image. For example, a grayscale image has only one channel, while an RGB image has three.
- C defines the height of the image.
- D defines the width of the image.

While the definitions above talk about images, they can be extended for image-like objects. Such objects are 3D volumes and have been extensively described in the previous chapters.
Explanation of the computational nodes follows below:

- *data*: This node feeds the data into the network. Its shape is $[1, 1, 50, 50]$. As a result, the input to this network is a 50×50 grayscale image.
- *conv_1*: This node performs a convolution to the input volume, which turns out to be an image, using a 5×5 kernel. It generates 32 output maps, which means that there are 32 different kernels that perform convolution. The depth of each kernel equals the depth of the input volume, thus each output map needs $1 \times 5 \times 5$ weight elements for the generation of each output map. In total, $[32 \times 1 \times 5 \times 5]$ weight elements are needed. Because padding is zero and stride is one, sliding the kernel across the image will generate 46 output pixels in each dimension. This means that the size of the output volume will be $[1 \times 32 \times 46 \times 46]$.
- *pool_1*: This node is a pooling node and does not need weight parameters to operate. It changes the spatial dimension of the input volume $[1 \times 32 \times 46 \times 46]$, reducing it to $[1 \times 32 \times 23 \times 23]$.
- *out*: This is a fully connected node that generates a vector of 7 elements and needs a matrix of weights to operate. In particular, it needs a $[7 \times (1 \times 32 \times 23 \times 23)]$ matrix, because the input 3D volume collapses into a vector and gets multiplied with a weights matrix whose size is given above. This matrix-vector multiplication results into a $[7 \times 1]$ matrix, i.e. a 7-element vector.

Finally notice that bias is also added to the result of each computational node, before it is fed to the next node. For convolutional nodes, the same value is added to each element of an output map. This value changes between output maps. For fully connected nodes, a bias vector is added to the result of the matrix-vector multiplication.

3.1.2 Storage of the weights required by the CNN

The previous subsection makes clear the need for storing the weights of the convolutional and fully connected nodes. These weights are stored inside C files in the following manner:

```
struct conv_weights
conv_1_weights = {
    .masks = {
        .depth = 1,
        .height = 5,
        .width = 5,
        .element_size = 2,
        .data = conv_1_masks_raw
    },
    .biases = {
        .depth = 32,
        .height = 1,
        .width = 1,
        .element_size = 2,
        .data = conv_1_biases_raw
    }
};
```

```
struct fc_weights
out_weights = {
    .matrix = {
        .depth = 1,
        .height = 7,
        .width = 32 * 23 * 23,
        .element_size = 2,
        .data = out_weights_raw
    },
    .biases = {
        .depth = 1,
        .height = 7,
        .width = 1,
        .element_size = 2,
        .data = out_biases_raw
    }
};
```

Notice the following:

- The dimensions of depth, height and width match those explained in the previous subsection.
- The `element_size` field defines the length of each element in the input data. The value 2 means that 16-bit floating point numbers are given as weights.
- `data` field holds a pointer to the actual weight data, which are:

```
#define DBUF __attribute__((section(".ddr_direct.data"), aligned (16)))

fp16 DBUF conv_1_masks_raw[32*1*5*5]             = { /* Data here */ };
fp16 DBUF conv_1_biases_raw[32]                  = { /* Data here */ };
fp16 DBUF hidden_out_weights_raw[7*(32 * 23 * 23)] = { /* Data here */ };
fp16 DBUF hidden_out_biases_raw[7]               = { /* Data here */ };
```

3.1.3 Provided API

Currently the implementation provides the Application Programming Interface (API) presented in table 3.1. This API allows the construction of several different CNNs, due to its modular design. The programmer is advised to carefully read the example source code 3.3 in order to make sure that has a very good grasp of how all of it fit together.

Table 3.1: *CNN implementation API*

C function	Brief description
`struct network_node *network_node_source_create(...);`	Defines the input data source
`struct network_node *network_node_conv_create(...);`	Creates a convolutional computational node and attaches it to the network
`struct network_node *network_node_pool_create(...);`	Creates a pooling computational node and attaches it to the network
`struct network_node *network_node_fc_create(...);`	Creates a fully connected computational node and attaches it to the network
`void network_execute(network_node *);`	Executes the network

Note: "..." indicate parameters that specify the number of SHAVEs, the type of computational kernel, padding, stride, weights (if necessary), etc.

3.2 Detailed explanation of the API

The current CNN implementation is flexible enough to allow the execution of various different CNNs. For this reason, there is be a way to assemble the desired neural network, using the provided API. After the assembly, the network is preprocessed and several necessary actions are performed. The network preprocessing occurs only once and brings the network to a state where it can accept input and process data over and over again.

3.2.1 API internals

The whole implementation is written in C, however the API follows an object oriented approach. Each computational node, such as convolution or pooling is obligated to implement the following API:

Source code 3.1: *API of each computational node of the CNN, leon/network/node.h*

```c
#ifndef __NETWORK_NODE_H__
#define __NETWORK_NODE_H__

#include "node_defines.h"

struct network_node {
    struct data_blob *(*get_output_data)(struct network_node *);
    void (*execute)(struct network_node *);

    struct network_node *next;
};

struct data_blob *get_output_data(struct network_node *);
void network_execute(struct network_node *);
```

```
15
16 #endif
```

As seen from source code 3.1, each node implements the following:

- The `struct network_node` keeps a pointer to two functions and also has a `next` pointer, which is used to link several network nodes together to make a linked list. Currently the implementation does not support the execution of arbitrary Directed Acyclic Graphs (DAGs). The reason is because a DAG requires the existence of a scheduler, which is a very complicated piece of software on each own.

- `get_output_data`, which uses the `get_output_data` pointer of the `network_node` structure, with the purpose of returning the data produced by the current node. This function facilitates the linkage of several nodes in a chain. Each next node uses this function to access the data of the previous node in the architecture.

- `network_execute`, which uses the `execute` pointer of the `network_node` structure to initiate the computation of the current node.

  Reiterating, each computational node implements two functions. The first function is `network_execute` and the second is `get_output_data`. At the same time, nodes use several other functions to implement their functionality. However, these functions are not exposed to the rest of the code.

Using this primitive data layout the programmer is able to instantiate the computational nodes. Each type of node fills the fields of the `network_node struct` in a different way, thus providing different functionality. The instantiation functions are:

- `network_node_source_create`: With this function the programmer is able to create the input data source that will be fed into the network for computation. It returns a `struct network_node` pointer. All the instantiation functions - mentioned below - follow the same pattern, so this is a good place to present some code.

Source code 3.2: *leon/network/node_source.c*

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <string.h>
4
5 #include "node_defines.h"
6 #include "node.h"
7
8 #include "node_source.h"
9
10 static struct data_blob *
11 get_output_data_source(struct network_node *node)
12 {
13     struct network_node_source *n = (struct network_node_source *)node;
14
```

```
15     return &(n->output_blob);
16 }
17
18 static void
19 execute_source(struct network_node *node) {
20      ERROR("Cannot execute a source node");
21 }
22
23 struct network_node *
24 network_node_source_create(struct data_blob *output_data_blob)
25 {
26     struct network_node_source *source = malloc(sizeof(*source));
27
28     // Connect the methods
29     source->node.get_output_data = get_output_data_source;
30     source->node.execute = execute_source;
31
32     memcpy(&(source->output_blob), output_data_blob,
33                                 sizeof(struct data_blob));
34
35     return &source->node;
36 }
```

Source code 3.2 shows the following:

1. The `network_node_source_create` allocates (line 26) and prepares (lines 29-32) the data structure presented in source code 3.1.
2. The actual implementations of `get_output_data` and `execute` are not exposed, since they are prefixed with the keyword `static`.
3. The input and output data are modeled as 3D volumes with the use of the `struct data_blob`, which is:

   ```
   typedef u8 byte_t;

   struct data_blob {
       int depth;
       int height;
       int width;
       int element_size;
       byte_t *data;
   };
   ```

   Data of a 3D volume are stored as contiguous arrays in row-major order. In a row-major order, the consecutive elements of a row reside next to each other. While the term alludes to the rows of a two-dimensional array, i.e. a matrix, the orders can be generalized to arrays of any dimension by noting that the term row-major is equivalent to lexicographic order.

- network_node_conv_create: With this function the programmer is able to create a convolutional node and attach it to the network. Thus, during the creation of the node, the options below are available:

  1. out_maps: The number of output maps that will be generated.
  2. ddr_function: The specific routine that will perform the convolution. The name of this routine is exported by the SHAVE array library which will be described in the next chapter. For now, it is sufficient to know that such a name looks like "MV_conv5x5s1hhhh" for convolution with 5×5 kernel and unit stride.
  3. padding_v: Is the vertical padding used during convolution.
  4. padding_h: Is the horizontal padding used during convolution. Details about padding are provided in the previous chapter.
  5. with_relu: Indicates whether the convolution will apply the ReLU function to the output 3D volume.
  6. alignment: This is an optimization parameter used for increasing the performance. The numbers allowed are powers of 2, such as 1, 2, 4, 8, etc. Explanation about this parameter will be given in the chapter after next. For now, the programmer is suggested to make a bruteforce search in order to find the value that performs the best.
  7. coalescing_num: This is also an optimization parameter, which will be explained in the chapter after next. The programmer is suggested to start from the number 1 and keep increasing the value until execution fails due to memory exhaustion.
  8. shaves_no: This value defines how many SHAVEs will be used to execute the convolution. Increasing this number will most likely reduce the execution time. The way this operation is parallelized will be explained in the chapter after next. There are in total 12 SHAVEs, so shaves_no $\in [1, 12]$. The programmer is suggested to start from the number 1 and move sequentially to the number 12. The point to stop increasing this number is when little to no gain is observed.
  9. struct conv_weights *weights: Points to the weights needed by the convolution. This struct defines a 3D volume and follows the same principles as the struct data_blob. An example will be given shortly that will clarify how it is used.
  10. struct network_node *prev_node: This pointer is needed to create the linked list of nodes. The list is important, because it defines the data flow, i.e. how the output data of one node are used as the input data in the next node.

- network_node_pool_create: With this function the programmer is able to create a pooling node and attach it to the network. Thus, during the creation of the node, the options below are available:

1. `ddr_function`: The specific routine that will perform the pooling. The name of this routine is exported by the SHAVE array library which will be described in the next chapter. For now, it is sufficient to know that such a name looks like "`MV_maxPool2x2s2hh`" for 2×2 max pooling with 2×2 stride.
2. `padding_v`: Is the vertical padding used during pooling.
3. `padding_h`: Is the horizontal padding used during pooling. Details about padding are provided in the previous chapter.
4. `shaves_no`: This value defines how many SHAVEs will be used to execute the pooling. Increasing this number will most likely reduce the execution time. The way this operation is parallelized will be explained in the chapter after next. There are in total 12 SHAVEs, so `shaves_no` $\in [1, 12]$. The programmer is suggested to start from the number 1 and move sequentially to the number 12. The point to stop increasing this number is when little to no gain is observed.
5. `struct network_node *prev_node`: This pointer is needed to create the linked list of nodes. The list is important, because it defines the data flow, i.e. how the output data of one node are used as the input data in the next node.

- `network_node_fc_create`: With this function the programmer is able to create a fully connected node and attach it to the network. Thus, during the creation of the node, the options below are available:

  1. `ddr_function`: The specific routine that will perform the matrix-vector multiplication. For now, the only available option is "`MV_matvecmul_hhhh`".
  2. `with_relu`: Indicates whether the matrix-vector multiplication will apply the ReLU function to the output matrix.
  3. `shaves_no`: This value defines how many SHAVEs will be used to execute the matrix-vector multiplication. Increasing this number will most likely reduce the execution time. The way this operation is parallelized will be explained in the chapter after next. There are in total 12 SHAVEs, so `shaves_no` $\in [1, 12]$. The programmer is suggested to start from the number 1 and move sequentially to the number 12. The point to stop increasing this number is when little to no gain is observed.
  4. `struct fc_weights *weights`: Points to the matrix needed by the matrix-vector multiplication. This `struct` defines a 3D volume (although only two dimensions are used) and follows the same principles as the `struct data_blob`. An example will be given shortly that will clarify how it is used.
  5. `struct network_node *prev_node`: This pointer is needed to create the linked list of nodes. The list is important, because it defines the data flow, i.e. how the output data of one node are used as the input data in the next node.

Source code 3.3 demonstrates how the provided API is used to construct a CNN.

Source code 3.3: *Snippet of leon/network/network.c*

```
static struct network_node *data;
static struct network_node *conv_1;
static struct network_node *pool_1;
static struct network_node *out;

void prepare_network()
{
    struct data_blob
    input_data = {
        .depth = 1,
        .height = 50,
        .width = 50,
        .element_size = 2,
        .data = input_data_raw
    };

    data = network_node_source_create(&input_data);
    conv_1 = network_node_conv_create(32,                  // Output maps number
                                      MV_conv5x5s1hhhh,    // Convolution type
                                      0,                   // Vertical padding
                                      0,                   // Horizontal padding
                                      1,                   // With/without ReLU
                                      2,                   // Alignment
                                      6,                   // Coalescing Number
                                      12,                  // Shaves number
                                      &conv_1_weights,     // Weights required
                                      data);               // Previous node

    pool_1 = network_node_pool_create(MV_maxPool2x2s2hh,// Pooling type
                                      0,                   // Vertical padding
                                      0,                   // Horizontal padding
                                      5,                   // Shaves number
                                      conv_1);             // Previous node

    out = network_node_fc_create(MV_matvecmul_hhhh,
                                      1,                   // With/without ReLU
                                      6,                   // Shaves number
                                      &out_weights,        // Weights required
                                      pool_1);             // Previous node
}

void execute_network() {
    network_execute(conv_1);
    network_execute(pool_1);
    network_execute(out);
}
```

As seen from source code 3.3, there are two steps involved. The first step is the execution of `prepare_network`. It uses data stored in DDR, shapes them into 3D volumes and creates the nodes required by the specific CNN architecture. This step is also called

preprocessing, since everything necessary during execution that can be calculated in advance, is being computed. The next step involves the execution of a CNN with the provided input image. Using preprocessing, the time spent during execution is minimized, which is a very desired feature for the following reasons:

- The preprocessing step is done only once during initialization and helps to reduce the time during execution. A lot of applications are real-time, meaning that they need to fetch and process images continuously and with high frame rate, making preprocessing essential.
- When the CNN is not used frequently, one approach would be to turn off Myriad2 completely. However, Myriad2 provides advanced power management features, one of them being sleep mode. Thus, in this scenario the SoC could be put into sleep mode, which keeps the data in memory intact. As a result, the step of preprocessing does not need to be performed more than once, saving both power and time.

In conclusion, preprocessing is a very important step and there are multiple scenarios - such as those described above - that benefit greatly from it. Therefore, in every implementation of a CNN API the preprocessing step should become an indispensable part.

Chapter 4

Description of the source code peripherals

This chapter provides information about the code residing in the CMX and DDR memory and explains how this code is used by the SHAVE processors. However, the first step is to configure the Myriad2 SoC, by setting up the processor frequency and the caches. The purpose is to describe the need for the existence of this code and how several parts of it are bound together. It is important to emphasize that the present chapter is not going to explain in detail how convolution, pooling and inner product are written. Such details will be the objective of the next chapter.

4.1 Memory Layout

4.1.1 Why CMX is not enough

The paradigm proposed by the MDK examples regarding the way to write applications for Myriad2 is as follows:

- Leon OS processor is used to communicate with peripherals, such as USB and Ethernet. This is because Leon OS provides extensive interrupts support that make it suitable for being able to host the RTEMS operating system. As a result, it is suggested to place the operating system in DDR - that is much larger - posing no space constraints to the developer of the application.

- Leon RT processor is used to communicate with other peripherals, such as cameras and microphones. This is because Leon RT provides extensive hardware support for multimedia peripherals and relating protocols. Also, this processor in closer to the SIPP engine and can manage the SHAVE processors directly, making it suitable for managing the execution of parallel tasks in the SHAVEs. As a result, it is suggested to place the code of this processor in CMX, if it is small enough. Otherwise, the only remaining option is utilizing DDR for code of Leon RT.

- SHAVE processors are used for the computational intensive tasks. These processors are the main unit of work in Myriad2. They perform the bulk of the computations, thus speed is an essential property. For that reason, it is strongly recommended to place SHAVE code in CMX exclusively. Although this would be great, it

is almost impossible to achieve is larger applications. Implementation of CNNs is such an application and compromises need to be made.

4.1.2 Proposed memory map

Many modern neural networks are quite large for an embedded device. The CMX memory available for each SHAVE is only 128KB. A large network requires multiple variations of common operations such as convolution and pooling. For instance, conv7x7, conv5x5, pool3x3, pool2x2 with different amounts of striding are required. For improved performance, these variations are written in assembly language to fully exploit the SIMD capabilities of the ISA provided by Myriad2. As a result, placing them inside CMX leaves little to no space available for local buffers and other logic. Therefore CMX memory presents a limitation - at a very early stage - on the capability of Myriad2 to run larger networks.

Another important reason that justifies the relocation of code away from CMX is the existence of local buffers. The main idea of processing the data is based on the paradigm of bringing data from DDR to the CMX, which is close to the SHAVEs, processing the data, and finally writing the results back to the DDR. This requires the presence of local buffers to store intermediate data and other temporary results before and after the processing step. Also, local buffers help to improve the overall performance, since the bandwidth of SHAVEs to the CMX is practically unlimited.

The reasons described above justify the use of another paradigm, different from the paradigm proposed by the MDK examples. Figure 4.1 shows the proposed memory map.

Before explaining fig. 4.1 in detail, description of the memory structure is required. Memory areas of Myriad2 are shown in table 4.1.

Table 4.1: *Memory areas of Myriad2*

Memory	Size	LEON Access Cost	SHAVE access cost	Start Address
CMX	2 MB	Low	Low	0x70000000
DDR	128 MB	High Low when cache hit	High for random access Moderate when L2 hit Low for L1 hit	0x80000000

In particular, table 4.2 gives a deeper look in the CMX. In fact, CMX is comprised of several smaller SRAM blocks, which make it extremely fast. In other words, CMX is much like a cache, but it is manually controlled by the programmer. The CMX memory of 2 MB may be considered as 16x128 KB “slices”.

A couple of notes regarding CMX are worth mentioning at this point:

- Each SHAVE has higher bandwidth/lower power access to its “own” local slice.
- Slice locality follows SHAVE number: Shave0 is assigned the lowest 128 KB of CMX, ..., Shave11 is assigned to slice 11.

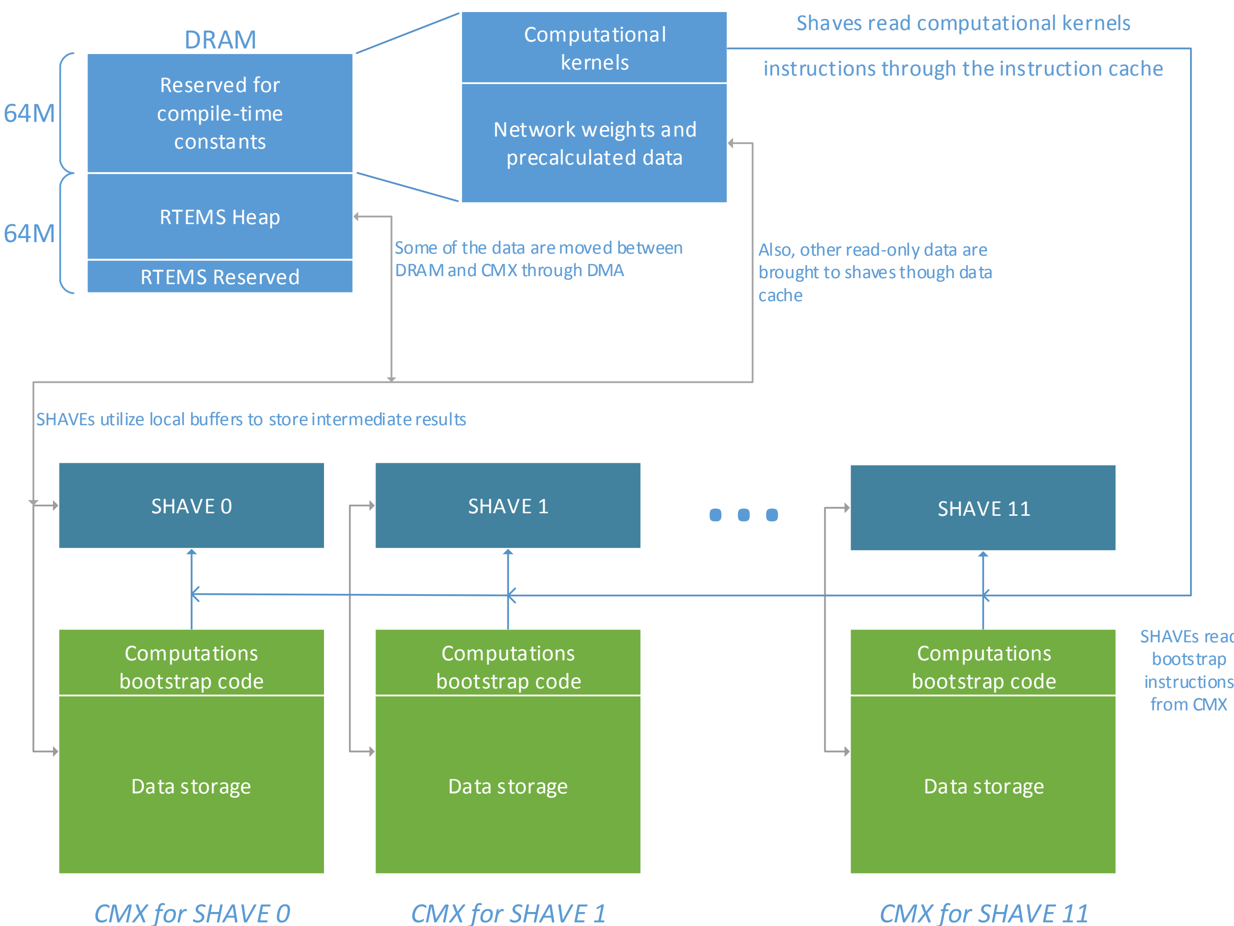


Figure 4.1: *Proposed memory map that is suitable for CNNs*

- Slices 12 to 15 are not tied to any SHAVE. They may be freely used for any other purposes.

Important to notice that there is also a possibility of accessing data from memory in an uncached manner. Both the DDR and the CMX memory have uncached views of the memory address space. In the case of DDR, addresses having MSB “0” for example: 0x8******* represent cached views while addresses starting with MSB “1”, for example 0xC******* represent uncached views. For CMX, 0x78****** represent uncached views and 0x70****** represent cached views. This feature allows easy sharing of control data between the Leon OS and SHAVEs.

The following bullets explain in detail fig. 4.1:

- DDR is split into two areas. The lowest 64MB are assigned to the Leon OS processor that runs the RTEMS operating system. Not so much space is needed by RTEMS itself. Most of this space is used by the application, that performs memory allocation operations to keep the data generated by the CNN nodes. The highest 64MB of DDR are used to store the network parameters. These parameters are learned during the training phase of the CNN and are used to perform the com-

Table 4.2: *CMX memory overview*

Slice	Start Address	End Address
0	0x70000000	0x7001FFFF
1	0x70020000	0x7003FFFF
2	0x70040000	0x7005FFFF
3	0x70060000	0x7007FFFF
4	0x70080000	0x7009FFFF
5	0x700A0000	0x700BFFFF
6	0x700C0000	0x700DFFFF
7	0x700E0000	0x700FFFFF
8	0x70100000	0x7011FFFF
9	0x70120000	0x7013FFFF
10	0x70140000	0x7015FFFF
11	0x70160000	0x7017FFFF
12	0x70180000	0x7019FFFF
13	0x701A0000	0x701BFFFF
14	0x701C0000	0x701DFFFF
15	0x701E0000	0x701FFFFF

putation of several types of nodes. Not every type of node needs such parameters to perform its computation. Convolutional and fully connected nodes need these parameters and call them weights. On the other hand, pooling nodes do not need such parameters at all.

- CMX is assigned to the SHAVEs the usual way. Each shave is assigned its own local slice to utilize during the computation. The remaining slices - slices 12 to 15 - are used by all the shaves to store shared parameters. More details will be provided in the following sections.

- Each CMX slice is mostly used for data, rather than code. The code inside the SHAVEs is minimal and its purpose it to act as an entry point to the actual code that needs to run. The actual code resides in RAM and the code residing in CMX tries to reach the appropriate part of the code in RAM needed for the particular computation. That is why the figure refers to the entry point code as bootstrap code.

- In order to increase performance, utilization of cache subsystem is needed. Myriad2 provides cache hierarchies for Leon OS, Leon RT and the SHAVE processors. The goal is to use cache of SHAVEs, in order to diminish the impact of accessing the DDR from these processors. The cache is utilized the following ways:

  1. Instruction cache: It is used for executing the code that describes the computation. Every computational node, such as convolution or pooling may come in different flavors, each one optimized for a particular class of the input size.

Small CNNs do not require large size of code to execute, which makes it possible to fit this code inside CMX. However, for larger CNNs this approach is not viable. A general solution that can support the code size of each CNN, without sacrificing most of the performance, is using the instruction cache. Also, another reason that makes cache a very attractive choice is that each computational node is usually run in parallel from multiple SHAVE processors. As a result, the exact same code is executed by several SHAVEs. This temporal locality of accesses is a clear indication that cache can perform well.

2. Data cache: Data cache is mostly used to increase the throughput of the DDR. Myriad2 provides an advanced DMA engine that can transfer data asynchronously between DDR and CMX. However, the resources of this engine are finite and need to be used wisely. The DMA engine is used for transferring the output data of the previous computational node to the current computational node and also for transferring the output data of the current computational node to the next computational node. These operations exhaust the resources of the DMA engine. However, several nodes need extra trained parameters (e.g. weights) on top of the input data to operate. In particular, the weights needed by convolutional nodes are the kernel masks. Due to the nature of convolution and the optimized code used (that will be described in detail in the next chapter), these weights are needed in small quantities every once in a while, making the data cache a suitable choice for this kind of data. As a result, DDR data are transfered to/from CMX both implicitly - though cache - and explicitly - through calls to the DMA engine.

3. Note: The several cache hierarchies in Myriad2 are not related to each other through a cache coherence protocol. Cache coherence needs to be maintained by the programmers themselves. However, the proposed paradigm uses cache only for read-only data, making the need for keeping the caches coherent unnecessary.

4.1.3 Creating the memory map in code

In order to create the memory map described previously, the GNU Linker needs to be used [14]. The linker script only describes how each slice of CMX memory is split for data and code and defines the regions of memory used by Leon OS and Leon RT.

Source code 4.1: *scripts/ld/custom.ldscript*

