Implementation of Computational Intensive Convolutional Neural Networks on Embedded Devices with Limited Resources

Υλοποίηση Έντονων Υπολογιστικά Συνελικτικών Νευρωνικών Δικτύων σε Ενσωματωμένες Αρχιτεκτονικές με Περιορισμένους Πόρους

Μπαρτσώκας Αναστάσιος

Επιβλέπων: Δημήτριος Σούντρης

Αν. Καθηγητής Ε.Μ.Π

Αθήνα Ιούνιος 2018

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

Implementation of Computational Intensive Convolutional Neural Networks on Embedded Devices with Limited Resources

Υλοποίηση Έντονων Υπολογιστικά Συνελικτικών Νευρωνικών Δικτύων σε Ενσωματωμένες Αρχιτεκτονικές με Περιορισμένους Πόρους

Μπαρτσώκας Αναστάσιος

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

Εγκρίθηκε από την τριμελή εξεταστική επιτροπή τον Ιούνιο του 2018.

(Υπογραφή)	*(Υπογραφή)*	*(Υπογραφή)*
.....	.....	.....
Δημήτριος Σούντρης	Κιαμάλ Πεκμεστζή	Κωνσταντίνος Σιώζιος
Αν. Καθηγητής Ε.Μ.Π	Καθηγητής Ε.Μ.Π	Επ. Καθηγητής Α.Π.Θ

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

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. Μπαρτσώκας

Ιούνιος 2018

Περίληψη

Τα τελευταία χρόνια γίνονται συνεχώς έρευνες γύρω από την ανάπτυξη των **Τεχνητών Νευρωνικών Δικτύων** (ΤΝΔ). Τα ΤΝΔ είναι εμπνευσμένα από βιολογικούς οργανισμούς κι έχουνε ξεπεράσει κατα πολύ σε απόδοση τις πρηγούμενες μορφές τεχνητής νοημοσύνης. Ένα ΤΝΔ αποτελείται από απλούς υπολογιστικούς κόμβους διασυνδεδμέμους μεταξύ τους, οι οποίοι είναι εκπαιδευμένοι να αντιδρούνε σε ερεθίσματα. Τα ΤΝΔ που παίρνουνε ως είσοδο εικόνες ονομάζονται **Συνελικτικά Νευρωνικά Δίκτυα** (ΣΝΔ) και χρησιμοποιούνται για την επίλυση προβλημάτων όρασης υπολογιστών, όπως είναι η αναγωνώριση αντικειμένων, η εύρεση θέσης των αντικειμένων κ.α.
Τα τελευταία χρόνια τα ΤΝΔ έχουνε εισαχθεί στον κόσμο των **ενσωματωμένων συσκευών**, καθώς είναι ιδιαίτερα σημαντικό τόσο για τους δημιουργούς λογισμικού όσο και εντυπωσιακό για τους χρήστες να έχουνε τη δυνατότητα να αναγνωρίζουνε αντικείμενα με τις συσκευές τους π.χ. κινητά τηλέφωνα.
Στόχος της παρούσας διπλωματικής είναι η εξέλιξη ενός συστήματος εκτέλεσης ΣΝΔ στο ενσωματωμένο σύστημα **Myriad2**. Η μηχανή υποστηρίζει βαθιά δίκτυα με μεγάλο πλήθος ομότιμων κόμβων, που έχουνε εκπαιδευτεί σε τεράστια σύνολα δεδομένων. Αυτό σημαίνει ότι το σύνολο δεδομένων των δικτύων είναι πολύ μεγάλο για τη μνήμη τη ενσωματωμένης συσκευής, η οποία όμως έχει το πλεονέκτημα **χαμηλής κατανάλωσης ενέργειας** ανα μονάδα υπολογισμού. Η αρχιτεκτονική της Myriad 2 συνίσταται απο 12 VLIW επεξεργαστές, χτισμένους γύρω από μια μικρή και γρήγορη μνήμη και άλλους που έχουνε συντονιστικό ρόλο. Από τη φύση τους τα ΣΝΔ απαιτούνε διαχείριση τεράστιου όγκου δεδομένων που σημαίνει και μεταφορά δεδομένων μεταξύ των μνημών. Τα βαθιά ΣΝΔ περιέχουνε συνελικτικές στρώσεις με αρκετές παραμέτρους, οι οποίες έχουνε ως σκοπό να μειώνουνε τις απαιτήσεις σε μνήμη του δικτύου. Αυτές υλοποιήθηκανε με αποδοτικό τρόπο, ακόμα και με τη χρήση συμβολικής γλώσσας, εμβαθύνοντας σε τεχνικές εκμετάλλευσης της αρχιτεκτονικής. Επιπλέον αναπτύχθηκε **νέος τρόπος υλοποίησης των συνελικτικών στρώσεων**, προκειμένου να μειώσει τον χρόνο εκτέλεσης στρώσεων με συγκεκριμένες προδιαγραφές, αλλά και τη συνολική κατανάλωση ενέργειας. Τέλος, το γεγονός ότι το σύστημα είναι πολυεπεξεργαστικό, αυξάνει την πολυπλοκότητα ακόμα περισσότερο.
Τα παραπάνω επεκτείνονται σε κάθε ενσωματωμένο επεξεργαστή, ο οποίος υποστηρίζει εκτέλεση νευρωνικών δικτύων καθώς οι μεθοδολογίες που χρησιμοποιήθηκανε εφαρμόζωνται και έξω από τη Myriad2.
Τέλος αναπτύχθηκε μια εφαρμογή πραγματικού χρόνου, η οποία κάνει ταξινόμηση σύμφωνα με το ImageNet.

Λέξεις Κλεδιά

Μηχανική μάθηση, Συνελικτικά νευρωνικά δίκτυα, Βαθιά νευρωνικά δίκτυα, Ενσωματωμένα συστήματα, Πολυεπεξεργαστικά συστήματα, Myriad 2, Νέος τρόπος συνέλιξης, Εφαρμογή Πραγματικού Χρόνου.

Abstract

During the recent years, science community is focusing on the field of **Artificial Neural Networks** (ANN). ANN function by following a similar way with biological computational models and they are more efficient than older classification models. An ANN contains simple computational nodes linked between them, which are trained to activate when they have the appropriate signal. ANNs which accept images as input on their nodes, are named **Convolutional Neural Networks** (CNNs) and they are known for solving computer vision problems such as image classification, object detection etc.
During the last years, it is surprising that ANNs tend to enter more and more into **embedded world** as it is so important to recognize objects on images, both for the application developers and for the users of devices. A typical example of an embedded device, which uses image classification is mobile phones.
The aim of this diploma thesis is to extend a CNN engine to test **Deep** Neural Networs in **Myriad2**. The CNN engine is able to deploy Neural Networks which have been trained in large datasets like Imagenet. This means that the memory demands of the application are enormous. Placing all these data inside memories of the device remains a challenge. The main advantage of Myriad2 is the **little memory consumption** per computation unit. In order to imlement the new version of the CNN Engine importance was given to the Myriad2 architecture. More specifically, this device comprises 12 VLIW processors built around a small yet fast memory block and 2 RISC processors with an arbitrator role. Naturally, CNNs require the transfer of large amount of information. Deep CNNs contain convolutional layers with parameters, which decrease memory demands like group, striding or even 1x1 convolution layers. All these were implemeneted efficiently, even using Assembly Language, taking adantage of Myriad2 Architecture. Furthermore, a different approach using **General Matrix To Matrix Multiplication (GEMM)** for testing convolutional layers was implemented. Finally, the fact that the system is SIMD increases the complexity even further. The basic ideas and methodologies which have been followed for the CNN implementation in Myriad2 can be used in other embedded devices too.
Finally, a real-time application for ImageNet's Classification has been implemented.

Key Words

Machine Learning, Convolutional Neural Networks, Deep Neural Networks, Embedded Systems, Multi-processing Systems, Myriad 2, GEMM Approach for Convolution, Real Time Classification

Αφιέρωση

Σε γονείς και φίλους,

Ευχαριστίες

Καταρχάς θα ήθελα να ευχαριστήσω του καθηγητή κ. Σούντρη Δημήτριο 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 του Δρ. Λάζαρο Παπαδόπουλο για την καθοδήγησή του και τη συνεργασία που είχαμε. Η συνεργασία μου μαζί τους θεωρώ ότι με μετέτρεψε από φοιτητή σε μηχανικό. Επιπλέον ένα μεγάλο ευχαριστώ αξίζουν οι γονείς μου για τη στήριξή τους όλα αυτά τα χρόνια.

Ξεχωριστά ευχαριστήρια κατανέμονται σε κάποια άτομα που μαζί ανακαλύψαμε και ανακαλύπτουμε του ορισμό της λέξης ¨φιλία¨.

Αθήνα, Ιούνιος 2018

Μπαρτσώκας Αναστάσιος

Περιεχόμενα

Κατάλογος Σχημάτων

Κατάλογος Πινάκων

Κεφάλαιο 1

Υλοποίηση Εντονων Υπολογιστικά Συνελικτικών Νευρωνικών Δικτύων σε Ενσωματωμένες Αρχιτεκτονικές με Περιορισμένους Πόρους

1.1 Εισαγωγή στη Μηχανική Μάθηση

Η Μηχανική μάθηση είναι ένα υποπεδίο της επιστήμης των υπολογιστών που αναπτύχθηκε από τη μελέτη της αναγνώρισης προτύπων και της υπολογιστικής θεωρίας μάθησης στην τεχνητή νοημοσύνη. Ο Tom M. Mitchell πρότεινε έναν πιο επίσημο ορισμό που χρησιμοποιείται ευρέως [19]: «Ένα πρόγραμμα υπολογιστή λέγεται ότι μαθαίνει από εμπειρία Ε ως προς μια κλάση εργασιών Τ και ένα μέτρο επίδοσης Π, αν η επίδοσή του σε εργασίες της κλάσης Τ, όπως αποτιμάται από το μέτρο Ρ, βελτιώνεται με την εμπειρία Ε». Ο τεράστιος όγκος δεδομένων και πληροφοριών που υπάρχει σήμερα μας οδήγησε στην ανάπτυξη αυτοδίδακτων αλγορίθμων, οι οποίοι είναι σε θέση να αντλούν γνώση από τα διαθέσιμα δεδομένα. Το πρωτόγνωρο χαρακτηριστικό της μηχανικής μάθσης είναι δεν απαιτεί την ανθρώπινη παρέμβαση για τη μοντελοποίηση των κανόνων που περιγράφουν τα υπό μελέτη δεδομένα. Ο λόγος είναι ότι οι ίδιοι οι αλγόριθμοι κατασκευάζουνε τα μοντέλα περιγραφής των δεδομένων και τα βελτιώνουνε συγκρίνοντας τα με αυτά που έχουνε ως στόχο. Η μηχανική μάθηση έχει αξιοθαύμαστες εφαρμογές στην καθημερινότητα, αφού είναι ο λόγος που υπάρχουνε αποτελεσματικές μηχανές αναζήτησης, εφαρμογές αναγνώρισης εικόνας και ήχου, ευφυή ρομπότ και αυτοκαθοριζόμενα αυτοκίνητα κ.α.

1.1.1 Κατηγορίες εργασιών Μηχανικής Μάθησης

Οι εργασίες μηχανικής μάθησης συνήθως ταξινομούνται σε τρεις μεγάλες κατηγορίες ανάλογα με τη φύση του εκπαιδευτικού «σήματος» ή την «ανατροφοδότηση» που είναι διαθέσιμα σε ένα σύστημα εκμάθησης. Αυτές σύμφωνα με [19] είναι:

- **Επιτηρούμενη μάθηση** (αλλιώς επιβλεπόμενη μάθηση ή μάθηση με επίβλεψη): Το υπολογιστικό πρόγραμμα δέχεται τις παραδειγματικές εισόδους καθώς και τα επιθυμητά αποτελέσματα από έναν «δάσκαλο», και ο στόχος είναι να μάθει έναν γενικό κανόνα προκειμένου να αντιστοιχίσει τις εισόδους με τα αποτελέσματα.

- **Μη επιτηρούμενη μάθηση** (αλλιώς επίβλεπτη μάθηση ή μάθηση χωρίς επίβλεψη): Χωρίς να παρέχεται κάποια εμπειρία στον αλγόριθμο μάθησης, πρέπει να βρεί την δομή των δεδομένων εισόδου. Η μη μεπιτηρούμενη μάθηση μπορεί να είναι αυτοσκοπός (ανακαλύπτοντας κρυμμένα μοτίβα σε δεδομένα) ή μέσο για ένα τέλος (χαρακτηριστικό της μάθησης). 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 υπόλοιπες κατηγορίες μηχανικής μάθησης.

- **Ενισχυτική μάθηση**: Ένα πρόγραμμα υπολογιστή αλληλεπιδρά με ένα δυναμικό περιβάλλον στο οποίο πρέπει να επιτευχθεί ένας συγκεκριμένος στόχος (π.χ. η οδήγηση ενός οχήματος), χωρίς κάποιος δάσκαλος να του λέει ρητά αν έχει φτάσει κοντά στο στόχο του. Ένα άλλο παράδειγμα είναι να μάθει να παίζει ένα παιχνίδι εναντίον κάποιου αντιπάλου.

Μια άλλη κατηγοριοποίηση των προβλημάτων μηχανικής μάθησης, όπως βλέπουμε στο [19] προκύπτει όταν κάποιος θεωρήσει το επιθυμητό αποτέλεσμα του σύστηματος μηχανικής μάθησης.

- Στην **ταξινόμηση**, τα δεδομένα εισόδου χωρίζονται σε δύο ή περισσότερες κλάσεις, και η μηχανή πρέπει να κατασκευάσει ένα μοντέλο, το οποίο θα αντιστοιχίζει τα δεδομένα σε μία ή περισσότερες (multi-label ταξινόμηση) κλάσεις. Αυτό συνήθως εμπίπτει στην επιτηρούμενη μάθηση. Τα φίλτρα spam είναι ένα παράδειγμα ταξινόμησης, όπου οι είσοδοι είναι τα emails ή άλλα μηνύματα και οι κλάσεις είναι ¨spam¨ και ¨όχι spam¨.

- Στην **παλινδρόμηση**, επίσης πρόβλημα επιτηρούμενης μάθησης, τα αποτελέσματα είναι συνεχή και όχι διακριτά.

- Στην **συσταδοποίηση**, ένα σύνολο εισόδων πρόκειται να χωριστεί σε ομάδες. Σε αντίθεση με την ταξινόμηση, οι ομάδες δεν είναι γνωστές εκ των προτέρων, καθιστώντας αυτόν τον διαχωρισμό τυπική εργασία μη επιτηρούμενης μάθησης.

- Στην **εκτίμηση πυκνότητας** βρίσκει την κατανομή των δεδομένων εισόδου σε κάποιο χώρο.

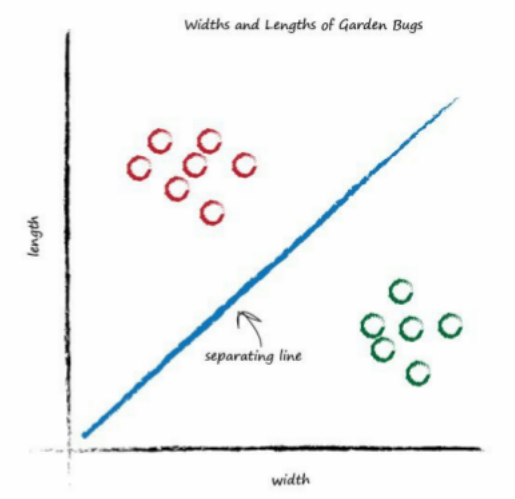
- Σε προβλήματα **μείωσης διαστασιμότητας** (dimensionality reduction), τα δεδομένα απλοποιούνται και αντιστοιχίζονται σε ένα χώρο λιγότερων διαστάσεων. Το στατιστικό μοντέλο θεμάτων (Topic modeling) είναι ένα σχετικό πρόβλημα, όπου η μηχανή καλείται να βρει έγγραφα που καλύπτουν παρόμοια θέματα από ένα σύνολο εγγράφων γραμμένων σε φυσική γλώσσα.

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. Η ταξινόμηση ενός αντικειμένου σε μια κατηγορία γίνεται με την ανάθεση μιας ετικέτας σε αυτό. Οι ετικέτες μπορούν να πάρουν 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 είδους εντόμων. Σε αυτή την περίπτωση, ένα σύνολο δεδομένων εκπαίδευσης που περιέχει διαστάσεις ως παραδείγματα για ανναγνώριση κατηγορίας εντόμων είναι ένα σημείο εκκίνησης. Έπειτα, εάν ο χρήστης εισαγάγει διαστάσεις, μέσω μιας συσκευής εισόδου, το μοντέλο πρόβλεψης θα είναι σε θέση να εκτιμήσει το σωστό έντομο με κάποια ακρίβεια. Ωστόσο, ο αλγόριθμος δε θα είναι σε θέση να αναγνωρίσει επιτυχώς οποιαδήποτε έντομα, εάν αυτά δεν υπάρχουν στο σύνολο εκπαίδευσής του.
Το παρακάτω σχήμα 1.1 μας δείχνει την ιδέα της δυαδικής ταξινόμησης, υποθέτοντας ότι έχουν δοθεί 13 δείγματα κατά το στάδιο της εκπαίδευσης: 7 δείγματα έχουν την ετικέτα της κατηγορίας κάμπια (κόκκινοι κύκλοι) και 6 δείγματα έχουν την ετικέτα των κοκκινέλιδων (πράσινοι κύκλοι). Σε αυτό το σενάριο, το σύνολο δεδομένων είναι δισδιάστατο, το οποίο σημαίνει, ότι κάθε δείγμα εμπεριέχει δύο τιμές x1 και x2. Ένας αλγόριθμος (επιτηρούμενης) μηχανικής μάθησης μπορεί να χρησιμοποιηθεί για να μάθει ένα κανόνα - το σύνορο της απόφασης που αναπαρίσταται με τη μπλε γραμμή - που διαχωρίζει τις δύο κατηγορίες και ταξινομεί τα νέα δεδομένα σε μία από τις κατηγορίες δεδομένων των τιμών x1 και x2. Το παράδειγμα έχει εμνευστεί από το βιβλίο [14].

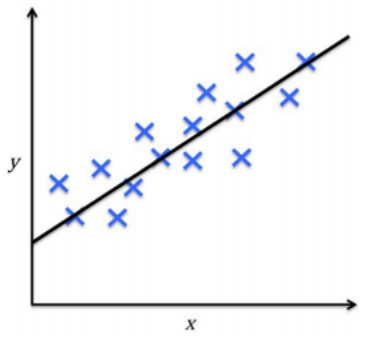
1.1.3 Παλινδρόμηση στην επιτηρούμενη μάθηση: Προβλέποντας συνεχή αποτελέσματα

Η προηγούμενη υποενότητα έδειξε πως ο στόχος της ταξινόμησης είναι η ανάθεση μη διατεταγμένων ετικετών σε αντικείμενα. Ένα δεύτερο είδος της επιτηρούμενης μάθησης είναι η πρόβλεψη συνεχών αποτελεσμάτων, που είναι γνωστό στον κλάδο της στατιστικής ως ανάλυση παλινδρόμησης. Στην ανάλυση



Σχήμα 1.1: Δυαδική ταξινόμηση στην επιτηρούμενη μάθηση.

παλινδρόμησης σύμφωνα με το [13], 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 παρακάτω σχήμα 1.2 αποτυπώνει την ιδέα της γραμμικής παλινδρόμησης. Δεδομένων μιας μεταβλητής πρόβλεψης ξ και μιας μεταβλητής απόκρισης ψ, 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.



Σχήμα 1.2: Γραμμική παλινδρόμηση στην επιτερούμενη μάθηση.

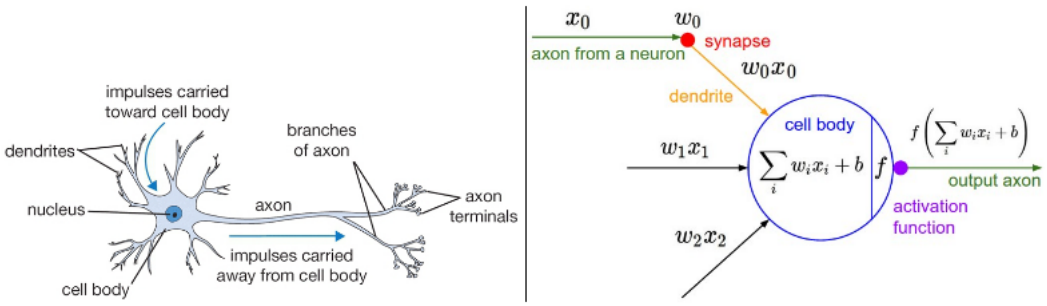
1.2 Νευρωνικά Δίκτυα

Νευρωνικό δίκτυο [20] ονομάζεται ένα κύκλωμα διασυνδεδεμένων νευρώνων. Στην περίπτωση βιολογικών νευρώνων, πρόκειται για ένα τμήμα νευρικού ιστού.
Στην περίπτωση τεχνητών νευρώνων, πρόκειται για ένα αφηρημένο αλγοριθμικό κατασκεύασμα το οποίο εμπίπτει στον τομέα της υπολογιστικής νοημοσύνης. Στόχος του νευρωνικού δικτύου είναι η επίλυση

κάποιου υπολογιστικού προβλήματος, ή της υπολογιστικής νευροεπιστήμης. Επιπλέον στόχος μπορεί να είναι η υπολογιστική προσομοίωση της λειτουργίας των βιολογικών νευρωνικών δικτύων με βάση κάποιο μαθηματικό μοντέλο τους.
Η περιοχή των Νευρωνικών Δικτύων αρχικά εμπνεύστηκε κυρίως από το στόχο της μοντελοποίησης των βιολογικών νευρωνικών συστημάτων, αλλά από τότε αποκλίνει και γίνεται ζήτημα μηχανικής και επιτυγχάνοντας καλά αποτελέσματα στις εργασίες Machine Learning.

1.2.1 Βιολογικός Νευρώνας

Η βασική υπολογιστική μονάδα του εγκεφάλου είναι ένας **νευρώνας** [10]. Περίπου 86 δισεκατομμύρια νευρώνες μπορούν να βρεθούν στο ανθρώπινο νευρικό σύστημα και συνδέονται με περίπου 10^{14} - 10^{15} συνάψεις. Το παρακάτω διάγραμμα 1.3 δείχνει ένα σχέδιο καρτών ενός βιολογικού νευρώνα (αριστερά) και ένα κοινό μαθηματικό μοντέλο (δεξιά). Κάθε νευρώνας λαμβάνει σήματα εισόδου από τους δενδρίτες και παράγει σήματα εξόδου κατά μήκος του (μοναδικού) άξονα του. Ο άξονας τελικά εκλέγεται και συνδέεται μέσω των συνάψεων με τους δενδρίτες άλλων νευρώνων. Στο υπολογιστικό μοντέλο ενός νευρώνα, τα σήματα που ταξιδεύουν κατά μήκος των αξόνων x0 αλληλεπιδρούν πολλαπλά w0x0 με τους δενδρίτες του άλλου νευρώνα με βάση τη συναπτική αντοχή σε αυτή τη συνάφεια w0. Η ιδέα είναι ότι οι συναπτικές δυνάμεις (τα βάρη w) μπορούν να εκπαιδευτούν και να ελέγξουν την επιρροή βοηθητικά(θετικό βάρος) ή ανασταλτικό (αρνητικό βάρος) ενός νευρώνα σε ένα άλλο. Στο βασικό μοντέλο, οι δενδρίτες μεταφέρουν το σήμα στο σώμα των κυττάρων, όπου όλοι συγκεντρώνονται. Εάν το τελικό ποσό είναι πάνω από ένα συγκεκριμένο όριο, ο νευρώνας μπορεί να πυροδοτήσει, στέλνοντας μια ακίδα κατά μήκος του άξονα του. Στο υπολογιστικό μοντέλο, υποθέτουμε ότι οι ακριβείς χρονισμοί των ακίδων δεν έχουν σημασία και ότι μόνο η συχνότητα της πυροδότησης μεταδίδει πληροφορίες. Με βάση αυτή την ερμηνεία κώδικα ρυθμού, μοντελοποιήσαμε την ταχύτητα πυροδότησης του νευρώνα με μια συνάρτηση ενεργοποίησης f, η οποία αντιπροσωπεύει τη συχνότητα των αιχμών κατά μήκος του άξονα. Ιστορικά, μια κοινή επιλογή της συνάρτησης ενεργοποίησης είναι η σιγμοειδής συνάρτηση σ, δεδομένου ότι λαμβάνει μια πραγματική τιμή εισόδου (την ισχύ του σήματος μετά το άθροισμα) και την αντιστοιχεί σε εύρος μεταξύ 0 και 1.

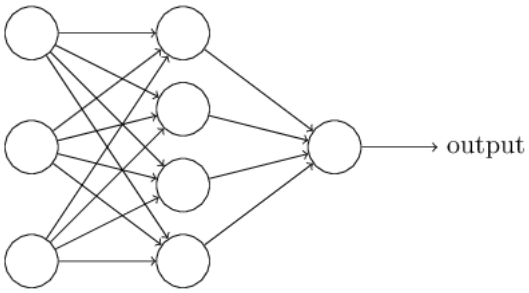


Σχήμα 1.3: Βιολογικός Νευρώνας αριστερά με το μαθηματικό μοντέλο δεξιά.