```
1  MEMORY
2  {
3    SHV0_CODE (wx)  : ORIGIN = 0x70000000 + 0 * 128K,       LENGTH = 32K
4    SHV0_DATA (w)   : ORIGIN = 0x70000000 + 0 * 128K + 32K, LENGTH = 96K
5
6    SHV1_CODE (wx)  : ORIGIN = 0x70000000 + 1 * 128K,       LENGTH = 32K
7    SHV1_DATA (w)   : ORIGIN = 0x70000000 + 1 * 128K + 32K, LENGTH = 96K
8
9    SHV2_CODE (wx)  : ORIGIN = 0x70000000 + 2 * 128K,       LENGTH = 32K
```

```
10   SHV2_DATA (w)    : ORIGIN = 0x70000000 + 2 * 128K + 32K, LENGTH = 96K
11
12   SHV3_CODE (wx)   : ORIGIN = 0x70000000 + 3 * 128K,       LENGTH = 32K
13   SHV3_DATA (w)    : ORIGIN = 0x70000000 + 3 * 128K + 32K, LENGTH = 96K
14
15   SHV4_CODE (wx)   : ORIGIN = 0x70000000 + 4 * 128K,       LENGTH = 32K
16   SHV4_DATA (w)    : ORIGIN = 0x70000000 + 4 * 128K + 32K, LENGTH = 96K
17
18   SHV5_CODE (wx)   : ORIGIN = 0x70000000 + 5 * 128K,       LENGTH = 32K
19   SHV5_DATA (w)    : ORIGIN = 0x70000000 + 5 * 128K + 32K, LENGTH = 96K
20
21   SHV6_CODE (wx)   : ORIGIN = 0x70000000 + 6 * 128K,       LENGTH = 32K
22   SHV6_DATA (w)    : ORIGIN = 0x70000000 + 6 * 128K + 32K, LENGTH = 96K
23
24   SHV7_CODE (wx)   : ORIGIN = 0x70000000 + 7 * 128K,       LENGTH = 32K
25   SHV7_DATA (w)    : ORIGIN = 0x70000000 + 7 * 128K + 32K, LENGTH = 96K
26
27   SHV8_CODE (wx)   : ORIGIN = 0x70000000 + 8 * 128K,       LENGTH = 32K
28   SHV8_DATA (w)    : ORIGIN = 0x70000000 + 8 * 128K + 32K, LENGTH = 96K
29
30   SHV9_CODE (wx)   : ORIGIN = 0x70000000 + 9 * 128K,       LENGTH = 32K
31   SHV9_DATA (w)    : ORIGIN = 0x70000000 + 9 * 128K + 32K, LENGTH = 96K
32
33   SHV10_CODE (wx)  : ORIGIN = 0x70000000 + 10 * 128K,       LENGTH = 32K
34   SHV10_DATA (w)   : ORIGIN = 0x70000000 + 10 * 128K + 32K, LENGTH = 96K
35
36   SHV11_CODE (wx)  : ORIGIN = 0x70000000 + 11 * 128K,       LENGTH = 32K
37   SHV11_DATA (w)   : ORIGIN = 0x70000000 + 11 * 128K + 32K, LENGTH = 96K
38
39   /* The CMX_DMA section must be between the following addreses
40    *   0x78000000 + 12 * 128K
41    *        and
42    *   0x78000000 + 13 * 128K
43    */
44   CMX_DMA_DESCRIPTORS  (wx) : ORIGIN = 0x78000000 + 12 * 128K , LENGTH = 12K
45   CMX_OTHER (wx) : ORIGIN = 0x70000000 + 12 * 128K + 12K, LENGTH = 256K - 12K
46
47   LOS (wx) : ORIGIN = 0x80000000, LENGTH = 64M
48   LRT (wx) : ORIGIN = 0x70000000 + 14 * 128K, LENGTH = 256K
49
50   DDR_DATA (wx) : ORIGIN = 0x80000000 + 64M, LENGTH = 64M
51
52 }
53
54 INCLUDE myriad2_leon_default_elf.ldscript
55 INCLUDE myriad2_shave_slices.ldscript
56 INCLUDE myriad2_default_general_purpose_sections.ldscript
```

With this script the following arrangement is made:

- 32KB of CMX are used for code and 96KB of CMX are used for data in each SHAVE.

- 12KB of CMX are used explicitly by the DMA Engine.
- 244KB of CMX are used for other purposes. In particular, this space will be used for placing shared parameters used by the SHAVEs.
- 64MB of DDR are used by the Leon OS and the RTEMS operating system.
- 256KB of CMX are used by the Leon RT. In fact, for this application Leon RT is not needed. However this space is provided for future extension of the implementation.
- Finally, the rest 64MB of DDR are used for placing parameters/weights of the CNN.

In order to force the MDK to use this file as the default linker script the statement

```
LinkerScript = ./scripts/ld/custom.ldscript
```

needs to be added to the Makefile.

4.2 Setting up Myriad2 SoC

This section is concerned about the proper way to configure the Myriad2 hardware. Care must be taken, because there are multiple drivers involved in the process. The programmer is encouraged to delve into the details of the driver implementations for better understanding of the code provided by the MDK. The final goal is to reach a state where source code 4.2 can be executed.

Source code 4.2: *leon/main.c*

```
1 #include <stdlib.h>
2 #include <stdio.h>
3
4 #include <rtems.h>
5 #include <app_config.h>
6
7 #include <DrvLeon.h>
8 #include <OsDrvTimer.h>
9
10 extern u32 jumpTable;
11 u32 jumpTableAddr;
12
13 void POSIX_Init (void *args)
14 {
15     UNUSED(args);
16
17     // This needs to be an uncached store.
18     jumpTableAddr = (u32)&jumpTable;
19
20     s32 sc;
21
22     // Initiallize Clocks, DDR and enable subsystems
23     sc = InitClocksAndMemory();
```

```
24      if (sc) {
25          puts("initClocksAndMemory failed");
26          exit(sc);
27      }
28
29      sc = InitShaveL2C();
30      if (sc) {
31          puts("InitShaveL2C failed");
32          exit(sc);
33      }
34
35      sc = ConfigShaveL2C();
36      if (sc) {
37          puts("ConfigShaveL2C failed");
38          exit(sc);
39      }
40
41      sc = OsDrvTimerInit();
42      if (sc) {
43          printf("Error initializing time driver.\n");
44          exit(sc);
45      }
46
47      printf("System frequency: %d Mhz\n\n",
48                  DrvCprGetClockFreqKhz(SYS_CLK, NULL) / 1000);
49
50      prepare_network();
51
52      while (1) {
53          read_image();
54          execute_network();
55          return_results();
56      }
57
58      exit(0);
59  }
```

A couple of explanations are needed:

- The code executed by Leon OS is placed into the directory `leon`. As a result, the proper compiler is selected automatically by the MDK build system.
- The entry point of the whole application is named `POSIX_Init`, as shown in line 13. This name is mandatory if the programmer wants to use the RTEMS operating system with thread support.
- Line 10 uses the extern symbol `jumpTable` which then passes it to the global variable `jumpTableAddr`. The purpose of `jumpTable` is to support execution of SHAVE code from DDR. The next sections explain in detail how this variable is used.

- Lines 20-45 deal with the hardware setup. There is nothing more to be said about it right now, because it will be discussed shortly in detail.
- Since Leon OS uses the RTEMS operating system, the include `rtems.h` is needed, as shown in line 4.
- Finally, lines 50-56 hint the required operations for the CNN implementation. The `prepare_network` call is called once and prepares the CNN. Lines 52-55 indicate that the CNN is capable of reading new data, processing them and returning the results. There are various ways to communicate data in and out of Myriad2. It is left to the programmer to achieve such functionality. For this reason, among these lines, only line 54 will be analyzed.

4.2.1 Setting up RTEMS

RTEMS is provided in precompiled form by Movidius. In order to activate RTEMS, the following lines of Makefile code are necessary:

```
MV_SOC_OS = rtems
RTEMS_BUILD_NAME = b-prebuilt
```

The next step is to write the code for setting up RTEMS itself, which is independent of the target platform (Myriad2 in this case). The schema proposed for accomplishing this involves the creation of a new directory `leon/config`. Afterwards the code presented in source code 4.3 is required.

Source code 4.3: *leon/config/rtems_config.c*

```
1  #include <rtems.h>
2  #include "rtems_config.h"   // Describes configuration of clock and cache
3
4  // User extension to be able to catch abnormal terminations
5  // This function is attached to RTEMS inside the rtems_config.h header.
6   static void Fatal_extension(
7                               Internal_errors_Source the_source,
8                               bool is_internal,
9                               uint32_t the_error
10                              )
11 {
12     switch (the_source)
13     {
14     case RTEMS_FATAL_SOURCE_EXIT:
15         if (the_error)
16             printk("Exited with error code %d\n", the_error);
17         break; // normal exit
18     case RTEMS_FATAL_SOURCE_ASSERT:
19         printk("%s : %d in %s \n",
20                ((rtems_assert_context * )the_error)->file,
21                ((rtems_assert_context * )the_error)->line,
22                ((rtems_assert_context * )the_error)->function);
```

```
23          break;
24      case RTEMS_FATAL_SOURCE_EXCEPTION:
25          rtems_exception_frame_print((const rtems_exception_frame *) the_error);
26          break;
27      default:
28          printk("\nSource %d Internal %d Error %d  0x%X:\n",
29                 the_source,
30                 is_internal, the_error, the_error);
31          break;
32      }
33  }
```

This piece of code is mandatory and its form is suggested by the examples of the MDK. Its purpose is to define the behavior of RTEMS in case of failure. Such failure could be a driver malfunction or an unhandled hardware exception. Notice the include statement in line 2. This is an important line that defines a large part of the hardware configuration.

The version of RTEMS shipped with the MDK comes with a *board support package* (BSP) that is capable of configuring Myriad2 with simple RTEMS directives. These directives are presented in the source code 4.4.

Source code 4.4: *leon/config/rtems_config.h*

```
1  #ifndef LEON_RTEMS_CONFIG_H_
2  #define LEON_RTEMS_CONFIG_H_
3
4  #ifndef _RTEMS_CONFIG_H_
5  #define _RTEMS_CONFIG_H_
6
7  #include "app_config.h"
8
9  /*
10  * (Code removed for brevity)
11  */
12
13 // Program the booting clocks
14 // Clock configuration at startup
15 BSP_SET_CLOCK(OSC_CLOCK_KHZ,     // Reference oscillator used
16               APP_CLOCK_KHZ,     // PLL0 Target Frequency
17               1,                 // Master Divider Numerator
18               1,                 // Master Divider Denominator
19               CSS_CLOCKS,    //CSS Clocks
20               MSS_CLOCKS,    // MSS Clocks
21               UPA_CLOCKS,    // UPA Clocks
22               CLOCKS_NONE,   // SIPP Clocks
23               CLOCKS_NONE    // AUX Clocks
24 );
25
26 // Program the L2C at startup
27 BSP_SET_L2C_CONFIG( 1,                    // Enable (1) / Disable (0)
28                     L2C_REPL_LRU,         // Either L2C_REPL_LRU (default),
29                                           //        L2C_REPL_PSEUDO_RANDOM,
```

```
30                                      //         L2C_REPL_MASTER_INDEX_REP
31                                      //      or L2C_REPL_MASTER_INDEX_MOD
32                      0,              // Cache ways
33                      L2C_MODE_COPY_BACK, // Either L2C_MODE_COPY_BACK
34                                      //      or L2C_MODE_WRITE_TROUGH
35                      0,              // Number of MTRR registers to program
36                      NULL            // Array of MTRR configuration
37 );
38
39 #endif // _RTEMS_CONFIG_H_
40 #endif // LEON_RTEMS_CONFIG_H_
```

The main feature of source code 4.4 is the configuration of the different clocks of the Myriad2 SoC. Lines 15-24 configure the various clocks. In particular, the `OSC_CLOCK_KHZ` statement declares the main frequency that is commonly set to 480 or 600 MHz. The next three lines declare the frequency of the DDR RAM, which (the frequency) must be compatible with the main frequency. Currently the DDR is set up at the maximum possible frequency. The next lines enable or disable specific units inside the Myriad2 hardware. The statements in these lines are actually macros, which are defined in the file `app_config.h` included in line 7. Indeed, the source code 4.5 contains this information.

Source code 4.5: *Snippet of leon/config/app_config.h*

```
1 #ifndef _APP_CONFIG_H_
2 #define _APP_CONFIG_H_
3
4 #include <OsDrvCprDefines.h>
5 #include <OsDrvShaveL2Cache.h>
6
7 /*
8  * (Code removed for brevity)
9  */
10
11 #define APP_CLOCK_KHZ      (600000)
12 #define OSC_CLOCK_KHZ      (12000)
13
14 // CSS clocks
15 #define CSS_CLOCKS (\
16                     DEFAULT_CORE_CSS_DSS_CLOCKS        | \
17                     DEV_CSS_GETH                       | \
18                     DEV_CSS_I2C0        )
19
20 // Enable needed Shave clocks, CMXDMA, Shave L2Cache and UPA Control interfaces
21 #define UPA_CLOCKS (DEV_UPA_SH0       | \
22                     DEV_UPA_SH1       | \
23                     DEV_UPA_SH2       | \
24                     DEV_UPA_SH3       | \
25                     DEV_UPA_SH4       | \
26                     DEV_UPA_SH5       | \
27                     DEV_UPA_SH6       | \
28                     DEV_UPA_SH7       | \
```

```
29                     DEV_UPA_SH8        | \
30                     DEV_UPA_SH9        | \
31                     DEV_UPA_SH10       | \
32                     DEV_UPA_SH11       | \
33                     DEV_UPA_SHAVE_L2   | \
34                     DEV_UPA_CDMA       | \
35                     DEV_UPA_CTRL       )
36
37 // Enable clocks needed for Leon and busses access
38 #define MSS_CLOCKS     (DEV_MSS_APB_SLV      | \
39                         DEV_MSS_APB2_CTRL    | \
40                         DEV_MSS_RTBRIDGE     | \
41                         DEV_MSS_RTAHB_CTRL   | \
42                         DEV_MSS_LRT          | \
43                         DEV_MSS_TIM          | \
44                         DEV_MSS_LRT_DSU      | \
45                         DEV_MSS_LRT_L2C      | \
46                         DEV_MSS_LRT_ICB      | \
47                         DEV_MSS_AXI_BRIDGE   | \
48                         DEV_MSS_MXI_CTRL     )
49
50 #endif
```

Continuing on source code 4.4, lines 27-36 configure the L2 cache of the Leon OS processor. The comments in the source code explain these lines in detail. However some reference to MTRR is needed. *Memory Type Range Registers* (MTRRs) are a set of processor supplementary capabilities control registers that provide system software with control of how accesses to memory ranges by the CPU are cached. It uses a set of programmable model-specific registers (MSRs) which are special registers provided by most modern CPUs.

4.2.2 Switching off power islands

There are 20 power islands in the Myriad2 chip. Power islands can be turned off to save dynamic and leakage power if not in use. For a CNN implementation this is a very handy feature and can be exploited to lower the power required to process an input image. Several operations are I/O bounded, meaning that there is not gain in parallelizing their execution across all the 12 SHAVEs. In such case, turning power islands off can be only a benefit. Source code 4.6 shows that the Myriad2 turns all SHAVEs off during the initialization process.

Source code 4.6: *Snippet of leon/config/app_config.c regarding power management*

```
1 int InitClocksAndMemory(void)
2 {
3     u32 sc;
4
5     tyAuxClkDividerCfg appAuxClkCfg[] = {
6         // Give the UART an SCLK that allows it to generate an output baud
```

```
7            // rate of of 115200 Hz (the uart divides by 16)
8            {AUX_CLK_MASK_UART, CLK_SRC_REFCLK0, 96, 625},
9
10           {0, 0, 0, 0}, // Null Terminated List
11       };
12
13       // Configure the system
14       sc = OsDrvCprInit();
15       if (sc) {
16           puts("OsDrvCprInit failed");
17           return sc;
18       }
19
20       sc = OsDrvCprOpen();
21       if (sc) {
22           puts("OsDrvCprOpen failed");
23           return sc;
24       }
25
26       sc = OsDrvCprAuxClockArrayConfig(appAuxClkCfg);
27       if (sc) {
28           puts("OsDrvCprAuxClockArrayConfig failed");
29           return sc;
30       }
31
32       // Null means default configuration. This forces DDR to reset and activate.
33       DrvDdrInitialise(NULL);
34
35       // Also force the following subsytems to reset and activate
36       sc = OsDrvCprSysDeviceAction(UPA_DOMAIN, DEASSERT_RESET, UPA_CLOCKS);
37       if (sc) {
38           puts("OsDrvCprSysDeviceAction failed");
39           return sc;
40       }
41
42       sc = OsDrvCprSysDeviceAction(MSS_DOMAIN, DEASSERT_RESET, MSS_CLOCKS);
43       if (sc) {
44           puts("OsDrvCprSysDeviceAction failed");
45           return sc;
46       }
47
48       // Switch off shave power islands
49       sc = OsDrvCprTurnOffShaveMask(-1);
50       if (sc) {
51           puts("OsDrvCprTurnOffShaveMask failed");
52           return sc;
53       }
54
55       sc = OsDrvCprPowerTurnOffIsland(POWER_ISLAND_USB);
56       if (sc) {
57           puts("OsDrvCprPowerTurnOffIsland failed");
58           return sc;
```

```
59     }
60
61     return OS_MYR_DRV_SUCCESS;
62 }
```

Some explanations for source code 4.6 are:

- Line 14 initializes the Clock-Power-Reset (CPR) driver that controls the power islands of the Myriad2 chip.
- Line 33 initializes the DDR. The purpose of this line is to reset the DDR content. This helps during development time, since values from previous runs are not preserved, assisting the programmer to track bugs.
- Finally, some power islands are turned off. More precisely, all the SHAVEs and the USB unit are turned off. However, there is a caveat. In order for the power islands to work properly, it is important to enable all required power islands beforehand. For example, since all the SHAVEs are required for the CNN implementation, the source code 4.4 in conjunction with the snippet 4.5 enables all SHAVEs in advance. However, afterwards, the CPR drivers turns them off, making it possible to turn them back on when necessary. If the programmer tried to enable the SHAVEs though the CPR driver without configuring them using the RTEMS BSP routine, an error would occur.

4.2.3 Setting up SHAVEs cache

The initialization of cache that is performed in source code 4.4 is only for the Leon OS processor. The CNN implementation also requires the initialization of the cache subsystem used by the SHAVE processors. For this reason, it provides a far more advanced software driver that is capable of separating the cache into several partitions. Source code 4.7 contains the necessary commands.

Source code 4.7: *Snippet of leon/config/app_config.c regarding SHAVE cache initialization*

```
1  #define L2CACHE_CFG       (SHAVEL2C_MODE_NORMAL)
2
3  int InitShaveL2C(void)
4  {
5      s32 sc;
6
7      sc = OsDrvShaveL2CacheInit(L2CACHE_CFG);
8      if (sc) {
9          puts("OsDrvShaveL2CacheInit failed");
10         return sc;
11     }
12
13     // Reset the L2 cache partition configuration internal structure (The L2
14     // cache configuration registers are left unmodified).
```

```
15      sc = OsDrvShaveL2CResetPartitions();
16      if (sc) {
17          puts("OsDrvShaveL2CResetPartitions failed");
18          return sc;
19      }
20
21      return OS_MYR_DRV_SUCCESS;
22  }
```

To understand the necessity of the source code 4.7, some clarification is required. The driver responsible for setting up the partitions works in the following manner: It keeps an internal structure that describes the partitioning schema. This structure is reset and then built as the programmer desires. Afterwards, the partition schema is instantiated into the hardware with another driver call that will be shown shortly. The macro `L2CACHE_CFG` configures the cache behavior. The available options are:

- `SHAVEL2C_MODE_DIRECT`: In this mode the L2 cache acts as a 128KB SRAM at address 0x40000000.
- `SHAVEL2C_MODE_NORMAL`: In this mode the L2 cache acts as a cache only for the 0x80000000-0xbfffffff address space of DDR.
- `SHAVEL2C_MODE_BYPASS`: In this mode the L2 cache is bypassed completely.
- `SHAVEL2C_MODE_CACHED_ALL`: In this mode the L2 cache acts as a cache for the full DDR address space.

According to the requirements of the CNN implementation, the most suitable choice is `SHAVEL2C_MODE_NORMAL`. Details suggesting this choice are given in the following chapter, which describes how cache can be efficiently used by the SHAVEs.

Finally, the actual set up of the partitions is shown in source code 4.8.

Source code 4.8: *Snippet of leon/config/app_config.c regarding SHAVE cache partitions configuration*

```
1   int ConfigShaveL2C(void)
2   {
3       s32 sc;
4
5       // ID of the last allocated partition
6       int last_part_id = -1;
7
8       sc = OsDrvShaveL2CGetPartition(SHAVEPART128KB, &last_part_id);
9       if (sc) {
10          puts("OsDrvShaveL2CGetPartition failed");
11          return sc;
12      }
13
14      for (int i = 1; i <= 6; i++) {
15          sc = OsDrvShaveL2CGetPartition(SHAVEPART16KB, &last_part_id);
```

```
16         if (sc) {
17             puts("OsDrvShaveL2CGetPartition failed");
18             return sc;
19         }
20     }
21 
22     for (int i = 0; i < 12; i++) {
23         sc = OsDrvShaveL2CSetNonWindowedPartition(i, 0,
24                                 NON_WINDOWED_INSTRUCTIONS_PARTITION);
25         if (sc) {
26             puts("OsDrvShaveL2CSetNonWindowedPartition failed");
27             return sc;
28         }
29     }
30 
31     for (int i = 0, partId = 1; i < 6; i += 2, partId++) {
32         sc = OsDrvShaveL2CSetNonWindowedPartition(i, partId,
33                                         NON_WINDOWED_DATA_PARTITION);
34         if (sc) {
35             puts("OsDrvShaveL2CSetNonWindowedPartition failed");
36             return sc;
37         }
38 
39         sc = OsDrvShaveL2CSetNonWindowedPartition(i+1, partId,
40                                         NON_WINDOWED_DATA_PARTITION);
41         if (sc) {
42             puts("OsDrvShaveL2CSetNonWindowedPartition failed");
43             return sc;
44         }
45     }
46 
47     sc = OsDrvShaveL2CacheAllocateSetPartitions();
48     if (sc) {
49         puts("OsDrvShaveL2CacheAllocateSetPartitions failed");
50         return sc;
51     }
52 
53     for (int i = 0; i <= last_part_id; i++) {
54         sc = OsDrvShaveL2CachePartitionInvalidate(i);
55         if (sc) {
56             puts("OsDrvShaveL2CachePartitionInvalidate failed");
57             return sc;
58         }
59     }
60 
61     return OS_MYR_DRV_SUCCESS;
62 }
```

The source code 4.8 configures the L2 cache of the SHAVEs using 7 out of 8 partitions. The first partition is set to be 128KB, while the next 7 partitions are set to be 16KB each. This makes a total of 224KB of cache, while the total cache size is 256KB.

More precisely:

- Lines 22-29 assign the instruction port of each SHAVE to point to the first partition. This means that the first partition is going to be used as instruction cache and will be shared among all SHAVEs.
- Lines 31-45 assign the Load-Store Unit (LSU) (or data) port of each SHAVE to point to one of the six remaining partitions. In particular, every pair of consecutive SHAVEs will have their LSU port pointing at the same partition. SHAVE 0 and 1 will use the second partition, …, SHAVE 10 and 11 with use the seventh partition.
- Lines 47-51 will instantiate the cache configuration defined in the previous lines. This means the internal data structure of the cache driver is stored to hardware registers.
- Finally, lines 53-59 invalidate the cache, in order to make sure there are no stale cache entries in the memory hierarchy of the SHAVEs.

4.3 SHAVE code residing in CMX

Ideally, CMX is the best memory in term of performance and power consumption that is available to the SHAVE processors. However, due to its limited size, only the most essential parts of data and code can be placed there. In general, CMX is suitable for:

- Placing the entire SHAVE code, if it is able to fit. If not, a less space consuming choice is to place routines that need to be extremely fast. For example, a memory allocator is such a choice. Also, widely used utility routines and some in-line routines could also be placed there.
- Placing code that is used to bootstrap each computational node. It is a mandatory requirement to place some code inside CMX in order to be able to begin the execution. This code can then refer to other code that is placed in either CMX or DDR.
- Finally, parameters to CMX routines are also important. During the preprocessing stage of the CNN, all parameters to the computational nodes are calculated. These parameters are then provided to the computational routines that run in SHAVEs. Due to the complexity of the routines, the size of the parameters is quite large (e.g. 128 bytes), so not carefully handling them can result in a not-so-small time penalty during the execution stage. However, by placing these data inside CMX the time penalty vanishes.

The following subsections will provide code for the contents of the CMX.

4.3.1 Memory allocator code

A memory allocator is an important part of the CNN implementation. Allocation operations, such as malloc, are not available for SHAVE processors. Also, it is common that these operations are optimized for allocating large blocks of memory, making them of limited use for SHAVEs, because of their performance. The requirements of a memory allocator for the SHAVE processors used in the CNN implementation are quite easy to meet. First, the allocator needs to be extremely fast. Second, it not necessary for the allocator to be able to free. As a result the implementation of 4.10 is compact and concise.

Source code 4.9: *shave/cmx/memory.h*

```
1 #ifndef __MEMORY_H__
2 #define __MEMORY_H__
3 #include <ddr_functions.h>
4
5 // Size of the memory pool (in bytes) available for each shave
6 #define MEMORY_POOL 95*1024
7
8 void setAlignedMem(int shaveId, J_FUNCPTR_T jumpTable);
9 void *getAlignedMem(int alignment, int bytes);
10
11 #endif//__MEMORY_H__
```

Source code 4.10: *shave/cmx/memory.c*

```
1 #include <swcWhoAmI.h>
2 #include <mv_types.h>
3 #include <stddef.h>
4
5 #include "memory.h"
6 #include <ddr_functions.h>
7
8 // Statically allocate the maximum available space available for local buffers
9 static u8 __attribute__(( aligned (128))) mem[MEMORY_POOL];
10 static u32 nextAddress;
11 static PRINTF_PTR printf;
12 static int primaryShave;
13
14 void setAlignedMem(int shaveId, J_FUNCPTR_T jumpTable)
15 {
16     nextAddress = (u32)mem;
17     printf = (PRINTF_PTR) jumpTable(CM_printf);
18     primaryShave = shaveId;
19 }
20
21 void *getAlignedMem(int alignment, int bytes)
22 {
23     u32 residue = nextAddress % (u32)alignment;
24     u32 ret = nextAddress + (residue ? ((u32)alignment - residue) : 0);
```

```
25     nextAddress = ret + bytes;
26 
27     if ((nextAddress - (u32)mem) > sizeof(mem))
28         if (primaryShave == (swcWhoAmI() - PROCESS_SHAVE0))
29             printf("Error: Trying to allocate memory space that exceeds local"
30                    " memory pool capacity!\n");
31 
32     return (void *) ret;
33 }
```

The way the allocator works is as follows:

- Each shave statically allocates a fixed amount of memory that is used as a memory pool. This pool is used as the memory that is available to each shave for performing allocations. Each shave can perform multiple allocations, however the total size allocated cannot exceed the size of the pool. The pool size is defined in line 11 of the source code 4.9. Notice that the number placed there is compatible with the size defined in the linker script, shown in source code 4.1.
- The `setAlignedMem` function is responsible for initializing the allocator. It needs to be called from each SHAVE separately and before any call to `getAlignedMem` is performed. This function requires also the argument `jumpTable`, a function that makes it possible for SHAVEs to refer to code that resides in DDR. For example, `printf` is a function that would make sense to be placed in DDR.
- The `getAlignedMem` makes the allocation and returns a pointer to the allocated space. This function supports aligned allocation, a very important feature that can boost performance of execution. Allocations may fail if the memory pool has been exhausted. In such case, the `printf` function prints the appropriate message.
- Finally, notice that `setAlignedMem` also requires a `shaveId` argument. This is because the `printf` function is not thread safe, so only one SHAVE at a time can use it. By providing a `shaveId`, only the specified SHAVE is responsible for using `printf` in case of memory pool exhaustion. A more sophisticated mechanism using lock could be used, but this approach meets the needs.

4.3.2 Bootstrap code

The bootstrap code acts as an entry point for code execution from the SHAVEs. Its size needs to be small in order to fit inside the reserved CMX space used for code.

Source code 4.11: *shave/cmx/entry.c*

```
1 #include <svuCommonShave.h>
2 #include <swcCdma.h>
3 #include <swcWhoAmI.h>
4 
5 #include <stddef.h>
```

```
6 #include <mv_types.h>
7
8 #include <moviVectorUtils.h>
9
10 #include "../ddr/ddr_conv.h"
11 #include "../ddr/ddr_pool.h"
12 #include "../ddr/ddr_fc.h"
13
14 #include <ddr_functions.h>
15 #include "memory.h"
16
17 #define DMA_TASKS_MAX 3
18 #define DMA_REFS_MAX 3
19
20 dmaTransactionList_t __attribute__((section(".cmx.cdmaDescriptors")))
21 task[DMA_TASKS_MAX], *ref[DMA_REFS_MAX];
22
23 //#define DRIVER_ROUTINES_INSIDE_CMX
24
25 #ifdef DRIVER_ROUTINES_INSIDE_CMX
26 #include "../ddr/utils.c"
27 #include "../ddr/ddr_conv.c"
28 #include "../ddr/ddr_pool.c"
29 #include "../ddr/ddr_fc.c"
30 #endif
31
32 void
33 shave_conv(conv_info *info,
34            u32 firstMapNo,
35            u32 lastMapNo,
36            J_FUNCPTR_T jumpTable
37 )
38 {
39     int shaveId = swcWhoAmI() - PROCESS_SHAVE0;
40
41     conv_context context = {
42         .dma = (dma_context){
43             .dmaInitRequester = dmaInitRequester,
44             .dmaCreateTransactionFullOptions = dmaCreateTransactionFullOptions,
45             .dmaStartListTask = dmaStartListTask,
46             .dmaWaitTask = dmaWaitTask,
47             .task = task,
48             .ref = ref
49         },
50
51         .com = (common_context){
52             .shaveId = shaveId,
53             .jumpTable = jumpTable
54         },
55
56         .mem = (memory_context){
57             .setAlignedMem = setAlignedMem,
```

```
58                .getAlignedMem = getAlignedMem
59            },
60
61            .info = info,
62        };
63
64  #ifndef DRIVER_ROUTINES_INSIDE_CMX
65      CONV_DDR_PTR conv_ddr = (CONV_DDR_PTR) jumpTable(CM_conv_ddr);
66  #endif
67
68      conv_ddr(firstMapNo, lastMapNo, &context);
69
70      SHAVE_HALT;
71  }
72
73  void
74  shave_pool(
75      pool_info *info,
76      u32 firstMapNo,
77      u32 lastMapNo,
78      J_FUNCPTR_T jumpTable
79  )
80  {
81      /*
82       * (Code omitted for brevity)
83       */
84
85  #ifndef DRIVER_ROUTINES_INSIDE_CMX
86      POOL_DDR_PTR pool_max_ddr = (POOL_DDR_PTR) jumpTable(CM_pool_max_ddr);
87      POOL_DDR_PTR pool_ave_ddr = (POOL_DDR_PTR) jumpTable(CM_pool_ave_ddr);
88  #endif
89
90      switch (info->type) {
91          case pooling_AVE:
92              pool_ave_ddr(firstMapNo, lastMapNo, &context);
93              break;
94          case pooling_MAX:
95              pool_max_ddr(firstMapNo, lastMapNo, &context);
96              break;
97      }
98
99      SHAVE_HALT;
100 }
101
102 void
103 shave_fc(
104     fc_info *info,
105     int firstLineNo,
106     int lastLineNo,
107     J_FUNCPTR_T jumpTable
108 )
109 {
```

```
110     /*
111      * (Code omitted for brevity)
112      */
113
114 #ifndef DRIVER_ROUTINES_INSIDE_CMX
115     FC_DDR_PTR fc_ddr = (FC_DDR_PTR) jumpTable(CM_fc_ddr);
116 #endif
117
118     fc_ddr(firstLineNo, lastLineNo, &context);
119
120     SHAVE_HALT;
121 }
```

The way the entry points for SHAVEs are set up is as follows:

- It contains the functions `shave_conv`, `shave_pool`, `shave_fc`, each one acting as an entry point to the computations regarding convolutional, pooling and fully connected nodes, respectively. The aforementioned computations receive an argument named `context` that contains all the required parameters for performing the computation. As seen from lines 41-61, convolutions need to make use of the DMA engine, the memory allocator and the `jumpTable`. Also, the `info` argument is passed, which is responsible for parameters concerning the computation itself. For example, contents of this argument are the address of the kernel masks needed for performing the convolution, the address of the input data etc. The three functions follow the same pattern. For this reason, only the function `shave_conv` is shown completely.

- In line 23, there is the preprocessor macro `DRIVER_ROUTINES_INSIDE_CMX`. This macro alters the place where the computational routines are placed. If not defined, the computational routines are placed in DDR. This is the default behavior and the only solution for the general case, where many different computational routines are needed. Otherwise, the computational routines are placed inside CMX. The existence of this feature is there to assess the performance penalty of accessing SHAVE code from DDR. In fact, if the macro is not commented the performance gain is insignificant, justifying that placing the SHAVE code inside DDR is not a disastrous choice.

- Lines 17-21 define the structures required by the DMA engine. These structures are placed in a special place of CMX memory that is dictated be the hardware itself. This is accomplished by using the compiler attribute syntax which positions the structures in the section `.cmx.cdmaDescriptors` that is placed at the region `CMX_DMA_DESCRIPTORS` (referring to source code 4.1) by the MDK build system. It is not only an obligation to place DMA related code inside CMX, but also a safety concern. Because this piece of code is actually a driver and is already provided by the MDK, placing it inside CMX is guaranteed for it to operate correctly. On the other hand, using DDR for this code can lead to bugs difficult to track down.

- Notice that each entry point requires two more arguments. For example, convolution requires the arguments `firstMapNo` and `lastMapNo`. These arguments are used for parallelization. More precisely, each SHAVE receives a different parameter of these arguments, indicating that each one produces a different part of the output. The precise parallelization strategy has bee explained in a previous chapter.

It is suggested to use the GNU Linker garbage collection option for code and data residing in CMX [8]. Keep in mind that the MDK uses two compilers for generating code. The first compiler is used for code that is executed by Leon OS and Leon RT, while the second compiler is used for code that is executed by the SHAVE processors. These compilers are completely separated and do not communicate with each other. The linker garbage collection mechanism tries to find code and data that with no reference to. However the search for references is confined within each compiler generated code separately. For example, if a function is defined but never used, then the executable file will never contain this function. Also, if a function is defined in SHAVEs and only used by Leons, again the executable file will never contain this function. As a result, the SHAVE computational node entry points, that are referenced only by the Leons, are garbage collected. To circumvent this mechanism the Makefile needs to contains some extra instructions.

The specific lines of the Makefile that handle this functionality and perform the garbage collection are presented in source code snippet 4.12.

Source code 4.12: *Snippet of Makefile for entry points garbage collection*

```
1 ENTRYPOINTS1 = -e shave_conv -u shave_pool -u shave_fc --gc-sections
2 $(CNNApp).mvlib : $(SHAVE_CNN_OBJS) $(PROJECT_SHAVE_LIBS)
3     $(ECHO) $(LD) $(MVLIBOPT) $(ENTRYPOINTS1)          \
4                               $(SHAVE_CNN_OBJS)        \
5                               $(PROJECT_SHAVE_LIBS)    \
6                               $(CompilerANSILibs)      -o $@
```

The `-e` option defines the entry point. If there are multiple entry points, as in the case of the current implementation, then only one of them, whose choice is arbitrary - is used with the `-e` option. The `-u` option forces the symbol following it to be entered in the output file as an undefined symbol. This prevents the garbage collector from erasing it. If fact, by replacing the `-e` with `-u` would make no difference. However it is left this way for readability and better understanding by programmers not familiar with the GNU Linker.

Finally it is worth commenting on the `info` argument. Each computational node requires its own type of the `info` argument. However, these arguments contain essentially the same information, so only the case of convolution will be presented.

The information required is expressed with the struct of source code 4.13.

Source code 4.13: *info argument used by convolution*

```
1 #ifndef __CONV_API_H__
2 #define __CONV_API_H__
```

```
3
4 #include <mv_types.h>
5 #include <network_config.h>
6
7 typedef struct {
8     // Input data place
9     u8 *input;
10    int input_channel_offset;
11    u8 inputBPP;
12
13    // Output data place
14    u8 *output;
15    int output_channel_offset;
16    u8 outputBPP;
17
18    u8 *conv_weights;
19    int conv_weights_offset;          // Offset between different masks
20    int conv_weights_channel_offset;  // Offset between layer of the same
        mask
21
22    // Input DMA
23    int in_src_addr, in_src_width, in_src_stride;
24    int in_dst_addr, in_dst_width, in_dst_stride;
25    int in_buffer_elements, in_elements;
26
27    // Output DMA
28    int out_src_addr, out_src_width, out_src_stride;
29    int out_dst_addr, out_dst_width, out_dst_stride;
30    int out_buffer_elements, out_elements;
31
32    /*
33     * (Other fields are removed for brevity)
34     */
35
36 } conv_info;
37
38 #endif//__CONV_API_H__
```

Source code 4.13 is pretty much self explanatory. It is just reiterated that convolution requires input and output, as well as weight parameters. Also, instructions for the DMA engine are provided to specify the way data will be moved from DDR to CMX and vice versa. During convolution the DMA transfers 2D images. Someone could argue that the only information necessary for DMA is the size of the data and the width of the image. Sadly, this is not the case, because of several optimization steps that are involved along the way and will be described in the next chapter.

The data that fill the `info` struct are computed during the preprocessing stage of the CNN, that is performed by the Leon OS. Again, for convolution this is presented in source code 4.14.

Source code 4.14: *How and where the info parameter is prepared*

```
1 /*
2  * .....
3  */
4
5 static conv_info __attribute__((section(".cmx_direct.data"), aligned (16)))
6 info_list[NETWORK_MAX_CONV_NODES];
7
8 static int info_list_occupied = 0;
9
10 /*
11  * .....
12  */
13
14 static void
15 prepare_arguments(struct network_node_conv *n) {
16
17     if (info_list_occupied == NETWORK_MAX_CONV_NODES)
18         ERROR("Network does not support any more convolution nodes");
19     conv_info *info = &info_list[info_list_occupied++];
20
21     if (info == NULL)
22         ERROR("Could not allocate!");
23
24     info->input = (u8*)(n->input_data.data);
25     info->input = (u8*)(n->input_data.data);
26     info->input_channel_offset = n->input_data.width * n->input_data.height;
27     info->inputBPP = n->input_data.element_size;
28
29     info->output = (u8*)(n->output_data.data);
30     info->output_channel_offset = n->output_data.width * n->output_data.height;
31     info->outputBPP = n->output_data.element_size;
32
33     /*
34      * ...
35      */
36
37     info->conv_weights = (void *)((u32)(n->weights.masks.data) &
           (u32)0x8fffffff);
38     info->conv_weights_channel_offset = n->kernel_size * n->kernel_size;
39     info->conv_weights_offset = info->conv_weights_channel_offset *
           info->channels;
40
41     info->conv_biases = (void *)((u32)(n->weights.biases.data) &
           (u32)0x8ffffff);
42     info->kernelBPP = n->weights.masks.element_size;
43
44     conv_prepare_dma(info,
45                      n->input_data.height, n->input_data.width,
46                      info->kernel_h, info->kernel_w,
47                      n->padding_v, n->padding_h,
48                      n->stride, n->stride,
49                      n->alignment
```

```
50     );
51 
52     n->tensor = info;
53 }
54 
55 /*
56  * .....
57  */
```

The following lines of snippet 4.14 are worth mentioning:

- Lines 5-8 allocate the space in CMX memory where the parameters will be placed. The `.cmx_direct.data` section instructs the linker to place this section in the `CMX_OTHER` region (referring to source code 4.1) and access it in an uncached manner. This ensures that the Leon OS will indeed update the contents of the CMX immediately. Although this piece of code is executed by the Leon OS, even this processor does not support dynamic memory allocation for CMX. A simple - yet effective - way is to statically allocate the space in advance. Making this allocation statically is not a problem, since this value can be easily altered if a certain CNN requires more convolutional nodes.

- The function `prepare_arguments` receives as argument a high level description of the type of convolution and the data to apply it to. Afterwards it builds the specific parameters needed by the computational routines of the SHAVEs. Notice lines 37 and 39. These lines convert the addresses for the weights needed by the computation, such that the access to them will be though the cache subsystem of the SHAVE processors.

4.4 SHAVE code and data residing in DDR

This section will describe in detail the pieces of code and data - used by SHAVEs - that have been placed in DDR, as well as the way this is accomplished. In fact, the procedure that compiles and places the SHAVE code in DDR is not directly supported by the MDK, as a result it is a bit cumbersome and needs attention.

4.4.1 Trained parameters

These parameters are generated by Caffe during the training phase of the network. They are usually called weights and are essential for performing the computations involving convolutional and fully connected nodes of the CNN. It is important to note that the parameters grow larger in size for deeper CNNs and can quickly fill the entire DDR. As explained earlier, parameters for convolution are read through the cache subsystem by the SHAVEs. On the other hand, parameters for fully connected nodes are read through DMA, because they need to be available to the computational routines in large bulks.

The parameters generated by Caffe are stored in a special format called *binary proto* that was introduced by Google in their protobuf library. Caffe performs the training using 32-bit floating point numbers (IEEE 754), while the SHAVEs are configured to use 16-bit floating point numbers (fp16 for short). The SHAVEs operate at a lower precision, even though they support 32-bit floating point numbers (fp32 for short), because they can perform more computations in the same amount of cycles with this configuration. This compromise is not a problem by itself, because CNNs are - in general - numerically robust. However, the loss in precision will be put under test in the chapter analyzing the experimental results on specific networks. Since the entire CNN operates at fp16, a way to convert from fp32 to fp16 is needed.

The MDK provides the functionality of f32 to/from f16 conversion, because Leon OS and Leon RT do not support fp16 numbers. The conversion code is located at the directory `common/components/Fp16Convert` and is used to create the python module shown in source code 4.15.

Source code 4.15: *FpConvert python module setup.py*

```
1 #!/usr/bin/python2.7
2 # -*- coding: UTF-8 -*-
3
4 # To install to local user directory type:
5 # python2.7 setup.py install --user
6
7 from distutils.core import setup, Extension
8 import numpy as np
9
10 ext_modules = [ Extension('FpConvert', sources = ['Fp16Convert.c',
      'module.c']) ]
11
12 setup(
13         name = 'FpConvert',
14         version = '1.0',
15         include_dirs = [np.get_include()], #Add Include path of numpy
16         ext_modules = ext_modules
17       )
```

From source code 4.15 it is visible that the module uses the `Fp16Convert.c` file which is provided by the MDK. Also, it uses the `module.c` file that is shown in source code 4.16.

Source code 4.16: *FpConvert python module module.c*

```
1 #include <Python.h>
2 #include <numpy/arrayobject.h>
3
4 #include <stdio.h>
5 #include <stdlib.h>
6 #include "Fp16Convert.h"
7
8 static PyObject*
9 fp32tofp16 (PyObject *dummy, PyObject *args)
```

```
10 {
11     PyObject *arg1 = NULL;
12     PyArrayObject *arr1 = NULL;
13 
14     if ( !PyArg_ParseTuple(args, "O", &arg1) ) return NULL;
15 
16     arr1 = (PyArrayObject*)PyArray_FROM_OTF(arg1, NPY_FLOAT32, NPY_IN_ARRAY);
17     if (arr1 == NULL) goto fail2;
18 
19     int nd = PyArray_NDIM(arr1);   //number of dimensions
20 
21     if (nd > 1) {
22         PyErr_SetString(PyExc_RuntimeError,
23                         "Input array must be 1-dimensional (i.e. flattened).");
24 
25         goto fail2;
26     }
27 
28     float acc = 0.0f;
29     float *elem = NULL;
30 
31     PyObject* list = PyList_New(arr1->dimensions[0]);
32     if (list == NULL) goto fail1;
33 
34     for (int i = 0; i < arr1->dimensions[0]; ++i) {
35         elem = (float*)PyArray_GETPTR1(arr1, i);
36 
37         PyList_SetItem(list, i, Py_BuildValue("I", f32Tof16(*elem)));
38     }
39 
40     Py_DECREF(arr1);
41 
42     return list;
43 
44 fail1:
45     Py_XDECREF(list);
46 
47 fail2:
48     Py_XDECREF(arr1);
49 
50     return NULL;
51 }
52 
53 static struct PyMethodDef methods[] = {
54     {"f32Tof16", fp32tofp16, METH_VARARGS, "Convert float32 to float16 hex
         equivalent"},
55     {NULL, NULL, 0, NULL}
56 };
57 
58 PyMODINIT_FUNC
59 initFpConvert (void)
60 {
```

```
61     (void)Py_InitModule("FpConvert", methods);
62     import_array();
63 }
```

Compiling the python module is very simple and requires the following terminal command:

```
$ python2.7 setup.py install --user
```

Afterwards, it is possible to import the `FpConvert` module in python and use its `f32Tof16` function. The script of source code 4.17 can then run without a problem.

Source code 4.17: *Python script that converts Caffe trained parameters*