Με άλλα λόγια, κάθε νευρώνας εκτελεί ένα εσωτερικό γινόμενο με την είσοδο και τα βάρη του, προσθέτει την βάση και εφαρμόζει τη μη γραμμικότητα (ή τη λειτουργία ενεργοποίησης), σε αυτή την περίπτωση το sigmoid σ(χ) = 1/(1+ $\varepsilon^{-\chi}$).

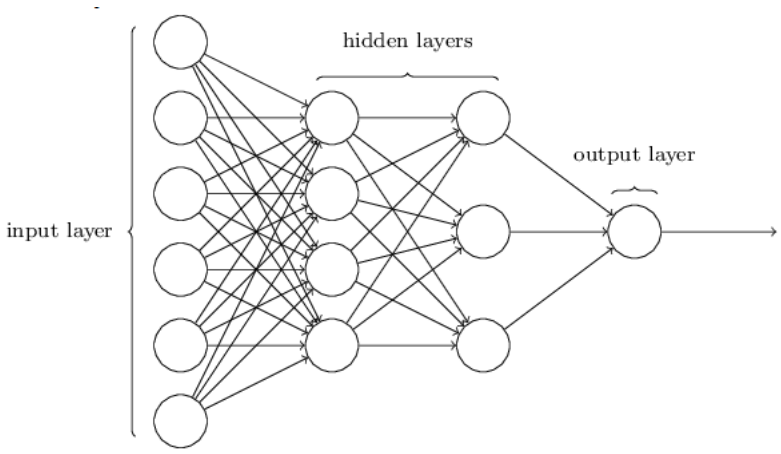
1.3 Αρχιτεκτονική Νευρωνικών Δικτύων

Στο 1.4 παρουσιάζεται ένα **νευρωνικό δίκτυο**, από το οποίο θα αντλήσουμε τα εξής:



Σχήμα 1.4: Νευρωνικό Δίκτυο με ένα κρυφό στρώμα.

Το αριστερά στρώμα ονομάζεται στρώμα εισόδου και αποτελείται από νευρώνες που ονομάζονται νευρώνες εισόδου. Αντίστοιχα, το δεξί στρώμα καλείται στρώμα εξόδου και περιέχει νευρώνες εξόδου, όπου στην περίπτωσή μας βλέπουμε μόνο ένα. Το μεσαίο στρώμα καλείται ¨κρυμμένο¨, αφού οι νευρώνες του δεν είναι ούτε εισόδου ούτε εξόδου. Τα ¨βαθιά¨ νευρωνικα δίκτυα αποτελούνται από τουλάχιστον ένα ¨κρυφό¨ στρώμα. Στα κανονικά νευρωνικά δίκτυα, ο 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, όπως βλέπουμε στο 1.5.



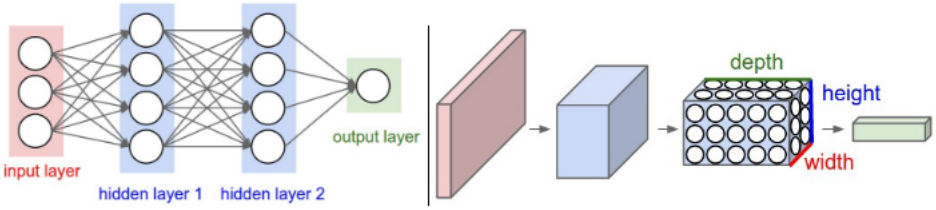
Σχήμα 1.5: Ένα ΝΔ τριών στρώσεων με 6 εισόδους, 2 κρυφές στρώσεις με 4 και 3 νευρώνες η καθεμία αντίστοιχα τη στρώση εξόδου με 1 νευρώνα.

1.4 Συνελικτικά Νευρωνικά Δίκτυα

Τα **συνελικτικά νευρωνικά δίκτυα** (ΣΝΔ) είναι παρόμοια με τα Νευρωνικά Δίκτυα, καθώς αποτελούνται από νευρώνες που διαθέτουνε βάρη και βάσεις. Κάθε νευρώνας δέχεται μερικές εισόδους, υπολογίζει ενα εσωτερικό γινόμενο πριν από κάποιες άλλες πράξεις και δίνει την έξοδο. Επίσης, τα ΣΝΔ εξακολουθούν να έχουν μια λειτουργία απώλειας (π.χ. SVM / Softmax) στο τελευταίο (πλήρως συνδεδεμένο) στρώμα, εξακολουθώντας να ισχύουν αυτά που είδαμε παραπάνω για τα Νευρωνικά Δίκτυα. Επί της ουσίας, ένα ΣΝΔ είναι ένα μοντέλο Νευρωνικού Δικτύου που έχει σχεδιαστεί αποκλειστικά για την αναγνώριση δισδιάστατων αντικειμένων, παρουσιάζοντας υψηλό βαθμό αναλλοίωτης συμπεριφοράς κατά την μετάθεση, κλιμάκωση, στρέβλωση και άλλες παραμορφώσεις της εισόδου. Αυτό που διαφοροποιεί τα ΣΝΔ είναι ότι τόσο η είσοδος όσο και το σετ δεδομένων που έχει εκπαιδεύσει το δίκτυο αποτελούνται από εικόνες.

1.4.1 Επισκόπηση Αρχιτεκτονικής Συνελικτικών Νευρωνικών Δικτύων

Τα τεχνητά νευρωνικά δίκτυα δεν αποδίδουν καλά με εικόνες [10]. Στο CIFAR-10, οι εικόνες έχουν μέγεθος μόνο 32x32x3 (32 πλάτος, 32 ύψος, 3 κανάλια χρώματος), έτσι ένας μόνο νευρώνας σε ένα πρώτο κρυφό πλήρως συνδεδεμένο στρώμα ενός νευρικού δικτύου θα έχει $32 * 32 * 3 = 3072$ βάρη. Το ποσό αυτό εξακολουθεί να φαίνεται διαχειρήσιμο αλλά σε δίκτυα με μεγαλύτερες εικόνες ως είσοδο προκαλεί σημαντικό ζήτημα. Είναι σαφές ότι το πλήρες συνδεδεμένο στρώμα συνδυάζεται με τεράστιο όγκο παραμέτρων που οδηγούνε σε υπερχείλιση.
Τα συνελικτικά νευρωνικά δίκτυα επωφελούνται από το γεγονός ότι η είσοδος αποτελείται από εικόνες και περιορίζουνε την αρχιτκετονική τους με τρόπο που βγάζει περισσότερο νόημα. Πιο συγκεκριμένα σε αντίθεση με τα κοινά Νευρωνικά Δίκτυα οι στρώσεις ενός ΣΝΔ έχουνε νευρώνες διατεταγμένους σε 3 διαστάσεις: πλάτος, ύψος και 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. Για παράδειγμα, οι εικόνες εισόδου στο CIFAR-10 είναι ένας όγκος εισόδου ενεργοποιήσεων και ο όγκος έχει διαστάσεις 32x32x3 (πλάτος, ύψος, βάθος αντίστοιχα). Οι νευρώνες σε ένα στρώμα θα συνδεθούν μόνο σε μια μικρή περιοχή του στρώματος πριν από αυτό, αντί για όλους τους νευρώνες με πλήρως συνδεδεμένο τρόπο. Επιπλέον, το τελικό στρώμα εξόδου για το CIFAR-10 θα έχει διαστάσεις 1x1x10, διότι μέχρι το τέλος της αρχιτεκτονικής του ΣΝΔ θα μειώσουμε την πλήρη εικόνα σε ένα μόνο διάνυσμα βαθμολογίας κλάσης, διατεταγμένο κατά μήκος της διάστασης βάθους.



Σχήμα 1.6: Αριστερά, ένα ΝΔ 2 ¨κρυφών¨ στρώσεων. Δεξιά, ένα ΣΝΔ με νευρώνες 3 διαστάσεων. Κάθε στρώση μετασχηματίζει τον 3Δ όγκο εισόδου σε ένα 3Δ όγκο εξόδου απο νευρώνες. 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.

1.4.2 Βασικές στρώσεις Συνελικτικών Νευρωνικών Δικτύων

Παρακάτω αναφέρονται κάποια από τα βασικά είδη στρώσεων που συναντώνται σε ΣΝΔ, σύμφωνα με [10]. Συνελικτικές στρώσεις, πλήρως συνδεδεμένες στρώσεις, στρώσεις συγκέντρωσης και κανονικοποίησης. Αυτές οι στρώσεις διατάσσονται με διάφορους τρόπους και δημιουργούνε αρχιτεκτονικές ΣΝΔ. Σε όλα τα δίκτυα παρατηρείται και η στρώση εισόδου που η λειτουργία της είναι να αντιγράφει την είσοδο της στρώσης στην έξοδο. Λεπτομέρειες για τις υπόλοιπες στρώσεις:

- **Στρώση Εισόδου** Εμπεριέχει τα δεδομένα εισόδου, που είναι οι τιμές των εικονοστοιχείων της προεπεξεργασμένης εικόνας εισόδου. Το βάθος της στρώσης εισόδου είναι το ίδιο με το πλήθος των καναλιών της εικόνας εισόδου.

- **Συνελικτική Στρώση** Δίνει την έξοδο των νευρώνων που είναι συνδεδεμένοι σε τοπικές περιοχές της εισόδου, υπολογίζοντας το εσωτερικό γινόμενο μεταξύ των βαρών των πυρήνων και της περιοχής που αντιστοιχεί στην είσοδο. Αν για παράδειγμα γίνει συνέλιξη με 12 φίλτρα, η έξοδος θα είναι 12 x w x h, όπου w,h το πλάτος και το ύψος εξόδου αντίστοιχα.

- **Συγκεντρωτική Στρώση** Πραγματοποεί υποδειγματοληψία με διάφορους τρόπους κατά μήκος των τοπικών διαστάσεων (πλάτος, ύψος).

- **Ανορθωτική Στρώση** Η δουλεία αυτής της στρώσης είναι να υπολογίζει το max(0,x). Αυτή η στρώση δεν επηρεάζει καθόλου τις διαστάσεις εισόδου καθώς η συνάρτηση εφαρμόζεται σε κάθε στοιχείο ξεχωριστά.

- **Στρώσεις κανονικοποίησης** Η Κανονικοποίηση προκαλεί παραλλαγές στη φωτεινότητα, υπολογίζοντας μέσους όρους, αφαιρέσεις, κάνοντας υψιπερατά φιλτραρίσματα αλλά και τοπικές εξομαλύνσεις της αντίθεσης και άλλα χαρακτηριστικά των εικόνων.

- **Πλήρως Συνδεδεμένες Στρώσεις** Χρησιμοποιούνται για να υπολογίσουνε τα σκορ των κατηγοριών. Μετατρέπουνε την είσοδο σε ένα διάνυσμα όπου οι διαστάσεις του είναι όσες και οι κατηγορίες της ταξινόμησης. Οι νευρώνες αυτής της στρώσης συνδέονται με κάθε νευρώνα του πρηγούμενου σταδίου.

1.5 Μονάδα Επεξεργασίας Εικόνας Myriad2

Η υλοποίηση της μηχανής εκτέλεσης ΝΔ έγινε στο πολυεπεξεργαστικό σύστημα Myriad2. Η μονάδα επεξεργασίας εικόνας αναπτύχθηκε από τη Movidius Ltd., η οποία έγινε μέλος της ομάδας Perceptual Computing Group της Intel, με στόχο την δημιουργία ευφυών συσκευών σε εφαρμογές όρασης υπολογιστών. Το κύριο χαρασκτηριστικό της πλατφόρμας είναι ότι καταφέρνει να προσφέρει εξαιρετικά χαμηλή κατανάλωση ενέργειας σε πολύ ¨βαριές¨ εφαρμογές αναγνώρισης προτύπων, όπως είναι η εκτέλεση ΣΝΔ.

1.5.1 Χαρακτηριστικά Αρχιτεκτονικής Myriad2

Τα κυριότερα χαρακτηριστικά της Myriad2 είναι:

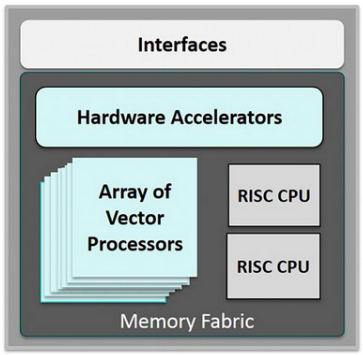
- **Σχεδιασμός πολύ χαμηλής ισχύος:** Κάνοντας τη κατάλληλη για χρήση σε φορητές συσκευές, που η αυτονομία της μπαταριάς είναι κυρίαρχη παράμετρος.

- **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.

- **Επιπλέον Λεπτομέρειες:** Η αρχιτεκτονική της Myriad2 περιλαμβάνει ενα σετ από διεπαφές, από οπτικούς επιταχυντές, 12 διανυσματικούς VLIW επεξεργαστές που λέγονται SHAVE και μια μνήμη που σε συνδυασμό με τους υπόλοιπους πόρους προσφέρει τη δυνατότητα αποδοτικής κατανάλωσης ισχύος.



Σχήμα 1.7: Συνοπτική Περιγραφή της Αρχιτεκτονικής Myriad2.

Μια πιο λεπτομερής περιγραφή της αρχιτεκτονικής δίνεται παρακάτω στην 1.8:

- Ο **Leon OS** είναι ο κύριος επεξεργαστής, καθώς μετά την εκκίνηση της συσκευής η εκτέλεση του προγράμματος αρχίζει από αυτό τον επεξεργαστή, ο οποίος ανήκει στο υποσύστημα CPU subsystem (CSS) που είναι η κύρια μονάδα επικοινωνίας με τον εξωτερικό κόσμο μέσω των εξωτερικών περιφερειακών επικοινωνίας: I2C, I2S, SPI, UART, GPIO, ETH και USB3.0. Η κύρια μονάδα ελέγχου του CSS είναι ο επεξεργαστής LeonOS (LOS), που διαθέτει αρκετά μεγάλες κρυφές μνήμες, επιτρέποντας τη δυνατότητα εκτέλεσης ενός μοντέρνου λειτουργικού συστήματος πραγματικού χρόνου RTOS.

- Ο **Leon RT** είναι ο επιπρόσθετος SPARC επεξεργαστής και ανήκει στο υποσύστημα Media subsystem (MSS), μια δομική μονάδα που επιτρέπει διασύνδεση με συσκευές εικόνας, όπως αισθητήρες εικόνας, οθόνες LCD, ελεγκτές HDMI κ.λ.π.

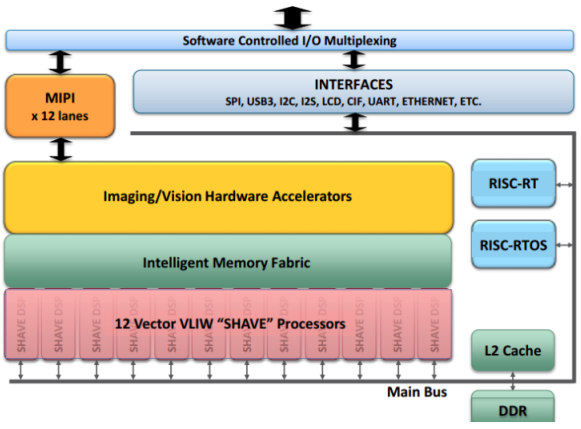
- Οι **SHAVEs** έχουν υπολογιστικό ρόλο και διευθύνονται από τους 2 SPARC επεξεργαστές. Προκειμένου να προσφέρουν υψηλή απόδοση και χαμηλή κατανάλωση περιέχουν ευρείς register-files, όπου σε συνδυασμό με παραλληλία τύπου SIMD μεγιστοποιούνε την επίδοση ανα μονάδα κατανάλωση ισχύος.

- Η **DDR** είναι η κύρια μνήμη του συστήματος και οι 2 SPARC επεξεργαστές αναφέρονται σε αυτή τη μνήμη με μικρό κόστος, λόγω των επιλεγμένων μεγεθών της κρυφής μνήμης. Είναι τοποθετημένη εκτός της ψηφίδας, που σημαίνει ότι οι 14 επεξεργαστές χρησιμοποιούν τον ίδιο ελεγκτή για να την προσπελάσουν και έχουν επιπλέον κόστος.

- **CMX**: Πρόκειται για τη συντομογραφία του Connection Matrix, το οποίο δικαιολογείται από το γεγονός ότι η CMX αποτελείται από αρκετές μικρότερες μονάδες SRAM, με συνολικό μέγεθος τα 2 MB. Κάθε επεξεργαστής SHAVE έχει ξεχωριστές θέσεις 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.

- **CMX DMA Controller**: Αυτός ο ελεγκτής βρίσκεται ανάμεσα του διαύλου MXI των 128-bit και της μνήμης CMX. 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.

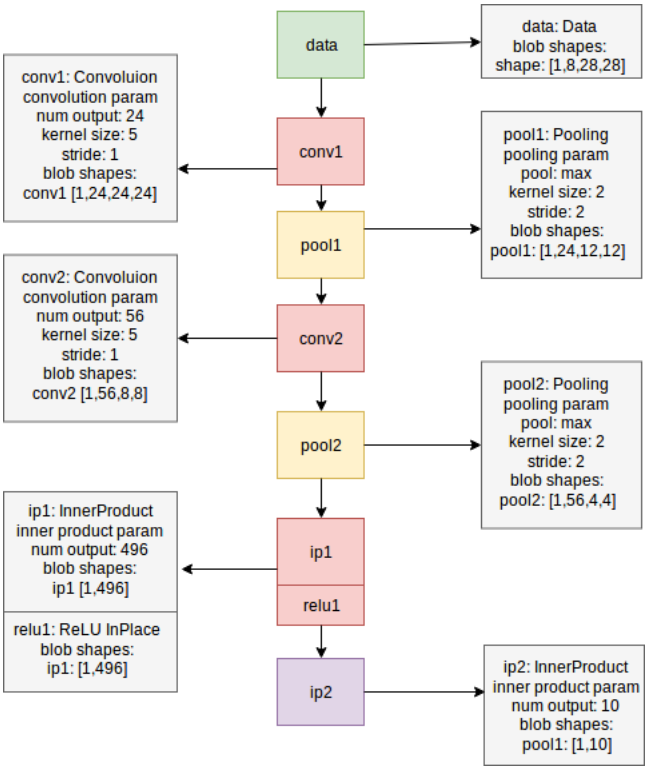
- **SIPP**: Πρόκειται για ένα μηχανισμό υλικού/λογισμικού που χρησιμοποιείται από τη Myriad2, με σκοπό την αποδοτική δρομολόγηση εργασιών ψηφιακής επεξεργασίας εικόνας. Αυτός ο μηχανισμός είναι βασισμένος σε επεξεργασία μορφής σωλήνωσης και χρησιμοποιεί τα φίλτρα υλικού που παρέχονται από την Myriad2,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



Σχήμα 1.8: Οι δομές της Αρχιτεκτονικής της Myriad2.

1.6 Συνοπτική Περιγραφή Προηγούμενης Έκδοσης

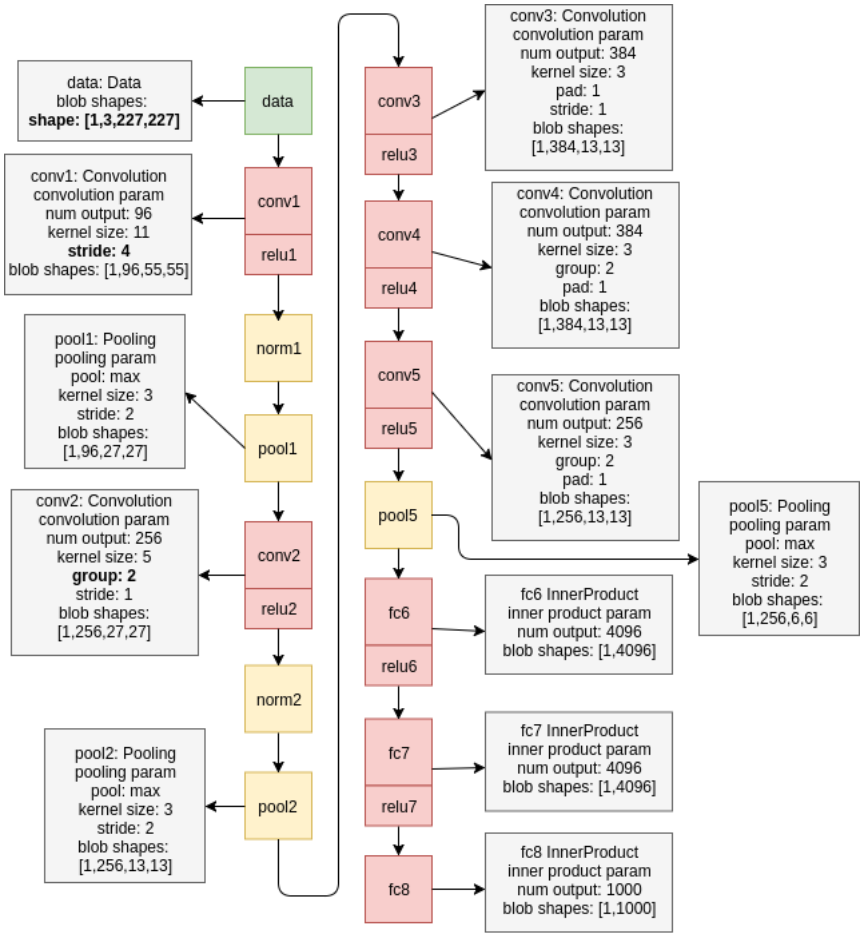
Αρχικά για την εκτέλεση των νευρωνικών δικτύων, εισαγάγουμε στατικά στην κύρια μνήμη τόσο την εικόνα εισόδου μετά την προεπεξεργασία που χρειάζεται, όσο και τα βάρη του νευρωνικού δικτύου σε κατάλληλη μορφή από το Caffe. Στην συνέχεια, δημιουργούνται οι κόμβοι του δικτύου ανάλογα με το είδος τους και αρχίζουνε να εκτελούνται σειριακά γράφοντας ο καθένας την έξοδο του στην είσοδο του επόμενου. Αξίζει να σημειωθεί το γεγονός ότι τα δεδομένα πρέπει να μεταφέρονται από την κύρια μνήμη DDR στη CMX που είναι κοντά στους SHAVEs, επειδή εκεί γίνονται γρήγορα οι υπολογισμοί και στη συνέχεια τα αποτελέσματα να επιστρέφονται. Αυτές οι μεταφορές επιτυγχάνονται μέσω του DMA ελεγκτή για ταχύτητα. Ένα παράδειγμα εκτέλεσης νευρωνικού δικτύου με την προηγούμενη έκδοση της μηχανής αποτελεί το παρακάτω **Lenet-8**.



Σχήμα 1.9: Lenet8 με 7 στρώσεις.

1.7 Συνεισφορές για την Υποστήριξη μεγαλύτερων Δικτύων

Στόχος ήτανε να εκτελεστούνε βαθιά ΣΝΔ και να επιτευχθεί ταξινόμηση με βάσει το σύνολο δεδομένων ImageNet, αρχίζοντας από το AlexNet [7] νικητή του διαγωνισμού ImageNet Large Scale Visual Recognition Challenge το 2012, η αρχιτεκτονική του οποίου φαίνεται και παρακάτω στο 1.17. Κάθε κανάλι της εικόνας εισόδου αυτού του δικτύου περιλαμβάνει δεδομένα τα οποία δε χωράνε στην μνήμη CMX του πολυεπεξεργαστή για να μπορέσουνε να γίνουνε οι υπολογισμοί. Αυτό έχει ως αποτέλεσμα, η εικόνα να χρειάζεται προεπεξεργασία ώστε να σταλθεί από την κύρια μνήμη. Επιπλέον σε αυτό το δίκτυο λόγω τεράστιου όγκου δεδομένων χρησιμοποιούνται παράμετροι, οι οποίοι μειώνουνε τον όγκο των δεδομένων και βοηθάνε στην ακρίβεια, όπως αποδείχθηκε. Τέτοιου είδους παράμετροι αποτελούνε οι group, striding. Μια επιπλέον τεχνική που χρησιμοποιήθηκε σε άλλα επόμενα ΣΝΔ που γίνανε train στο ImageNet ήτανε η χρήση των 1x1 συνελικτικών στρώσεων. Ο λόγος είναι ότι η διαστάσεις εξόδου θα είναι ίσες με την είσοδο, οπότε αν το πλήθος των φίλτρων είναι μικρότερο από ότι το βάθος της εισόδου, το νευρωνικό δίκτυο περιορίζει τους νευρώνες του και τις συνδέσεις τους, με άλλα λόγια τα βάρη του. Η υλοποίηση του 1x1 συνελικτικού φίλτρου έγινε με συμβολική γλώσσα για λόγους απόδοσης.



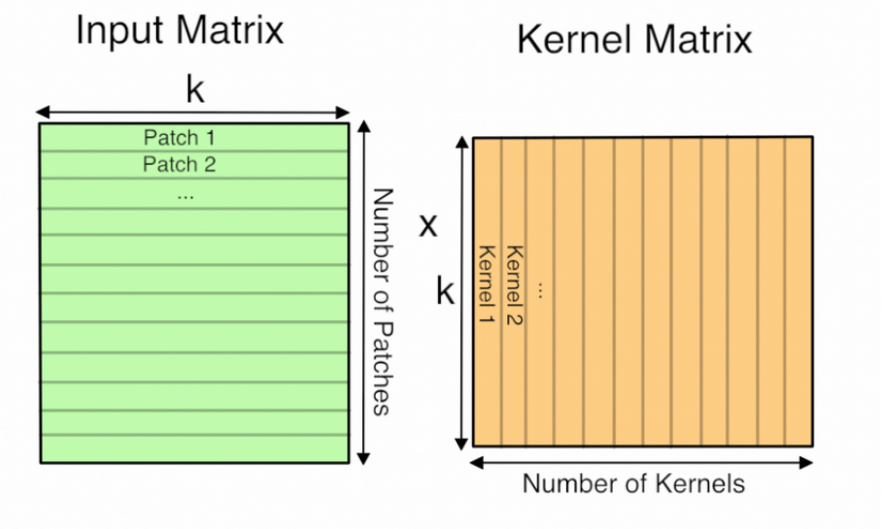
Σχήμα 1.10: Η αρχιτεκτονική του AlexNet.