```
 1 #!/usr/bin/python2.7
 2 # -*- coding: UTF-8 -*-
 3
 4 import numpy as np
 5 import caffe
 6 import FpConvert
 7
 8 SUPPORTED_LAYERS = ['Input', 'Convolution', 'Pooling', 'InnerProduct', 'ReLU']
 9
10 caffe.set_mode_cpu()
11
12 net = caffe.Net('path/to/.prototxt', 'path/to/.caffemodel', caffe.TEST)
13
14 GENERATE_BLOBS = True
15
16 DATA_C = []
17
18 def array2fp16(array):
19     return FpConvert.f32Tof16(list(array.flatten()))
20
21 def tup2mult(t):
22     return '*'.join(map(str, t))
23
24 def list2str(t):
25     return ', '.join(map(str, t))
26
27 def node_with_weights_name(net):
28     for k, _ in net.params.items():
29         yield k
30
31 def generate_buffer(node_data, buf_prefix):
32     shape = node_data.data.shape
33
34     blob = ''
35     if GENERATE_BLOBS:
36         blob = list2str(array2fp16(node_data.data))
37
```

```
38     DATA_C.append(
39         "fp16 DATA_BUFFER " + buf_prefix +
               "_raw[{}]{};".format(tup2mult(shape), ' = { \n' + blob + '\n}'))
40
41 n = node_with_weights_name(net)
42
43 # Process input data
44 try:
45     node_data = net.blobs['data']
46     shape = node_data.data.shape
47     generate_buffer(node_data, 'input_data')
48 except IndexError:
49     raise IndexError("Input data node must be named 'data'")
50
51 # Process the rest layers with blobs
52 for l in net.layers:
53     x = l.blobs
54
55     if l.type not in SUPPORTED_LAYERS:
56         raise NotImplementedError('Only ' +
57                                   str(SUPPORTED_LAYERS) + ' are supported')
58
59     if len(x) > 0:
60         node_type = l.type
61         node_name = next(n)
62         weight = x[0]
63         bias = x[1]
64
65         if node_type == 'Convolution':
66             generate_buffer(weight, node_name + '_masks')
67             generate_buffer(bias, node_name + '_biases')
68
69         elif node_type == 'InnerProduct':
70             generate_buffer(weight, node_name + '_weights')
71             generate_buffer(bias, node_name + '_biases')
```

Note that the script of source code 4.17 is a simplified version of the actual script, in order to explain the key elements, without sacrificing much space. Lines that need attentions are:

- Line 8: Contains the Caffe layers that the script is able to parse. Caffe is a large framework with many different layers of computation for CNNs. The goal of this script is to be able to support the subset of operations that is required by the current CNN implementation in Myriad2.
- Line 12: With this line Caffe parses the `.prototxt` file, that contains the description of the network and loads the trained parameters included in the `.caffemodel` file.
- Lines 31-39: These lines make the conversion from fp32 to fp16. They store the result in the `DATA_C` list in the form of text, which is then printed to generate C code

that is embedded inside the CNN source code. This will become clearer with the next piece of source code.

Then complete version of source code 4.17 exports the trained parameters in C code. A snippet of such exporting is shown in source code 4.18.

Source code 4.18: *Converted Caffe trained parameters*

```
1  #include "node_defines.h"
2
3  #define DATA_BUFFER __attribute__((section(".ddr_direct.data"), aligned (16)))
4
5  fp16 DATA_BUFFER input_data_raw[1*1*50*50] = {
6  15636, 15521, 15681, ...
7  };
8
9  fp16 DATA_BUFFER conv_masks_raw[32*1*5*5] = {
10 13040, 13579, 13540, ...
11 };
12
13 fp16 DATA_BUFFER conv_biases_raw[32] = {
14 47182, 45891, 47157, ...
15 };
16
17 fp16 DATA_BUFFER fc_weights_raw[200*1152] = {
18 11484, 40758, 9013, ...
19 };
20
21 fp16 DATA_BUFFER fc_biases_raw[200] = {
22 43883, 39963, 43263, ...
23 };
```

The following parts of source code 4.18 are worth mentioning:

- Line 3: The section `.ddr_direct.data` is placed at the region `DDR_DATA` (referring to source code 4.1) by the MDK build system. Also, the access to the data is uncached, making sure no caches are used without explicitly stated by the programmer.
- It is reiterated that the source code presented is compiled by the Leon compiler. As a result, the `fp16` data type is actually an alias for `uint16_t`. That is why the data defined by the arrays are integer numbers. These integers are in fact the binary representation of 16-bit floating point numbers, which are generated by the script of source code 4.17.
- The ellipses shown are just for brevity, because the data of each array are quite large. Array names prefixed with `conv` refer to required parameters for convolutional nodes. On the other hand, array names prefixed with `fc` refer to required parameters for fully connected nodes. These prefixes are just a convention, there is no special semantic meaning to them.

4.4.2 The Jump Table

Although MDK comes with mechanisms for feeding SHAVEs with instructions residing in DDR, these solutions ended up not being useful for the needs of a CNN implementation. These mechanisms are designed for dynamically replacing the application running on the SHAVEs, however, the time to perform the switch is substantial. This leads to the development of a simpler approach based on the Dynamic Shave Loading source code provided by the MDK. The general idea behind the developed schema is the following: The `jumpTable` is a function that acts as the entry point for SHAVE code in DDR. With this function the programmer is able to "jump" between different functions of the DDR code, that are finally executed on the SHAVEs. In other words, the `jumpTable` exports the position of all the required SHAVE functions that are placed in DDR. The exact implementation of the `jumpTable` will be presented shortly. For now, the process of placing SHAVE compiled code in DDR will be presented.

The main difficulty is to instruct the MDK to compile a set of files with the SHAVE compiler - named moviCompiler - and then place the resulting object file in DDR. Thus, the Makefile is extended, as show in source code 4.19.

Source code 4.19: *Makefile extension code that places SHAVE instructions in DDR*

```
1  ENTRYPOINTS2 =
2  $(ddrApp).mvlib : $(SHAVE_ddrApp_OBJS) $(PROJECT_SHAVE_LIBS)
3      $(ECHO) $(LD) $(MVLIBOPT) $(ENTRYPOINTS2)          \
4                              $(SHAVE_ddrApp_OBJS)      \
5                              $(PROJECT_SHAVE_LIBS)     \
6                              $(CompilerANSILibs)      -o $@
7
8  $(ddrApp)_shvdlib.text : $(ddrApp).shvdlib
9      $(ECHO) $(OBJCOPY) -O binary --only-section=.dyn.text $< $@
10
11 $(ddrApp)_shvdlib.data : $(ddrApp).shvdlib
12     $(ECHO) $(OBJCOPY) -O binary --only-section=.dyn.data $< $@
13
14 SHVDLIB_DEPS := $(ddrApp)_shvdlib.text $(ddrApp)_shvdlib.data
15 $(ddrApp)_shvdlib.combined : $(SHVDLIB_DEPS)
16     $(ECHO) (diff=$(shell cat $(ddrApp)_shvdlib.text | wc -c); \
17     dd if=/dev/zero bs=1 count=$$((0x00100000 - $$diff))       \
18     >> $(ddrApp)_shvdlib.text)
19     $(ECHO) cat $(SHVDLIB_DEPS) > $@
20
21 $(ddrApp)_bin.o : $(ddrApp)_shvdlib.combined
22     $(ECHO) $(OBJCOPY) -I binary --rename-section          \
23      .data=.ddr.data --redefine-sym                         \
24     _binary_$(subst /,_,$(subst .,_,$<))_start=jumpTable   \
25     -O elf32-sparc -B sparc $< $@
```

The lines of source code 4.19 are explained as follows:

- Lines 1-6: These lines generate the relocatable SHAVE object file using moviCom-

piler. Normally the `.mvlib` file that is produced would be placed in CMX.

- Lines 8-9: The prerequisite of this rule is a `.shvdlib` file that is generated from rules provided by the MDK. This file contains two sections. The first section is `.dyn.text` and contains compiled code. The second section is `.dyn.data` and should contain constant data, such as a lookup table or `const` C variables. The action of the rule uses the GNU Linker script `scripts/ld/lib.ldscript` (source code 4.20) to resolve relative addresses and place the code of section `.dyn.text` in absolute address space. The result is stored in the target file of the rule.

- Lines 10-11: These lines perform the same operation as the two previous lines do, although this time for the section `.dyn.data`.

- Lines 14-19: Absolute addresses have now been resolved and two files have been produced, namely files with suffixes `_shvdlib.text` and `_shvdlib.data`. This rule pads the end of the `_shvdlib.text` file with zeros to create a new file of size 0x00100000 bytes, i.e 1MB, and afterwards combines the resulting file with the `_shvdlib.data` file. The padding operation will make the code generation to fail if the `_shvdlib.text` grows larger than 1MB, thus attention should be paid to ensure this value changes if the code surpasses the size of 1MB.

- Lines 21-25: The library is now compiled with moviCompiler and is ready to be placed in DDR [15]. The MDK provides a method to insert raw binary files into the executable .elf file (that is sent to Myriad2) and this method is exploited in these lines to insert machine code instead. The code is placed in the `.ddr.data` section and the symbol `jumpTable` is introduced to point at the beginning of the code. This symbol is actually introduced by the `objcopy` tool called at line 22 and will be used to call library functions from other parts of the CNN application.

- In conclusion, the above description tries to clarify the procedure where the piece of code generated by the object files `$(SHAVE_ddrApp_OBJS)` is converted into a `.mvlib` library. From this library is then converted to `.shvdlib` library that contains two sections, one for text and one for data. The two sections are slightly manipulated and their address are converted from relative to absolute ones. The resulting file, with suffix `_shvdlib.combined` is then `objcopy`ed so that it is included in the executable `.elf` that is sent to Myriad2. The `objcopy` tool essentially copies the `_shvdlib.combined` file and pastes it in the `.elf` file, at the begging of section `.ddr.data`. At the same time, it introduces a new symbol into the `.elf` file that is called `jumpTable` (see line 24). As a result, the symbol `jumpTable` is available inside the C source code of Leon OS. However, in order for this symbol to be useful, it is important to point to a function of our choice. This function is named the same way and is presented in source code 4.21.

The tokens 0x00100000 and `jumpTable` of the lines 17 and 24 need special attention, since they are tightly coupled with some other parts of the application. Source code 4.1 specifies that the `.ddr.data` section begins at address 0x80000000 + 64MB, i.e. at

address 0x84000000 and has length of 64MB. With these in mind, the source code 4.20 contains the magic numbers 0x84000000 and 0x84100000.

Source code 4.20: *scripts/ld/lib.ldscript*

```
/*Linker script file which sets all dynamic code*/
SECTIONS {
  /*Setting up shave slices memory with absolute addresses*/
  . = 0x84000000;
  .dyn.text ALIGN(4) : {
        *(S.text*)
        *(S.__TEXT__sect)
        *(S.__MAIN__sect)
        *(S.init*)
  }
  . = 0x84100000;
  .dyn.data : {
        *(S.data*)
        *(S.rodata*)
        *(S.__DATA__sect*)
        *(S.__STACK__sect*)
        *(S.__static_data*)
        *(S.__HEAP__sect*)
        *(S.__T__*)
  }
}
```

Reiterating, source code 4.20 instructs the GNU linker to resolve the relative addresses to absolute ones. The `.dyn.text` section is placed in address 0x84000000, which is the same address as that defined in 4.1. It is important to ensure these two addresses match. Also, the `.dyn.data` section is placed in address 0x84100000 that is 0x00100000 bytes apart from the `.dyn.text` section. This value matches the value in line 17 of source code 4.19.

The Makefile needs to be extended further more, in order to make use of the `lib.script` of source code 4.20. This is accomplished by placing the following lines in the Makefile:

```
LDDYNOPT =-L . -L ./scripts --nmagic -s
LDDYNOPT+=-T ./scripts/ld/lib.ldscript

LDSYMOPT =-L . -L ./scripts --nmagic
LDSYMOPT+=-T ./scripts/ld/lib.ldscript
```

Finally the `jumpTable` token of 4.19 should match the name of the function in 4.21.

Source code 4.21: *shave/ddr_functions.c*

```
#include "ddr_functions_exports.h"

FUNCPTR_T jumpTable(int i)
{
```

```
5       struct lib_function func = lib[i];
6
7       switch (func.category) {
8       case func_common:
9           return func.cat.cm.func;
10      case func_conv:
11          return func.cat.conv.func;
12      case func_pool:
13          return func.cat.pool.func;
14      case func_acc:
15          return func.cat.acc.func;
16      case func_fc:
17          return NULL;
18      }
19
20      return 0;
21  }
```

More specifically it is important to ensure the `jumpTable` symbol of the intermediate file is placed at address 0x84000000. To satisfy such restriction the source file ddr_functions.c contains only this function and nothing else. Also, it is the first file that is compiled among the other files that generate the DDR SHAVE instructions. To check wether the symbol `jumpTable` is placed at address 0x84000000, one way is to verify if the command `sparc-myriad-elf-readelf -a $(ddrApp)_sym.o | grep -i jumpTable` produces non-blank output, where $(ddrApp) is defined in the Makefile of the application.

The purpose of the `jumpTable` is to provide an entry point to the DDR SHAVE code. This entry point is identified as an extern symbol by the Leon OS compiler and is used across multiple files of the application. The reason this process is not automated is because it would require changes to provided MDK code. This would make the solution to break with every new MDK release. The current approach does not break with new MDK releases, although requires attention during development time.

4.4.3 Operation of the `jumpTable`

The previous subsection described the procedure necessary to put SHAVE compiled code in DDR. The result of this procedure is that the `jumpTable` symbol is accessible by Leon OS and also points to a function with the same name. How this function is actually used will be described in this section.

The `jumpTable` function, shown in source code 4.21 takes an integer argument and indexes the array `lib` with it. The array contains pointers to a `struct` and the argument selects one of those pointers. The `struct` is presented in source code 4.22.

Source code 4.22: *Snippet of shared/ddr_functions.h*

```
1 #ifndef _DDR_FUNCTIONS_H_
2 #define _DDR_FUNCTIONS_H_
```

```
3
4 /*
5  * (Code removed for brevity)
6  */
7
8 struct lib_common {
9     char *name;
10    FUNCPTR_T func;
11 };
12
13 enum lib_func_cat {func_common, func_conv, func_pool, func_acc, func_fc};
14
15 struct lib_function {
16     enum lib_func_cat category;
17     union {
18         struct lib_conv conv;
19         struct lib_pool pool;
20         struct lib_acc acc;
21         struct lib_fc fc;
22         struct lib_common cm;
23     } cat;
24 };
25
26 /*
27  * (Code removed for brevity)
28  */
29
30 #define MV_conv3x3s1hhhh 0
31 #define MV_conv3x3s2hhhh 1
32
33 /*
34  * (Code removed for brevity)
35  */
36
37 #define CM_pool_ave_ddr 38
38 #define CM_pool_max_ddr 39
39 #define CM_fc_ddr 40
40
41 #endif
```

In the source code 4.22 notice the following:

- The type of `struct` of array `lib` is shown at line 15. From there it is seen that each library function has a category, and data that describe each specific library function. Lines 18-22 indicate that are currently supported five different categories. The last category is named `cm` and includes library functions that do not fit in any other category. For example, `printf` function belongs to this category.

- Each one the `structs` in lines 18-22 contains a field that stores the pointer to the actual function. In the case of category `cm`, this is clearly shown in line 10. The

reader is encouraged to cross reference these lines with the line 9 of source code 4.21.

- Finally, lines 30-40 are symbolic names for numbers 0 to 40. Each symbolic name corresponds to an index of the library array. Thus, it is mandatory for the corresponding numbers to grow sequentially, starting from 0. For example, if the `jumpTable(MV_conv3x3s1hhhh)` call is made, then the `jumpTable` function will return the index 0 of the library array. The symbolic name `MV_conv3x3s1hhhh` is a mnemonic for the programmer.

The last bullet above indicates that there must be a mechanism populating the library array with new functions. Indeed this is done in source code 4.23.

Source code 4.23: *Snippet of shared/ddr_functions_exports.h*

```
1 #ifndef __DDR_FUNCTIONS_EXPORTS_H__
2 #define __DDR_FUNCTIONS_EXPORTS_H__
3
4 #ifdef __MOVICOMPILE__
5 #include <mv_types.h>
6 #include <stdio.h>
7 #include <stdlib.h>
8 #include <strings.h>
9
10 #include <accumulateFp16.h>
11
12 #include <convolution3x3Fp16ToFp16.h>
13 #include <convolution5x5Fp16ToFp16.h>
14
15 /*
16  * (Code removed for brevity)
17  */
18
19 #include "../shave/ddr/asm/vecVecDotProduct32.h"
20
21 #include "../shave/ddr/ddr_conv.h"
22 #include "../shave/ddr/ddr_pool.h"
23 #include "../shave/ddr/ddr_fc.h"
24
25 #endif //__MOVICOMPILE__
26
27 #include "ddr_functions.h"
28
29 /*
30  * (Code removed for brevity)
31  */
32
33 #ifdef __MOVICOMPILE__
34 #define LIB_FUNC_CM(f_name, f_func)                                     \
35     {.category = func_common,                                           \
36      .cat = {                                                           \
37         .cm = {                                                         \
```

```
38                  .name = #f_name,                                          \
39                  .func = (FUNCPTR_T)f_func                                 \
40            }                                                               \
41        }                                                                   \
42      }
43 #else
44 #define LIB_FUNC_CM(f_name, f_func)                                         \
45      {.category = func_common,                                             \
46        .cat = {                                                            \
47            .cm = {                                                         \
48                  .name = #f_name,                                          \
49                  .func = (FUNCPTR_T)NULL                                   \
50            }                                                               \
51        }                                                                   \
52      }
53 #endif
54
55 struct lib_function lib[] = {
56
57      LIB_FUNC_CONV(MV_conv3x3s1hhhh, hhhh, k3x3,
58                                          stride1, mvcvConvolution3x3Fp16ToFp16_asm),
59
60      LIB_FUNC_CONV(MV_conv3x3s2hhhh, hhhh, k3x3,
61                                          stride2, mvcvConvolution3x3s2hhhh_asm),
62
63 /*
64  * (Code removed for brevity)
65  */
66
67      LIB_FUNC_CM(CM_pool_ave_ddr, pool_ave_ddr),
68      LIB_FUNC_CM(CM_pool_max_ddr, pool_max_ddr),
69      LIB_FUNC_CM(CM_fc_ddr, fc_ddr),
70
71 };
72
73 #endif//__DDR_FUNCTIONS_EXPORTS_H__
```

Source code 4.23 requires a couple of explanations:

- Lines 55-71 define the library array that is used by the `jumpTable` function. Notice the symbolic names appearing here are those defined in the source code 4.22. The symbolic names are wrapped around preprocessor macros that will be explained below. Each macro defines an array element. The important detail here is that the integer value of each wrapped symbolic name corresponds to the same position of the initializer list of `lib`. For example, `MV_conv3x3s2hhhh` is the symbolic name for the number 1, it is wrapped around the macro `LIB_FUNC_CONV` and appears as the *second* element (i.e. index 1) in the initializer list of `lib`.

- Macros such as `LIB_FUNC_CONV` or `LIB_FUNC_CM` are used to create elements in the library array. For brevity, only the `LIB_FUNC_CM` macro is shown completely. The

purpose of the macro is to properly initialize the `struct` presented in source code 4.22. Notice that this macro is defined twice. It is defined for the first time in lines 34-42. These lines are activated if the `__MOVICOMPILE__` directive is defined, which happens when this code is compiled with moviCompiler. Otherwise, if this code is compiled with the gcc compiler used for Leon OS, then lines 44-52 are activated instead.

- The purpose of doubly defined macros is to declare and initialize the library array `lib` twice, once for each compiler. Thus, the first definition of lib is done for the SHAVEs. The lib is constructed and placed in DDR, since this file is included in the source code 4.21 containing the `jumpTable` function. Therefore, the `jumpTable` function associates the symbolic names, such as `MV_conv3x3s2hhhh` with the actual implementation of code, which is `mvcvConvolution3x3s2hhhh_asm` in this case. The second definition of `lib` is done for Leon OS. The reason for the second definition is to make sure Leon OS code can associate symbolic names with a description. For example, the `MV_conv3x3s2hhhh` name represents - among other information - a convolution of kernel 3x3 and striding 2x2. This can be easily inferred by a human, straight from the name. However, the program relies in the description associated with the symbolic name to get the same information. Such description is useful for validation purposes during the preprocessing stage of the CNN. For instance, if a fully connected node is provided with the symbolic name `MV_conv3x3s2hhhh`, an error will occur, since the application is able to conclude that this library function is suitable only for convolutional nodes.
- Again, notice the macro `LIB_FUNC_CM`. When src__MOVICOMPILE__ directive is defined, the last argument of the macro stores the actual function implementation in the array library `lib`. This is shown in line 39 of source code 4.23. However, for the same library in Leon OS, the last argument of this macro has no effect, as seen in line 49 of source code 4.23.
- Finally, the reader is advised to pay attention to the first lines of source code 4.23 and more specifically the `include` directives. These directives introduce the declarations of function that will be included in the array library. `include`s of lines 10, 12 and 13 are provided by the MDK. In order to be able to access them, the following lines need to be added in the Makefile:

```
SHAVE_COMPONENTS = yes
ComponentList_SVE += kernelLib/MvCV
```

Also, lines 19, 20-23 include functions provided by the implementation, which include assembly written code and C code. Among this code are the routines describing the computation of convolutional, pooling and fully connected nodes. It is advised to place only thread-safe functions in the DDR array library, because these functions may be executed by multiple SHAVEs at the same time. If, for example,

a function depends on global variables, calling it from multiple SHAVEs simultaneously would create unexpected and absurd behavior.

In conclusion, the DDR library of the implementation uses code provided by the MDK, as well as code written in the directories `shave/ddr` and `shave/ddr/asm`. Because it is useful for the library to be accessible by the SHAVEs and the Leon OS, the source code describing and building the library is put in the directory `shared`. The MDK build system automatically makes code of this directory available to both compilers, thus both SHAVEs and Leon OS.

Chapter 5

Optimization of CNN computational nodes

This chapter presents in detail the implementation of the SHAVE source code that performs the computation of convolution, pooling, and matrix-vector multiplication. The goal is to describe the specifics of each operation and the optimization steps that improve the execution time.

5.1 DMA CMX Driver

The DMA engine is utilized extensively inside the computational nodes, transferring data between CMX and DDR. Therefore, understanding the functionality of the DMA engine exposed by the respective driver is essential. Each DMA transfer is performed thought the issue of a transaction. For the purposes of the CNN implementation, 2D transactions are needed, since the transferred data are shaped as images.

There are several driver functions for declaring 2D transactions:

- `dmaCreateTransaction`: This is the simplest form and can only copy contiguously laid data and place them contiguously at the destination. For example, such function is useful when transferring complete image channels.
- `dmaCreateTransactionSrcStride`: This form can copy non-contiguously laid data and place them contiguously at the destination.
- `dmaCreateTransactionDstStride`: This form can copy contiguously laid data and place them non-contiguously at the destination.
- `dmaCreateTransactionFullOptions`: This form is the most general. It can copy non-contiguously laid data and place them non-contiguously at the destination.

The concept of contiguous and non-contiguous data layout is expressed through the "stride" term. Figure 5.1 illustrates the use of a 2D striding transaction [9]. The goal is to copy the rectangle named "DMA SRC DATA" from the "SRC Start Address" and place it in the rectangle "DMA DST DATA' at "DST Start Address". However, both rectangles are not contiguously laid out in memory, since they are embedded into larger rectangles. In this illustration the Source Line stride (SRC STRIDE) differs from the Destination Line

Stride (DST STRIDE) and could represent a part of an image being cropped from one frame and placed inside another frame which is of different dimensions.

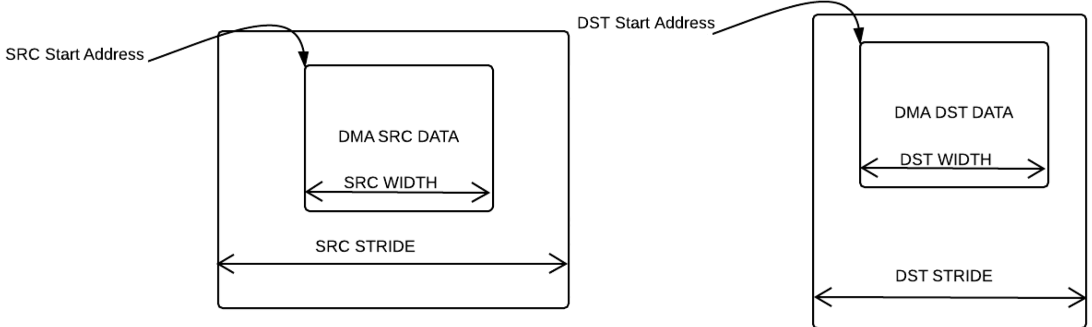


Figure 5.1: *2D striding illustration.*

Also, the Destination Width can be programmed to a different value than Source Width. This is illustrated in figure 5.2.

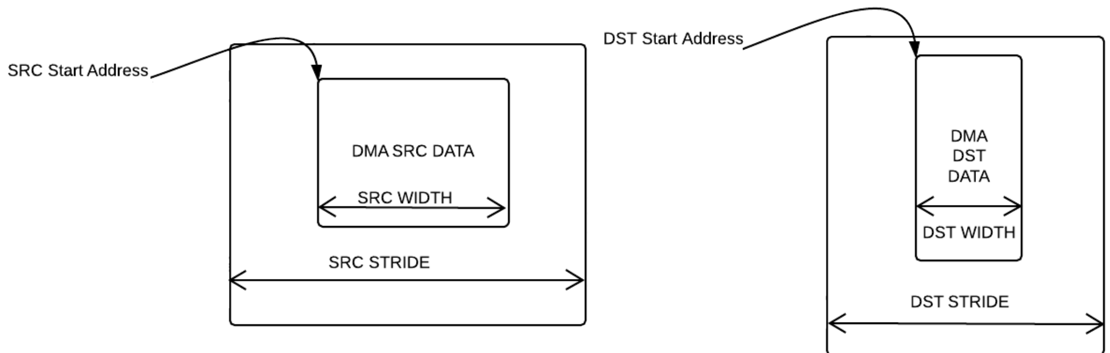


Figure 5.2: *2D striding transaction with different* `DST_WIDTH`, `SRC_WIDTH`.

A snippet using the driver is shown below, where an contiguous 2D rectangle is transferred from DDR to CMX.