Το 2010 έγινε, για πρώτη φόρα, ο διαγωνισμός ILSVRC, ο πρώτος με τόσο μεγάλο όγκο δεδομένων: Συγκεντρώθηκαν 14 εκατομμύρια εικόνες από το Διαδίκτυο και άνθρωποι σημείωσαν τι θεωρούν ότι φαίνεται σε κάθε εικόνα (γάτα, σκύλος, παραλία, βουνό κτλ). Κατόπιν, το «σημείωσαν» και οι αλγόριθμοι. Στον διαγωνισμό αυτό, το σφάλμα, ακόμη και των έως τότε πιο εξελιγμένων αλγορίθμων της επιστημονικής κοινότητας, ήταν 25%. Για μία στις τέσσερις εικόνες, αυτό που θεωρούσαν οι άνθρωποι ότι απεικονίζεται στις εικόνες δεν ήταν καν στις πρώτες πέντε επιλογές των συστημάτων. Όλα αυτά, μέχρι το 2012. Το 2012 ο Alex Krizhevsky, όπως είδαμε με το AlexNet κέρδισε το ILSVRC, μειώνοντας το σφάλμα κατά 10 ολόκληρες μονάδες, από το 25%, στο 15,3%. Τότε, όλοι οι σχετιζόμενοι, εταιρείες και επιστημονική κοινότητα, άρχισαν να δίνουν ξανά σημασία στα ΤΝΔ. Πλέον, τα επίπεδα αυξάνονται, και τα δίκτυα ονομάζονται βαθιά (deep) όπως βλέπουμε παρακάτω. Επιπλέον, ενώ αυτά τα τεχνητά νευρωνικά δίκτυα βασίστηκαν στην ίδια αρχή λειτουργίας με τους τεχνητούς νευρώνες του 1970, τώρα η εκπαίδευση και τα μαθηματικά τους είχαν πολλά στοιχεία γραμμικής άλγεβρας. Έτσι, αν μη τι άλλο για να σηματοδοτηθεί η διαφορά, άρχισε να κυριαρχεί ο όρος Deep Learning. Από το 2012 και μετά, δεν υπάρχει κανένας νικητής στον ILSVRC, και γενικά σχεδόν καμία δημοσίευση για επεξεργασία εικόνας, που να μη χρησιμοποιεί ΣΝΔ. Η έκδοσης της μηχανής εκτέλεσης ΣΝΔ υποστηρίζει τα εξής δίκτυα: **AlexNet, GoogleNet, NiN-ImageNet, SqueezeNet, VGGNet, ZFNet**

1.8 Εναλλακτικός τρόπος Υλοποίησης της Συνελικτικής Στρώσης

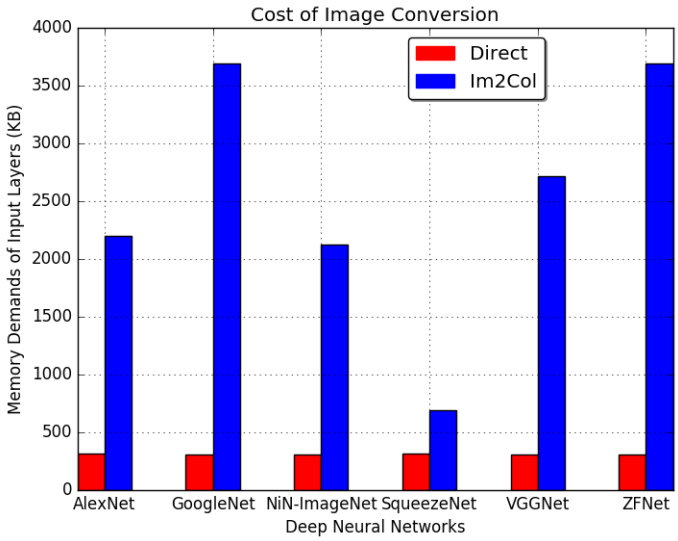
Μετά την εκτέλεση των παραπάνω ΣΝΔ του διαγωνισμού ImageNet παρατηρήθηκε ότι τα φίλτρα που υλοποιούνε συνέλιξη με πυρήνα 1x1 είναι αρκετά αργά σε σχέση με τα υπόλοιπα λόγω περιορισμένων λειτουργιών και η συμβολική γλώσσα δε μπορεί να γραφτεί χωρίς κύκλους καθυστέρησης. Έτσι, έγινε η ανάπτυξη ενός **νέου τρόπου που χρησιμοποεί γραμμική αλγεβρα** και συγκεκριμένα γινόμενο πίνακα

με πίνακα για να υλοποιηθεί. Αυτό για να συμβεί βέβαια, πρέπει πρώτα να μετατραπούνε τόσο η είσοδος όσο και τα βάρη σε δισδιάστατους πίνακες στην κατάλληλη μορφή όπως φαίνεται στο σχήμα 1.11 και τότε το γινόμενό τους θα δώσει το κατάλληλο αποτέλεσμα.



Σχήμα 1.11: Μετατροπή σε δισδιάστατους πίνακες και γινόμενο.

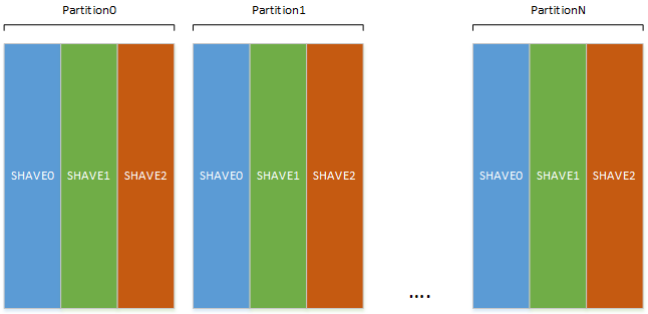
Στην περίπτωση όπου το stride είναι μεγαλύτερο από το kernel η μετατροπή σε 2 διαστάσεις μεγαλώνει την είσοδο πολύ σε απαιτήσεις μνήμης, όπως βλέπουμε στην εικόνα 1.12 για τις πρώτες στρώσεις των δικτύων που εκτελούμε.



Σχήμα 1.12: Κόστος μετατροπής σε δισδιάστατη μορφή.

Ο παραπάνω λόγος οδηγεί σε **τμηματοποίηση της εισόδου και κατάλληλη δρομολόγηση** ώστε να χωράνε τα δεδομένα στην μικρή και γρήγορη μνήμη CMX. Τέλος το DMA της εξόδου πρέπει να γίνεται κατάλληλα ώστε τα δεδομένα να αποθηκεύονται κατά βάθος και να δίνονται στο επόμενο στρώμα κατάλληλα, ανεξάρτητα από τον τρόπο σύμφωνα με τον οποίο θέλουμε να γίνει η συνέλιξη. Έστω ότι στην περίπτωση ενός παραδείγματος το συνελικτικό φίλτρο έχει διαστάσεις (1x1x64) και η είσοδος είναι διαστάσεων (64x56x56). Οι απαιτήσεις σε μνήμη της εισόδου μετά την δισδιάστατη μετατροπή ειναι: $64 * 1 * 1 = 64$ όσον αφορά τις στήλες και $56 * 56 = 3136$ γραμμές. Αυτό σημαίνει ότι οι συνολικές

απαιτήσεις είναι $3136 * 64 * 2 = 401408$ bytes. Η τμηματοποίηση θα γίνει όπως φαίνεται στην παρακάτω εικόνα 1.13 με 4 κομμάτια και 3 SHAVEs, σύμφωνα με τον αλγόριθμο που χρησιμοποείται για το πόσο χωράει στην κάθε μνήμη. Ο κάθε shave θα αναλάβει $(3136/12) * 64 * 2 = 33536$ bytes περίπου.



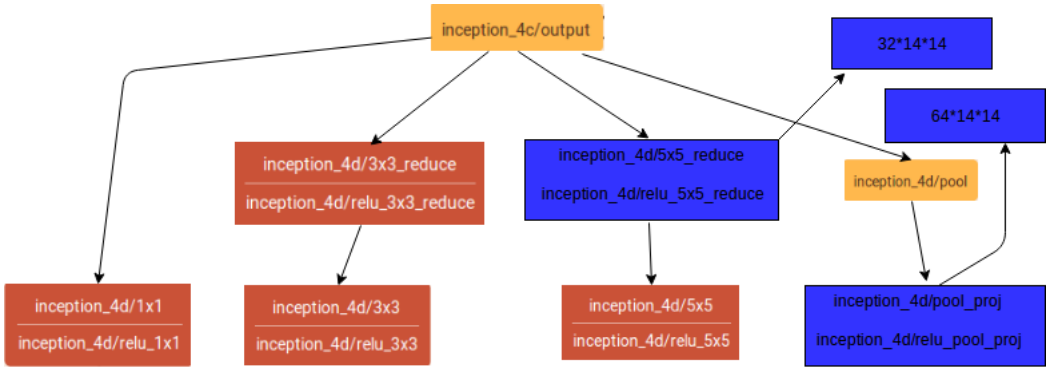
Σχήμα 1.13: Παράδειγμα τμηματοποίησης εισόδου/εξόδου.

1.9 Αξιολόγηση της Υλοποίησης

Αυτή η ενότητα παρουσιάζει την εκτέλεση κάποιων νευρωνικών δικτύων, όπως AlexNet, GoogleNet, SqueezeNet και VGGNet και τις μετρήσεις που ελήφθησαν όταν αυτό εκτελέστηκε τόσο στη **Myriad2** όσο και στον επεξεργαστή **Nvidia Tegra X1** με ή χωρίς χρήση της βιβλιοθήκης cuDNN, η οποία είναι βιβλιοθήκη που επιταχύνει την εκτέλεση των ΣΝΔ με τη χρησιμοποίηση των πόρων της κάρτας γραφικών. Τέλος κάποιες μετρήσεις ελήφθησαν και στο **Movidius Neural Compute Stick**, το οποίο αποτελείται από το ίδιο υλικό (Myriad2), αλλά το πακέτο λογισμικού είναι κλειστού κώδικα. Επιπλέον, κάποιες στρώσεις χρησιμοποιούνε τον άμεσο τρόπο συνέλιξης και κάποιες άλλες αυτόν της γρραμμικής άλγεβρας, ο οποίος δίνει καλύτερα αποτελέσματα σε κάποιες περιπτώσεις τόσο σε κατανάλωση ενέργειας όσο και σε χρόνο εκτέλεσης.

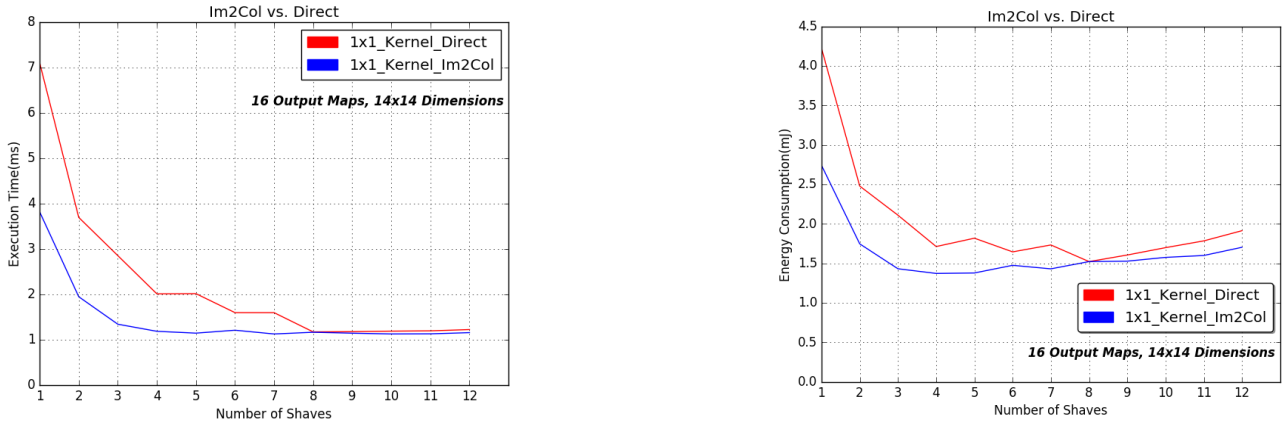
1.9.1 Πλεονεκτήματα κάθε τρόπου Συνέλιξης

Είναι γεγονός ότι η υπολογιστική συνάρτηση της συνέλιξης με τη μετατροπή σε δισδιάστατο χωρίο έχει υλοποιηθει με συνάρτηση πολλαπλασιασμού διανύσματος με πίνακα. Αυτό σημαίνει ότι για κάθε εσωτερικό γινόμενο διανυσμάτων πολλαπλασιάζονται στοιχεία πλήθους $kernel_{width} * kernel_{height} * input_{feature-maps}$ και το πλήθος των εσωτερικών γινόμενων διανυσμάτων είναι $output_{width} * output_{height} * output_{feature-maps}$. Στο GoogleNet υπάρχουνε στρώσεις που διαθέτουνε τέτοιου είδους χαρακτηριστικά, όπως φαίνεται στο 7.9 και αυτές κερδίζουν σε χρόνο εκτέλεσης με το δεύτερο τρόπο υλοποίησης. Από την άλλη πλευρά αποδεικνύεται ότι με μεγαλύτερους πυρήνες από 1x1 ο τρόπος άμεσης συνέλιξης κερδίζει σε χρόνο εκτέλεσης. Επιπλέον παρατηρείται συνεχές κέρδος σε χρόνο εκτέλεσης όσο αυξάνονται οι υπολογιστικές μονάδες στον τρόπο άμεσης συνέλιξης ενώ στον άλλο αυτό έχει να κάνει με τις διαστάσεις του δισδιάστατου πίνακα. Στον πρώτο τρόπο η παραλληλοποίηση γίνεται κατά βάθος ενώ στον άλλο κατά πλάτος του πίνακα. Όταν το πλάτους είναι μικρό, όσο αυξάνονται οι επεξεργαστές απλά αναλαμβάνουνε μικρότερα τμήματα συνεχώς και κάνουνε λίγους υπολογισμούς, έτσι ο χρόνος εκτέλεσης αναλώνεται σε μεταφορές μεταξύ των μνημών, εκτός αν το ύψος είναι αρκετά μεγάλο για να εξισορροπεί τη διαφορά και τότε παρατηρείται βελτίωση.

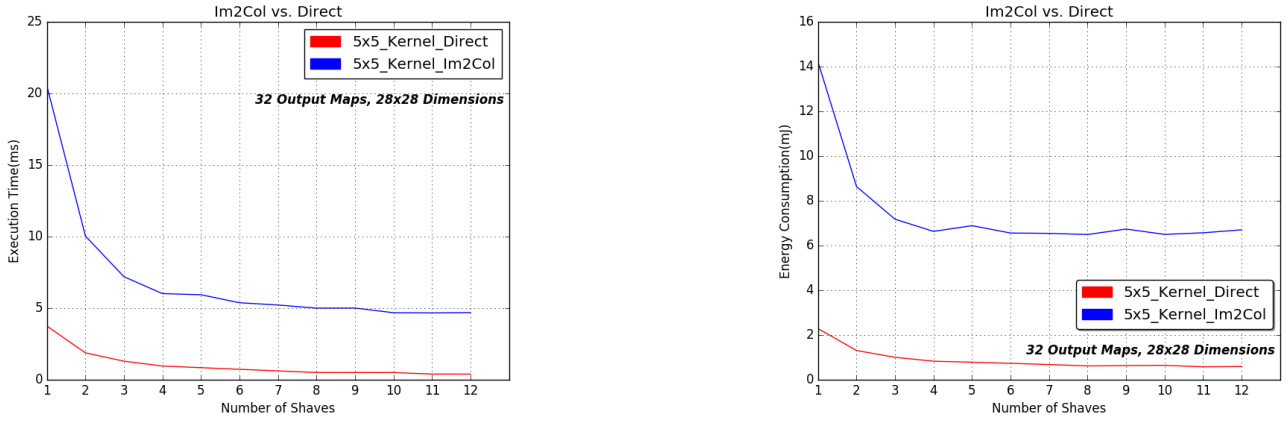


Σχήμα 1.14: Κόμβοι που επιταχύνονται σε σχέση με την άμεση συνέλιξη.

Παρακάτω βλέπουμε πως **κλιμακώνουνε οι 2 τρόποι συνέλιξης σε σχέση με την αύξηση των επεξεργαστικών μονάδων**, πρώτα για μικρό πυρήνα συνέλιξης και στη συνέχεια για μεγάλο, αριστερά φαίνονται οι χρόνοι εκτέλεσης και δεξιά η κατανάλωση ενέργειας:



Σχήμα 1.15: Κέρδος Im2Col σε σχέση με Direct



Σχήμα 1.16: Κέρδος Direct σε σχέση με Im2Col

1.9.2 Μετρήσεις

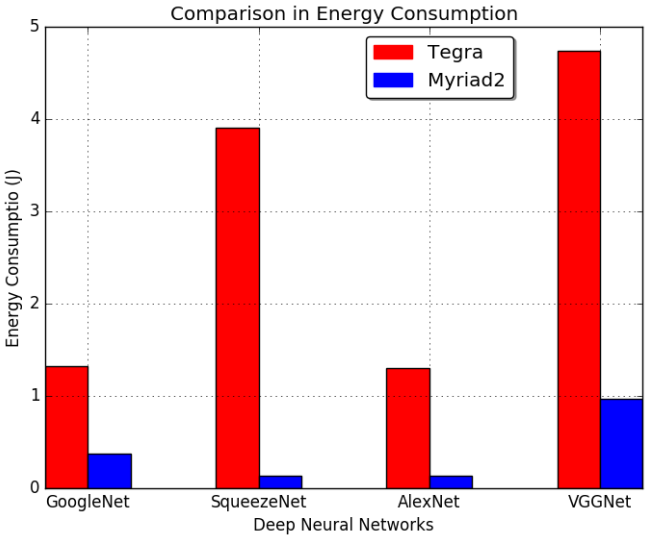
Ακολουθούνε μετρήσεις και συγκρίσεις μεταξύ των ενσωματωμένων συσκευών για 4 από τα πιο γνωστά ΣΝΔ. Το ένα είναι το AlexNet, το οποίο έμεινει στην ιστορία ως το πρώτο που έφτασε σε τόσα υψηλά ποσοστά ευστοχίας προβέψεων και άλλαξε την ιστορία για τα ΣΝΔ και το δεύτερο που έχουμε ήδη αναφέρει το GoogleNet άλλαξε τη νοοτροποία των ¨βαθιών¨ νευρωνικών δικτύων και αφαίρεσε τις βαριές

σε μνήμη πλήρως συνδεδεμένες στρώσεις. Επιπλέον, το SqueezeNet είναι κατάλληλο για να τρέχει σε ενσωματωμένες συσκευές λόγω της μικρών απαιτήσεων μνήμης που έχει. Τέλος το VGGNet έχει αρκετές διαφορετικές εκδόσεις, οι οποίες χρησιμοποιούνται σε άλλα είδους δίκτυα όπως Region-CNNs. Η παρούσα υλοποίηση θα συγκριθεί με το Movidius Neural Compute Stick. Τέλος, τα δίκτυα εκτελούνται και με το Caffe, στην ενσωματωμένη συσκευή Nvidia Tegra TX1 με επεξεργαστή Quad ARM A57 με ή χωρίς τη χρήση της GPU. Έγινε χρήση μόνο του ενός core του Quad ARM A57 για να φανεί το κόστος των πράξεων. Το Caffe εκτελείται σε Quad ARM® A57/2 MB L2 επεξεργαστή με ή χωρίς NVIDIA Maxwell ™,

Πίνακας 1.1: Χρόνοι εκτέλεσης ms και Συγκρίσεις Υλοποιήσεων

CNNs	Current Implementation	Movidius NCS	Caffe in Quad ARM A57	Caffe with CuDNN
AlexNet	98.3	96.27	7518	22.2
GoogleNet	246.1	99.04	16836	180
SqueezeNet	85.5	50.26	8961	695.38
VGGNet	586	733.50	7587	85.8

256 CUDA cores. Το συμπέρασμα από τις εκτελέσεις των δικτύων είναι ότι το Caffe με τη χρήση της GPU πετυχαίνει καλύτερους χρόνους σε κάποπια δίκτυα, αλλά δεν αποδίδει πολύ καλύτερα συγκριτικά με τη Myriad2. Αυτό στηρίζεται και στο γεγονός ότι ο παράγοντας της κατανάλωσης ενέργειας είναι μια τάξη μεγέθους πάνω ακόμα και με τη χρήση της GPU. Επίσης, φαίνεται ότι το Caffe ακόμα και με τη χρήση της GPU υστερεί στα δίκτυα που έχουν παράλληλους κόμβους, λόγω μη αξιόλογης παραλληλοποίησης και εκμετάλλευσης των πόρων. Το σίγουρο είναι ότι στη GPU επιτυγχάνεται καλύτερη παραλληλοποίηση στην πλήρως συνδεδεμένη στρώση.



Σχήμα 1.17: Χαμηλή κατανάλωση ενέργειας της Myriad2 σε σχέση με το Caffe στη TegraTX1.

Στον πίνακα 1.2 παρουσιάζονται τα αποτελέσματα της παρούσας υλοποίσης για διάφορα νευρωνικά δίκτυα:

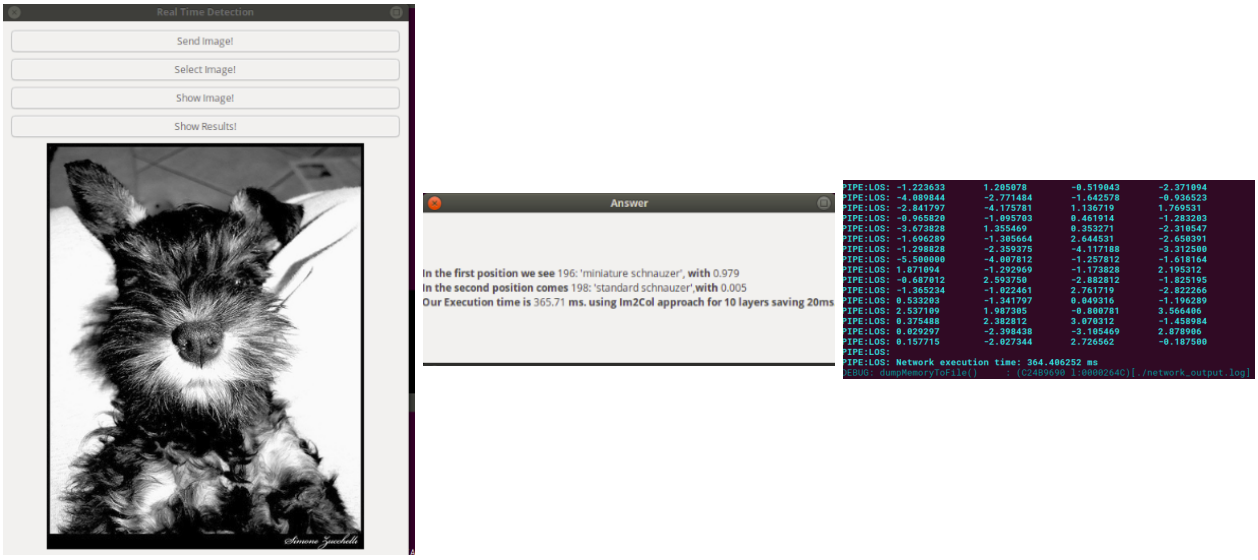
1.9.3 Εφαρμογή Πραγματικού Χρόνου Αναγνώρισης

Τέλος όλα τα παραπάνω συνδυάστηκαν και επιτεύχθηκε μια **εφαρμογή αναγνώρισης σε πραγματικό χρόνο**. Αυτή δέχεται οποιαδήποτε εικόνα, την επεξεργάζεται κατάλληλα χωρίς την χρήση της βιβλιοθήκης του Caffe αλλά με ισοδύναμες συναρτήσεις που υλοποιηθήκανε με python. Στη συνέχεια ξεκινάει επικοινωνία με τη χρήση sockets με τον υπολογιστή που βρίσκεται η Myriad2 και στέλνει την εικόνα μέσω του Ethernet. Το κομμάτι της επικοινωνίας στο socket γίνεται με τη χρησιμοποίηση συναρτήσεων

Πίνακας 1.2: Εκτελέσεις Νευωνικών Δικτύων του ImageNet

CNN	Execution Time(ms)	Energy Consumption(mJ)	layers	memory(MB)
AlexNet	98.3	125.6	13	117
GoogleNet	249.1	365.2	83	16.6
NiN-imagenet	244	335.7	16	15.5
SqueezeNet	85.5	126.7	38	4.68
VGG	586	961	16	276
ZFnet	99	130.3	13	121

μετατροπής του μηνύματος από Python σε C και αντίστροφα. Τα μηνύματα εισόδου η επεξεργασμένη εικόνα εισόδου και της εξόδου τα αποτελέσματα της ταξινόμησης και ο χρόνος εκτέλεσης. Παράδειγμα της εφαρμογής για την οποία υλοποιήθηκε και GUI σε Python G.T.K φαίνεται παρακάτω. Η εικόνα εισόδου είναι από το σύνολο δεδομένων ImageNet γι' αυτό η πρόβλεψη είναι τόσο επιτυχής και απόλυτα συμβατή με αυτή του Caffe. Ο χρόνος εκτέλεσης είναι για τυχαίο συνδυασμό από processing units και τρόπους υλοποίησης της συνέλιξης.