```
ref[0] = dmaCreateTransactionFullOptions(
      id,
      &task[0],
      input_src_addr,                    // src
      input_dst_addr,                    // dst
      line_width * number_of_lines,      // byte length
      line_width,                        // src line width
      line_width,                        // dst line width
      line_width,                        // src stride
      input_dst_stride);                 // dst stride
dmaStartListTask(ref[0]);
```

where:

- `input_src_addr` refers to the memory location in DDR where the rectangle resides.

- `input_dst_addr` refers to the CMX local buffer location the rectangle will be embedded.
- `line_width` refers to the length (in bytes) of each line of the rectangle.
- `number_of_lines` refers to number of input lines.
- `input_dst_stride` configures the bytes skipped at the end of each line transferred to the local buffer. If this value is equal to `line_width`, then the lines are placed in memory contiguously, without skipped byte in-between.
- Notice that *src line width* and *dst line width* are equal to `line_width`. This means the shape of the input rectangle will not change during the transfer. Also, the *src stride* is equal to `line_width`, which means that the input rectangle is contiguously laid out in memory.

5.2 Convolution

This section will describe in detail the implementation of convolution. It will focus on the optimization steps that reduce the execution time, by employing several techniques to achieve the final result.

5.2.1 Parallelization schema

Convolution is the most intensive operation in a CNN architecture. In the general case, it is performed in an input 3D volume multiple times, each time with a different kernel. For the purposes of a CNN implementation, parallelizing convolution with respect to the output (feature) maps is a suitable choice. More precisely, each SHAVE utilizes the whole input 3D volume and generates some of the output maps. An example is shown in figure 5.3.

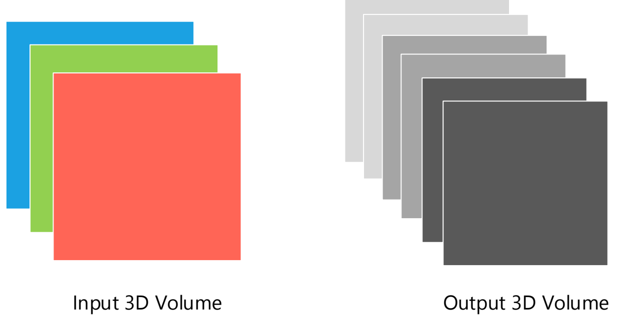


Figure 5.3: *Parallelization of convolution in an RGB image*

For the purposes of this example, the input 3D volume is supposed to be an RGB image and 3 SHAVEs are used. The convolution computation generates 6 output maps. If the computation is spread among 3 SHAVEs, then each SHAVE will generate 2 output

maps. The first SHAVE generates the light gray output maps, the second SHAVE generates the (normal) gray output maps and the third SHAVE generates the dark gray output maps. Remember that each output map requires all three input channels and a different set of weights, among the rest of the output maps.

This parallelization schema does not require the use of locks, because only read-only data are shared. This is of great benefit in an embedded platform, since locks and synchronization of accesses tend to increase power consumption.

5.2.2 Convolution in assembly

An important step towards the improvement of time is writing critical parts of the computation in assembly language. MDK already provides assembly routines for several types of convolution, and all of them share the same interface. For example, one such routine has the form:

```
void conv(half** in, half** out, half mask[], u32 inWidth);
```

Suppose, for the sake of the example, that this routine performs a 5x5 convolution. The code above is explained in detail:

- `half`: This is an alias for the 16-bit floating point datatype supported by the SHAVE processors. Because each SHAVE can operate on 128-bit vectors with SIMD instructions, this means that each vector can be set up to contain 8 `half` numbers.
- `mask`: Convolution requires a kernel to perform the operation. This array parameter requires the same number of elements as the size of convolution. The 5×5 convolution would require 25 array elements.
- `in`: Each one of these routines requires the appropriate amount of input lines to generate just one output line. The number of required input lines depends of the size of the convolution kernel. The 5×5 convolution would require 5 input lines. `in` is actually an array of pointers, each one pointing to a different line.
- `out`: Although the routines generate only one line, the output parameter has the same format as the input one. `out` pointer array needs to contain one element only.
- `inWidth`: This defines the width of the input line. Because the assembly routine utilizes SIMD instructions, it is important to bare in mind that `inWidth` needs to be a multiple of 8. If this is not the case, usually `inWidth` is rounded down to the closest multiple of 8, however this is not always the case (there is a discrepancy among the MDK routines). Also, convolution routines exhibit another peculiarity, namely they access data beyond the input buffer passed to them. This is shown at the following C code that performs a - not optimized - 5×5 convolution:

```
void conv(half** in, half** out, half mask[], u32 inWidth)
{
    int x, y;
```

```
    unsigned int i;
    half* lines[5];
    half sum;

    //Initialize lines pointers
    lines[0] = in[0];
    lines[1] = in[1];
    lines[2] = in[2];
    lines[3] = in[3];
    lines[4] = in[4];

    //Go on the whole line
    for (i = 0; i < inWidth; i++){
        sum = 0.0;
        for (x = 0; x < 5; x++)
        {
            for (y = 0; y < 5; y++)
            {
                sum += lines[x][y - 2] * conv[x * 5 + y];
            }
            lines[x]++;
        }

        out[0][i] = sum;
    }
}
```

Finally, it is explained how SIMD instructions can be used to perform convolution. Suppose a 3×3 convolution with input:

$$in = \begin{bmatrix} a_{1,1} & a_{1,2} & a_{1,3} & a_{1,4} & a_{1,5} & a_{1,6} & a_{1,7} & a_{1,8} & a_{1,9} & a_{1,10} \\ a_{2,1} & a_{2,2} & a_{2,3} & a_{2,4} & a_{2,5} & a_{2,6} & a_{2,7} & a_{2,8} & a_{2,9} & a_{2,10} \\ a_{3,1} & a_{3,2} & a_{3,3} & a_{3,4} & a_{3,5} & a_{3,6} & a_{3,7} & a_{3,8} & a_{3,9} & a_{3,10} \end{bmatrix}$$

and kernel:

$$k = \begin{bmatrix} b_{1,1} & b_{1,2} & b_{1,3} \\ b_{2,1} & b_{2,2} & b_{2,3} \\ b_{3,1} & b_{3,2} & b_{3,3} \end{bmatrix}$$

Then, the convolution can be performed as follows:

Source code 5.1: *Explanation of SIMD version for 3×3 convolution*

Input: *In* (*Three input lines*)

Output: *Out* (*Single output line*)

Declare $k_1 = \begin{bmatrix} b_{1,1} & b_{1,1} & \cdots & b_{1,1} \end{bmatrix}$ (*8 elements*)

Declare $k_2 = \begin{bmatrix} b_{1,2} & b_{1,2} & \cdots & b_{1,2} \end{bmatrix}$ (*8 elements*)

Declare $k_3 = \begin{bmatrix} b_{1,3} & b_{1,3} & \cdots & b_{1,3} \end{bmatrix}$ (*8 elements*)

Declare $k_4 = \begin{bmatrix} b_{2,1} & b_{2,1} & \cdots & b_{2,1} \end{bmatrix}$ (*8 elements*)

Declare $k_5 = \begin{bmatrix} b_{2,2} & b_{2,2} & \cdots & b_{2,2} \end{bmatrix}$ (*8 elements*)
Declare $k_6 = \begin{bmatrix} b_{2,3} & b_{2,3} & \cdots & b_{2,3} \end{bmatrix}$ (*8 elements*)
Declare $k_7 = \begin{bmatrix} b_{3,1} & b_{3,1} & \cdots & b_{3,1} \end{bmatrix}$ (*8 elements*)
Declare $k_8 = \begin{bmatrix} b_{3,2} & b_{3,2} & \cdots & b_{3,2} \end{bmatrix}$ (*8 elements*)
Declare $k_9 = \begin{bmatrix} b_{3,3} & b_{3,3} & \cdots & b_{3,3} \end{bmatrix}$ (*8 elements*)
Declare $Acc = \begin{bmatrix} 0 & 0 & \cdots & 0 \end{bmatrix}$ (*8 elements*)

▷ All operations below are single element-wise assembly instructions
$Acc+ = \begin{bmatrix} a_{1,1} & a_{1,2} & \cdots & a_{1,8} \end{bmatrix} \times k_1$
$Acc+ = \begin{bmatrix} a_{1,2} & a_{1,3} & \cdots & a_{1,9} \end{bmatrix} \times k_2$
$Acc+ = \begin{bmatrix} a_{1,3} & a_{1,4} & \cdots & a_{1,10} \end{bmatrix} \times k_3$
$Acc+ = \begin{bmatrix} a_{2,1} & a_{2,2} & \cdots & a_{2,8} \end{bmatrix} \times k_4$
$Acc+ = \begin{bmatrix} a_{2,2} & a_{2,3} & \cdots & a_{2,9} \end{bmatrix} \times k_5$
$Acc+ = \begin{bmatrix} a_{2,3} & a_{2,4} & \cdots & a_{2,10} \end{bmatrix} \times k_6$
$Acc+ = \begin{bmatrix} a_{3,1} & a_{3,2} & \cdots & a_{3,8} \end{bmatrix} \times k_7$
$Acc+ = \begin{bmatrix} a_{3,2} & a_{3,3} & \cdots & a_{3,9} \end{bmatrix} \times k_8$
$Acc+ = \begin{bmatrix} a_{3,3} & a_{3,4} & \cdots & a_{3,10} \end{bmatrix} \times k_9$
▷ Now *Acc* contains the result of 8 output elements

Note that 5.1 tries to clarify the approach used, not describe precisely the assembly instructions chosen. In fact, the assembly version is far more sophisticated in terms of operations used. Finally, this example does not show the loop that employs the described technique whenever the input width is larger.

Finally, this subsection makes clear that the assembly routines used for convolution are far from performing convolution in a 3D volume. In fact, these routines cannot even perform convolution in a (2D) rectangle, which means that a lot of work needs to be done.

5.2.3 Optimization: Reduced number of routine calls

The first step towards the implementation of convolution for 3D volumes is performing convolution in a (2D) rectangle. The results of several convolutions on rectangles will be combined to generate an output map. For the rest of the section, the terms input/output rectangle and input/output channel will be used interchangeably.

The traditional approach towards convolution on rectangles is using the assembly routine to produce one output line, and repeat this process iteratively, to generate all the output lines dictated by the input rectangle and the type of convolution.However, an alternative approach is employed. Input data are carefully placed inside local CMX buffers to compute the result making a single call. The characteristics of this optimiza-

tion are:

- Only one call is required irrespectively of the size of the input channel.
- Multiple calls are especially wasteful for small input channel sizes. For these sizes, each input line to the convolution routine is not large enough to benefit from the SIMD capabilities of Myriad2.
- Assembly language uses delay slots in branch instructions. Tight loops, as those generated by the multiple calls approach, cannot use the delay slots. This results in wasted cycles and thus inferior performance.

The remaining of this subsection will explain how this mechanism operates through the presentation of meticulously designed diagrams. It is important to emphasize that the present optimization is only concerned about a single input channel, and does not account for the relation between other input channels. For the sake of the example, it is assumed that the height of the input channel is 6 and a 5×5 convolution with zero-padding and unit stride is applied.

First of all, the input channel is laid out in memory in row-major order, much like C stores 2D arrays in memory. This is clearly shown in figure 5.4. Due to the fact that the convolution kernel is 5×5 and because the input channel height is 6, the height of the output rectangle will be 3. Notice, in this figure, that the lines are shown symbolically.

Figure 5.4: *Layout of input channel in memory*

No elements are presented, because a convolution routine that generates one output line is already provided.

Normally three calls to the convolution routine would be necessary. This is shown in pseudocode 5.2. The first call uses the first 5 lines to generate the first output line. The `inWidth` argument is equal to the number of elements in each line. The remaining two calls follow the same pattern. This optimization is based on the following observation: The same pointer variable in each call to `conv` changes sequentially. For example, the second pointer variable in the first call to `conv` points to “Line 1”. The same variable in

Source code 5.2: *Multiple-call convolution on input channel*

```
1: Call conv with input: Lines 0-4, output: Outline 0, width: Line elements
2: Call conv with input: Lines 1-5, output: Outline 1, width: Line elements
3: Call conv with input: Lines 2-6, output: Outline 2, width: Line elements
```

the second call to `conv` points to "Line 2". Finally, the same variable in the third call to `conv` points to "Line 3". This is true for all the pointer variables, e.g. the fifth pointer variable points to "Line 4", then "Line 5" and finally "Line 6".

As a result, if the pointers are placed as show in figure 5.5, making a single call to `conv` with `inWidth` equal to 3 times the number of elements in each line, gives equivalent results. Notice, that the pointers in the first line of pseudocode 5.2 and the pointers shown in figure 5.5, point to the same locations. Since the `inWidth` is now tripled, each pointer points to three times more data. For example, "Ptr1" points to the memory block "Line 1", "Line 2", "Line 3".

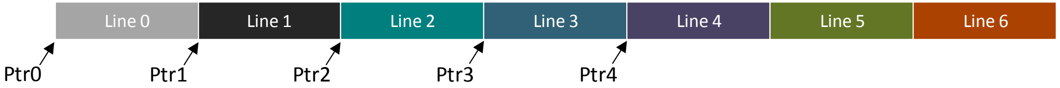


Figure 5.5: *Layout of input channel in memory with pointers placed appropriately*

It is now becoming clear how one call can calculate the result of three output lines. Each pointer starts at the appropriate location and is able to see the next input lines as it moves to the right. This is because data are laid out continuously in memory. Effectively, the convolution routine views the data as presented in figure 5.6, although each line is present in memory only once.

Ptr0→	Line 0	Line 1	Line 2
Ptr1→	Line 1	Line 2	Line 3
Ptr2→	Line 2	Line 3	Line 4
Ptr3→	Line 3	Line 4	Line 5
Ptr4→	Line 4	Line 5	Line 6

Figure 5.6: *How the convolution routine "sees" the data in memory*

Comparing figure 5.6 with figure 5.4 it is visible that sliding horizontally the 5×5 convolution kernel across the figure 5.6 is the same as sliding horizontally the 5×5 convolution kernel three times - starting from "Line 0", then "Line 1" and finally "Line

2" - in figure 5.4. A concrete example is given below with smaller sizes:

$$\text{Input as in fig. 5.4} \equiv I_1 = \begin{bmatrix} 3 & 3 & 2 & 1 & 0 \\ 0 & 0 & 1 & 3 & 1 \\ 3 & 1 & 2 & 2 & 3 \\ 2 & 0 & 0 & 2 & 2 \\ 2 & 0 & 0 & 0 & 1 \end{bmatrix} \qquad Kernel \equiv K = \begin{bmatrix} 0 & 1 & 2 \\ 2 & 2 & 0 \\ 0 & 1 & 2 \end{bmatrix}$$

$$\text{Input as in fig. 5.6} \equiv I_2 = \left[\begin{array}{ccccc|ccccc|ccccc} 3 & 3 & 2 & 1 & 0 & 0 & 0 & 1 & 3 & 1 & 3 & 1 & 2 & 2 & 3 \\ 0 & 0 & 1 & 3 & 1 & 3 & 1 & 2 & 2 & 3 & 2 & 0 & 0 & 2 & 2 \\ 3 & 1 & 2 & 2 & 3 & 2 & 0 & 0 & 2 & 2 & 2 & 0 & 0 & 0 & 1 \end{array}\right]$$

The output of the convolution $I_1 * K$ is:

$$I_1 * K = \begin{bmatrix} 12 & 12 & 17 \\ 10 & 17 & 19 \\ 9 & 6 & 14 \end{bmatrix}$$

While the output of convolution $I_2 * K$ is:

$$I_2 * K = \left[\begin{array}{ccc|cc|ccc|cc|ccc} 12 & 12 & 17 & 14 & 10 & 10 & 17 & 19 & 23 & 17 & 9 & 6 & 14 \end{array}\right]$$

One drawback of this method is that garbage results are generated when the convolution kernel passes from one column to another, as seed in vector $I_2 * K$. However, the gain from the reduction of routine calls outperforms the waste generated between intermediate columns. This is especially true for small kernels, that appear in abundance in CNNs. Also, for increase in performance it is better to align the distance between successive pointers at a 16-byte boundary. In other words, it is better $Ptr1 - Ptr0 \equiv 0 \pmod{128}$, $\ldots Ptr3 - Ptr4 \equiv 0 \pmod{128}$. To achieve such alignment it is important to pad each input line with junk elements. Padding can be performed efficiently using DMA stride, as explain at the beginning of the chapter.

5.2.4 Optimization: Reduced number of DMA transfers

Because local buffers have limited space, it is necessary to bring the same input multiple times during the execution of convolution. It would be beneficial if these transfer could be reduced, for a couple of reasons. First, less DMA transfers lower the power consumption. Second, DDR memory can serve fewer transaction more easily, than being clogged by a large number of requests. The normal approach to convolution is described as follows:

The focus on pseudocode 5.3 is at the for-loops of lines 1 and 2. The approach described here brings all the input channels from memory to generate every output map. The optimization of this subsection tries to mitigate this problem. The idea is to make better use of each input channel before bringing the next input channel. That is,

Source code 5.3: *Normal approach to DMA transfers in convolution*

```
1: for map in output_maps do
2:     for channel in input_channels do
3:         Bring input channel with DMA
4:         Apply 2D convolution using channel
5:         Accumulate the result of 2D convolution to map buffer
6:     end for
7:     Send map buffer back with DMA
8: end for
```

every input channel participates in partial calculation of multiple output maps before it is replaced by the next input channel. This is shown in pseudocode 5.4.

Source code 5.4: *Coalescing approach to DMA transfers in convolution*

```
1: Split output_maps into several groups
2: for group in output_maps_groups do
3:     for channel in input_channels do
4:         Bring input channel with DMA
5:         for map in group do
6:             Apply 2D convolution using channel
7:             Accumulate the result of 2D convolution to map buffer
8:         end for
9:     end for
10:    Send the group of map buffers back with DMA
11: end for
```

The optimization is called coalescing, because multiple memory accesses to the same data are merged together. Essentially, the two loops of pseudocode 5.3 have been swapped. Lines 7 and 10 of pseudocode 5.4 indicate that multiple local buffers are needed to store the output maps, before they are returned back to DDR. In other words, DMA transfers are reduced by using multiple local buffers efficiently. Output maps are split into groups due to the limited amount of local buffers. If there was enough space to simultaneously keep all the generated output maps in local buffers, then each input channel would be brought in (with DMA) only once.

The careful reader will soon realize that pseudocodes 5.3 and 5.4 do not mention anything about the convolution kernel masks. This omission is deliberate, in order to simplify the description of the optimization. In reality, kernel masks are accessed by the SHAVEs thought the data cache subsystem. This has a couple of benefits:

- Although DMA would be the ideal choice, the hardware supporting it has limited amount of DMA agents. In fact the are four DMA agent that are responsible for serving DMA requests. When multiple SHAVEs are running in parallel, which is usually the case, it is more wise to task DMA engine with doing larger data block transfers.
- The access pattern to the kernel masks is sequential, which means that spacial locality of data favors the cache utilization.

- Another reason that justifies the use of cache is an extension of the current parallelization schema. When performing convolution in spatially large input channels, each SHAVE could be assigned to a different part of the input channel. This means that all the shaves use the same kernel masks at the same time to calculate the output maps. In other words, the access pattern to kernel masks exhibits temporal locality.
- Finally, utilizing both DMA and cache for data transfers fully exploits the bandwidth of DDR, since the requests for data are both explicit and implicit. This also simplifies the algorithm of convolution.

Note: Myriad2 has multiple processors and multiple cache hierarchies. However, it lacks a cache coherence protocol to maintain the synchronization of the same data across those caches. This makes cache management a burden for the programmer. Nevertheless, fetching kernel masks through cache does not come with these problems, because the data are read-only and are placed in a permanent place in memory.

5.2.5 Optimization: DMA transfers are "hidden" in computation

Myriad2 DMA engine operates in an asynchronous way. This means that a DMA request can be made, and while data are being transferred between DDR and CMX, other computations can be performed. Source code 5.5 show this functionality.

Source code 5.5: *Asynchronous operation of DMA engine*

```
1  ref = dmaCreateTransactionFullOptions( ...parameters... );
2
3  // Send request to DMA engine to begin transfer of data
4  dmaStartListTask(ref);
5
6  // Perform computations here
7  do_computations();
8
9  // Wait for DMA request to finish
10 dmaWaitTask(ref);
```

It is important to emphasize a limitation of the DMA engine. It is not possible to issue another DMA request from the same SHAVE, while some other DMA request is already active. For example, the `do_computations()` function in code 5.5 cannot contain a DMA request. The optimization presented tries to exploit this asynchronous behavior. If the `do_computations()` function takes longer time than the DMA transfer running simultaneously, then the DMA transfer is as if it never happened in terms of time spent to it. In other words, DMA transfers are "hidden" in computation. This technique is used both for transferring data from DDR to CMX and vice versa. A simplified version of code will be presented to clarify and comment on various aspects of the optimization. Note that two versions will be presented to assist the understanding of the technique.

Source code 5.6: *First version of "hiding" DMA inside computation*

```
1 // Loop over the number of maps
2 u32 mapNo = 0;
3 while (mapNo < info->outputMapsNo)
4 {
5     // Coalescing
6     for (u32 i = 0; i < info->coalescing_num; i++)
7         bzero(localOutput[i], outputBufferBytes);
8
9     // Swap prefetchBuffer between localInput and localInput1
10    prefetchBuffer = localInput;
11    inputPtr = inputPtr2;
12
13    // Prefetch the first ping pong input buffer to warm up
14    u8* inputAddr = (u8*)info->input[0];
15
16    ref[0] = dmaCreateTransactionFullOptions( ... );
17    dmaStartListTask(ref[0]);
18
19    // Now go in loop
20    for (int channelIdx = 0;
21            channelIdx < info->channelsNo; channelIdx++) {
22        // Ensure our last input DMA task completed
23        dmaWaitTask(ref[0]);
24
25        prefetchBuffer = (void *) ((u32)prefetchBuffer ^
              (u32)toggleInputBuffer);
26
27        // Prefetch next buffer if not the last one
28        if ((channelIdx+1) < (info->channelsNo)) {
29            inputAddr = (u8*)info->input[channelIdx+1];
30
31            ref[0] = dmaCreateTransactionFullOptions( ... );
32
33            dmaStartListTask(ref[0]);
34            // Do not wait for DMA completion here;
35            // We do this at top of next loop.
36        }
37
38        inputPtr = (void *) ((u32)inputPtr ^ (u32)toggleInputPtr);
39
40        // Coalescing
41        for (u32 i = 0; (mapNo+i) < info->outputMapsNo &&
42                            i < info->coalescing_num; i++) {
43            // Convolution and accumulation
44            do_convolution_and_accumulation();
45        }
46    } // end for channels in map
47
48    // Apply the bias
49    if (info->convBiases != NULL) {
50        // Coalescing
```