Σχήμα 1.18: Παράδειγμα ταξινόμησης με εφαρμογή πραγματικού χρόνου, όπου αριστερά φαίνεται το γραφικό περιβάλλον της εφαρμογής, στη μέση τα αποτελέσματα που επιστρέφει και δεξιά κάποια από τα αποτελέσματα της ταξινόμησης της Myriad2 για το GoogleNet που εκτελείται.

Κεφάλαιο 2

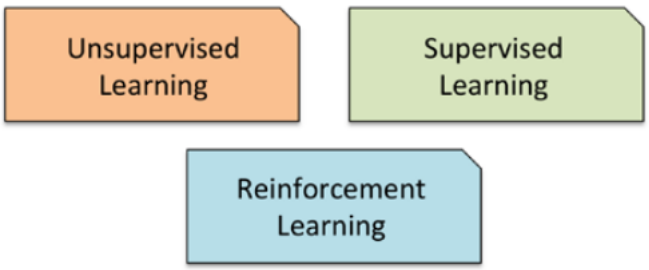
Building Intelligent Machines

2.1 Machine Learning

In this age of modern technology, there is one resource that there is 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 rulesand 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 even greater role in our everyday life. Thanks to machine learning applications can be seen in our every day life. Examples of these applications are 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 [13].

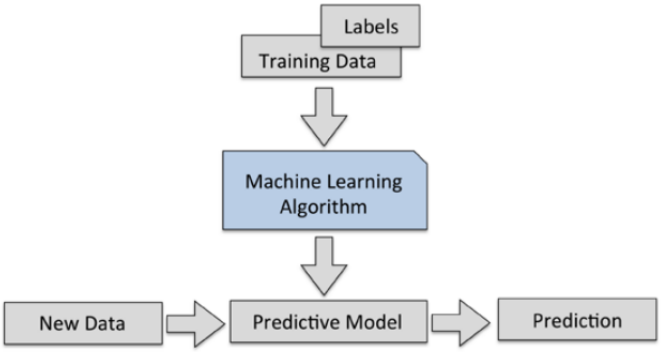
2.1.1 Three Different Types of Machine Learning

The three types of machine learning are: **supervised learning, unsupervised learning** and **reinforcement learning**.



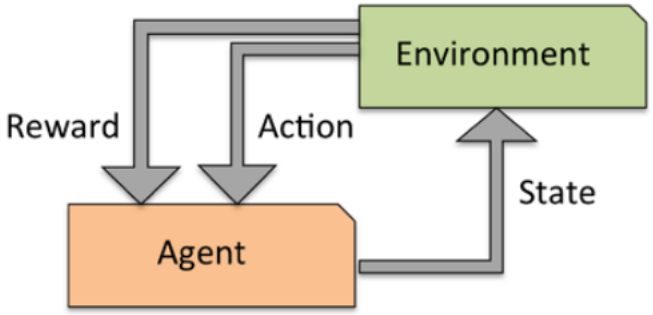
Σχήμα 2.1: Three Types of Machine Learning.

The main goal in supervised learning is to learn a model from labeled training data that allows us 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. Considering the example of e-mail spam filtering, a model can be trained using a supervised machine learning algorithm on a corpus of labeled e-mail, e-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 task. Another subcategory of supervised learning is regression, where the outcome signal is a continuous value:



Σχήμα 2.2: Supervised Machine Learning.

Another type of machine learning is reinforcement learning. In reinforcement learning, the goal is to develop a system (agent) that improves its performance based on interactions with the environment. Since the information about the current state of the environment typically also includes a so-called reward signal, reinforcement learning can be thought 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. A popular example of reinforcement learning is a chess engine. Here, the agent decides upon a series of moves depending on the state of the board (the environment), and the reward can be defined as win or lose at the end of the game:



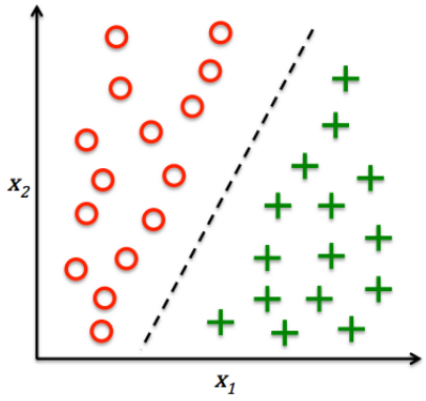
Σχήμα 2.3: Reinforcement Learning.

In supervised learning, the right answer is known beforehand the training of our model, and in reinforcement learning, a measure of reward for particular actions is defined by the agent. In unsupervised learning, however, dealing with unlabeled data or data of unknown structure is usual. With 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.

2.1.2 Classification and Regression in Supervised Learning

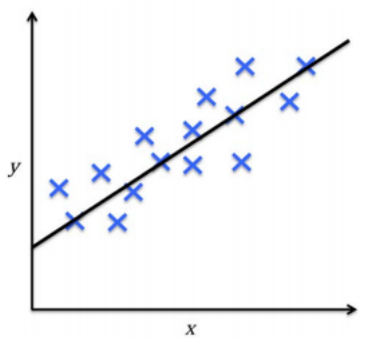
Classification is a subcategory of supervised learning where the goal is to predict the categorical class labels of new instances based on past observations. Those class labels are discrete, unordered values that can be understood as the group memberships of the instances. The previously mentioned example of e-mail-spam detection represents a typical example 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 can be collected that consists of multiple handwritten examples of each letter in the alphabet. 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, our machine learning system would be unable to correctly recognize any of the digits zero to nine, for example, if they were not part of our training dataset.
The following figure 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, our dataset is two-dimensional, which means that each sample has two values associated with it: x1 and x2 . Now, a supervised machine learning algorithm can be used to learn a rule that 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 x1 and x2 values:



Σχήμα 2.4: Binary classification Task.

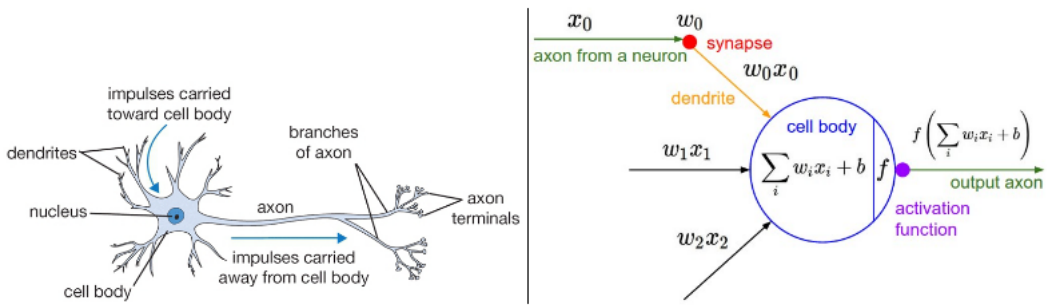
In **regression analysis**, a number of predictor (explanatory) variables and a continuous response variable (outcome) is given, and a relationship between those variables that allows us to predict an outcome as stated in [13] is trying to be found. For example, let's assume that there is interest in 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 can be used as training data to learn a model that uses the study time to predict the test scores of future students who are planning to take this test. The following figure illustrates the concept of linear regression. Given a predictor variable x and a response variable y, a straight line to this data is fitted that minimizes the distance, most commonly the average squared distance, between the sample points and the fitted line. The intercept and slope learned from this data can be used in order to predict the outcome variable of new data:



Σχήμα 2.5: Linear Regression.

2.2 Artificial Neural Networks

The area of Neural Networks has originally been primarily inspired by the goal of modeling biological neural systems, but has since diverged and become a matter of engineering and achieving good results in Machine Learning tasks. The basic computational unit of the brain is a **neuron**. Approximately 86 billion neurons can be found in the human nervous system and they are connected with approximately $10^{14} - 10^{15}$ synapses. The diagram below shows a cartoon drawing of a biological neuron (left) and a common mathematical model (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. x0) interact multiplicatively (e.g. w0x0) with the dendrites of the other neuron based on the synaptic strength at that synapse (e.g. w0). 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 can be 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 with an activation function f can be modeled. This function represents the frequency of the spikes along the axon.

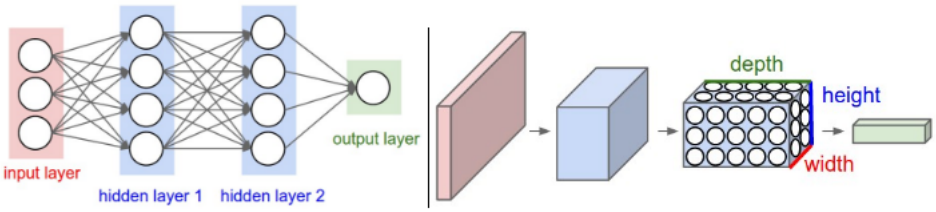


Σχήμα 2.6: Biological neuron (left) and its Mathematical model (right).

2.2.1 ANNs Architecture

Neural Networks as neurons in graphs. Neural Networks are modeled as collections of neurons that are connected in an acyclic graph. In other words, 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 an amorphous blobs of connected neurons, Neural Network 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. Below are two example Neural Network topologies that use a stack of fully-connected layers:



Σχήμα 2.7: Left: A 2-layer Neural Network (one hidden layer of 4 neurons (or units) and one output layer with 2 neurons), and three inputs. Right: A 3-layer neural network with three inputs, two hidden layers of 4 neurons each and one output layer. Notice that in both cases there are connections (synapses) between neurons across layers, but not within a layer.

Notice that when the N-layer neural network is referred, the input layer is not measured. Therefore, a single-layer neural network describes a network with no hidden layers (input directly mapped to output). In that sense, you can sometimes hear people say that logistic regression or SVMs are simply a special case of single-layer Neural Networks. You may also hear these networks interchangeably referred to as "Artificial Neural Networks" (ANN) or "Multi-Layer Perceptrons" (MLP). Many people do not like the analogies between Neural Networks and real brains and prefer to refer to neurons as units.
Output layer. Unlike all layers in a Neural Network, the output layer neurons most commonly do not have an activation function (or you can think of them 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). Modern Convolutional Networks contain on orders of 100 million parameters and are usually made up of approximately 10-20 layers (hence deep learning). However, the number of effective connections is significantly greater due to parameter sharing. More on this in the Convolutional Neural Networks module.

2.3 Convolutional Neural Networks

Convolutional Neural Networks [10] are very similar to ordinary Neural Networks from the previous chapter: they are made up of neurons that have learnable weights and biases. Each neuron receives some inputs, performs a dot product and optionally follows it with a non-linearity. The whole network still expresses a single differentiable score function: from the raw image pixels on one end to class scores at the other. And they still have a loss function (e.g. SVM/Softmax) on the last (fully-connected) layer and all the tips/tricks, which were developed for learning regular Neural Networks still apply. ConvNet architectures make the explicit assumption that the inputs are images, which allows us to encode certain properties into the architecture. These then make the forward function more efficient to implement and vastly reduce the amount of parameters in the network. As seen in the previous chapter, Neural Networks receive an input (a single vector), and transform it through a series of hidden layers. Each hidden layer is made up of a set of neurons, where each neuron is fully connected to all neurons in the previous layer, and where neurons in a single layer function completely independently and do not share any connections. The last fully-connected layer is called the "output layer" and in classification settings it represents the class scores.
Regular Neural Nets don't scale well to full images. In CIFAR-10, images are only of size 32x32x3 (32 wide, 32 high, 3 color channels), so a single fully-connected neuron in a first hidden layer of a regular

Neural Network would have 32*32*3 = 3072 weights. This amount still seems manageable, but clearly this fully-connected structure does not scale to larger images. For example, an image of more respectable size, e.g. 200x200x3, would lead to neurons that have 200*200*3 = 120,000 weights. Moreover, it is useful to have several such neurons, so the parameters would add up quickly! Clearly, this full connectivity is wasteful and the huge number of parameters would quickly lead to overfitting. Convolutional Neural Networks take advantage of the fact that the input consists of images and they constrain the architecture in a more sensible way. In particular, unlike a regular Neural Network, the layers of a ConvNet 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 full Neural Network, which can refer to the total number of layers in a network.) For example, the input images in CIFAR-10 are an input volume of activations, and the volume has dimensions 32x32x3 (width, height, depth respectively). As quickly is going to be proved, 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. Moreover, the final output layer would for CIFAR-10 have dimensions 1x1x10, because by the end of the ConvNet architecture the full image into a single vector of class scores will be reduced. This image will be arranged along the depth dimension.

2.3.1 Layers of CNNs

Convolutional Layer is the core building block of a Convolutional Network that does most of the computational heavy lifting.
Convolution layer's parameters consist of a set of learnable filters. Every filter is small spatially (along width and height), but extends through the full depth of the input volume. For example, a typical filter on a first layer of a ConvNet might have size 5x5x3 (i.e. 5 pixels width and height, and 3 because images have depth 3, the color channels). During the forward pass, each filter across the width and height of the input volume is slided (more precisely, convolved) and compute dot products between the entries of the filter and the input at any position. As the filter is slided over the width and height of the input volume a 2-dimensional activation map that gives the responses of that filter at every spatial position is produced. Intuitively, the network will learn filters that activate when they see some type of visual feature such as an edge of some orientation or a blotch of some color on the first layer, or eventually entire honeycomb or wheel-like patterns on higher layers of the network. Now, there is an entire set of filters in each CONV layer (e.g. 12 filters), and each of them will produce a separate 2-dimensional activation map. All these activation maps along the depth dimension are stacked and produce the output volume.
When dealing with high-dimensional inputs such as images, it was proven that it is impractical to connect neurons to all neurons in the previous volume. Instead, it is better each neuron to be 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 (equivalently this is the filter size). 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) are treated and the depth dimension: The connections are local in space (along width and height), but always full along the entire depth of the input volume.
Right now it has been specified how many neurons there are in the output. In order to see how they are arranged the following **hyperparameters** have to be explained according to [10]:

- The **depth** of the output volume is a hyperparameter: it corresponds to the number of filters that is going to be used. Each filter is 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. A set of neurons that are all looking at the same region of the input is a depth column.

- The **stride** with which the filter is slided. When the stride is 1, then the filters are moved one pixel at a time. When the stride is 2 (or uncommonly 3 or more, though this is rare in practice) then the filters jump 2 pixels at a time. This will produce smaller output volumes spatially.

- Sometimes it will be convenient to pad the input volume with zeros around the border. The size of this **zero-padding** is a hyperparameter. The nice feature of zero padding is that it will allow us to control the spatial size of the output volumes (most commonly this parameter is used to exactly preserve the spatial size of the input volume so the input and output width and height are the same).

- **Parameter sharing** scheme is used in Convolutional Layers to control the number of parameters. The number of parameters can dramatically be reduced by making one reasonable assumption: That if one feature is useful to compute at some spatial position (x1,y1), then it should also be useful to compute at a different position (x2,y2).

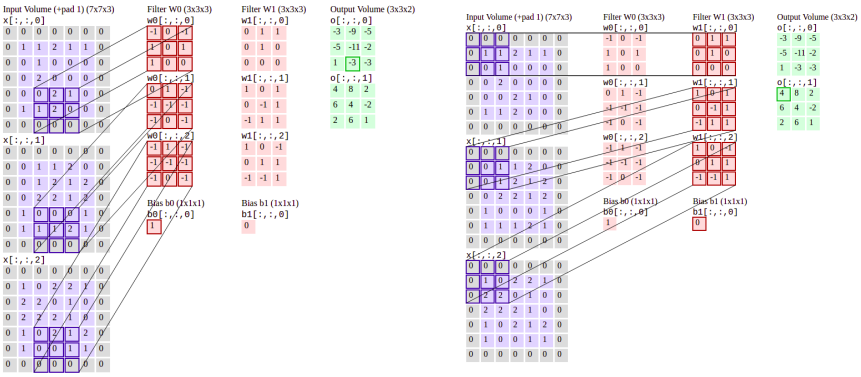
A convolutional's layer example is shown below, where parameters have the following values:

- The depth of the output volume is equal with the number of different convolutional filters, which means that it is 2.

- Striding parameter is equal with the unit as the filters slide one pixel per time and the output is a square of 3x3 dimensions.

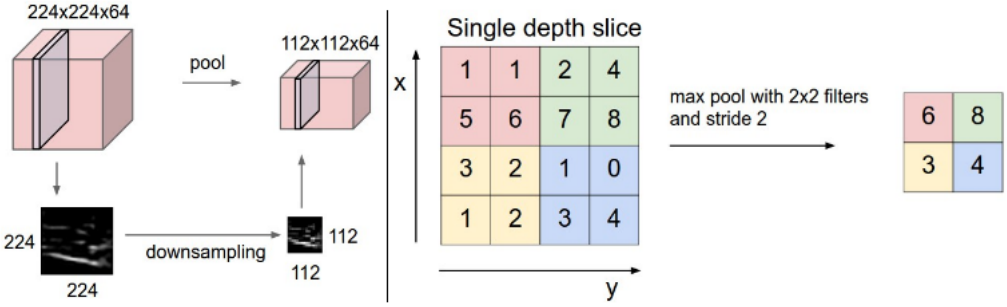
- Here a zero-padding parameter is used and its value is one. The reason is that the output has to keep information which will be used along the way of the CNN's execution.

- From the example is is not obvious to define if a sharing parameter is used. The most known is grouping parameter, whose functionality is going to be explained later.



Σχήμα 2.8: Example of Direct Convolution, in the left first output map, right second one.

It is common to periodically insert a **Pooling Layer** in-between successive Conv layers in a ConvNet architecture. Its function is to progressively reduce the spatial size of the representation to reduce the amount of parameters and computation in the network, and hence to also control overfitting. The Pooling Layer operates independently on every depth slice of the input and resizes it spatially, using operations like Max, Average, etc. The most common form is a pooling layer with filters of size 2x2 applied with a stride of 2 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 max over 4 numbers (little 2x2 region in some depth slice). It is worth noting that there are only two commonly seen variations of the max pooling layer found in practice: A pooling layer with $kernel_{size} = 3, stride = 2$ (also called overlapping pooling), and more commonly $kernel_{size} = 2, stride = 2$. Pooling sizes with larger receptive fields are too destructive.
Last but not least, in addition to max pooling there is **General pooling** too. The pooling units can also perform other functions, such as average pooling or even L2-norm pooling. Average pooling was often used historically but has recently fallen out of favor compared to the max pooling operation, which has been shown to work better in practice.



Σχήμα 2.9: Pooling layers downsamples the Input.

Neurons in a **fully connected layer** have full connections to all activations in the previous layer, as seen in regular Neural Networks. Their activations can hence be computed with a matrix multiplication followed by a bias offset.
Many types of **normalization layers** have been proposed for use in ConvNet architectures, sometimes with the intentions of implementing inhibition schemes observed in the biological brain. However, these layers have since fallen out of favor because in practice their contribution has been shown to be minimal, if any.

2.4 Development of CNNs

After the establishment of the CNNs for object's classification applications, a lot of research was invested to find ways not only to classify one object in an image but many objects and also detect them. Object detection is the task of finding the different objects in an image and classifying them. This function happens with the use of algorithms known as **Regions with CNNs** (R-CNNs). The goal of R-CNN is to take in an image, and correctly identify where the main objects in the image are, so a R-CNN functios as follows:

- Input: Image
- Output: Bounding boxes + labels for each object in the image.

R-CNN creates these bounding boxes, or region proposals, using a process called Selective Search which you can read about in [4]. An R-CNN arcitecture uses the following steps:

- Generate a set of proposals for bounding boxes.
- Run the images in the bounding boxes through a pre-trained AlexNet.
- Run the box through a linear regression model to output tighter coordinates for the box once the object has been classified.

Testing R-CNNs has the drawback of slow velocity due to many executions, one for every bounding box. In order to overcome this, researchers were driven to **Fast** and **Faster R-CNNs** as stated in [4] using techniques such as Region of Interest Pooling in order to decrease the overall execution time.

Κεφάλαιο 3

Introduction to CAFFE, Myriad2 and Tegra Jetson TX1

This chapter is going to introduce basic terminology about the software and the hardware used for the **CNN implementation**. Useful information about the ImageNet's CNNs is provided as well.

3.1 Convolutional Architecture for Fast Feature Embedding

Caffe is a deep learning framework made with expression, speed, and modularity in mind. It is developed by Berkeley AI Research (BAIR) and by community contributors. Yangqing Jia created the project during his PhD at UC Berkeley. Caffe is released under the BSD 2-Clause license. Caffe is an anvanced framework with the following advantages as stated in [9]:

- **Expressive architecture** encourages application and innovation. Models and optimization are defined by configuration without hard-coding. Switch between CPU and GPU by setting a single flag to train on a GPU machine then deploy to commodity clusters or mobile devices.

- **Extensible code** fosters active development. In Caffe's first year, it has been forked by over 1,000 developers and had many significant changes contributed back. Thanks to these contributors the framework tracks the state-of-the-art in both code and models.

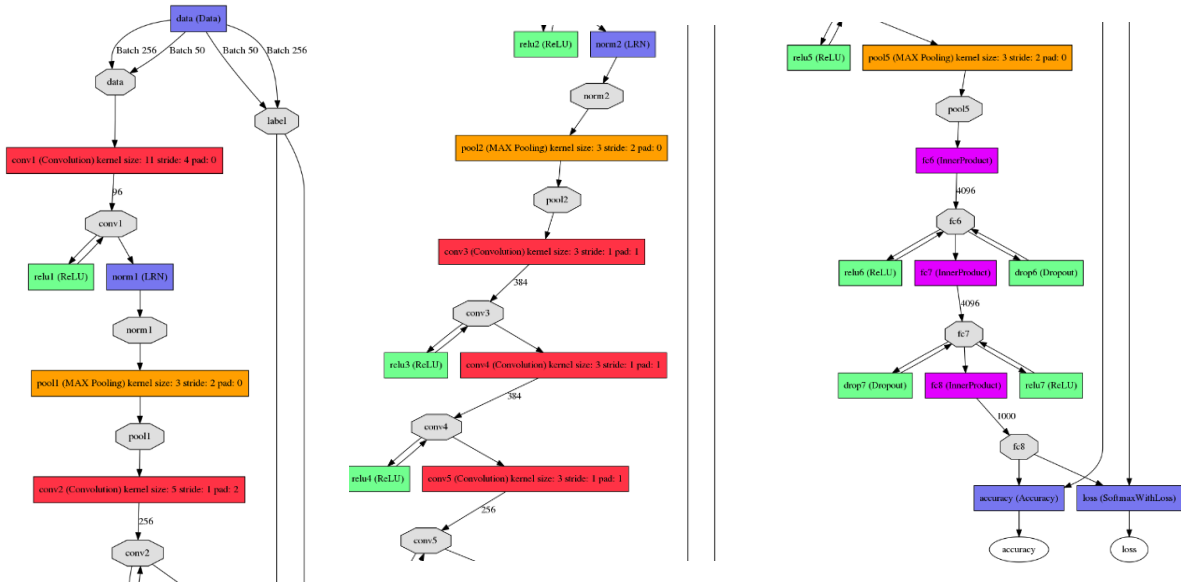
- The framework is a BSD-licensed C++ library with Python and MATLAB bindings for training and deploying general-purpose convolutional neural networks and other deep models efficiently on commodity architectures. Caffe fits industry and internet-scale media needs by CUDA GPU computation, processing over 40 million images a day on a single K40 or Titan GPU (2.5 ms per image).

3.1.1 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 (example in fig.3.1), 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.

3.1.2 Training a Network

Caffe **trains** models by the fast and standard stochastic gradient descent algorithm. Figure shows a typical example of a Caffe network **(AlexNet)** for visual recognition tasks. During training, a data layer fetches the images and labels from disk, passes it through multiple layers such as convolution, pooling, normalization and inner products 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 "models/bvlc_alexnet/train_val.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.



Σχήμα 3.1: Training Prototxt of AlexNet, where the colored boxes represent layers and the gray octagons represent data blobs produced by or fed into the layers.

3.1.3 Testing of a Network

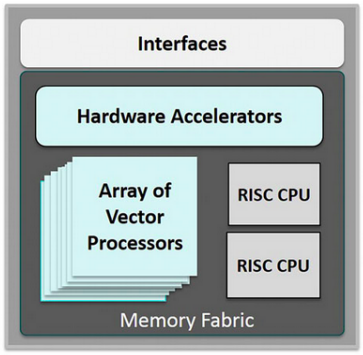
Caffe supports **testing** of any Convolutional Neural Network by using two files. The first file named **deploy.prototxt** desrcibes the architecture of the Network. The .prototxt file has two different formats for serialized data (textual or binary). The text format is human-readable and modifiable (and the corresponding files usually have the extension .prototxt), but it takes up a lot more space than the binary format. On the other hand, **.caffemodel** file comes as a result from the Network training and it contains both the weights of the appropriate layers(e.g. Convolutional and InnerProduct) and the biases. This file is on binary form in order to allocate less memory. Furthermore, except for these two files that need to be provoked, an appropriate **input** in every Convolutional Neural Network has to be given as well. This input is an image, which is provided through preprocessing with Python Libraries.