```
51          for (u32 i = 0; (mapNo+i) < info->outputMapsNo &&
52                               i < info->coalescing_num; i++) {
53              apply_bias();
54          }
55      } // end for apply the bias
56
57      // Output the result with coalescing
58      for (u32 i = 0; mapNo < info->outputMapsNo &&
59                          i < info->coalescing_num; i++, mapNo++) {
60          ref[1] = dmaCreateTransactionFullOptions( ... );
61
62          dmaStartListTask(ref[1]);
63          dmaWaitTask(ref[1]);
64      }
65  }
```

The code 5.6 contains optimizations mentioned in subsection 5.2.4. These optimizations have not been removed because the next version of the current optimization will exploit them to better "hide" the DMA transfers in computation. Focus is on specific lines:

- Line 3: `mapNo` seems to increase one unit at a time. However, if coalescing is used the increase happens in larger amounts at once, as seen in line 59.
- Lines 10 and 11: Double buffering for the input data is used. While DMA transfers data from DDR to CMX in one buffer, at the same time the computation is performed in another buffer.
- Lines 16 and 17: The first input channel needs to be transfered from DDR. This transfer cannot be hidden in computation. Nevertheless, the next transfers will happen simultaneously with computations.
- Lines 20-36: These lines bring data in (with DMA) and perform 2D convolution at the same time. Hiding of DMA transfers happens inside this loop.
- Lines 58-64: Calculated output channels are returned to DDR. At the current form, these lines act as a barrier reducing performance, because the DMA operation is not accompanied by computation. Another drawback of this version is that the larger the coalescing, the more time these lines take, because DMA is the only operation happening. It is clear that improvements can be made regarding these lines.

Loop unrolling is used to "hide" the DMA transfers from CMX to DDR. More specifically, the last iteration of the for-loop in lines 20-46 of code 5.6 is moved below the for-loop. Also, another loop-unrolling occurs at the for-loop in lines 41-45 of code 5.6. As the result, lines 48-64 of this code need to change to the code presented in 5.7.

Source code 5.7: *Second version of "hiding" DMA inside computation*

```
// Loop over the number of maps
u32 mapNo = 0;
while (mapNo < info->outputMapsNo)
{
    // Code here remains unchanged.
    // Code omitted for brevity.

    actual_coalescing = MIN(info->coalescing_num, info->outputMapsNo - mapNo);

    // Loop-unroll the last channelIdx
    // Ensure our last input DMA task completed
    dmaWaitTask(ref[0]);

    inputPtr = (void *) ((u32)inputPtr ^ (u32)toggleInputPtr);

    // Convolution and accumulation
    do_convolution_and_accumulation();

    if (info->convBiases != NULL) {
        apply_bias();
    }

    // Also, loop-unroll the for-loop inside the channelIdx for-loop
    for (u32 i = 1; i < actual_coalescing; i++) {
        ref[1] = dmaCreateTransactionFullOptions( ... );

        dmaStartListTask(ref[1]);

        // Convolution and accumulation
        do_convolution_and_accumulation();

        if (info->convBiases != NULL) {
            apply_bias();
        }

        dmaWaitTask(ref[1]);
    }

    ref[1] = dmaCreateTransactionFullOptions( ... );

    dmaStartListTask(ref[1]);
    dmaWaitTask(ref[1]);

    mapNo += actual_coalescing;
}
```

The characteristic of unrolling both loops are:

- As shown in code 5.6, the last iteration of the for-loop in lines 20-46 does not bring new data. In fact, this iteration only makes computations and then the bias is applied (shown in lines 48-55). Thus, the bias term can be moved inside the last

iteration code. So, lines 48-55 of the code 5.6 move inside for-loop in lines 41-45. However, this movement happens at the iteration only, where loop unrolling has occurred. This is shown at lines 17-22 and lines 30-34 of code 5.7.

- Next, from code 5.6 is seen that it is possible to improve the lines 58-64. During the last iteration it would be better if the order of convolution, accumulation, application of bias, and DMA transfer of output map to DDR was followed for each output map. Instead in 5.6, the DMA transfer happens at the end for multiple maps together. To perform the idea described here, another loop unrolling is necessary. Notice that the for-loop in lines 24-37 of code 5.7 starts from index 1 instead of 0.

- The benefit of performing the change described in 5.7 is great. The execution time is lowered and at the same time the amount of coalescing can be increased dropping the execution time even more. In contrast to the code in 5.6, where the DMA transfer of output maps acted as a barrier before processing the next group of output maps, the code in 5.7 does not suffer from this behavior. As a result, if local CMX buffer have enough space to support larger coalescing numbers, then it is better to do so.

5.3 Pooling

This section will describe in detail the implementation of pooling. In fact, pooling is a simplified version of convolution, since there are a couple of similarities between them. First, both convolution and pooling operate in 2D rectangles, where the generation of an output element requires several input elements. Second, pooling accepts as input a 3D volume and generates an output 3D volume, much like convolution does. However there is a caveat, since each output map generated by pooling requires data from a single input channel. In contrast, convolution requires all the input 3D volume to generate a single output map. This section will present the optimization applied to pooling that reduces the execution time.

5.3.1 Parallelization schema

Pooling is not a computationally intensive operation, compared to convolution, since each input element is used exactly once to participate in the generation of a single output element. For the purposes of a CNN implementation, parallelizing pooling with respect to the output (feature) maps is a suitable choice. More precisely, each SHAVE uses some of the input channels to generate the respective output maps. An example is shown in figure 5.7.

As shown in this example, the input 3D volume consists of 6 input channels and 3 SHAVEs are used. The pooling computation generates 6 output maps. If the computation is spread among 3 SHAVEs, then each SHAVE will generate 2 output maps. The first SHAVE generates the light and dark gray output maps, the second SHAVE generates the

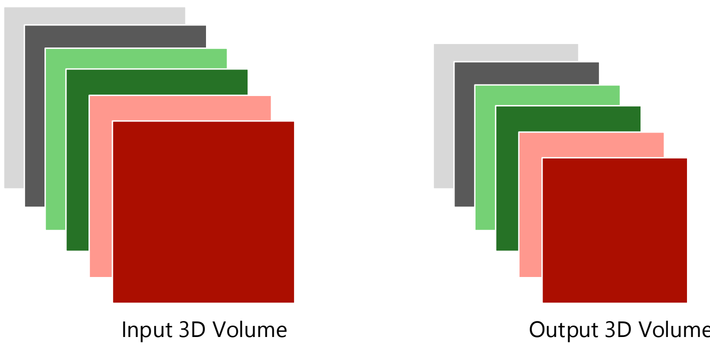


Figure 5.7: *Parallelization of pooling in an 3D volume*

light and dark green output maps and the third SHAVE generates the light and dark red output maps. Remember that each output map, in order to be produced, requires the respective input channel, i.e. the input channel of the same color.

This parallelization schema does not require the use of locks, because there is no data sharing. This is of great benefit in an embedded platform, since locks and synchronization of accesses tend to increase power consumption.

5.3.2 Pooling in assembly

As with convolution, it is important to use an assembly optimized routine for the time-critical parts of the computation. MDK already provides assembly routines for several types of pooling, and all of them share the same interface. For example, one such routine has the form:

```
void pool(half** in, half** out, half mask[], u32 outWidth);
```

Suppose, for the sake of the example, that this routine performs a 2x2 pooling with stride 2. The code above is explained in detail:

- `half`: This is an alias for the 16-bit floating point datatype supported by the SHAVE processors. Because each SHAVE can operate on 128-bit vectors with SIMD instructions, this means that each vector can be set up to contain 8 `half` numbers.
- `in`: Each one of these routines requires the appropriate amount of input lines to generate just one output line. The number of required input lines depends of the size of the pooling kernel. The 2×2 pooling would require 2 input lines. `in` is actually an array of pointers, each one pointing to a different line.
- `out`: Although the routines generate only one line, the output parameter has the same format as the input one. `out` pointer array needs to contain one element only.
- `outWidth`: This defines the width of the output line. Because the assembly routine utilizes SIMD instructions, it is important to bare in mind that `outWidth` needs to

be a multiple of 8. If this is not the case, usually `outWidth` is rounded down to the closest multiple of 8, however this is not always the case (there is a discrepancy among the MDK routines). For better understanding, the following C code that performs a - not optimized - 2×2 max pooling with stride 2 is given:

```
void pool(half** in, half** out, u32 outWidth)
{
    half max;
    u32 tl = 0;
    u32 i;

    for (i = 0; i < outputWidth; i++)
    {
        tl = i * 2;
        //[X][ ]
        //[ ][ ]
        max = src[0][tl];

        //[ ][X]
        //[ ][ ]
        max = (max < src[0][tl + 1]) ? src[0][tl + 1] : max;

        //[ ][ ]
        //[X][ ]
        max = (max < src[1][tl]) ? src[1][tl] : max;

        //[ ][ ]
        //[ ][X]
        max = (max < src[1][tl +1]) ? src[1][tl + 1] : max;
        dst[0][i] = max;
    }
}
```

Notice that the source code above refers to max pooling. The operation of average pooling is not provided in a separate assembly routine, since it is equivalent to convolution with proper weights in the kernel masks. For example, 3×3 average pooling can be performed with a 3×3 convolution routine, where the weight of each kernel mask element would be equal to $1/9$. However, using convolution for pooling requires a wrapper routines that converts the convolution assembly routine interface to a pooling one. It is suggested to write separate assembly routines for average pooling, for better performance and arithmetic precision.

5.3.3 Optimization: Reduced number of routine calls

This is the same optimization applied to convolution, although this time it is extended to support striding. The extension is straightforward and will be described though an example. It is important to emphasize that the present optimization is only concerned about a single input channel. For the sake of the example, it is assumed that the height

of the input channel is 6 and a 3×3 pooling (average or max makes no difference) with zero-padding and stride 2 is applied.

Again, as with convolution, suppose that the data are laid out in memory as shown in figure 5.8. Due to the fact that the pooling kernel is 3×3 with stride 2 and because the input channel height is 6, the height of the output rectangle will be 3. Notice, in

Figure 5.8: *Layout of input channel in memory*

this figure, that the lines are shown symbolically. No elements are presented, because a pooling routine that generates one output line is already provided.

For the specific pooling operation of the example, the first output line will use the input lines “Line 0” to “Line 2”, the second output line with use the input lines “Line 2” to “Line 4” and finally the third output line will use the lines “Line 4” to “Line 6”.

As a result, if the pointers are placed as show in figure 5.9, making a single call to `pool` with proper `outWidth`, gives equivalent results.

Figure 5.9: *Layout of input channel in memory with pointers placed appropriately*

More precisely, the `outWidth` should be equal to the output width that is generated if pooling is performed in a line with length thrice as much as the length of “Line 0”. Notice that since stride is 2, two separate buffers are required. This also means that two DMA

transfers are necessary to place the input data in these two buffers.

It is now becoming clear how one call can calculate the result of three output lines. Each pointer starts at the appropriate location and is able to see the next input lines as it moves to the right. Effectively, the pooling routine "sees" the data as presented in figure 5.10, although each line is present in memory only once.

Ptr0→	Line 0	Line 2	Line 4
Ptr1→	Line 1	Line 3	Line 5
Ptr2→	Line 2	Line 4	Line 6

Figure 5.10: *How the pooling routine "sees" the data in memory*

This extension generates garbage results, much like it is done in convolution. However, the gain from the reduction of routine calls outperforms the waste generated, which is especially true for small kernels, that appear in abundance in CNNs. Also, for increase in performance it is better to align the distance between successive pointers at a 16-byte boundary.

With this optimization in mind, the pooling computation is described in the source code 5.8.

Source code 5.8: *Main part of computation in pooling*

```
1 // Loop over the number of maps
2 u32 mapNo = firstMapNo;
3
4 while (mapNo < lastMapNo) {
5     for (int i = 0; i < info->in_buffers_num; i++) {
6         if (info->in_buffers[i].elements == 0) continue;
7
8         ref[0] = dmaCreateTransactionFullOptions(...);
9         dmaStartListTask(ref[0]);
10        dmaWaitTask(ref[0]);
11    }
12
13    pool((void**)inputPtr, (void**)&localOutput, info->out_buffer_elements);
14
15    ref[1] = dmaCreateTransactionFullOptions(...);
16
17    dmaStartListTask(ref[1]);
18    dmaWaitTask(ref[1]);
19
20    mapNo++;
21 }
```

Each iteration of the while-loop defined in line 4 transfers data from DDR to CMX, computes a single output map and returns the result data back to DDR. The for-loop

in lines 5-11 perform the input DMA transfer. Every iteration of this loop transfers a part of the input data and places them in a separate local CMX buffer. For instance, the example described previously requires two such local buffers, as seen in figure 5.9. Notice, that pooling does not employ the rest of optimization steps used in convolution. This is because the computational part, performed in line 13, is very brief. As a result, the overhead and the code complexity posed by these extra optimizations does not worth integrating them into the implementation.

5.4 Fully connected

This section will describe in detail the implementation of matrix-vector multiplication. Matrix-vector multiplication is the operation needed for the fully connected nodes of a CNN. Improving the performance of this operation is a challenge, since its characteristic are rather constraining.

5.4.1 Matrix-vector multiplication is I/O bounded

The main characteristic of this operation is that each element of the matrix is used exactly once. More precisely, every element participates in the computation that generates one element of the output vector. On the other hand, the input vector of the multiplication is needed by every row of the matrix. At the same time, the primitive operations dictated by the matrix-vector multiplication are simple, namely addition and multiplication. This offers little room for techniques used in convolution. In fact, the matrix-vector multiplication is I/O bounded, meaning that transferring the data from DDR to CMX and vice versa is more time consuming than the time required by the computation itself. As a side note, matrix-matrix multiplication offers greater reusability of the the same data, making it optimization-friendly.

5.4.2 Parallelization schema

For the purposes of a CNN implementation, parallelizing fully connected nodes by separating the input matrix into row bands is a suitable choice. More precisely, each SHAVE uses some rows of the input matrix to generate the respective elements in the output vector. An example is shown in figure 5.11.

For the purposes of this example, the input matrix is split into 3 bands, since 3 SHAVEs are used. Each SHAVE will use the whole input vector multiple (in this example it is 3) times to perform a vector-vector inner product operation with some rows of the input matrix. The first SHAVE generates the blue output vector elements, the second SHAVE generates the green output vector elements and the third SHAVE generates the red output vector elements.

This parallelization schema does not require the use of locks, too.

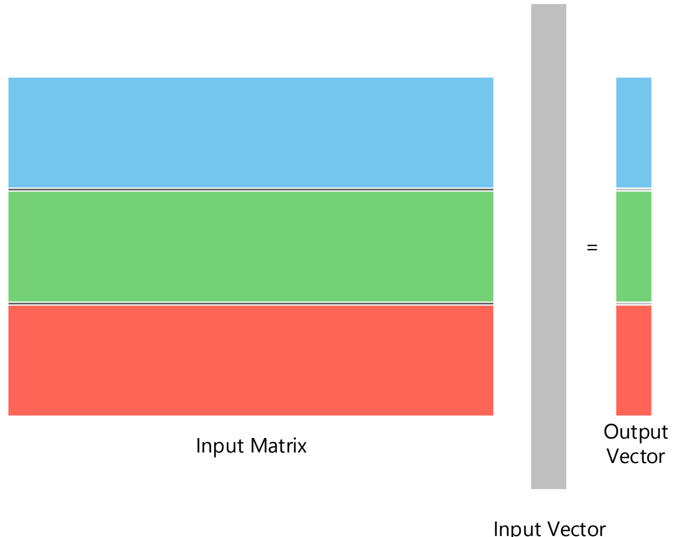


Figure 5.11: *Parallelization of fully connected nodes matrix-vector multiplication*

5.4.3 Optimized implementation

A simple, yet effective implementation of this operation is presented in source code 5.9. The optimizations used are the following:

- Double buffering is used. This way DMA transfers are "hidden" inside computation. As a result, the time of transferring the data is overlapped with the time of computation, which makes the effective time of transfer to be reduced.
- For supporting larger matrices, the input vector is not transferred to local buffers at once. Instead, it is split into segments that are transferred piece by piece. This will be explained shortly in more detail.

Source code 5.9: *Matrix-vector multiplication for the fully connected nodes*

```
1  // In each iteration, we process a part of all rows in the matrix
2  for (int width_start = 0; width_start < inputWidth;
3                                     width_start += seg_width) {
4
5      prefetchBuffer = (half *)localInput;
6      currentBuffer = (half *)localInput1;
7
8      true_seg_width = MIN(seg_width, inputWidth-width_start);
9      true_input_bytes = info->inputBPP * true_seg_width;
10
11     // Bring a part of the input vector and store it in the vector buffer.
12     ref[0] = dmaCreateTransaction(...);
13     dmaStartListTask(ref[0]);
14
15     // Round-up to true_seg_width to make it multiple of 8
16     true_seg_width8 = ((true_seg_width + 7) / 8) * 8;
```

```
17
18     dmaWaitTask(ref[0]);
19
20     // Bring a part of the first row assigned to the current SHAVE
21     // that is compatible with the vector buffer elements.
22     // Put the result into the prefetchBuffer.
23     ref[0] = dmaCreateTransaction(...);
24     dmaStartListTask(ref[0]);
25     dmaWaitTask(ref[0]);
26
27     // Perform the inner product of this part of the row with the vector
28     // buffer.
29     for (int lineNo = firstLineNo + 1; lineNo < lastLineNo; lineNo++) {
30         // Bring the same part of data of the next row.
31         // Put the result into the currentBuffer.
32         ref[0] = dmaCreateTransaction(...);
33         dmaStartListTask(ref[0]);
34
35         // Inner product
36         localOutput[lineNo - firstLineNo - 1] +=
37             (half)dot(vectorBuffer, prefetchBuffer, true_seg_width8);
38
39         // Swap buffers.
40         toggleInputBuffer = prefetchBuffer;
41         prefetchBuffer = currentBuffer;
42         currentBuffer = toggleInputBuffer;
43
44         dmaWaitTask(ref[0]);
45     }
46
47     // Inner product for the last row assigned to the current SHAVE.
48     localOutput[line_diff - 1] +=
49         (half)dot(vectorBuffer, prefetchBuffer, true_seg_width8);
50 }
51
52 // Apply bias
53 if (info->bias != NULL) {
54     half *bias = (half *)(info->bias);
55     for (int i = 0; i < lastLineNo - firstLineNo; i++) {
56         localOutput[i] += bias[i + firstLineNo];
57     }
58 }
59
60 // Apply ReLU (if needed)
61 if (info->with_relu) {
62     relu_inplace64_h(localOutput, line_diff64);
63 }
64
65 // Return results
66 ref[0] = dmaCreateTransaction(...);
67 dmaStartListTask(ref[0]);
```

```
68 dmaWaitTask(ref[0]);
```

Explanation of the source code 5.9 follows below:

- *Line 2*: This for-loop iterates over the segments of the input vector. With this iteration technique, it is possible to process matrices whose number of columns is very large. Each segment of the input vector is brought in (with DMA) exactly once. Afterwards, the inner product between this segment of the input vector and the respective part of a matrix row is computed. The result is stored inside a local buffer. Notice, that every inner product calculates part of the result of an element in the output vector. The complete result requires the inner product of the whole input vector, i.e. all the segments of the input vector.
- *Lines 5, 6*: These lines indicate the double buffering scheme. While one buffer is involved in a computation, the other buffer is fed with data.
- *Line 11*: Transfers the appropriate input vector segment. The first element of this segment is indicated by the `width_start` variable in line 2.
- *Line 23*: Fills the `prefetchBuffer` with part of the first row of the matrix band that is assigned to the current SHAVE. The number of matrix rows, which are assigned to the SHAVE, are indicated in the second for-loop (lines 29-45).
- *Lines 29-45*: These lines compute part of the output vector elements that are assigned to the current SHAVE. Each iteration uses the same segment of the input vector to perform the inner product with part of a different row of the input matrix. The result of each inner product is accumulated to the `localOutput` buffer. To better understand the way an output element is calculated, the following concrete example is provided:

  Suppose a SHAVE is responsible for the matrix-vector multiplication $A \times b$, where

$$A = \begin{bmatrix} 3 & 3 & 2 & 1 & 0 & 2 \\ 0 & 0 & 1 & 3 & 1 & 0 \\ 3 & 1 & 2 & 2 & 3 & 2 \\ 2 & 0 & 0 & 2 & 2 & 1 \end{bmatrix} \qquad b = \begin{bmatrix} 0 \\ 1 \\ 2 \\ 2 \\ 2 \\ 0 \end{bmatrix}$$

  Then, the output $y \equiv [y_1, y_2, y_3, y_4] = [9, 10, 15, 8]^T$, if it is assumed that each segment of the input vector is 3 elements long, is computed as follows:

$$y_1 = [3, 3, 2] \times [0, 1, 2]^T$$
$$y_2 = [0, 0, 1] \times [0, 1, 2]^T$$

$$y_3 = [3, 1, 2] \times [0, 1, 2]^T$$
$$y_4 = [2, 0, 0] \times [2, 2, 0]^T$$
$$y_1 += [1, 0, 2] \times [2, 2, 0]^T$$
$$y_2 += [3, 1, 0] \times [2, 2, 0]^T$$
$$y_3 += [2, 3, 2] \times [2, 2, 0]^T$$
$$y_4 += [2, 2, 1] \times [2, 2, 0]^T$$

Each iteration of the outer loop updates the result of all the components of the vector y. The inner for-loop is responsible for the computations involving the same segment of the input vector. In the example given above, the first 4 lines are computed by the first outer for-loop iteration and the next 4 lines are computed by the second outer for-loop iteration.

- *Line 32* refers to the double buffering technique. The data transfer is initiated there and the wait for it to finish happens at line 44. Between these lines the inner product is performed.
- *Lines 53-58* apply the bias vector. After the finish of the outer for-loop, the bias is applied. The whole part of the output vector has been produced and is kept in CMX memory, in array variable `localOutput`.
- *Lines 61-63*: The ReLU operation is applied in-place. This routine is written in C, rather than assembly, since it follows a canonical access pattern that the compiler is able to optimize by itself.

  This function is given below, to satisfy the curiosity of the reader:

```
void
relu_inplace64_h(half *ptr, int size)
{
    // +Inf in fp16 format
    int p_inf_bin = 0x00007C00;
    half p_inf = *((half *) &p_inf_bin);

    half8 v = {p_inf, p_inf, p_inf, p_inf, p_inf, p_inf, p_inf, p_inf};
    half8 *addr_h8 = (half8 *)ptr;

    for (int j = 0; j < size/8; j += 8) {
        addr_h8[j]   = __builtin_shave_cmu_clamp0_f16_rr_half8(addr_h8[j
             ], v);
        addr_h8[j+1] =
            __builtin_shave_cmu_clamp0_f16_rr_half8(addr_h8[j+1], v);
        addr_h8[j+2] =
            __builtin_shave_cmu_clamp0_f16_rr_half8(addr_h8[j+2], v);
        addr_h8[j+3] =
            __builtin_shave_cmu_clamp0_f16_rr_half8(addr_h8[j+3], v);
        addr_h8[j+4] =
            __builtin_shave_cmu_clamp0_f16_rr_half8(addr_h8[j+4], v);
```

```
        addr_h8[j+5] =
            __builtin_shave_cmu_clamp0_f16_rr_half8(addr_h8[j+5], v);
        addr_h8[j+6] =
            __builtin_shave_cmu_clamp0_f16_rr_half8(addr_h8[j+6], v);
        addr_h8[j+7] =
            __builtin_shave_cmu_clamp0_f16_rr_half8(addr_h8[j+7], v);
    }
}
```

- Finally, lines 66-68 return the `localOutput` back to the DDR.

Part III

Epilogue

Chapter 6

Evaluation of the Implementation

This chapter evaluates the CNN implementation, conducting a range of measurements with respect to different parameters. The explanation of the results will try to give a deeper insight on the limits posed by the hardware and the implementation itself.

6.1 Specific CNNs used

Several results of the evaluation are drawn from the execution of two specific convolutional neural networks. These networks are described below.

6.1.1 CIFAR10 Quick CNN

The first network used is the CIFAR10 Quick, that is presented in figure 6.1. In particular, this architecture is used for terrain classification from satellite images. More precisely, the NAIP dataset [10] is used, which consists of image patches each of size 28×28 and covering 6 land classes: barren land, trees, grassland, roads, buildings and water bodies.

Details about the architecture are imposed on this figure. CIFAR10 CNN was originally used for the classification of 10 categories, hence the number 10 in its name. In order to adapt this CNN for the NAIP dataset, that outputs 6 classes, the "num output" variable of the "ip2" node needs to be set at 6. Except that, the network presented in figure 6.1 and the original CIFAR10 Quick are identical.

6.1.2 nViso CNN

The second network is provided by the nViso company that specializes on emotion measurement technology. Due to licensing constraints, no further information has been made available, such as a dataset or other technical information. This CNN is used for the classification of emotions from facial expressions, that belong to the following categories: anger, disgust, fear, happiness, neutral, sadness and surprise. Figure 6.1 describes its architecture.

Figure 6.1: ***Left***: *CIFAR10 Quick CNN for terrain classification.* ***Right***: *nViso CNN for emotion classification from facial expressions.*

6.2 Measurements

This section will present several plots regarding the performance and the power consumption of the CNN implementation. Both metrics will be evaluated with respect to three different parameters, namely *input size*, *number of SHAVEs* and *frequency* of the Myriad2 SoC.

6.2.1 Different input sizes

For these measurements, the CIFAR10 Quick network is used. Its choice is arbitrary and other networks could be used instead.

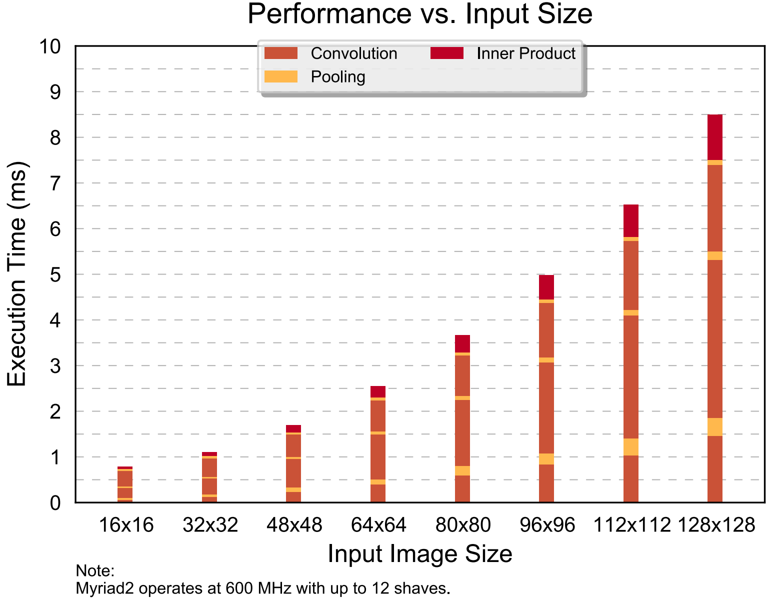


Figure 6.2: *Execution time of CIFAR10 Quick CNN for various input images.*

Notice, on figure 6.2, that the color coding of the bars is based on the color of the computational layers shown in figure 6.1. From this plot it is seen that convolution is the most time consuming operation, which justifies the effort to optimize it as much as possible. Also, observe that the height of the bars grows quadratically with respect to the input. In conclusion, convolutional layers are the most time demanding operations, followed by the fully connected layers and the pooling layers.

Figure 6.3 show the energy consumption needed for the execution of the same CNN. It is natural to expect that longer execution times require more energy, which is confirmed by the gathered data. Much like in figure 6.2, the energy consumption grows quadratically. This is justified by the fact that the average power consumption remains constant during execution and the execution time exhibits quadratic behavior. As a result, the product of power and time (which is energy) also behaves quadratically.

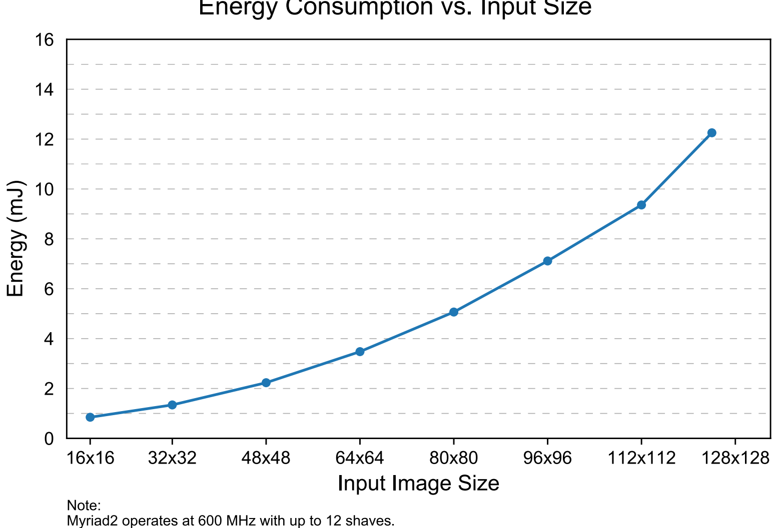


Figure 6.3: *Energy consumption of CIFAR10 Quick CNN for various input images.*

6.2.2 Different number of SHAVEs

This subsection will examine the time and energy requirements of the building blocks used by the CNNs presented in figure 6.1. These two metrics will be evaluated with respect to the number of SHAVEs executing each building block. It is a very important set of measurements, since they will reveal the inherent properties and the performance boundaries of the parallelization schema used by the computational nodes of the implementation.

Convolution

A typical example of convolution, that appear on many CNNs, is examined. In particular, the convolution is performed on an input volume of 64×64 spacial dimensions and 32 channels deep. The output volume contains 128 feature maps and its size is $64 \times 64 \times 128$.

Notice, from figure 6.4, the large times required for this computation. This, once more, confirms the observation mentioned on figure 6.2 and is justified by the fact that convolution involved in CNNs is a 3D operation, in contrast with regular convolution used for grayscale images (which is a 2D operation).

The absolute times presented in figure 6.4 have their merit, however scalability is of great interest. Figure 6.5 gives a different look at the data presented in figure 6.4. The results are extremely satisfactory, since the implementation is very close to the ideal case of linear scalability. The reason for such success is mostly because of the nature of the operation, which reuses a lot of the same data. As a result, convolution is a compute bounded operation, which means that the multiple SHAVE processors do not need to compete intensively for - the shared resource that is - DDR.

Another interesting result is the energy consumption of this operation. The measurements are plotted in figure 6.6.

Convolution: Performance vs. SHAVEs

Execution Time (ms)

Number of SHAVEs

Note:
Myriad2 operates at 600 MHz.
Convolution kernel is 5x5 (spatially), padding is 2 and stride is 1.
Convolution is performed on an 32x64x64 (DxHxW) input volume with 128 output feature maps.

Figure 6.4: *Time execution of convolution with respect to the number of SHAVEs.*

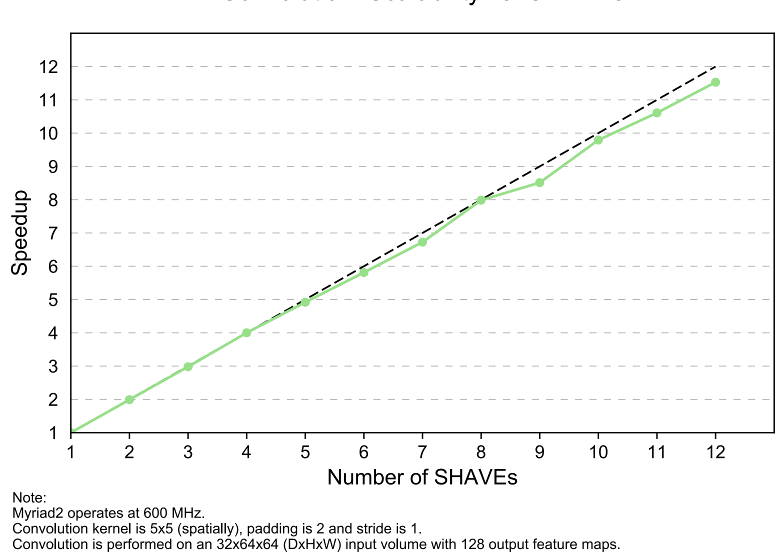


Figure 6.5: *Scalability convolution with respect to the number of SHAVEs*

Surprisingly, the energy consumption drops when increasing the number of SHAVEs. Even though usage of more SHAVEs increases the average power consumption, the shorter time of execution required for the operation results in less energy consumed overall. Notice, that this behavior is mostly because of the very good scalability of the operation.

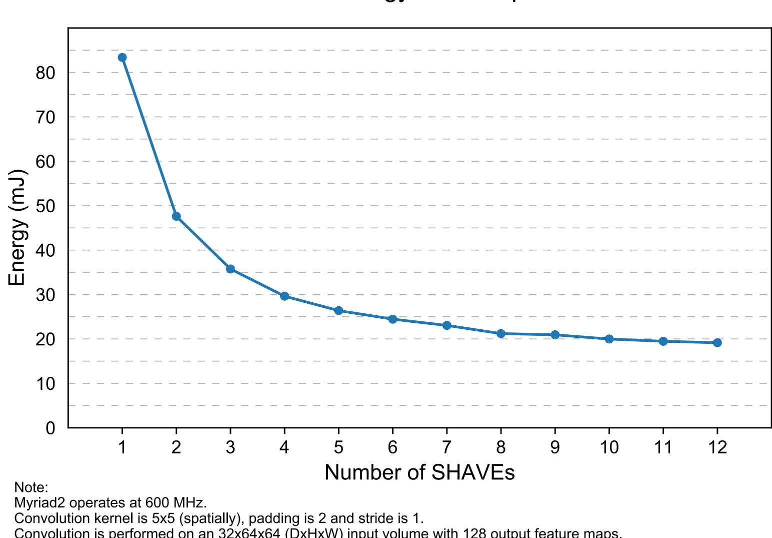


Figure 6.6: *Energy consumption of convolution with respect to the number of SHAVEs.*

Pooling

A typical example of pooling is examined, where the operation is performed on an input volume of 64×64 spacial dimensions and 128 channels deep.

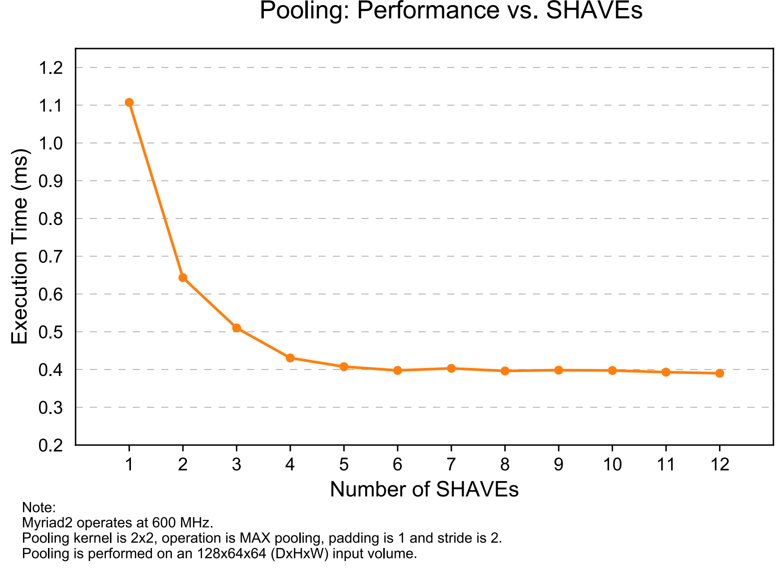


Figure 6.7: *Time execution of pooling with respect to the number of SHAVEs.*

Notice the absolute times of this operation. Even though the input volume is similar to the input volume of convolution shown in figure 6.4, their execution times differ by orders of magnitude. This is another indication of the large computational requirements of convolution, which makes pooling seem a trivial operation.

The scalability of pooling, shown in figure 6.8, is far from great. This kind of behavior is due to the I/O bounded nature of this operation. If fact, in the particular operation

Note:
Myriad2 operates at 600 MHz.
Pooling kernel is 2x2, operation is MAX pooling, padding is 1 and stride is 2.
Pooling is performed on an 128x64x64 (DxHxW) input volume.

Figure 6.8: *Scalability pooling with respect to the number of SHAVEs.*

shown in this figure, every element of the input volume is only used for the generation of a single element on the output volume. Also, the computation involved for generating each output element on the output volume is very simple. This leads to a very high and frequent demand for data residing in DDR, which ultimately (the DDR) reaches its maximum bandwidth.

Another interesting result is the energy consumption of this operation. Figure 6.9 indicates that increasing the number of SHAVEs, which perform the pooling operation in parallel, leads to increase in energy consumption. This is an expected result, since increasing the number of SHAVEs also increases the average power consumption. Higher power combined with inability for lowering the execution time when using a lot of SHAVEs, naturally leads to increase in energy consumption.

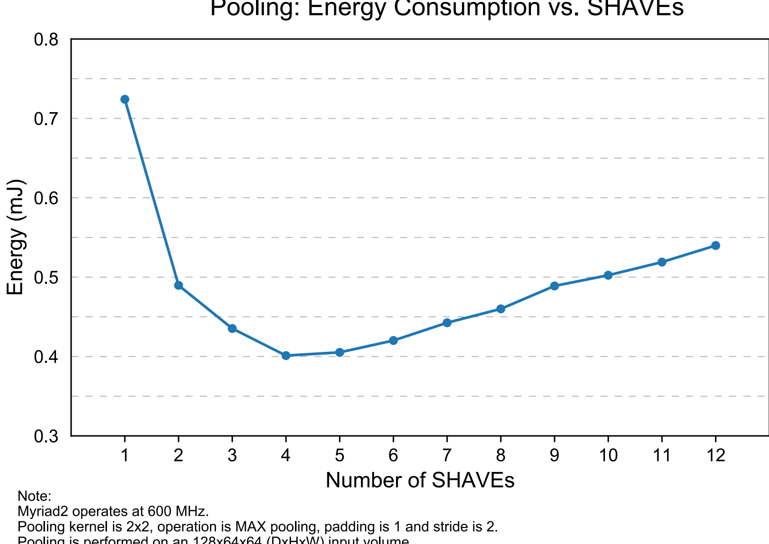


Figure 6.9: *Energy consumption of pooling with respect to the number of SHAVEs.*

Fully Connected

Finally, the same measurements are performed for a fully connected computational node, that involves the matrix-vector multiplication $[1000 \times 10240] \times [10240 \times 1]$.

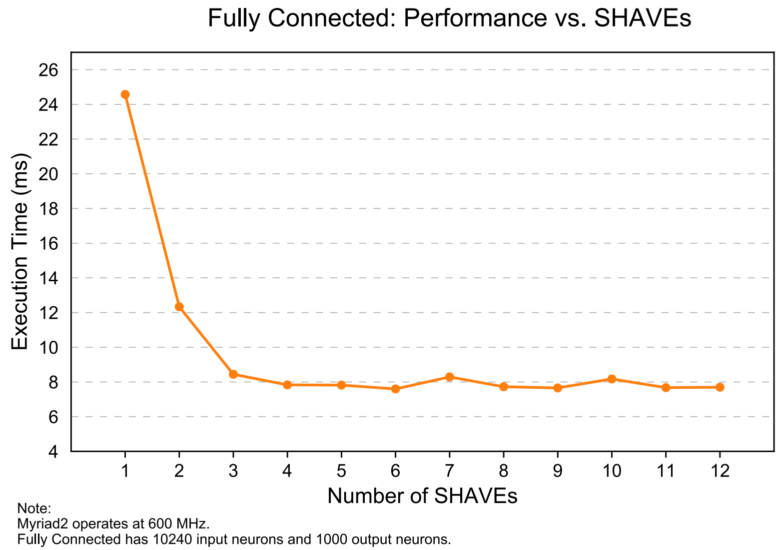


Figure 6.10: *Time execution of fully connected with respect to the number of SHAVEs.*

The behavior exhibited on figures 6.10, 6.10 and 6.12 is similar to the behavior of pooling. This is because both operations are I/O bounded. The operation involved in a fully connected node is matrix-vector multiplication, in which every element of the matrix participates in the computation of only one element of the output vector. Also, the computation in which it is involved is inner product, which Myriad2 can perform very efficiently in just one clock cycle per eight elements (with SIMD instructions). As a result, the time needed for data transfers is far larger that the time required by the

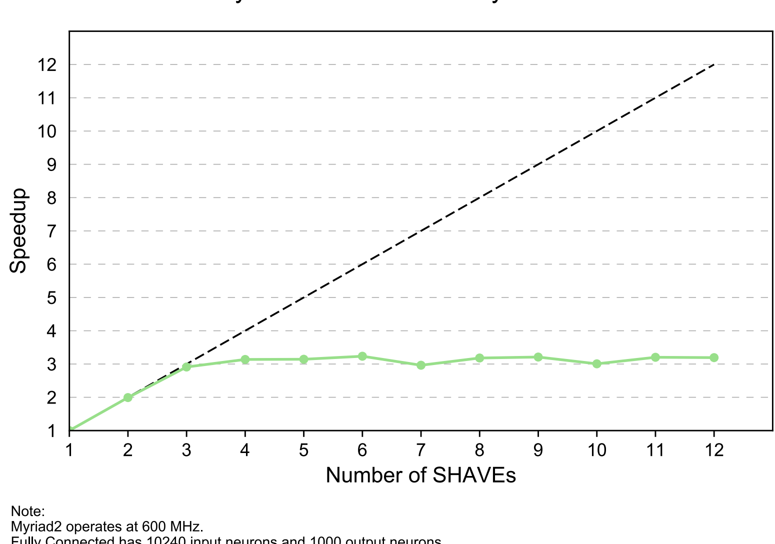


Figure 6.11: *Scalability fully connected with respect to the number of SHAVEs.*

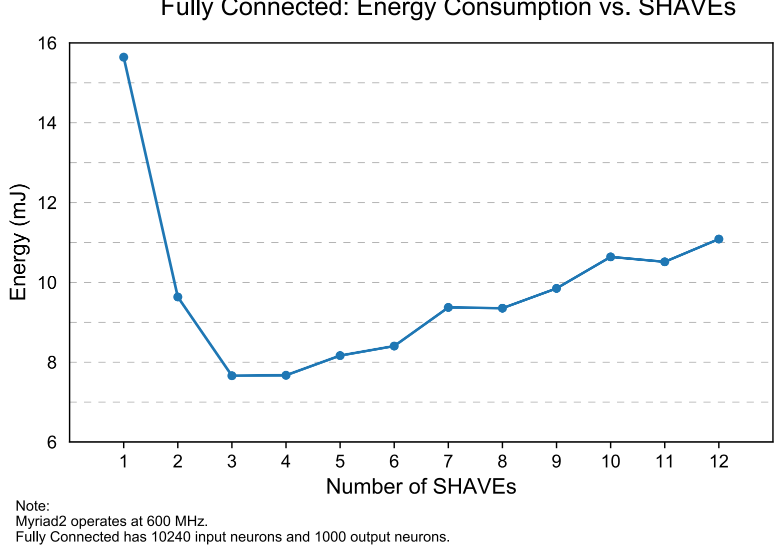


Figure 6.12: *Energy consumption of fully connected with respect to the number of SHAVEs.*

computation, leading to the presented plots.

In conclusion, observing the scalability plots of this subsection, it is obvious that it is not a good practice to execute an entire CNN dedicating all SHAVEs at each operation. This is because some of the operations cannot benefit from more processing power, due to their I/O bound limitations. As a result, dedicating all SHAVEs at each operation is only going to increase the energy consumption without any performance gain, which is a serious loss (especially) for an embedded platform. This explains the note "with up to 12 SHAVEs" on the figures 6.2 and 6.3, that indicates that performance is the first priority, but energy consumption is silently considered.

6.2.3 Different frequency

This subsection will examine the time and energy consumption of the CNN implementation, with respect to the SoC clock frequency. The CIFAR10 Quick architecture is selected as the test CNN, because it contains a balanced amount of the different operations (convolution, pooling, fully connected).

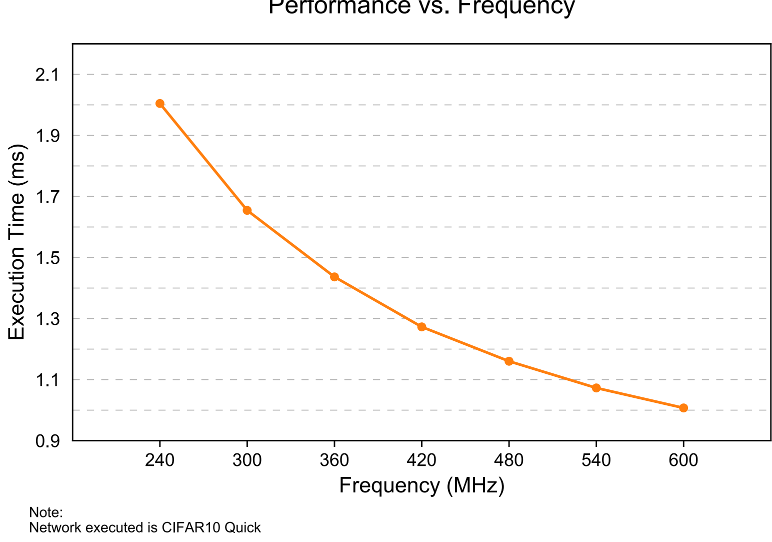


Figure 6.13: *Time execution of the CIFAR10 Quick CNN with respect to the SoC frequency.*

It is natural to expect faster execution with higher frequency, since this is predicted by the basic performance equation: CPU Time $=$ IC $\times$ CPI $\times$ CT, where:

- IC is the Instructions Count of the given program.
- CPI is the Clocks Per Instruction of the given processor.
- CT is the Clock Time of the given processor.

Notice that the different parameters above are not independent, since there are subtle architectural details that correlate them. For example, increasing CT will probably increase the CPI as well. However, in the general case this formula is true. As a result, increasing the SoC clock frequency is equivalent to decreasing the CT (because CT $= 1/$Frequency), which explains the behavior of the plot presented in figure 6.13.

Figure 6.14 shows the energy consumption of the same CNN. Surprisingly, the energy consumption increases as the frequency is lowered, since the CNN needs to run longer in order to complete the same amount of work, i.e. classify an input image. Again, the power consumption drops with the decrease of frequency, but the increased time of execution leads to increase in energy consumption overall.

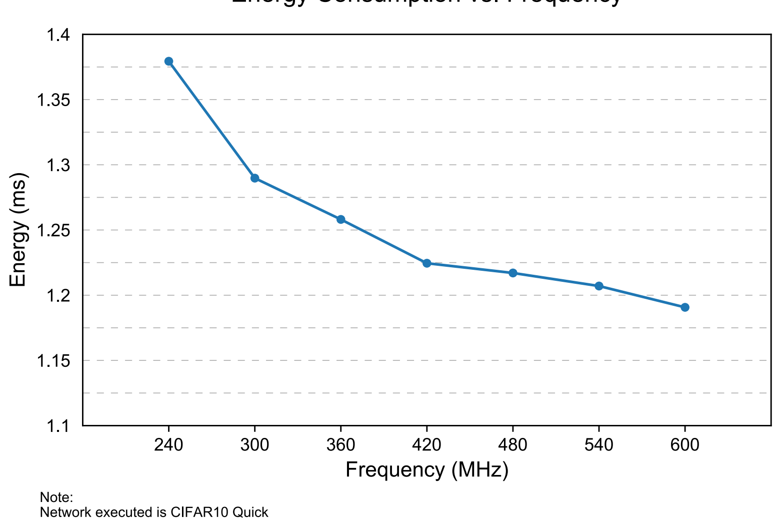


Figure 6.14: *Energy consumption of the CIFAR10 Quick CNN with respect to the SoC frequency.*

6.3 Comparison with other implementations

This section will compare the execution time of the two CNNs presented at the beginning of the chapter, using different devices and/or implementations. The current CNN implementation will be compared with Fathom NCS, which is essentially the same piece of hardware, accompanied by a closed source software package capable of executing neural networks on it. Also, caffe will be run on a x86 Intel CPU in order to get a filling of the execution times between multiple hardware devices. Figures 6.15 and 6.16 present the comparison.

First of all, notice that the current CNN implementation is executed on Myriad2, more precisely model MA2150. The Fathom NCS uses a slight improvement of this chip, which is MA2450, whose main advantage is larger and faster DDR RAM (512MB instead of 128MB, 933MHz instead of 533MHz). This makes the comparison more fair, since the hardware is almost identical. On the other hand, Caffe is executed on Intel Xeon E5-2650 v2 CPU, that comes with 20MB of cache hierarchies, has 8 cores, 16 threads and 2.60 GHz of base processor frequency. In general, this is an extremely powerful processor that is intended for server applications.

Caffe does not perform far better than the current CNN implementation. If the lack in performance of the current CNN implementation (in comparison with the performance on Caffe on the Intel Xeon processor) is not of critical importance, then this implementation should be used, because of the energy consumption. Myriad2 is designed for low power embedded applications, while the Intel Xeon processor is designed high performance server applications. This means that the energy consumption is not fair at all, although for the specific CNNs Myriad2 wins the energy test.

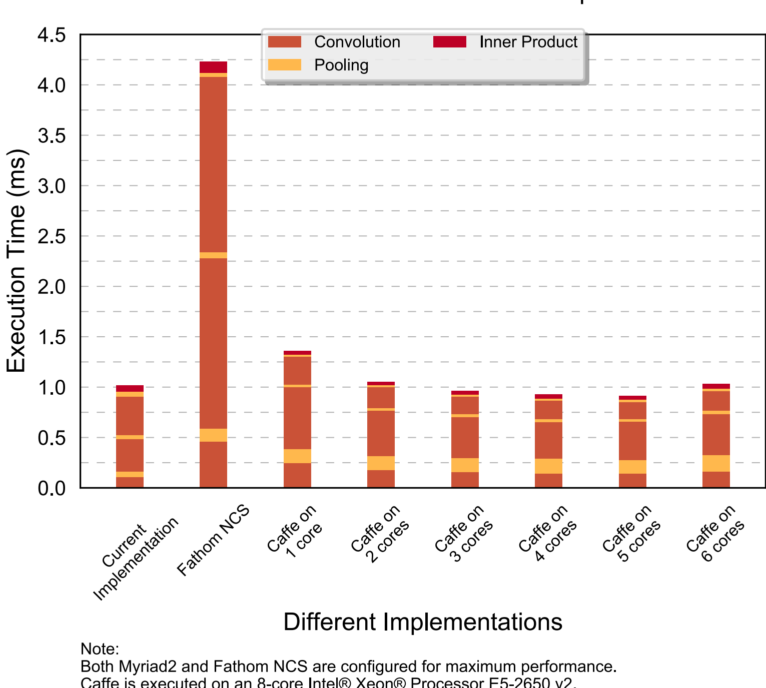


Figure 6.15: *Time execution of the CIFAR10 Quick CNN on different devices and/or implementations.*

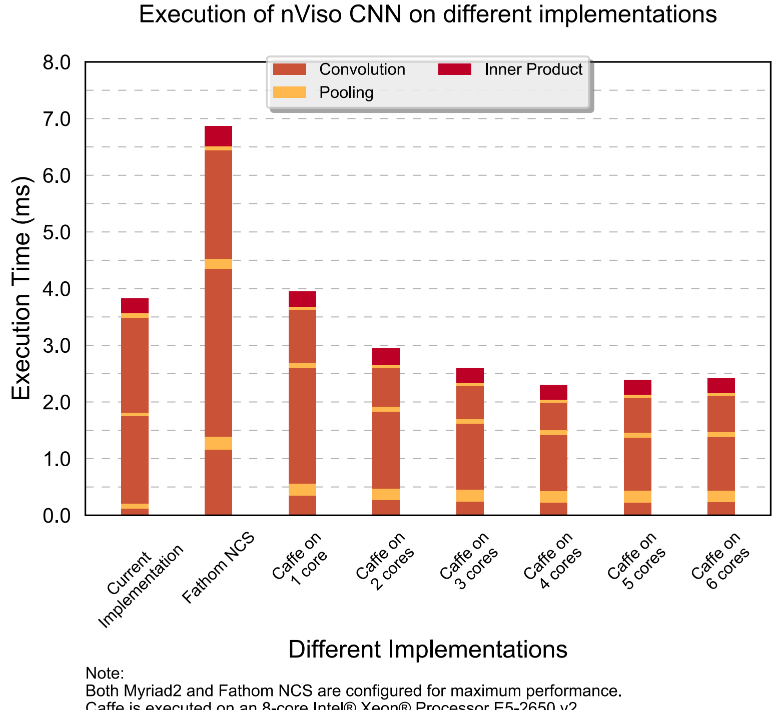