3.2 Description of Myriad2 multiprocessor SoC

The Myriad2 SoC [2] is developed by Movidius Ltd, that recently joined Intel's Perceptual Computing Group to accelerate adoption of visually intelligent devices. The Intel Movidius Myriad 2 VPU is the industry's first always-on vision processor. It delivers high-performance machine vision and visual awareness in severely power-constrained enviroments. Standing at the intersection of **low-power and high performance**, the Myriad 2 family of processors are transforming the capabilities of devices. Myriad 2 gives developers immediate access to its advanced vision processing core, while

allowing them to develop proprietary capabilities that provide true differentiation. Benefits that the Myriad 2 VPU offers are:

- An **ultra-low power design**. For mobile and connected devices where battery life is critical, Intel's Myriad™ 2 VPU 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**. Important is bringing vision technologies in connected devices closer to the capabilities of human vision. Intel's Myriad™ 2 VPU 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 Intel® Movidius™ Myriad™ 2. Our 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, Intel's Myriad™ 2 VPU was designed with a very small footprint that can easily be integrated into existing products.
- Additional Chip Details. The Intel® Movidius™ Myriad™ 2 architecture comprises a complete set of interfaces, a set of enhanced imaging/vision accelerators, a group of 12 specialized vector VLIW processors called SHAVEs, and an intelligent memory fabric that pulls together the processing resources to enable power efficient processing.



Σχήμα 3.2: Hardware Parts of Myriad2.

3.2.1 Uniqure VPU Architecture

A brief overview of the Myriad 2 common features are presented below:

- 12 x SHAVEVLIW vector processor, 2 x RISC processor
- There is 2 MB of on-chip RAM (CMX)
- 128/512 MB of in-package stacked DDR
- LEON RISC has 256 KB L2 cache memory
- LEON RT has 32 KB L2 cache memory
- Exceptionally high sustainable on-chip bandwidth
- SIPP Image Signal Processing hardware accelerators
- Wide range of IO peripherals interfaces, such as SPI, I2C, I2S, SDIO, Ethernet, USB

- Imaging interfaces, such as MIPI, CIF, LCD

The Myriad 2 family consists of the following socket revisions:

- MA2x5x { MA2150 / MA2155 / MA2450 / MA2455

The design principles for Intel® Movidius™ Myriad™ VPUs follows from a careful balance of programmable vector-processors, dedicated hardware accelerators and memory architecture for optimized data flow. Myriad VPUs feature a software-controlled, multi-core, multi-ported memory subsystem and caches which can be configured to allow a large range of workloads. This proprietary technology allows for exceptionally high sustainable on-chip data and instruction bandwidth to support the array of SHAVE processors, 2 CPUs and high-performance video hardware accelerators as extracted from [1]:

- **LeonOS** is the main processor, because after booting the device the execution of the program starts from OS, which belong to the CPU Sub System(CSS). The CSS have 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. Leon OS (LOS) RISC processor is the control unit of this block, 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.

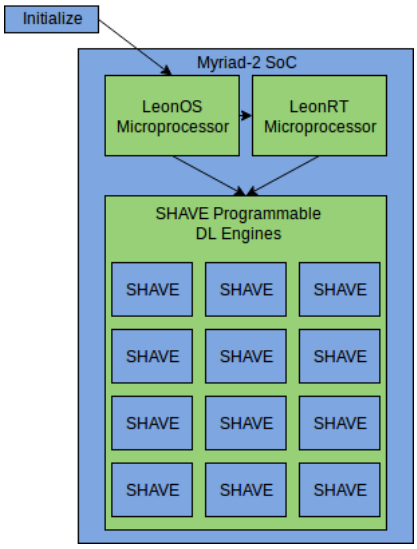
- **LeonRT** is the additional coordinator and belongs to the Media Sub System(MSS). The MSS is the architectural unit designed for allowing external connections with imaging devices (camera sensors, LCDs, HDMI controllers etc.) as well as allowing use of the HW filters available in Myriad 2. As such 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 coordination effort. As such the Myriad 2 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.

- **SHAVEs** have a computation-role and they are managed by the two SPARC processors. In order to guarantee sustained high performance and minimize power, the Movidius proprietary processor called SHAVE (Streaming Hybrid Architecture Vector Engine) contains wide and deep register-files coupled with a Variable-Length Long Instruction-Word (VLLIW) controlling multiple functional units including extensive SIMD capability for high parallelism and throughput at both a functional unit and processor level. The SHAVE processor is a hybrid stream processor architecture combining the best features of GPUs, DSPs and RISC with both 8/16/32 bit integer and 16/32 bit floating point arithmetic as well as unique features such as hardware support for sparse data structures. The architecture is designed to maximize performance-per-watt while maintaining ease of programmability, especially in terms of computer vision and machine learning workloads.

As far as the memory of the chip is concerned, Myriad 2 provides both DDR Memory and CMX Memory and ways to share data between them:

- **DDR** is the main memory of the chip, both LeonOS and LeonRT may execute from DDR with minimum penalty due to optimally choosen cache sizes. This is the place, where heavy data, like weights of a DNN can be stored statically.

- The **CMX** acronym comes from Connection Matrix, which belies the fact it is comprised of several smaller SRAM blocks. The CMX memory of 2 MB may be considered as 16x128 KB ‘slices’. Each SHAVE processor has preferential ports into a 128 KB slice of the CMX memory.

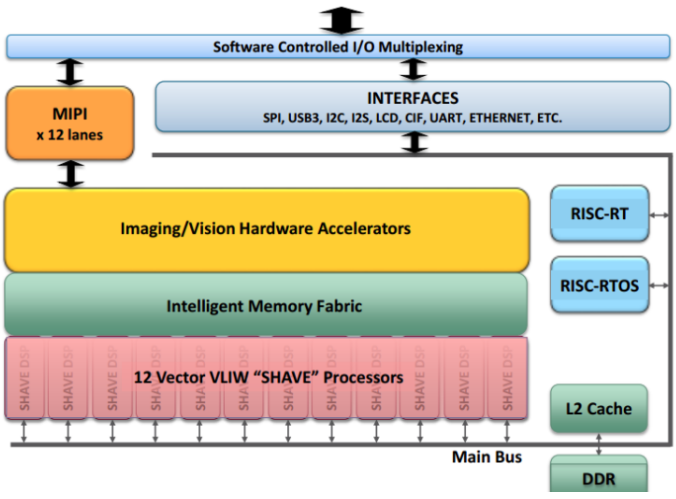


Σχήμα 3.3: Processors of Myriad 2.

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 resources. Usually, this area is used by CMX-DMA driver or even by HW SIPP filters usage or Leon OS timing critical code which would not be able to be kept in DDR.

- The **CMX DMA** resides between the 128-bit MXI bus and CMX memory. It provides for scheduling high-bandwidth data transfers between CMX and DRAM in either direction. It also supports data transfers from DRAM back to DRAM or from CMX to CMX, allowing data to be relocated within the same physical location. The CMX DMA engine processes a linked-list of DMA descriptors, which are created by the driver.

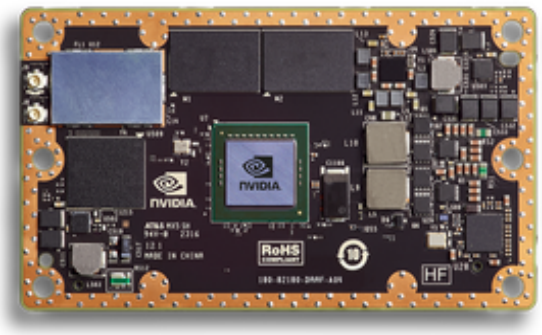
- **SIPP** is a proprietary software/hardware mechanism used by the Myriad2 processor to achieve highly optimized scheduling of Image Signal Processing (ISP) pipeline functionality. This mechanism is responsible for utilizing the HW filters provided by Myriad2 to achieve the best performance possible.



Σχήμα 3.4: Detailed Overview of Myriad's Hardware.

3.3 Description of NVIDIA Tegra Jetson TX1

A powerful supercomputer on a module, **Jetson TX1** is capable of delivering the performance and power efficiency needed for the latest visual computing applications according to [3]. It's built around the revolutionary NVIDIA Maxwell^{TM} architecture with 256 CUDA cores delivering over 1 TeraFLOPs of performance. 64-bit CPUs, 4K video endcode and decode capabilities, and a camera interface capable of 1400 MPix/s make this the best system for embedded deep learning, computer vision, graphics, and GPU computing.



Σχήμα 3.5: Jetson TX1 Module.

3.3.1 Building AI Applications with Tegra

NVIDIA JetPack SDK is the most comprehensive solution for building AI applications. Key Features in JetPack:

- **TensorRT** is a high performance deep learning inference runtime for image classification, segmentation, and object detection neural networks. It speeds up deep learning inference as well as reducing the runtime memory footprint for convolutional and deconvolutional neural networks.

- CUDA Deep Neural Network **(cuDNN)** library provides high-performance primitives for all deep learning frameworks. It includes support for convolutions, activation functions and tensor transformations.

- CUDA Toolkit provides a comprehensive development environment for C and C++ developers building GPU-accelerated applications. The toolkit includes a compiler for NVIDIA GPUs, math libraries, and tools for debugging and optimizing the performance of your applications.

- **VisionWorks** is a software development package for Computer Vision (CV) and image processing. It Includes VPI (Vision Programming Interface), a set of optimized CV primitives for use by CUDA developers. The NVX library enables direct access to VPI, and the OVX library enables indirect access to VPI via OpenVX framework.

- The NVIDIA® Tegra® Linux (L4T) Driver Package supports development on the Jetson Platform.

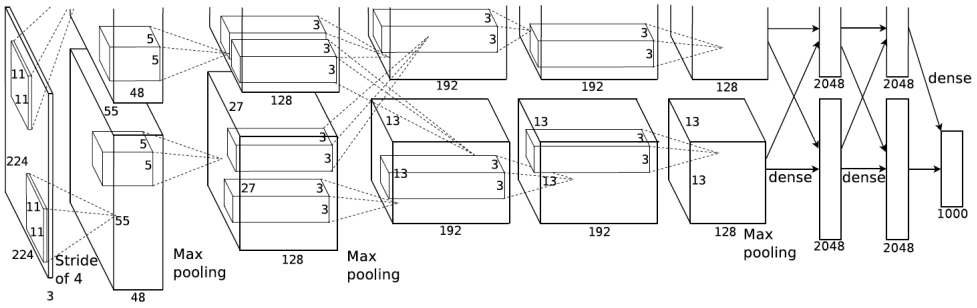
- **OpenCV** (Open Source Computer Vision Library) is an open source computer vision and machine learning software library. OpenCV was built to provide a common infrastructure for computer vision applications and to accelerate the use of machine perception in the commercial products. Being a BSD-licensed product, OpenCV makes it easy for businesses to utilize and modify the code.

3.4 Description of Imagenet CNNs

Convolutional neural networks are fantastic for visual recognition tasks. Good ConvNets are beasts with millions of parameters and many hidden layers. In fact, a bad rule of thumb is: 'higher the number of hidden layers, better the network'. Network architecture design is a complicated process and will take a while to learn and even longer to experiment designing on your own. Most ConvNets have huge memory and computation requirements, especially while training. Hence, this becomes an important concern. Similarly, the size of the final trained model becomes an important to consider if you are looking to deploy a model to run locally on mobile. As you can guess, it takes a more computationally intensive network to produce more accuracy. So, there is always a trade-off between accuracy and computation. Apart from these, there are many other factors like ease of training, the ability of a network to generalize well etc. The networks described below are the most popular ones and are presented in the order that they were published and also had increasingly better accuracy from the earlier ones as stated in [5].

3.4.1 AlexNet

The one that started it all (Though some may say that Yann LeCun's paper in 1998 was the real pioneering publication). This paper, titled [7], has been cited a total of 6,184 times and is widely regarded as one of the most influential publications in the field. Alex Krizhevsky, Ilya Sutskever, and Geoffrey Hinton created a "large, deep convolutional neural network" that was used to win the 2012 ILSVRC (ImageNet Large-Scale Visual Recognition Challenge). For those that aren't familiar, this competition can be thought of as the annual Olympics of computer vision, where teams from across the world compete to see who has the best computer vision model for tasks such as classification, localization, detection, and more. 2012 marked the first year where a CNN was used to achieve a top 5 test error rate of 15.4 percent. (Top 5 error is the rate at which, given an image, the model does not output the correct label with its top 5 predictions). The next best entry achieved an error of 26.2 percent, which was an astounding improvement that pretty much shocked the computer vision community. Safe to say, CNNs became household names in the competition from then on out.
In the paper, the group discussed the architecture of the network (which was called AlexNet). They used a relatively simple layout, compared to modern architectures. The network was made up of 5 conv layers, max-pooling layers, dropout layers, and 3 fully connected layers. The network they designed was used for classification with 1000 possible categories.



Σχήμα 3.6: AlexNet Architecture with two "streams", due to computationally expensive training process.

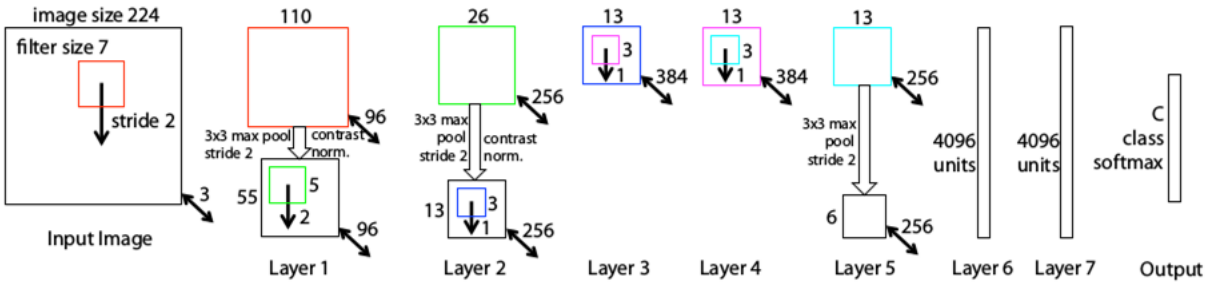
- Trained the network on ImageNet data, which contained over 15 million annotated images from a total of over 22,000 categories.
- Used ReLU for the nonlinearity functions (Found to decrease training time as ReLUs are several times faster than the conventional tanh function).

- Used data augmentation techniques that consisted of image translations, horizontal reflections, and patch extractions.
- Implemented dropout layers in order to combat the problem of overfitting to the training data.
- Trained the model using batch stochastic gradient descent, with specific values for momentum and weight decay.
- Trained on two GTX 580 GPUs for five to six days.

The neural network developed by Krizhevsky, Sutskever, and Hinton in 2012 was the coming out party for CNNs in the computer vision community. This was the first time a model performed so well on a historically difficult ImageNet dataset. Utilizing techniques that are still used today, such as data augmentation and dropout, this paper really illustrated the benefits of CNNs and backed them up with record breaking performance in the competition.

3.4.2 ZF Net

With AlexNet stealing the show in 2012, there was a large increase in the number of CNN models submitted to ILSVRC 2013. The winner of the competition that year was a network built by Matthew Zeiler and Rob Fergus from NYU. Named ZF Net [6], this model achieved an 11.2 percent error rate. This architecture was more of a fine tuning to the previous AlexNet structure, but still developed some very keys ideas about improving performance. Another reason this was such a great paper is that the authors spent a good amount of time explaining a lot of the intuition behind ConvNets and showing how to visualize the filters and weights correctly.
In this paper titled "Visualizing and Understanding Convolutional Neural Networks", Zeiler and Fergus begin by discussing the idea that this renewed interest in CNNs is due to the accessibility of large training sets and increased computational power with the usage of GPUs. They also talk about the limited knowledge that researchers had on inner mechanisms of these models, saying that without this insight, the "development of better models is reduced to trial and error". While currently there is a better understanding than 3 years ago, this still remains an issue for a lot of researchers! The main contributions of this paper are details of a slightly modified AlexNet model and a very interesting way of visualizing feature maps.



Σχήμα 3.7: ZF Net architecture.

- Very similar architecture to AlexNet, except for a few minor modifications.
- AlexNet trained on 15 million images, while ZF Net trained on only 1.3 million images.
- Instead of using 11x11 sized filters in the first layer (which is what AlexNet implemented), ZF Net used filters of size 7x7 and a decreased stride value. The reasoning behind this modification is that a smaller filter size in the first conv layer helps retain a lot of original pixel information

in the input volume. A filtering of size 11x11 proved to be skipping a lot of relevant information, especially as this is the first conv layer.

- As the network grows, also a rise in the number of filters used is observed.
- Used ReLUs for their activation functions, cross-entropy loss for the error function, and trained using batch stochastic gradient descent.
- Trained on a GTX 580 GPU for twelve days.
- Developed a visualization technique named Deconvolutional Network, which helps to examine different feature activations and their relation to the input space. Called "deconvnet" because it maps features to pixels (the opposite of what a convolutional layer does).

The basic idea behind how this works is that at every layer of the trained CNN, you attach a "deconvnet" which has a path back to the image pixels. An input image is fed into the CNN and activations are computed at each level. This is the forward pass. Now, let's say that there is a need to examine the activations of a certain feature in the 4th conv layer. The activations of this one feature map need to be stored and set all the other activations in the layer to 0.Then pass this feature map as the input into the deconvnet. This deconvnet has the same filters as the original CNN. This input then goes through a series of unpool (reverse maxpooling), rectify, and filter operations for each preceding layer until the input space is reached. ZF Net was not only the winner of the competition in 2013, but also provided great intuition as to the workings on CNNs and illustrated more ways to improve performance. The visualization approach described helps not only to explain the inner workings of CNNs, but also provides insight for improvements to network architectures.

3.4.3 Network in Network

It is interesting how the convolution filters are designed and how extracted features to class scores are mapped. This formed the basis of the Inception architecture. Two new concepts were introduced in this CNN architecture design according to [17]:

- MLPconv: Replaced linear filters with nonlinear Multi Linear Perceptrons to extract better features within the receipt field (see the figure above). This helped in better abstraction and accuracy.
- Global Average Pooling: Got rid of the fully connected layers at the end thereby reducing parameters and complexity. This was replaced by the creation of as many activation maps in the last layer as there are classes. This was followed by averaging these maps to arrive at final scores, which is passed to softmax. This is performant and more intuitive.

Network in network introduced the concept of having a neural network itself in place of a convolution filter. The input to this mini network would be the convolution, and the output would be the value of a neuron in the activation. Hence it does not alter the input/output characteristics of traditional filters. This mini network, called MLPconv, can then convolved over the input. The benefit of having such an arrangement is two-fold:

- It is compatible with the backpropagation logic of neural nets, thus this fits well into existing architectures of CNN's.
- It can itself be a deep model leading to rich separation between latent features.

In traditional CNN architectures, the feature maps of the last convolution layer are flattened and passed on to one or more fully connected layers, which are then passed on to softmax logistics layer for spitting out class probabilities. The issue with this approach is that it is hard to decode how the usual fully connected layers seen at the end of CNN architectures map to class probabilities. They are black boxes between the convolution layers and the classifier. They are also prone to overfitting and come with lots of parameters to train. An estimate says that the last FC layers contain 90 percent

of the parameters of the network. The last MLPconv layer produces as many activation maps as the number of classes being predicted. Then, each map is averaged giving rise to the raw scores of the classes. These are then fed to a SoftMax layer to produce the probabilities, totally making FC layers redundant. The advantages of this approach are:

- The mapping between the extracted features and the class scores is more intuitive and direct. The feature can be treated as category confidence.
- An implicit advantage is that there are no new parameters to train (unlike the FC layers), leading to less overfitting.
- Global average pooling sums out the spatial information, thus it is more robust to spatial translations of the input.

3.4.4 VGG Net

Simplicity and depth. That's what a model created in 2014 (weren't the winners of ILSVRC 2014) best utilized with its 7.3 percent error rate. Karen Simonyan and Andrew Zisserman of the University of Oxford created a 19 layer CNN that strictly used 3x3 filters with stride and pad of 1, along with 2x2 maxpooling layers with stride 2.

ConvNet Configuration					
A	A-LRN	B	C	D	E
11 weight layers	11 weight layers	13 weight layers	16 weight layers	16 weight layers	19 weight layers
input (224 × 224 RGB image)					
conv3-64	conv3-64 **LRN**	conv3-64 **conv3-64**	conv3-64 conv3-64	conv3-64 conv3-64	conv3-64 conv3-64
maxpool					
conv3-128	conv3-128	conv3-128 **conv3-128**	conv3-128 conv3-128	conv3-128 conv3-128	conv3-128 conv3-128
maxpool					
conv3-256 conv3-256	conv3-256 conv3-256	conv3-256 conv3-256	conv3-256 conv3-256 **conv1-256**	conv3-256 conv3-256 **conv3-256**	conv3-256 conv3-256 conv3-256 **conv3-256**
maxpool					
conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512 **conv1-512**	conv3-512 conv3-512 **conv3-512**	conv3-512 conv3-512 conv3-512 **conv3-512**
maxpool					
conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512 **conv1-512**	conv3-512 conv3-512 **conv3-512**	conv3-512 conv3-512 conv3-512 **conv3-512**
maxpool					
FC-4096					
FC-4096					
FC-1000					
soft-max					

Σχήμα 3.8: The 6 different versions of VGG Net, with configuration D giving the best results.

- The use of only 3x3 sized filters is quite different from AlexNet's 11x11 filters in the first layer and ZF Net's 7x7 filters. The authors' reasoning is that the combination of two 3x3 conv layers has an effective receptive field of 5x5. This in turn simulates a larger filter while keeping the benefits of smaller filter sizes. One of the benefits is a decrease in the number of parameters. Also, with two conv layers, we're able to use two ReLU layers instead of one.
- 3 conv layers back to back have an effective receptive field of 7x7.
- As the spatial size of the input volumes at each layer decrease (result of the conv and pool layers), the depth of the volumes increase due to the increased number of filters as you go down the network.

- Interesting to notice that the number of filters doubles after each maxpool layer. This reinforces the idea of shrinking spatial dimensions, but growing depth.

- Built model with the Caffe toolbox.

- Used scale jittering as one data augmentation technique during training.

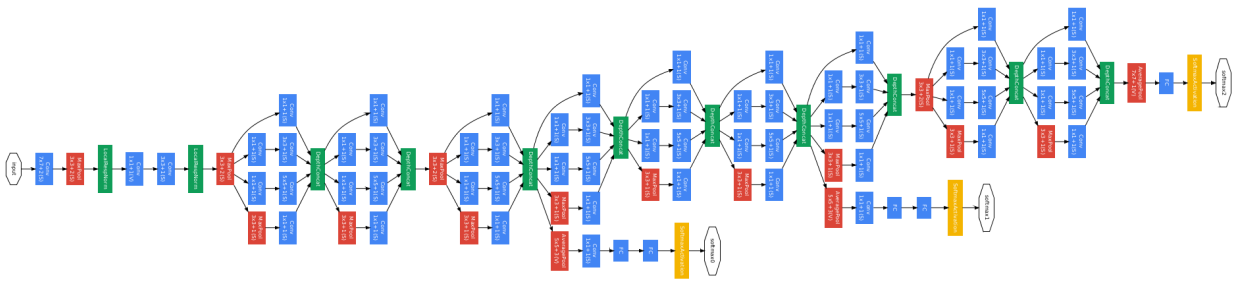
- Used ReLU layers after each conv layer and trained with batch gradient descent.

- Trained on 4 Nvidia Titan Black GPUs for two to three weeks.

VGG Net is one of the most influential papers [18], because it reinforced the notion that convolutional neural networks have to have a deep network of layers in order for this hierarchical representation of visual data to work. Keep it deep. Keep it simple.

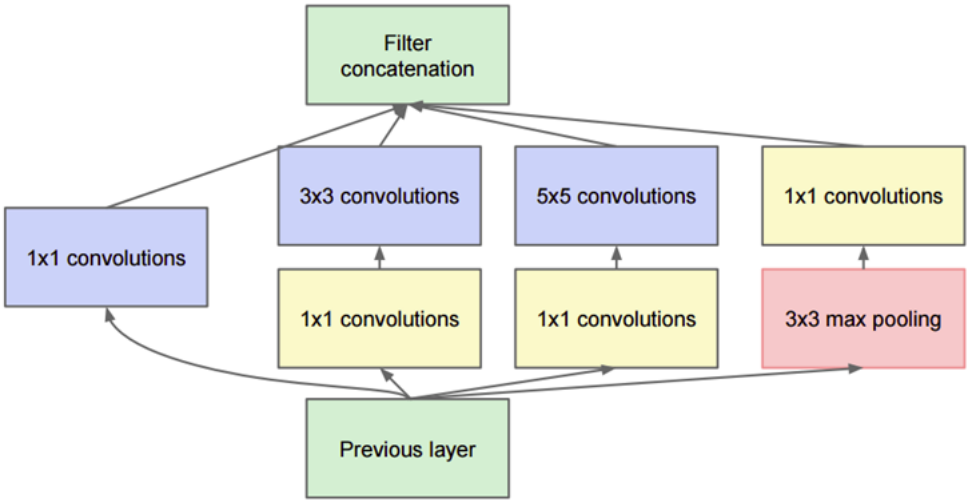
3.4.5 GoogleNet

The idea of simplicity in network architecture, Google threw it out the window with the introduction of the Inception module. GoogLeNet is a 83 layer CNN and was the winner of ILSVRC 2014 with a top 5 error rate of 6.7 percent. This was one of the first CNN architectures that really strayed from the general approach of simply stacking conv and pooling layers on top of each other in a sequential structure. The authors of the paper [12] also emphasized that this new model places notable consideration on memory and power usage.



Σχήμα 3.9: GoogleNet's Architecture.

After looking at the structure of GoogLeNet, someone can notice immediately that not everything is happening sequentially, as seen in previous architectures. Pieces of the network are happening in parallel. The box with the parallel layers is named inception module, let's take a closer look at what it's made of.