Figure 6.16: *Time execution of the nViso CNN on different devices and/or implementations.*

6.4 Accuracy

This section will take a look at the accuracy of the current CNN implementation. Accuracy is an important part of the evaluation, since the training of the CNN is done in 32-bit floating point arithmetic, while Myriad2 executes the forward step (i.e. the testing phase) operating in 16-bit floating point arithmetic. Thus, a question arises whether this decrease in precision affects accuracy and by how much. It is pointed out that the term accuracy is used to describe the magnitude of the arithmetic error in the output of the forward-step execution between Caffe and Myriad2. The two example networks presented in the beginning of this chapter are used, executing the forward step of some sample images. The results are presented in tabular form below. Keep in mind that Caffe results are annotated with “C”, while Myriad2 results are annotated with “M”. Also, the relative error of the Myriad2 result with respect to the Caffe result is annotated with “RE”.

The conclusion drawn from these tables is that the loss in precision in floating point arithmetic does not have a devastating effect on the output. More importantly, the results show that total order of the categories is preserved. In other words, there are no two categories in which the Caffe output is in different order compared to the Myriad2 output. The same inequalities that are true for the Caffe output are also true for the Myriad2 output. This is a very desirable property, since this way the prediction sequence (i.e. the sequence of probabilities assigned for each image) a made by Caffe is identical to the predicition sequence made by Myriad2.

Input \ Category		Building	Barren land	Trees	Grassland	Road	Water
	C	9.737597	0.028202	-12.598548	-8.100735	3.895604	-5.321511
	M	9.734375	0.028809	-12.59375	-8.09375	3.894531	-5.316406
	RE	0.033%	2.154%	0.038%	0.086%	0.028%	0.096%
	C	-3.427331	10.710045	-5.335921	2.318379	-3.615897	-8.866998
	M	-3.425781	10.710938	-5.335938	2.318359	-3.617188	-8.859375
	RE	0.045%	0.008%	0.000%	0.001%	0.036%	0.086%
	C	-7.987020	3.049417	14.191380	4.842995	0.034941	-3.474523
	M	-7.984375	3.052734	14.179688	4.839844	0.041656	-3.46875
	RE	0.033%	0.109%	0.082%	0.065%	19.218%	0.166%
	C	-3.890330	1.158329	-1.491262	8.688931	-3.650822	-5.344398
	M	-3.886719	1.158203	-1.489258	8.679688	-3.648438	-5.339844
	RE	0.093%	0.011%	0.134%	0.106%	0.065%	0.085%
	C	5.241163	0.519996	-8.747707	-4.886876	11.179963	-5.236880
	M	5.246094	0.518555	-8.742188	-4.886719	11.164062	-5.238281
	RE	0.094%	0.277%	0.063%	0.003%	0.142%	0.027%
	C	-8.714371	-0.579488	-0.774372	-2.972946	2.857591	18.699329
	M	-8.703125	-0.568359	-0.77832	-2.990234	2.875	18.6875
	RE	0.129%	1.920%	0.510%	0.582%	0.609%	0.063%

Table 6.1: *Comparison of accuracy of CIFAR10 Quick CNN layer "ip2" between Caffe and Myriad2 implementation.*

Input / Category		Anger	Disgust	Fear	Happiness	Neutral	Sadness	Surprise
	C	-17.850915	-2.357175	-6.676564	52.664138	-9.152770	-5.112073	-9.091076
	M	-17.843750	-2.351562	-6.683594	52.625000	-9.156250	-5.121094	-9.093750
	RE	0.040%	0.238%	0.105%	0.074%	0.038%	0.176%	0.029%
	C	50.073471	4.734987	-17.832698	-26.339084	-4.803456	4.931145	-13.976916
	M	50.09375	4.710938	-17.828125	-26.359375	-4.796875	4.941406	-13.976562
	RE	0.040%	0.508%	0.026%	0.077%	0.137%	0.208%	0.003%
	C	-20.158855	-33.927001	44.842117	-0.139439	-21.007415	-15.929889	64.673278
	M	-20.15625	-33.90625	44.84375	-0.145996	-21.015625	-15.921875	64.6875
	RE	0.013%	0.061%	0.004%	4.702%	0.039%	0.050%	0.022%
	C	28.647573	31.595224	-13.235262	-11.848876	-21.539316	-0.38034	-17.441757
	M	28.640625	31.609375	-13.25	-11.851562	-21.546875	-0.372559	-17.453125
	RE	0.024%	0.045%	0.111%	0.023%	0.035%	2.046%	0.065%
	C	-20.353111	-31.784439	48.527591	-2.38341	-20.314197	-17.113338	61.004989
	M	-20.34375	-31.75	48.5	-2.388672	-20.296875	-17.109375	60.96875
	RE	0.046%	0.108%	0.057%	0.221%	0.085%	0.023%	0.059%

Table 6.2: *Comparison of accuracy of nViso CNN layer "hidden_out" between Caffe and Myriad2 implementation.*

Chapter 7

Summary

7.1 Conclusion

This thesis, tried to develop an efficient CNN engine for the Myriad2 embedded multiprocessor. Instead of specializing explicitly on this specific processor, an effort was made, resulting in a more general methodology. The end result is a collection of steps that can be applied to any relevant embedded platform, helping the increase of performance and decrease of energy consumption.

One of the major problems that needed to be solved early on was the efficient management of given CMX memory. It was immediately obvious that CMX can offer a very large performance boost if exploited correctly. The proposed approach is depicted in figure 7.1. It results in about 50% increase of CMX space that is available for data.

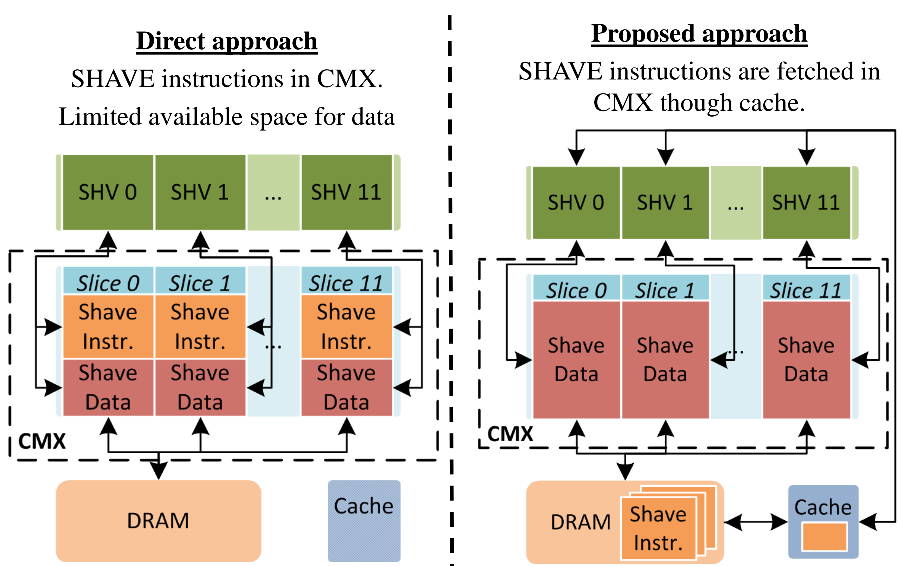


Figure 7.1: *Optimization 1: Increasing CMX available memory space for data.*

Due to extremely intense movement of data back and forth, from DDR to CMX and vice versa, a non wasteful access strategy was important. The key idea is to try to exploit the data as much as possible, before discarding them. This leads to reformulation of nested iterations, as depicted in figure 7.2. The proposed approach resulted in about 30%

reduction in execution time.

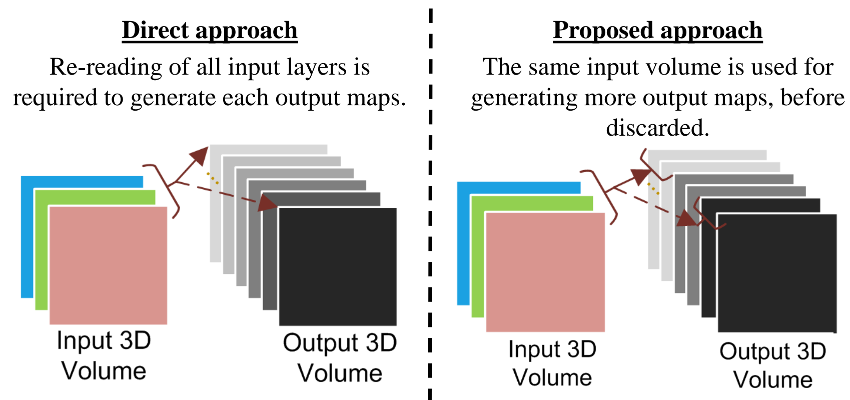


Figure 7.2: *Optimization 2: Reducing the number of required DMA transfers.*

In order to decrease the execution time even further, the DMA engine transfer time was overlapped with computation, making the transfer cost minimal. This requires double buffering and is especially true for compute bound operations, such as convolution. Notice that such a technique is not an extreme innovation, however it is a very important one. For instance, recent nVidia graphics processing units come with dual DMA engines, to assist the double buffering in the hardware level. The proposed approach is depicted in figure 7.3 and gives a performance boost of 20% for compute bound operations.

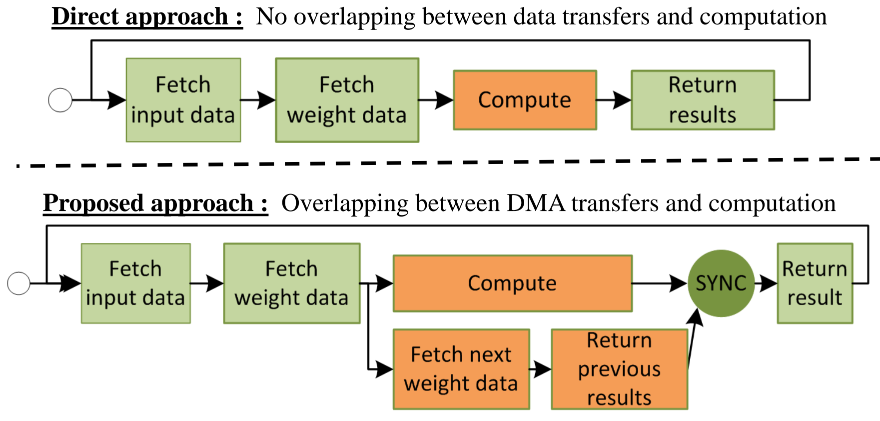


Figure 7.3: *Optimization 3: Reducing DMA transfers overhead.*

Finally, due to the hardware design of the Myriad2 processor, branches have a relatively large penalty, since there is no branch prediction mechanism, only a delay slot one. For this reason, measurements showed that short for-loops and many function calls have a measurable cost. Towards this direction, an effort was made to decrease function calls. Figure 7.4 explains how this was achieved for convolution. The reduction in the number of calls resulted in a 6% reduction in execution time.

In conclusion, many engineers around the world have developed several CNN frameworks targeting embedded platforms. Specifically, the ARM platform is the most popular

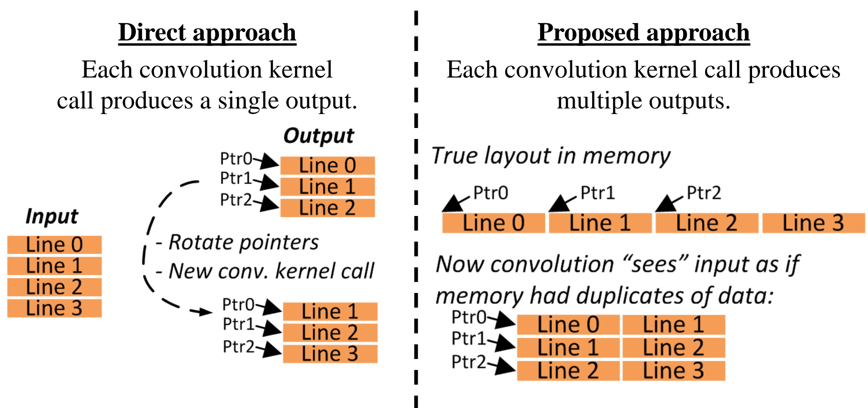


Figure 7.4: *Optimization 4: Reducing the number of convolution kernel calls.*

one. Our belief is that the techniques and methodology developed and tested in this thesis, will most definitely have a beneficial effect for these frameworks.

7.2 Future work

Throughout the course of the implementation of the CNN engine, several problems were faced. The proposed solutions of these problems are not always generic and the reason for this is the nature of these problems themselves. For example, the improvement in terms of performance and energy consumption of convolution requires the development of a wide range of different approaches, depending on the ratio of the kernel sizes over the input data, as well as the shape of them. This leads to vast design space, that makes the exhaustive evaluation of these decisions intractable. As a result, it is essential to consider advanced techniques for design space exploration, in order to haystack only the reasonable and worthy options for the problem at hand. For a future extension, the following seem to be of the greatest importance:

- *Extend the CNN engine, in terms of functionality*. Taking the Caffe framework as a complete CNN implementation, the first goal will be to become functionally equivalent with it. At the same time, recent state-of-the-art CNN architectures invented new CNN layers. This makes the effort of closing the functional gap between the current implementation and the latest developments in CNN architectures a daunting task.

- *Develop a powerful compiler for the graph model of the CNN*. This thesis focused mainly on optimizations that are closer to the hardware. However, another essential area of extension is transforming the graph model of a given CNN, into an intermediate format that is suitable for optimizations. Ideally, this format would be able to apply platform agnostic optimizations, such as merging of layer nodes in a CNN, as well as platform specific ones. For example, the choice of a particular implementation of a specific layer (e.g. convolutional) with respect to the parameters given to this layer, depends on the underlying hardware.

- *Consider the implementation of a sophisticated runtime system*. The fact that multiprocessors is the only solution to increasing computational power these days, makes runtime systems a valuable asset. In the context of a CNN engine, the main goal of such runtime system would be to make online decisions regarding the scheduling of computational layers. As already mentioned in previous chapters, CNNs consist of a mixture of both compute, as well as I/O bound operations. In general terms, it is best to parallelize these two types of operations, abiding by the data dependencies they have.

- *Autotuning*. It is common, for most computational layers of a CNN to have several tunable parameters, such as:

  1. The chunk size for transferring the data between CMX and DDR.
  2. The alignment boundary of blocks of data in DDR and CMX.
  3. etc.

Implementing more and more computational layers, makes the tuning of such parameters very difficult. Consequently, an automatic way of finding effective values for them is important. Most likely, this mechanism will follow several heuristics, that may differ from layer to layer and depend on hardware subtleties, because - once more - the dimensionality of the search space for these parameters may be extremely large.

- Finally, let as briefly discuss questions that are part of the design space:

  1. Is it better to use memcpy or the DMA engine to transfer data from DDR to CMX and vice versa? When memcpy is a much better choice?
  2. What is the best sharing scheme among the data already in the CMX, considering that each SHAVE is subjected to arbitration when trying to access a different CMX slice? When load stalls from sharing CMX data cost more than not sharing at all?
  3. When is double buffering a good solution, considering that decreases the effective available space of CMX? Is it better to process larger chunks of data at once and not overlap computation with data transfer?

  All the questions are reasonable, but the answers are not clear. In fact, the answers to these questions begin with the phrase ”It depends”, making the design space exploration essential.

Bibliography

[1] Sebastian Raschka. *Python machine learning : unlock deeper insights into machine learning with this vital guide to cutting-edge predictive analytics*. Packt Publishing, Birmingham, UK, 2015.

[2] Ian Goodfellow, Yoshua Bengio και Aaron Courville. *Deep Learning*. MIT Press, 2016. http://www.deeplearningbook.org.

[3] Andrej Karpathy. *Stanford University CS231n: Convolutional Neural Networks for Visual Recognition*.

[4] Gaurav Raina. *Deep Convolutional Network evaluation on the Intel Xeon Phi: Where Subword Parallelism meets Many-Core*. Μεταπτυχιακή διπλωματική εργασία, Eindhoven University of Technology, 2016.

[5] *Convolutional Neural Networks (CNNs): An Illustrated Explanation*. http://xrds.acm.org/blog/2016/06/convolutional-neural-networks-cnns-illustrated-explanation/. Accessed: 05-05-2017.

[6] Y. Jia, E. Shelhamer, J. Donahue, S. Karayev, J. Long, R. Girshick, S. Guadarrama και T. Darrell. *Caffe: Convolutional Architecture for Fast Feature Embedding*. *ArXiv e-prints*, 2014.

[7] *Myriad 2 MA2x5x Vision Processor*. https://uploads.movidius.com/1463156689-2016-04-29_VPU_ProductBrief.pdf. Accessed: 05-05-2017.

[8] Movidius Ltd. *Movidius Myriad2 Development Kit: Programmer's Guide (under non-disclosure license)*.

[9] Movidius Ltd. *Movidius Myriad2 MA215x Databook (under non-disclosure license)*.

[10] S. Basu, S. Ganguly, S. Mukhopadhyay, R. DiBiano, M. Karki και R. Nemani. *DeepSat - A Learning framework for Satellite Imagery*. *ArXiv e-prints*, 2015.

[11] Brian Dolhansky. *Artificial Neural Networks: Linear Multiclass Classification*. http://briandolhansky.com/blog/2013/9/23/artificial-neural-nets-linear-multiclass-part-3. Accessed: 05-05-2017.

[12] Tariq Rashid. *Make your own neural network : a gentle journey through the mathematics of neural networks, and making your own using the Python computer language*. CreateSpace Independent Publishing, United States, 2016.

[13] V. Dumoulin και F. Visin. *A guide to convolution arithmetic for deep learning*. *ArXiv e-prints*, 2016.

[14] John Levine. *Linkers and loaders*. Morgan Kaufmann, San Francisco, Calif. u.a, 2000.

[15] Milan Stevanovic. *Advanced C and C++ compiling*. Apress,Distributed to the Book trade worldwide by Springer, Berkeley, CA New York, NY, 2014.

Abbreviations

IEEE	Institute of Electrical and Electronics Engineers
CIFAR	Canadian Institute for Advanced Research
BVLC	Berkeley Vision and Learning Center
MNIST	Mixed National Institute of Standards and Technology
GNU	Gnu's Not Unix
API	Application Programming Interface
DAG	Directed Acyclic Graph
ANN	Artificial Neural Network
CNN	Convolutional Neural Network
MPL	Multi-Layer Perceptron
RGB	Red Greed Blue
FC	Fully Connected
SoC	System-on-Chip
CPU	Central Processing Unit
VPU	Vision Processing Unit
SHAVE	Streaming Hybrid Architecture Vector Engine
DRAM	Dynamic Random Access Memory
SRAM	Static Random Access Memory
CMX	Connection Matrix
DDR	Double Data Rate
DMA	Direct Memory Access
LSU	Load-Store Unit
CSS	CPU sub-system
MSS	Media sub-system
UPA	Microprocessor Array
MDK	Myriad2 Development Kit
LOS	Leon OS
LRT	Leon RT
CPR	Clock-Power-Reset