Σχήμα 3.10: Full Inception Module.

The bottom green box is our input and the top one is the output of the model (Turning the above picture right 90 degrees will offer the visualization of the model in relation to the last picture which shows the full network). Basically, at each layer of a traditional ConvNet, you have to make a choice of whether to have a pooling operation or a conv operation (there is also the choice of filter size). What an Inception module offers is that all of these operations can be performed in parallel. In fact, this was exactly the "naive" idea that the authors came up with. They thought to remove 1x1 convolutions, but following this way lead to too many outputs. Following this methodology would end up with an extremely large depth channel for the output volume. The way that the authors address this is by adding 1x1 conv operations before the 3x3 and 5x5 layers. The 1x1 convolutions (or network in network layer) provide a method of dimensionality reduction.

- Used 9 Inception modules in the whole architecture, with over 100 layers in total.
- No use of fully connected layers. They use an average pool instead, to go from a 7x7x1024 volume to a 1x1x1024 volume. This saves a huge number of parameters.
- Uses 12x fewer parameters than AlexNet.
- During testing, multiple crops of the same image were created, fed into the network, and the softmax probabilities were averaged to give us the final solution.
- There are updated versions to the Inception module.
- Trained on "a few high-end GPUs within a week".

GoogLeNet was one of the first models that introduced the idea that CNN layers didn't always have to be stacked up sequentially. Coming up with the Inception module, the authors showed that a creative structuring of layers can lead to improved performance and computationally efficiency. This paper has really set the stage for some amazing architectures that appeared in the latest years.

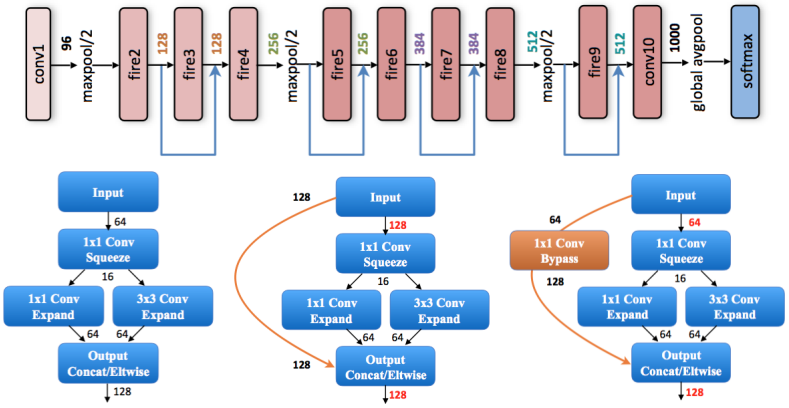
3.4.6 SqueezeNet

The paper of SqueezeNet [11] provides a smart architecture as well as a quantitative analysis. For the same accuracy of AlexNet, SqueezeNet can be 3 times faster and 500 times smaller. The main ideas of SqueezeNet are:

- Using 1x1(point-wise) filters to replace 3x3 filters, as the former only 1/9 of computation.

- Using 1x1 filters as a bottleneck layer to reduce depth to reduce computation of the following 3x3 filters.
- Downsample late to keep a big feature map.

The building brick of SqueezeNet is called fire module, which contains two layers: a squeeze layer and an expand layer. A SqueezeNet stacks a bunch of fire modules and a few pooling layers. The squeeze layer and expand layer keep the same feature map size, while the former reduce the depth to a smaller number, the later increase it. The squeezing (bottoleneck layer) and expansion behavior is common in neural architectures. Another common pattern is increasing depth while reducing feature map size to get high level abstract.



Σχήμα 3.11: SqueezeNet Architecture.

The squeeze module only contains 1x1 filters, which means it works like a fully-connected layer working on feature points in the same position. In other words, it doesn't have the ability of spatial abstract. As its name says, one of its benifits is to reduce the depth of feature map. Reducing depth means the following 3x3 filters in the expand layer has fewer computation to do. It boosts the speed as a 3x3 filter need as 9 times computation as a 1x1 filter. It is fact that too much squeezing limits information flow and too few 3x3 filters limits space resolution.

Κεφάλαιο 4

Basic Concept of CNN Engine

This chapter describes the process which is followed in order to make our CNN engine implementation able to support Convolutional Neural Networks. Furthermore, it provides the basic ideas of the old CNN engine's version and outlines both the Hardware and Software changes, which had to be done in order to support "deep" CNNs.

4.1 CNN Engine's Basic Ideas

First of all, it is important 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. The main idea begins by using the Caffe framework to import CNN's parameters, weights, biases and input data by taking advantage of CAFFE's dictionaries net.blobs and net.params. More specifically the CNN configuration defines the architecture and architectural parameters of the network. Examples of these parameters include:

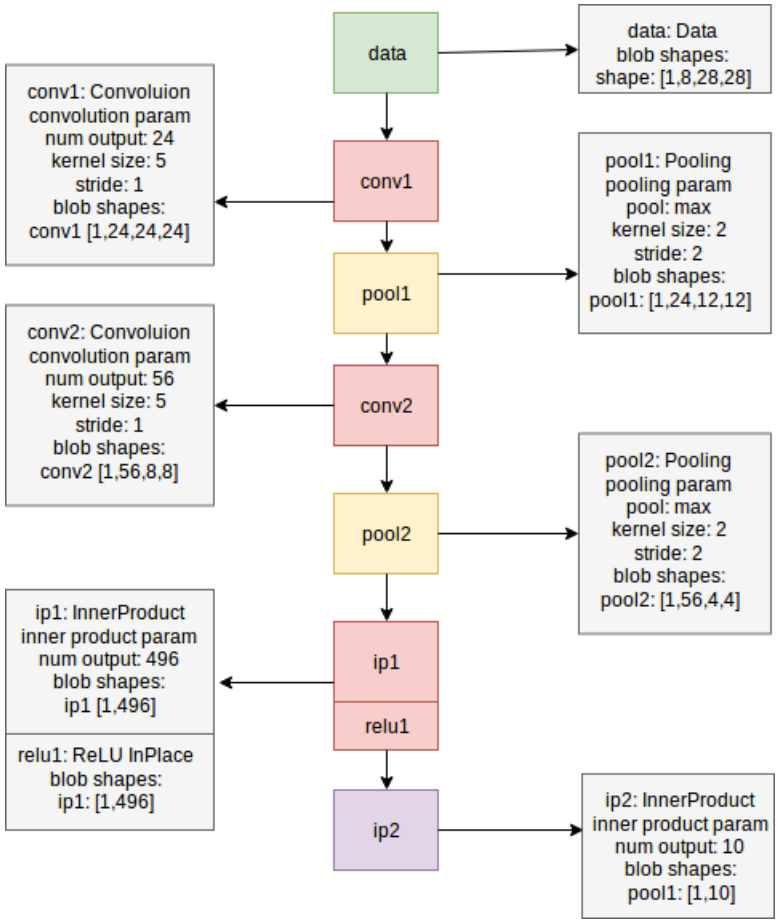
- Input data dimensions and channels (e.g. image size and colors)
- Number of convolutional filters
- Size of convolutional filters
- Pooling/downsampling size and method (e.g. max-pool or average)
- Number of convolution and pooling layers
- Size and number of fully connected layers
- Number of normalization layers
- Output representation size and type (e.g. the number of classes of the input dataset and the predicted class of an input sample)
- Some more parameters like function variables of the Jumptable and the output buffer, which will be explained later.

All these parameters and data (e.g. weights and biases) of the CNN Configuration are stored statically in the main memory of Myriad 2. After this, a method is called which creates all the nodes of the network initializing them with their parameters according to their layer type. Now, the execution of the network's nodes begin, which mean the following:

- Preprocess the Input appropriately with the specific way, which will be explained in chapter 5.
- Start SHAVEs, which execute their entry code from CMX until they use the Jumptable function. With the Jumptable function, they jump back to DDR in order to save their C code into DDR area.

- After these, SHAVEs transfer their data into local memories and call the computation assembly functions stated in CMX, before they return their results for the next layer's execution.

The above ideas were applied for CNNs with restricted number of layers and input dimensions like the one in 4.1.



Σχήμα 4.1: Lenet8 CNN.

In the image 4.1 the following layers and parameters are oserved:

- **data**: This layer accepts the input and copies it to the output without editing it. Lenet8 has an input blob of four dimensional shape [1,8,28,28]. In order to preprocess the input image and transform it into appropriate form for the network, an one dimensional picture of a digit between zero and nine is chosen. After this, the image is reshaped into 28x28 dimensions and eight copies of the reshaped digit are made.

- **conv**: Layers that perform convolution, include some parameters and a blob shape. Number of outputs represent the number of different convolutional filters and the depth dimension of the output as well. Kernel size represents the height and the width of the convolutional filter and the quantum quantity that each filter will convolve in the input image. Thus, stride shows the number with which the filter is slided, when the stride is 1 like in our fig. 4.1 filters are moved one pixel at a time. Blob shapes show always the output dimensions.

- **pool**: It is common to periodically insert a pooling layer in-between successive convolutional layers in a ConvNet architecture. Its function is to progressively reduce the spatial size of the representation in order to reduce the amount of parameters and computation in the network. This is made with max or average operation.

- **ip**: This is a fully connected node that generates a vector of Y elements and needs a matrix of weights to operate. In particular, it needs a [Y x previous output blob matrix], because the input 3D volume collapses into a vector and gets multiplied with a weights matrix. This matrix-vector multiplication results into a [Y x 1] matrix, i.e. a Y element vector.
- **ReLU**: ReLU can be attached in every layer and its role is to compute the maximum element between every element and zero. With other words it rejects negative values.

4.2 Hardware Attributes

In order to support the current implementation some drivers and **hardware attributes** had to be used. First of all, the Myriad2 Hardware had to be configured. The code starts to be executed by Leon OS as it 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 the RTEMS operating system with thread support is used. Below the two main hardware attributes are presented.

- The new Memory Layout of the Device which supports "deep" CNNs.
- The explanation of the CMX DMA Driver, which is used by every layer.

4.2.1 Memory Map of the Device

Deep neural networks are quite large for an embedded device. The CMX memory available for each SHAVE is maximum 128KB. A large network requires multiple variations of common operations such as convolution, pooling and inner product. For instance, conv11x11, conv7x7, conv5x5, conv3x3, conv1x1, 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. The presence of local buffers are required 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. Before seeing the memory map document of our CNN engine implementation memory regions of Myriad 2 should be clear, so they are presented in table 4.1.

Memory	**Size**	**LEON Access Cost**	**SHAVE Access cost**	**Start Address**
CMX	2 MB	Low	Low	0x70000000
DDR	128 MB 512 MB	High Low when cache hit	High random access Moderate L2 hit Low for L1 hit	0x80000000

Πίνακας 4.1: Memory areas of Myriad 2.

CMX is organized like is shown in table 4.2. 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.

- Local slices follow the same sequence with SHAVEs. SHAVE0 refers to the lowest 128 KB of CMX, SHAVE1 to next 128 KB, SHAVE11 is assigned to slice 11.
- Slices 12 to 15 are not tied to any SHAVE. They may be freely used for any other purposes, as for example allocate there variables from Leon code, in order that SHAVEs access them rapidly.

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.

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

Πίνακας 4.2: CMX slices appropriate for SHAVEs.

At this point the whole memory map of our initial CNN engine's edition in Myriad 2150 platform with DDR 128MB is presented.

Listing 4.1: Old memory map of Device

```
MEMORY
{
   SHV0_CODE (wx)   : ORIGIN = 0x70000000 + 0 * 128K,        LENGTH = 32K
   SHV0_DATA (w)    : ORIGIN = 0x70000000 + 0 * 128K + 32K, LENGTH = 96K

   SHV1_CODE (wx)   : ORIGIN = 0x70000000 + 1 * 128K,        LENGTH = 32K
   SHV1_DATA (w)    : ORIGIN = 0x70000000 + 1 * 128K + 32K, LENGTH = 96K

   SHV2_CODE (wx)   : ORIGIN = 0x70000000 + 2 * 128K,        LENGTH = 32K
   SHV2_DATA (w)    : ORIGIN = 0x70000000 + 2 * 128K + 32K, LENGTH = 96K

   SHV3_CODE (wx)   : ORIGIN = 0x70000000 + 3 * 128K,        LENGTH = 32K
   SHV3_DATA (w)    : ORIGIN = 0x70000000 + 3 * 128K + 32K, LENGTH = 96K

   SHV4_CODE (wx)   : ORIGIN = 0x70000000 + 4 * 128K,        LENGTH = 32K
   SHV4_DATA (w)    : ORIGIN = 0x70000000 + 4 * 128K + 32K, LENGTH = 96K

   SHV5_CODE (wx)   : ORIGIN = 0x70000000 + 5 * 128K,        LENGTH = 32K
   SHV5_DATA (w)    : ORIGIN = 0x70000000 + 5 * 128K + 32K, LENGTH = 96K

   SHV6_CODE (wx)   : ORIGIN = 0x70000000 + 6 * 128K,        LENGTH = 32K
   SHV6_DATA (w)    : ORIGIN = 0x70000000 + 6 * 128K + 32K, LENGTH = 96K

   SHV7_CODE (wx)   : ORIGIN = 0x70000000 + 7 * 128K,        LENGTH = 32K
   SHV7_DATA (w)    : ORIGIN = 0x70000000 + 7 * 128K + 32K, LENGTH = 96K

   SHV8_CODE (wx)   : ORIGIN = 0x70000000 + 8 * 128K,        LENGTH = 32K
   SHV8_DATA (w)    : ORIGIN = 0x70000000 + 8 * 128K + 32K, LENGTH = 96K

   SHV9_CODE (wx)   : ORIGIN = 0x70000000 + 9 * 128K,        LENGTH = 32K
   SHV9_DATA (w)    : ORIGIN = 0x70000000 + 9 * 128K + 32K, LENGTH = 96K

   SHV10_CODE (wx)  : ORIGIN = 0x70000000 + 10 * 128K,        LENGTH = 32K
   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    CMX_DMA_DESCRIPTORS  (wx) : ORIGIN = 0x78000000 + 12 * 128K , LENGTH = 12K
40    CMX_OTHER (wx) : ORIGIN = 0x70000000 + 12 * 128K + 12K , LENGTH = 256K - 12K
41
42    LOS (wx) : ORIGIN = 0x80000000, LENGTH = 64M
43    LRT (wx) : ORIGIN = 0x70000000 + 14 * 128K  LENGTH = 256K
44
45    DDR_DATA (wx) : ORIGIN = 0x80000000 + 64M, LENGTH = 64M
46
47  }
48
49  INCLUDE myriad2_leon_default_elf.ldscript
50  INCLUDE myriad2_shave_slices.ldscript
51  INCLUDE myriad2_default_general_purpose_sections.ldscript
```

From the above ldscript the following conclusions are extracted:

- 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 computation 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, 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.

4.2.2 Efficient Resource Management

It is a fact that Myriad 2 MA2150 platform contains limited main memory of 128MB. On the other hand talking about deep neural networks means 'heavy' memory demands. More specifically AlexNet and VGGNet, winners of Image Classification Contest, include weights of 117 MB and 276 MB respectively. These data are not able to fit in DDR memory of MA2150 so MA2450 platform should be provided. Following the main concept which is described above, proves that data should be sent into CMX in order to be processed and give us the output. Here appears a big challenge as every's SHAVE Memory slice is maximum 128 KB and memory space which is needed in CMX local buffers for both input and output feature maps are almost 500KB. The first contribution refers to the memory map which is renewed in order to take advantage of MA2450 platform's architecture and the limited demands of CMX SHAVE's code.

Listing 4.2: Renewed custom.ldscript

```
MEMORY
{
  SHV0_CODE (wx)   : ORIGIN = 0x70000000 + 0 * 128K,        LENGTH = 4K
  SHV0_DATA (w)    : ORIGIN = 0x70000000 + 0 * 128K + 4K,   LENGTH = 124K

  SHV1_CODE (wx)   : ORIGIN = 0x70000000 + 1 * 128K,        LENGTH = 4K
  SHV1_DATA (w)    : ORIGIN = 0x70000000 + 1 * 128K + 4K,   LENGTH = 124K

  SHV2_CODE (wx)   : ORIGIN = 0x70000000 + 2 * 128K,        LENGTH = 4K
  SHV2_DATA (w)    : ORIGIN = 0x70000000 + 2 * 128K + 4K,   LENGTH = 124K

  SHV3_CODE (wx)   : ORIGIN = 0x70000000 + 3 * 128K,        LENGTH = 4K
  SHV3_DATA (w)    : ORIGIN = 0x70000000 + 3 * 128K + 4K,   LENGTH = 124K

  SHV4_CODE (wx)   : ORIGIN = 0x70000000 + 4 * 128K,        LENGTH = 4K
  SHV4_DATA (w)    : ORIGIN = 0x70000000 + 4 * 128K + 4K,   LENGTH = 124K

  SHV5_CODE (wx)   : ORIGIN = 0x70000000 + 5 * 128K,        LENGTH = 4K
  SHV5_DATA (w)    : ORIGIN = 0x70000000 + 5 * 128K + 4K,   LENGTH = 124K

  SHV6_CODE (wx)   : ORIGIN = 0x70000000 + 6 * 128K,        LENGTH = 4K
  SHV6_DATA (w)    : ORIGIN = 0x70000000 + 6 * 128K + 4K,   LENGTH = 124K

  SHV7_CODE (wx)   : ORIGIN = 0x70000000 + 7 * 128K,        LENGTH = 4K
  SHV7_DATA (w)    : ORIGIN = 0x70000000 + 7 * 128K + 4K,   LENGTH = 124K

  SHV8_CODE (wx)   : ORIGIN = 0x70000000 + 8 * 128K,        LENGTH = 4K
  SHV8_DATA (w)    : ORIGIN = 0x70000000 + 8 * 128K + 4K,   LENGTH = 124K

  SHV9_CODE (wx)   : ORIGIN = 0x70000000 + 9 * 128K,        LENGTH = 4K
  SHV9_DATA (w)    : ORIGIN = 0x70000000 + 9 * 128K + 4K,   LENGTH = 124K

  SHV10_CODE (wx)  : ORIGIN = 0x70000000 + 10 * 128K,       LENGTH = 4K
  SHV10_DATA (w)   : ORIGIN = 0x70000000 + 10 * 128K + 4K,  LENGTH = 124K

  SHV11_CODE (wx)  : ORIGIN = 0x70000000 + 11 * 128K,       LENGTH = 4K
  SHV11_DATA (w)   : ORIGIN = 0x70000000 + 11 * 128K + 4K,  LENGTH = 124K

  CMX_DMA_DESCRIPTORS  (wx) : ORIGIN = 0x78000000 + 12 * 128K , LENGTH = 128K
  CMX_OTHER (wx) : ORIGIN = 0x70000000 + 13 * 128K  , LENGTH = 128K

  LOS (wx) : ORIGIN = 0x80000000, LENGTH = 2M
  LRT (wx) : ORIGIN = 0x80200000 + 13 * 128K  LENGTH = 2M

  DDR_DATA (wx) : ORIGIN = 0x80000000 + 4M, LENGTH = 400M

}
```

With the new script, the following are extracted:

- 4KB of CMX are used for code and 124KB of CMX are used for data in each SHAVE, in order to afford as much memory space is possible.
- 12KB of CMX are used explicitly by the DMA Engine.
- 128KB of CMX are used for other purposes. In particular, this space will be used for placing shared parameters used by the SHAVEs like the structs, which contain the network parameters.

- 2MB of DDR are used by the Leon OS and the RTEMS operating system, due to optimized code from the Leon Part.
- 2MB of DDR again are given to Leon RT, which will run the same code in order to count the energy consumption as will be explained below.
- Finally, 400MB of DDR are used for placing parameters/weights of the CNN.

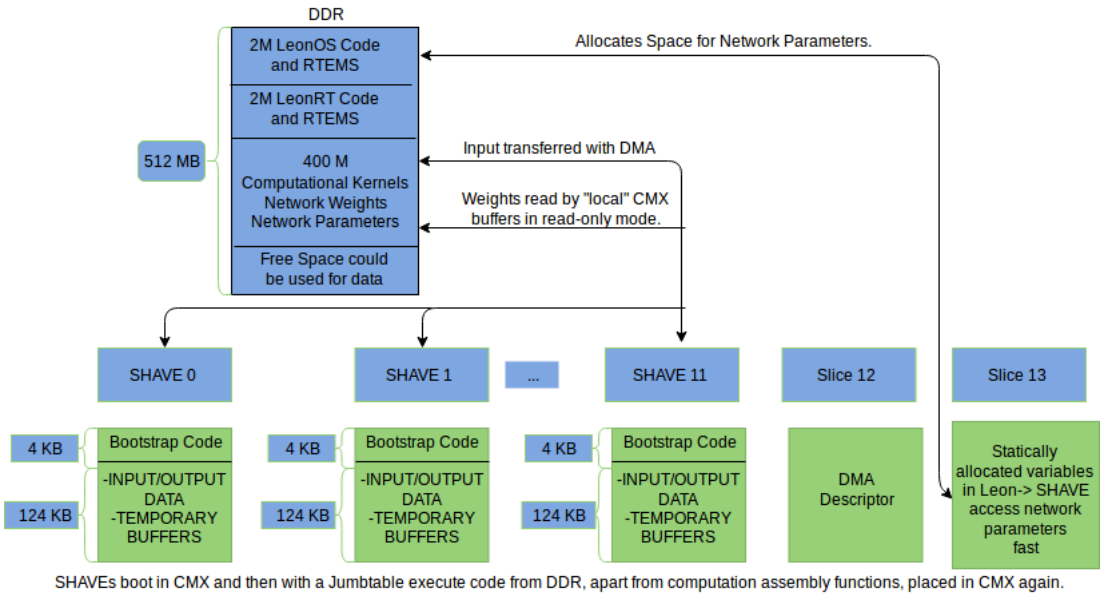


Figure 4.2: Engine's Memory Overview.

The new LinkerScript in contribution with the ideas, which are going to be presented in the next chapter are responsible for supporting 'really' Deep Neural Networks (e.g. GoogleNet with 83 Layers).

4.2.3 CMX DMA Driver

The DMA engine is utilized extensively inside the computational nodes, transferring data between DDR and CMX. Therefore, understanding the functionality of the DMA engine exposed by the respective driver is essential. Each DMA transfer is performed though 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 4.3 illustrates the use of a 2D striding transaction. 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. Also, the Destination

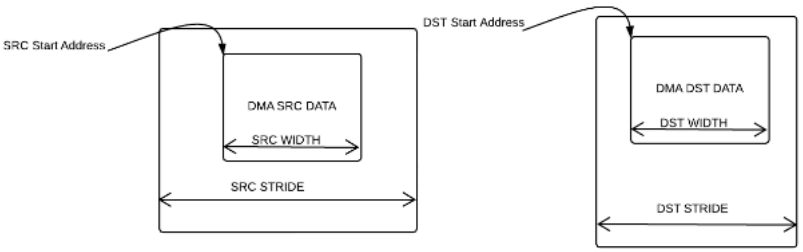


Figure 4.3: DMA transaction with same width.

width can be programmed to a different value than Source width. This is illustrated in fig. 4.4, where 2D data are transformed from row form into column form.

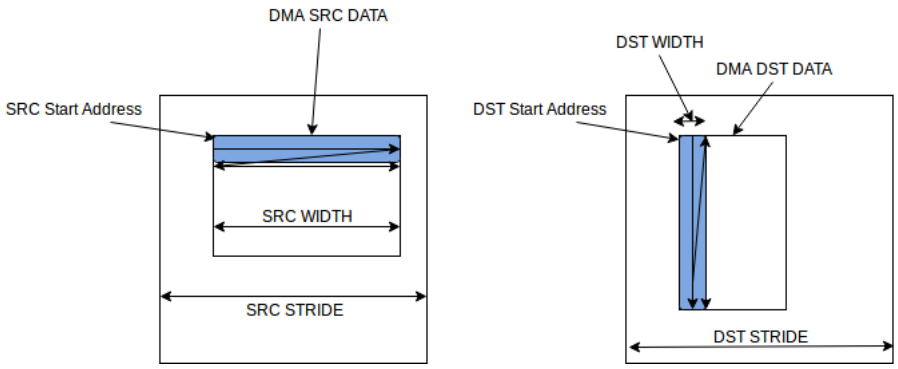


Figure 4.4: DMA transaction with different DST, SRC width, with conversion from row form into column one.

Also, a pseudocode of the driver is provided in Listing 4.3 in order to make the above transfer clear and includes the following variables:

Listing 4.3: Code description from row into column form.

```
ref[0] = dmaCreateTransactionFullOptions(
    id,
    &task[0],
    input_src_addr,          //src address
    input_dst_addr,          //dst address
    bytes_in_line * number_of_lines,    //bytes of 2d area
    bytes_in_line,           //src line width
    bytes_of_variable,       //dst line width
    bytes_in_line,           //src stride
    bytes_in_line);          //dst stride
dmaStartListTask(ref[0]);
```

- `input_src_address` refers to the memory location in DDR where the data rectangle is placed.
- `input_dst_address` is the memory location of the CMX buffer, in which the data rectangle will be placed.

- The variable `bytes_in_line` describes how many bytes of data are included in every line of the 2d rectangle.
- `number_of_lines` describes how many lines of data are included in the 2d rectangle, which are equal to the column size of the rectangle.
- `bytes_of_variable` is the dst line width, which means that data will be placed in column form as width size is equal with one element.
- Finally, when the src line width is equal with the dst line width the shape of the input rectangle does not change during transfer. Also, if src stride is equal to line width, data in DDR are placed contiguously in the main memory. The conclusion is that DMA driver is used in order to transfer data from DDR to CMX and the opposite way so let's have a detailed description of the two memories.

Chapter 5

Configure CNN Engine to Support ImageNet's Contest Deep CNNs

In this chapter is examined what happens after the initialization of the nodes till the execution of the Convolutional layers with the direct approach. All the nodes follow the same methodology, but other kind of layers do not contain so many optimization techniques due to the restricted number of computations. Also, the methodologies, which are used in order to test "deep" CNNs, are becoming clear. As stated above layer's execution support the following technique:

- Begin layer's execution from DDR side.
- Call the preprocess functions.
- Start SHAVEs to boot from CMX entry code.
- Use JumpTable in order to continue from code placed in DDR.
- Transfer Input DMA and after calling Assembly functions placed in CMX by using again the Jumptable transfer output with DMA.

5.1 Sequence of the Input Data in DDR

The basic idea for every's layer execution is the following: Transfer both input data and layer's parameters from the main memory into SHAVEs local memory, in order SHAVEs undertake the computations and afterwards transfer the results back into main memory. First of all, input data are statically stored into DDR as shown from Listing 5.1 in the Leon code.

Listing 5.1: Static allocation of Input data.

```
1 #define DDR_BUFFER __attribute__((section(".ddr_direct.data"), aligned (16)))
2
3 //------------------------Output buffer declaration-----------------------------//
4 fp16 DDR_BUFFER branch_output_buffer_0_0[maximum_size_of_blob];
5 fp16 DDR_BUFFER branch_output_buffer_0_1[maximum_size_of_blob];
```

These buffers are placed in DDR and they are double beacuse, in every linear neural network, the output of a previous layer is the input of the next one. So, two buffers are needed for exchanging the input and output data of every layer. The size of buffers has to be the maximum product of any blob dimensions in the network. Apart from input data, the weights and biases are statically stored into DDR buffers as well, but in specific address space, where SHAVE processors can have access too. The reason that this happens is that later, in the computation code, SHAVEs need to have access into weights and biases in order to execute convolution and fully conncected layers.
For every layer's representation structs are used, whose variables can be seen both from the main and local memories. In order to accomplish this, the structs are stored into the shared memory areas and their code into the "/shared" file. An example for the direct convolution's struct is presented below in Listing 5.2, which contains both layer's parameters and other information. This piece of information is becoming clear below, when is assigned with the appropriate variables.

Listing 5.2: Assignment of convolutional parameters

```
#ifndef __CONV_API_H__
#define __CONV_API_H__

#include <mv_types.h>

typedef struct{
    int src_addr;
    int dst_addr;
    int elements;
    int buffer_elements;
}conv_buffer_info;

typedef struct {
    u8 *input;
    u16 input_channel_offset;
    u8 inputBPP;

    u8 *output;
    u16 output_channel_offset;
    u8 outputBPP;

    u8 *conv_weights;
    u16 conv_weights_offset;
    u8 conv_weights_channel_offset;

    u8 *conv_biases;
    u8 kernelBPP;

    u16 channels;
    u32 ddr_function;
    u8 kernel_h;
    u8 kernel_w;

    u8 coalescing_num;
    u8 with_relu;

    int in_buffer_shift;
    u16 line_width;

    int out_src_addr, out_src_width, out_src_stride;
    int out_dst_addr, out_dst_width, out_dst_stride;
    int out_buffer_elements, out_elements;

    u16 maps;

    u8 c_group;
    u8 splits;

    int in_src_width, in_src_stride;
    int in_dst_width, in_dst_stride;

    u8 in_stride;
    int in_buffers_num;
    conv_buffer_info in_buffers[8];

} conv_info;

#endif
```

After the allocation of the input data, every node is created in Leon and is starting to be prepared for the execution as evey variable of the struct is assigned with the appropriate network's parameters. For example, below in Listing 5.3 the preparation code of a convolutional node is presented, where for one layer one struct is used as the information of the struct currently fits into local memories.

Listing 5.3: Assignment of convolutional parameters

```
if CONVOLUTION
u64 Convolution::execute(u8 *bottom_output_buffer, u16 &bottom_channels,
                         u16 &bottom_input_height, u16 &bottom_input_width){



            convolution_object->input = bottom_output_buffer;
            convolution_object->output = this->output_buffer;
            convolution_object->conv_weights = (u8*)((u32)(this->weight_pointer)
             & (u32)0x8...);
            convolution_object->conv_biases = (u8*)((u32)(this->bias_pointer)
             & (u32)0x8...);

            convolution_object->c_group = this->group;
            convolution_object->channels = bottom_channels / this->group;
            convolution_object->output_channel_offset = this->input_height
            * this->input_width;
            convolution_object->kernel_h = this->kernel_size;
            convolution_object->kernel_w = this->kernel_size;
            convolution_object->with_relu = this->ReLU_flag;
            convolution_object->in_stride = this->stride;

            convolution_object->input_channel_offset = bottom_input_height
            * bottom_input_width;

            ...

            convolution_object->splits = lines;
            convolution_object->maps = channels;

            convolution_object->coalescing_num = 1;
```

```
32
33              while (convolution_object->coalescing_num * ((this->input_height
34              * this->input_width)/lines) * 2 < 20000){
35                  convolution_object->coalescing_num ;
36              }
37
38              convolution_object->ddr_function = ddr_function;
39
40
41              conv_prepare_dma(...);
```

This "convolution object" struct is stored statically into CMX. This assumption gives speedup and guarantees that all SHAVE processors can refer to the variables of struct, in extension to all CAFFE parameters and every other attribute that is needed for the computation. For a convolutional node some variables, which are stated above and need to be described are:

- input is an address, which shows where in the main memory the input of this layer is stored. Basically, it is the output address of the previous node, except for the parallel nodes in CNNs like GoogleNet, SqueezeNet.
- offsets of the input and output, represent the bytes that input feature maps and output feature maps allocate. The reason that this information is needed as stated later to the computation of the layers.
- group, stride, channels and coalescing are parameters of the convolution and will be explained later in the detailed analysis of convolution.
- `conv_prepare_dma` is a function, which preprocesses the input and makes it ready to be sent with DMA into CMX, where SHAVE processors will compute the output. This funtion is responsible in order that a lot of optimization parameters are introduced.

As stated above, every layer has the need of one struct in order to be executed, but for "heavy" layers of Imagenet's CNNs like the input layers this can not be the case, as the memory demands of input and output feature maps exceed the local memories of SHAVEs. Following this assumption an array of structs is used, where every struct corresponds to a partition of the input and the output for which it is responsible. The following section proves the way that by preprocessing, both the input and its corresponding output partition are computed.
Before the preprocessing function for direct convolution and pooling is exlpained, is important for the reader to be familiar with these two operations. In order to accomplish this, a brief description is described below. 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 as figured below in 5.1.

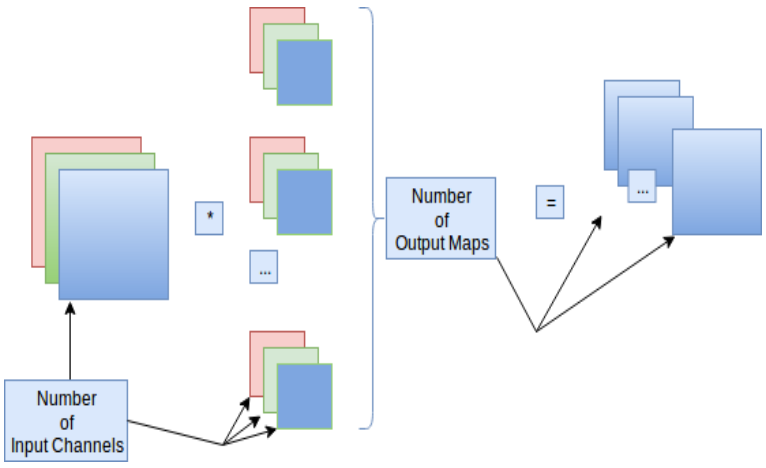


Figure 5.1: Convolution Operation.

For the purposes of this example the input has only three input channels. Every channel of these has to be convolved with every kernel. The output dimensions depend on the network parameters

like kernel size and striding parameter. Furthermore, in pooling operation for every output map only one input channel is needed, so it is a simpler operation. Let's see how these operations have been implemented in Myriad 2.

5.2 Preprocessing of the Input

First of all it must be clear that the preprocessing function has to be called for every single struct of the array. Preprocessing does not offer only optimizations, which will later be explained but also serves the correct execution of the CNN. Someone's first reaction in order to fit high definition images in local memories would be to break the input into tiles and send these tiles into CMX for computation. Let's display an example for an input image with input dimensions 10x10, kernel size equal to 3, striding parameter equals 1 and with no zero-padding:

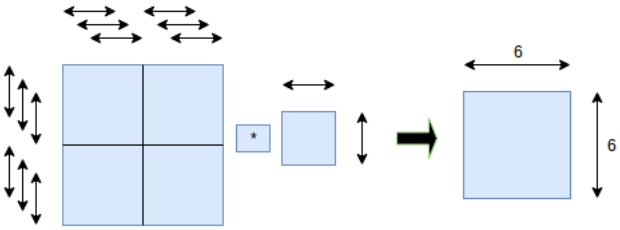


Figure 5.2: Wrong Produced Output.

From the figure 5.2 above it is concluded that convolution is not indepedent between the neighboring areas and there are some intersection areas, which lead to wrong results. In order to overcome this problem, a specific operation is followed. Our initial process is to compute the output dimensions of our node, which is possible knowing convolution's parameters. Output width and height are given with the following equations:

$$outputwidth = floorint(width + 2 * pad - kernel/stride) + 1$$

$$outputheight = floorint(height + 2 * pad - kernel/stride) + 1$$

After this, the real value of padding parameter has to be computed, because by partitioning the input and assigning different partitions into structs, the value of zero-padding in the boundary areas is changed. Also, CAFFE framework for its normal convolution's computations sometimes changes zero-padding value as stated in the example below. Let's say that an input with width equals 4, 3x1 kernel, striding equal to 2 and zero-padding parameter 1.

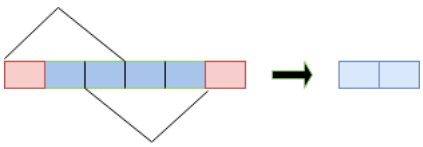


Figure 5.3: Real boundary padding.

The output dimensions are 2x1 so the value of right padding was not important. Under these circumstanses our real right padding value was equal to zero.
After the above operations, input and output dimensions are transformed into rectangular areas with coordinates, for example an input image with dimensions 100x100 will be transformed into an area with initial point $(x, y) = (0, 0)$ and final points $(x, y) = (100, 0), (x, y) = (0, 100), (x, y) = (100, 100)$. The ouput respectively (e.g. dimensions 64x64) will be transferred into a square area called 'window'

with $(0,0),(64,0),(0,64),(64,64)$ points. In addition to these, a base area is created, which hosts both the padding and the input elements with dimensions:

$$basewidth = inputwidth + pad_{left} + realpad_{right}$$

$$baseheight = inputheight + pad_{top} + realpad_{down}$$

As far as the output is concerned a reverse transformation in our 'window' area is made in order to find out in which region of the base area it corresponds, with the following equations(below only points of y ordinate,'height', are presented because it is the same for width):

$$res_{y0} = stride * (window_{y0} - out_{y0}) + base_{y0}$$

$$res_{y1} = stride * (window_{y1} - out_{y1}) + base_{y1}$$

With all these functions computing the input area from which the output square is produced is achieved. With other words, output is corresponded to a square described by four points and this is transformed into the input area. This square contains padding elements too, which are unnnecessary as they can be reproduced in CMX, escaping from addittional transfers which lead to DMA buffer 'overloading'. One solution to this problem is to compute the intersection area between the input area, which contains padding elemens, and input image, as presented below with pseudocode.

Listing 5.4: Select input image's elements without padding

```
void get_intersection(
    struct rect_plane *const intersection,
    const struct rect_plane *const area_input_image,
    const struct rect_plane *const area_reverse_transformed
)
{
    intersection->y1 = max(area_input_image->y1, area_reverse_transformed->y1)
    intersection->y2 = min(area_input_image->y2, area_reverse_transformed->y2)

    intersection->x1 = max(area_input_image->x1, area_reverse_transformed->x1)
    intersection->x2 = min(area_input_image->x2, area_reverse_transformed->x2)
}
```

Furthermore, alignment of all elements, which are contained in the intersection area, has to be done because the assembly functions convolve on one dimension. In the intersection area there are elements, which are not useful as shown below in figure 5.5.

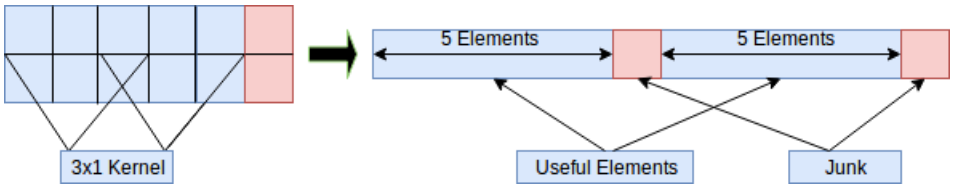


Figure 5.4: Input data alignment.

For the convolution function above, a 3x1 kernel with stride equals to 2 is applied in the rectangle input with dimensions 6x2. The result of this operation will be a 2x2 square, but when SHAVEs apply the convolution kernel they produce six elements from which two are 'junk'. The solution to this problem is simple as the ouput DMA transfer contains stride of the 'junk' elements, according to figure 4.3 source width equals to two and source stride equals to three. Generally, due to SHAVEs vector register unit sometimes 'junk' elements are added in the input. This happens, in order to ensure that the input line width is equal to a number, multiple of eight, which means multiple of 128 bytes as is the vector register's capacity. This optimization is called 'alignment' and it offers a speedup in our execution procedure in layers with small input dimensions. Also, here a trade-off between execution time and memory demands is observed, by adding 'junk' elements in computation. Finally, after the whole procedure of preprocess which is described above, every output partition regardless its shape can be tranformed into its input partition, in order to receive results compatible with CAFFE.

From the above, is becoming clear that the preprocess function finds for every output partition its corresponding input given only one id, regardless of the partition's shape. Below, the preprocess algorithm is presented with a flow diagram and a figure and a code which finds from the id which input/ouput slice is the appropriate.

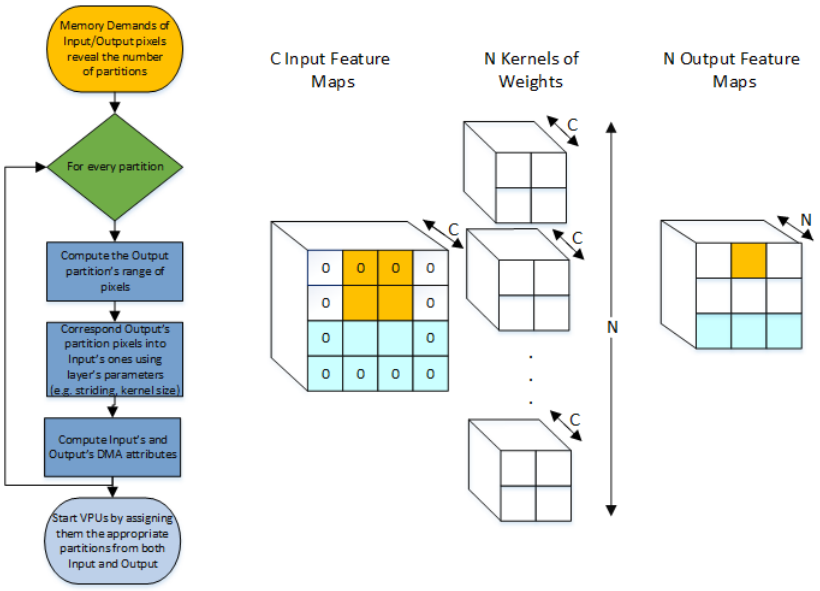


Figure 5.5: Preprocess Analysis.

Listing 5.5: Correspond right partitions from the id number

```
if((output_width%(tiles/2))==0){
    //every tile has the same width and height
    window_width = (out_width/tiles);
    window_height = (out_height/tiles);
    if(floor == 0){
        window_x = (out_width/tiles)*tile_id;
        window_y = 0;
    }
    else{
        window_y = (out_height/tiles)*floor;
        window_x = (out_width/tiles)*(tile_id - (tiles/floor));
    }
}
else{
    if(tile is the last of every floor){
        if(floor == 0){
            //last of the first floor takes the rest width
            window.width = out_width - (out_width/tiles);
            window.height = (out_height/tiles);
            window_x = (out_width/tiles)*(tile_id);
            window_y = 0;
        }
        else{
            //second floor takes rest width and rest height
            window.width = out_width - (out_width/tiles);
            window.height = out_height - (out_height/tiles);
            window_x = (out_width/tiles)*(tile_id - (tiles/floor));
            window_y = (out_height/tiles)*floor;
        }
    }
```

```
31      else{
32          window.width = (out_width/tiles);
33          if(floor == 0){
34              //every tile has the same width and height
35              window.height = (out_height/tiles);
36              window_x = (out_width/tiles)*(tile_id);
37              window_y = (out_height/tiles)*(tile_id);
38          }
39          else{
40              //every tile has constant width and the rest height
41              window.height = out_height - (out_height/tiles);
42              window_x = (out_width/tiles)*(tile_id - (tiles/floor));
43              window_y = (out_height/tiles)*floor;
44          }
45      }
46  }
```

5.2.1 Convolution's Layer Parameters

AlexNet, competed in the ImageNet Large Scale Visual Recognition Challenge in 2012 required group and striding parameters in order to be tested, which were not implemented in our previous CNN engine's version. After partitioning, parameters had to be implemented regardless of the number of tiles.
It is a fact that convolutional kernels are applied on one dimension, so in order to support **striding parameter** the appropriate indexes on SHAVEs had to be provided. In the section of preprocessing it became clear that the alignment of all lines hade to be made in order to use one DMA buffer and transfer all the elements. On the other hand, in order to use striding greater than 1 previous implementation is extended and an example is provided below in order to explain striding's approach. Imagine that the height and the width of an input channel is 7x7 and a 3x3 kernel with zero-padding and stride 2 is applied. The data are laid out in memory as shown in figure 5.6. Due to the fact that the convolution kernel is 3x3 with stride 2 and because the input channel height is 7, the height of the ouput rectangle will be 3. Notice, in this figure, that the lines are shown symbolically. No elements are presented, because a convolution routine that generates one output line is already provided.

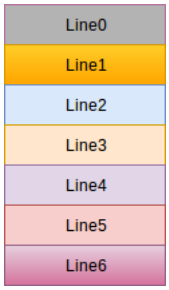


Figure 5.6: Layout of input channel in memory.

For the specific convolution 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". Preprocess function is responsible to separate the input in groups accordingly with the striding value, such as figure 5.7 shows.

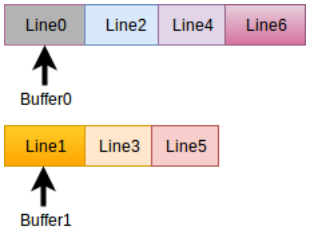


Figure 5.7: Layout of input channel in memory with DMA buffers.

In order that input pointers in CMX take the appropriate input lines for computation, as many pointers as output height are going to be used. 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 convolution routine "sees" the data as presented in figure 5.8, although each line is present in memory only once.

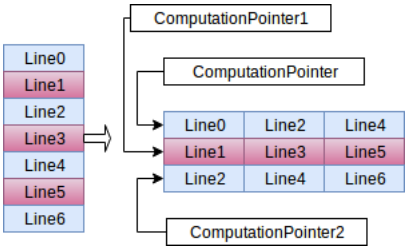


Figure 5.8: How the convolution routine "sees" data in memory.

This extension generates garbage results, much like it is done in convolution with stride 1. 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. In case that striding parameter is greater than 1, no double buffering is used, below a pseudocode from the SHAVEs part is presented, which will be later explained.

Listing 5.6: SHAVE's scheduling algorithm

```
for every line do
    for every dma buffer do
        dma input buffer
    end for
    for every input pointer do
        correspond input pointers with dma buffers
    end for
    for group in output_maps_groups do
        for channel in input_channels do
            Bring input channel with DMA
            for map in group do
                Apply 1D convolution using channel
                Accumulate the result of 1D convolution
            end for
        end for
        Send the group of map buffers back with DMA
    end for
end for
```

Filter groups (AKA **grouped convolution**) were introduced in the now seminal AlexNet paper in 2012. As explained by the authors [8], their primary motivation was to allow the training of the network over two Nvidia GTX 580 gpus with 1.5GB of memory each. It was noted that filter groups seemed

to consistently divide convolution layers into two separate and distinct tasks: black and white filters and colour filters. AlexNet trained with varying numbers of filter groups, from 1 (i.e. no filter groups), to 4. When trained with 2 filter groups, AlexNet was more efficient and yet achieves the same if not lower validation error as shown in the figure 5.9.

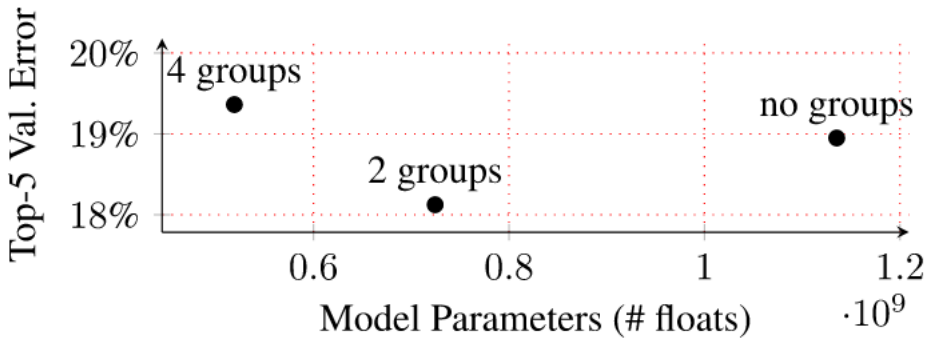


Figure 5.9: AlexNet's best results with group equals to 2.

Typical convolution filters in CNNs have full connections between the input and output feature maps. If the input feature map has c_{in} channels and the output feature map has c_{out} channels, the filter dimension is $h \times w \times c_{in} \times c_{out}$. This means that the height of the filter is h, the width is w, the channel depth is c_{in}, and there are c_{out} filters of corresponding shapes. Filter groups disconnect the connectivity between the input and output feature maps. For example, if n filter groups are applied, n uniform filter groups with c_{out}/n filters are used. Each filter group has a dimension of $h \times w \times c_{in}/n$, i.e. total filter dimension becomes $h \times w \times c_{in}/n \times cout$. Total parameters required for this convolution layer is n times smaller than that of the original full convolution layer. Group parameter ensures that the network has decreased memory demands for its weights. This happens as each of the filters in the grouped convolutional layer is now exactly half the depth, i.e. half the parameters and half the compute as the original filter. Let's introduce the first convolutional layer of AlexNet with group parameter equals to 2. Input dimensions are 96x27x27, kernels 256*48*5*5 and output equals to 256*27*27.

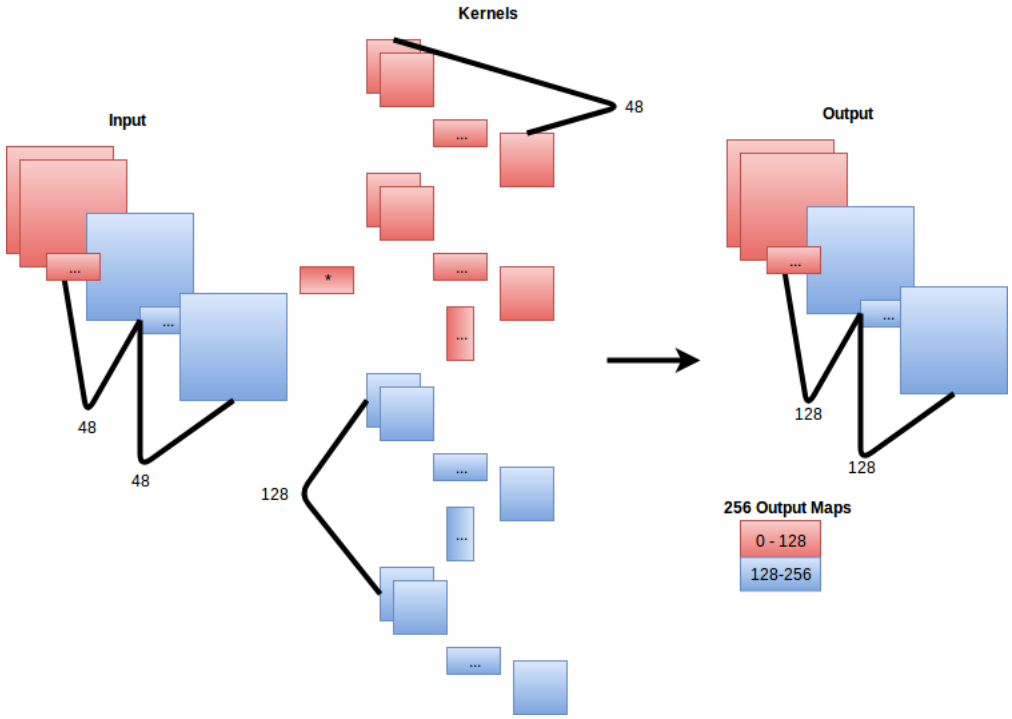


Figure 5.10: Detailed testing of AlexNet's second layer.

As far as the implementation in Myriad 2 is concerned, it is more flexible for speedup to fulfill group parameter in SHAVE's code. As stated before, every SHAVE is responsible for producing a specific number of output maps. The range of output maps will determine how many times the SHAVE code is going be executed. In other words, if both the first and last map are less than the whole maps divided by group, the execution will happen once. The same happens both if the first and last map are greater than the whole maps divided by group. Otherwise, the SHAVE code will be repeated twice with the appropriate weight parameters and input channels seen in 5.10. So according to the range of output maps, SHAVE's pseudocode is the following:

Listing 5.7: Group Implemntation

```
1  if (group != 1) do
2      if (firstmap && lastmap < maps/group) do
3          firstChannel = 0
4      elif (firstmap && lastmap>maps/group) do
5          firstChannel = channels/2
6      else
7          execution_times = 2
8          temporaryMap = lastmap
9          lastmap = maps/group
10         ...//first execution: firstmap->maps/group
11         ...//second execution: maps/group->lastmap
12     endif
13 end if
```

5.3 SHAVE's Implementation and Parallization Scheme

After computing the appropriate input/output partitions and assigning all the parameters it is time to start SHAVEs in order to make the computations. They are assigned with different output maps and the parallization scheme concerns the depth of the output maps as show in the figure 5.11 below.

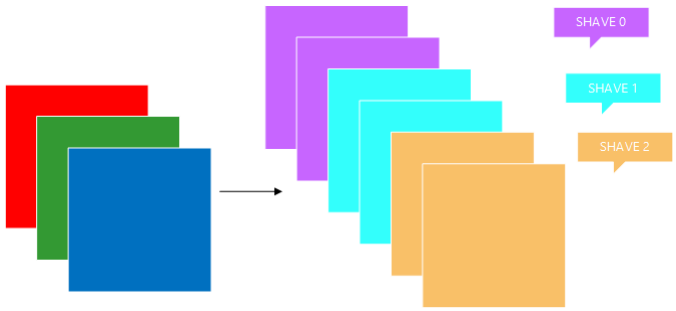


Figure 5.11: Parallization Scheme.

An input image with three channels and six output maps is given, so convolutional kernels should be six. Above three SHAVEs are used and every one of them is responsible for two ouput maps. The first SHAVE generates the purple output maps, the second SHAVE generates the white blue output maps and the third SHAVE generates brown output maps. Generally, every SHAVE undertakes output maps divided by number of shaves. If there is a remainder, it is shared one by one to every SHAVE until remainder equals to zero. This procedure is translated with the code below:

Listing 5.8: SHAVE's scheduling algorithm

```
1   u8 first_shave = 0;
2   u8 last_shave = shaves_used - 1;
3   u8 current_shave = 0;
4
5   u16 number_of_outputs_per_shave = channels / shaves_used;
6   u16 remained_outputs = channels % shaves_used;
7   u16 first_map = 0;
8   u16 last_map = 0;
9   for(u8 shave_index = 0; shave_index < shaves_used; shave_index++){
10
11      last_map += number_of_outputs_per_shave;
12      if(remained_outputs > 0){
13          last_map++;
14          remained_outputs--;
15      }
16      ResetShave(first_shave + shave_index);
17
18      StartShaveCC(first_shave + shave_index,
19                           (u32) startShave_conv[shave_index + first_shave],
20                           "iiii",
21                  convolution_object, first_map, last_map, jumpTableAddr);
22      first_map = last_map;
23      current_shave++;
24      if(last_map > channels){
25          break;
26      }
27  }
28
29  for (u8 shave_index = 0; shave_index < current_shave; shave_index++){
30      WaitShave(first_shave + shave_index);
31  }
```

From the code above, when function StartShaveCC is called, SHAVEs begin with a serial sequence. SHAVE processors start their execution with an entry function, code placed in CMX. The arguments of the StartShave function contain are presented below:

- shaveId, it depends of how many shaves for our application are going to be used. Shave index can take values from zero to eleven.
- entrypoint, is the address inside CMX that SHAVE processors will start to execute code, defined in Makefile as well.
- convolution-object is a struct which contains both network parameteres and useful attributes, which later will be explained.
- first-last map represent the range of output maps that every shave will undertake.
- jumpTableaddr is the address of Jumptable, which is used from SHAVEs in order to jump from the bootsrap code inside CMX into DDR. The opposite direction is followed later, when assembly kernels for computations are used.

Well, after the initialization of the SHAVEs, code is executed from the "/shave/cmx/entry.c" directory:

Listing 5.9: SHAVE's entry function in CMX

```
void shave_conv(conv_info **info, u32 firstMapNo, u32 lastMapNo,
                J_FUNCPTR_T jumpTable){

    int shaveId = swcWhoAmI() - PROCESS_SHAVE0;
    //attributes of convolution initialied
    conv_context context = {
        .dma = (dma_context){
            .dmaCreateTransactionFullOptions =
            dmaCreateTransactionFullOptions,
            .dmaStartTask = dmaStartTask,
            .dmaWaitTask = dmaWaitTask,
            .task = task,
        },
        .com = (common_context){
            .shaveId = shaveId,
            .jumpTable = jumpTable
        },
        .mem = (memory_context){
            .setAlignedMem = setAlignedMem,
            .getAlignedMem = getAlignedMem
        },
        .info = info,
    };
    CONV_DDR_PTR conv_ddr = (CONV_DDR_PTR) jumpTable(CM_conv_ddr);
    conv_ddr(firstMapNo, lastMapNo, &context);
    SHAVE_HALT;
}
```

The first argument refers to the different structs, which are passed as variables into the function and later will be passed into SHAVE's DDR side for the appropriate computations. The aforementioned computations receive an argument named context that contains all the required parameters for performing the computation. Convolution layers also need to make use of the DMA engine. This is accomplished by using the compiler attribute syntax, which positions the structures into the section .cmx.cdmaDescriptors that is placed at the region `CMX_DMA_DESCRIPTORS` 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. Also, here arguments for both memory allocaton and the Jumptable are shown.

5.3.1 JumpTable and SHAVE's Computations

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, providing 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.

Listing 5.10: "/shave/ddr/ddr$_{functions.c}$"

```
#include <ddr_functions_exports.h>
FUNCPTR_T jumpTable(int i)
```

```
3  {
4      struct lib_function func = lib[i];
5
6      switch (func.category) {
7
8      // Utility Functions
9      case func_common:
10         return func.cat.cm.func;
11     case func_conv:
12         return func.cat.conv.func;
13     case func_pool:
14         return func.cat.pool.func;
15     case func_acc:
16         return func.cat.acc.func;
17     case func_fc:
18         return func.cat.fc.func;
19     case func_lrn:
20         return func.cat.lrn.func;
21     case func_im2col:
22         return func.cat.im2col.func;
23     }
24
25     return 0;
26 }
```

Above, the reason that structs contain a variable `ddr_function` is shown, as SHAVEs have to know from which function placed in DDR will start their execution. Thus, after execution is started from the entry points in CMX, SHAVEs jump into the DDR functions in order to decrease SHAVE code demands and increase the ones for data.
It is a fact that in order to produce an output map in convolution, every input channel with the corresponding weights has to be multiplied and then accumulated. So for every output map that is going be produced DMA transfers for all input channels have to be implemented. According to the striding parameter's value, it determines the number of DMA buffers. Group parameter may increase the executions of the computation function. Finally, after calling assembly functions for computation and accumulating output maps will be returned with DMA engine. Below, the convolution function placed in DDR with a brief pseudocode is shown:

Listing 5.11: SHAVE's convolution function

```
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
```

Worth to mention here is two optimization techniques, which reduced the execution time:

- SHAVEs undertake the whole input channel so far and they are working on it as figured in 5.12. With the partitioning methodology a way to place the image classification's networks input into CMX has been found, but an efficient scheduling methodology on how SHAVEs will work at these partitions had to be implemented too. First of all, it is a fact that as partitions increase, the execution requirements of the layer increase so the minimum partition number has to be found. Supposing that our CNNs have input images of 3x227x227, an input channel requires almost 105 KB, but the demands for evey shave are bigger as space for an input double buffer and the output is allocated as well. The double buffering technique is used for speedup, as while DMA transfers data from DDR to CMX in one buffer, at the same time the computation is performed in another buffer.
  As far as scheduling is concerned, the first implementation was to use four convolution-object pointers instead of one. By this way, SHAVEs worked in series, by computing first line0, after

they undertake line1 5.12 and so on. The process described above is presented in the following figure, for four input lines, eight convolution kernels and four shaves. Different shades of gray symbolize output maps on which SHAVES 1,2,3 work, meanwhile the other colours refer to the ouput maps of SHAVE0.

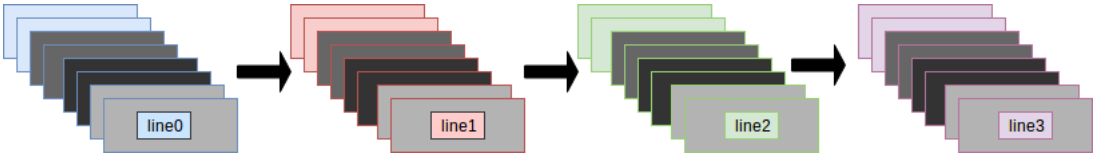


Figure 5.12: First parallization Scheme.

The above methodology ensures that SHAVEs are started once and work on input partitions in a serial manner. An optimization methodology has been applied here, as by giving an argument to SHAVEs a table of pointers describing the whole input channel SHAVEs are obligated to work on the whole input channel. This parallelization schema does not require the use of locks, because there is no data sharing and SHAVEs are SIMD. This is of great benefit in an embedded platform, since locks and synchronization of accesses tend to increase power consumption.

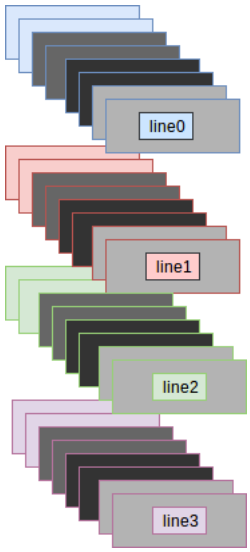


Figure 5.13: Final Optimized Parallization Scheme.

- Reduced number of DMA transfers [16], as 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 transfers 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.

Listing 5.12: SHAVE's convolution function

```
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
```

```
    end for
    Send the group of map buffers back with DMA
end for
```

5.3.2 Convolution 1x1 Kernel

Finally an assmebly function executed by SHAVEs in CMX will be presented. In order to deploy GoogleNet, it is necessary to use 1x1 kernel convolutional functions. GoogleNet uses a lot 1x1 convolution, almost in the half of its total layers, due to the reasons, which are desribed here:

- To reduce the dimensions inside GoogleNet's "inception module".
- To make network deep, as layers become almost double.
- To add more non-linearity by having ReLU immediately after every 1x1 convolution.

It can be seen from GoogleNet's description, that 1x1 convolutions, are specially used before 3x3 and 5x5 convolution to reduce the dimensions. It should be noted that a two step convolution operation can always be combined into one, but in this case and in most other deep learning networks, convolutions are followed by non-linear activation and hence convolutions are no longer linear operators and cannot be combined. These type of functions were not implemented in the source code of the Myriad 2 developement tool. Taking this into account assembly routines for 1x1 convolution have been implemented. As it was said in the subsection of the preprocess, before transfering input data with DMA it is important to align them because convolution routines are one dimensional. The computational function is easy as for every input channel, only one value of weight has to be provided as an argument. Moreover, because 1x1 kernels are applied only with stride equals to one, input and output dimensions are the same.

Listing 5.13: Pseudocode of 1x1 Convolution in High Level Language

```
function convolution1x1s1(in, out, conv, width)
    for every input_element in width do
        sum = 0
        sum += in[element] * conv[0]
        out[element] = sum
    end for
```

First of all, in order to test GoogleNet and SqueezeNet, which have 1x1convolutional layers, this routine was written in C language. Execution time of GoogleNet was near 2 seconds, where 1x1 convolutional nodes lasted almost 1.85 seconds. After this, SHAVE Assmebly function was written with serial commands using integer register file. This routine was like this:

Listing 5.14: Serial Assembly Pseudocode

```
function convolution1x1s1(in, out, conv, width)
    load reg1, out
    load reg2, conv
    load reg3, width
    loop:
        load reg0, in
        mul reg1, reg0, reg2
        st out, reg1
        add in, in, 1
        add out, out, 1
        decr reg3, reg3, 1
        brnez loop
    end loop
```

The above function was not so helpful as far as the execution time is concerned. If the width is equal to two hunderd for example, two hundred accesses to memory will take place. Generally, memory accesses and especially loading data from memory addresses is the slowest procedure in a program,

as it lasts the most clock circles. By taking this into account, vector register file was used, in order to load more than one elements parallel in vector registers and multiply them in parallel. Also, SHAVE's Internal architecture offers us the ability to use in parallel different units such as Branch and Load/Store unit for example.

Listing 5.15: Efficient SHAVE Assembly Pseudocode

```
1  function convolution1x1s1(in, out, conv, width)
2      load vec2, conv
3      load reg3, width
4      loop:
5          //load and increase address to show the next 8 elements
6          load vec1, in || add vec1, vec1, 1
7          //multiply every element from vec0 with-
8          //less significant number of vec2
9          mul vec4, vec1, vec2 || swizzle vec2
10         st out, vec4
11         shr reg3, reg3, 3
12         add out, out, 1
13         brnez loop
14     end loop
```

This routine had less no operations commands end led to speedup, as the less memory accesses and parallel multiplications led to 300 ms from almost 2 seconds. Despite the fact that the difference was remarkable, still improvement margins existed. This led to a new approach, which is shown below.

Chapter 6

General Matrix to Matrix Multiplication in Deep Learning

It is a fact that a lot of optimizations have been applied in order to speed up convolution. The results were remarkable but the final aim was to minimize execution time of 1x1 kernels, where the no operations commands of our assembly delayed the whole process. Taking this into account, a different way of convolution had to be implemented. More specifically, a way which is supported by General Matrix to Matrix Multiplication (GEMM) and uses the tecnhique of Image to Column (Im2Col) conversion.

6.1 Theoretical Analysis of Im2Col Convolution

General Matrix to Matrix Multiplication known as GEMM does exactly what it says on the tin, multiplies two input matrices together to get an output one as staten in [15]. The difference between it and the kind of matrix operations used to in the 3D graphics world is that the matrices it works on are often very big. For example, a single layer in a typical network may require the multiplication of a 256 row, 1,152 column matrix by an 1,152 row, 192 column matrix to produce a 256 row, 192 column result. Naively, that requires 57 million (256 x 1,152, x 192) floating point operations and there can be dozens of these layers in a modern architecture. It is a fact that there are networks that need several billion FLOPs to calculate a single frame. A diagram of how it works is shown below 6.1:

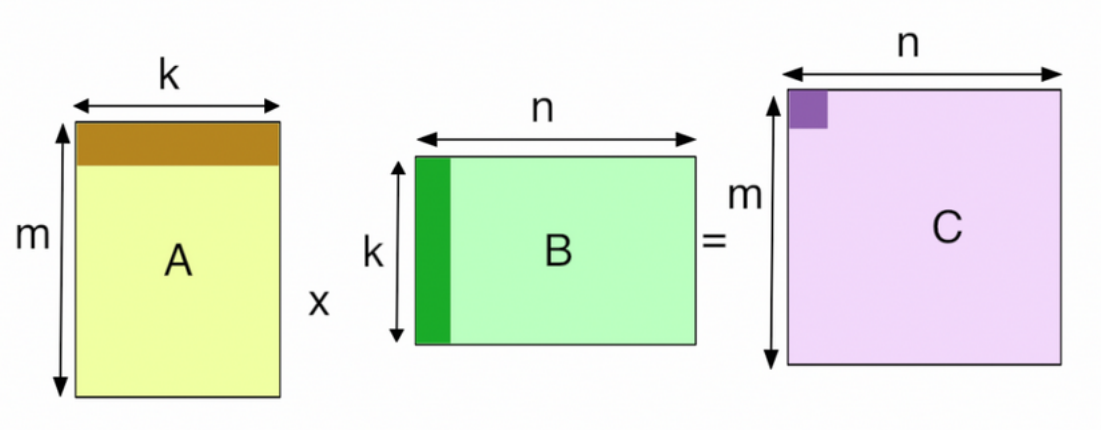


Figure 6.1: How GEMM works.

6.1.1 Fully-Connected Layers

Fully-connected layers are the classic neural networks that have been around for decades, and it's probably easiest to start with how GEMM is used for those. Each output value of a FC layer looks at every value in the input layer, multiplies them all by the corresponding weight it has for that input index, and sums the results to get its output. In terms of the diagram above, it looks like this:

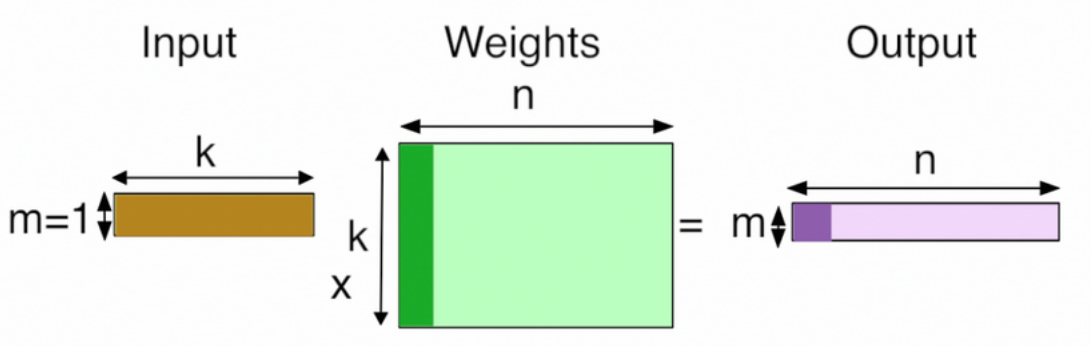


Figure 6.2: Fully Connected Layer.

There are 'k' input values, and there are 'n' neurons, each one of which has its own set of learned weights for every input value. There are 'n' output values, one for each neuron, calculated by doing a dot product of its weights and the input values.

6.1.2 Convolutional Layers with GEMM

Using GEMM for the convolutional layers is a lot less of an obvious choice. A conv layer treats its input as a two dimensional image, with a number of channels for each pixel, much like a classical image with width, height, and depth. Unlike the images used to dealing with though, the number of channels can be in the hundreds, rather than just RGB or RGBA! The convolution operation produces its output by taking a number of 'kernels' of weights. and applying them across the image. In the figure 6.3 is shown how an input image and a single kernel look like:

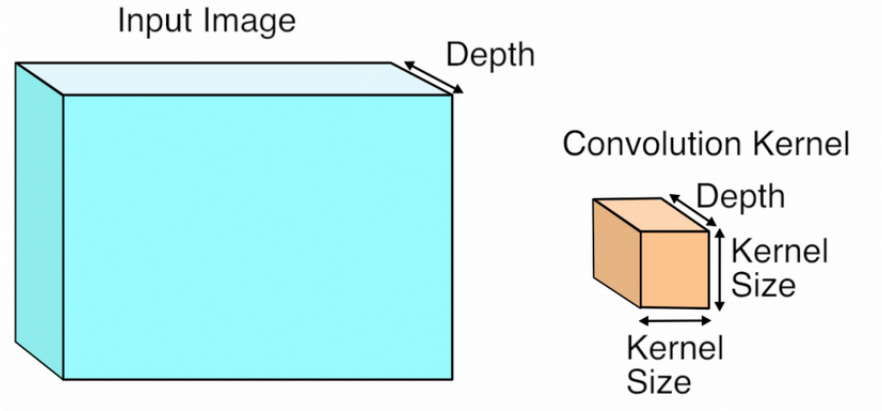


Figure 6.3: RGB Form of Input and Kernel.

Each kernel is another three-dimensional array of numbers, with the depth the same as the input image, but with a much smaller width and height, typically something like 7x7. To produce a result,

a kernel is applied to a grid of points across the input image. At each point where it's applied, all of the corresponding input values and weights are multiplied together, and then summed to produce a single output value at that point. Here's what that looks like visually:

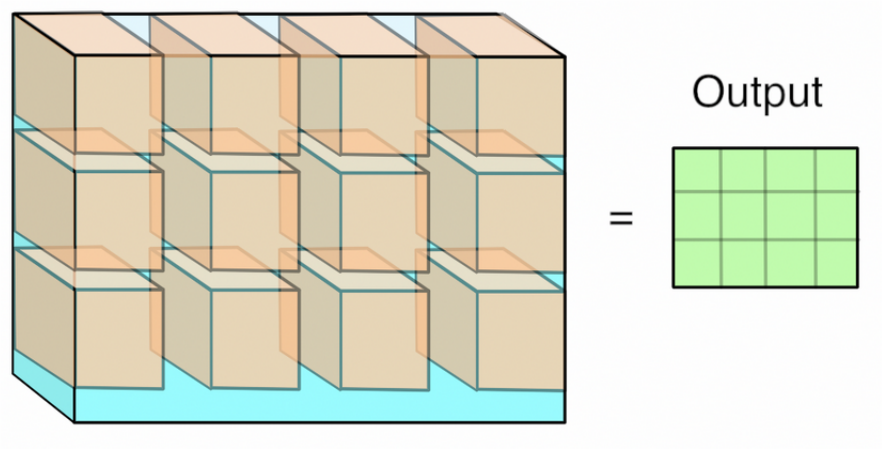


Figure 6.4: Kernels 'fit' in Input and provide Output.

This operation is something like an edge detector. The kernel contains a pattern of weights, and when the part of the input image it's looking at has a similar pattern it outputs a high value. When the input doesn't match the pattern, the result is a low number in that position. Here are some typical patterns that are learned by the first layer of a network, courtesy of the awesome Caffe and featured on the NVIDIA blog:

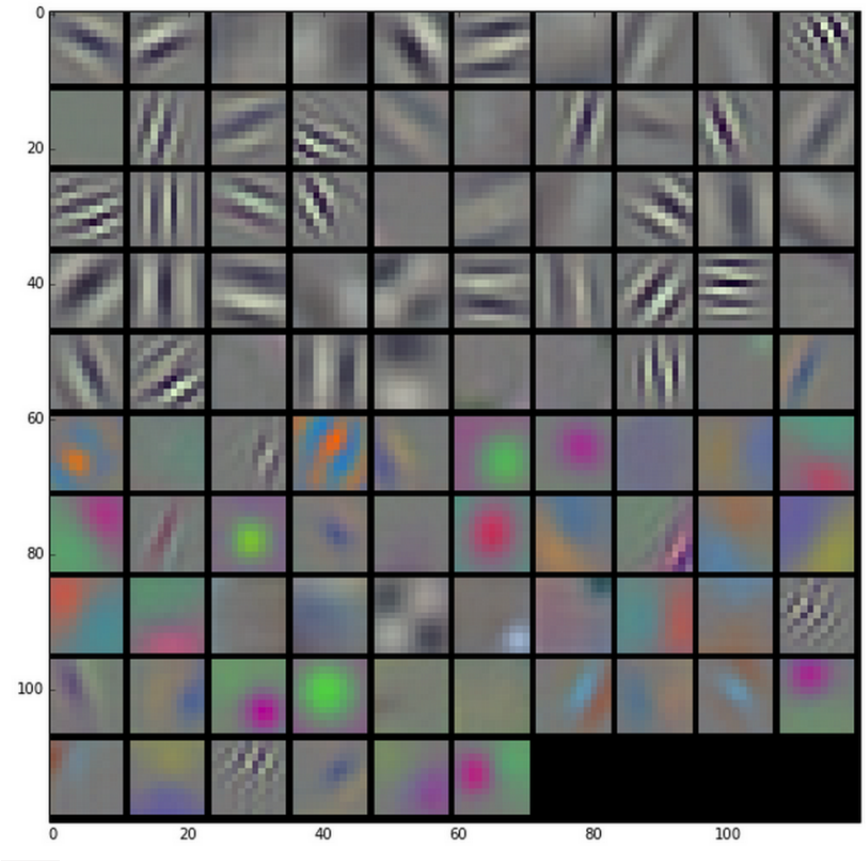


Figure 6.5: Kernels look for special patterns in Input.

Because the input to the first layer is an RGB image, all of these kernels can be visualized as RGB too, and they show the primitive patterns that the network is looking for. Each one of these 96 kernels is applied in a grid pattern across the input, and the result is a series of 96 two-dimensional arrays, which are treated as an output image with a depth of 96 channels. Someone can see how each one of these is a bit like an edge detector optimized for different important patterns in the image, and so each channel is a map of where those patterns occur across the input.
The key controlling factor for the grid in the figure 6.5 above is a parameter called 'stride', which defines the spacing between the kernel applications. For example, with a stride of 1, a 256x256 input image would have a kernel applied at every pixel, and the output would be the same width and height as the input. With a stride of 4, that same input image would only have kernels applied every four pixels, so the output would only be 64x64. Typical stride values are less than the size of a kernel, which means that in the diagram visualizing the kernel application, a lot of them would actually overlap at the edges.
As far as GEMM in convolution is concerned, it seems like quite a specialized operation. It involves a lot of multiplications and summing at the end, like the fully-connected layer, but it's not clear how or why this into a matrix multiplication for the GEMM should be turned.
The first step is to turn the input from an image, which is effectively a 3D array, into a 2D array that can be treat like a matrix. Where each kernel, which is applied, is a little three-dimensional cube within the image. So each one of those cubes of input values can be taken and be copied as a single column into a matrix. This is known as im2col, for image-to-column, from an original Matlab function, and here's how it seems:

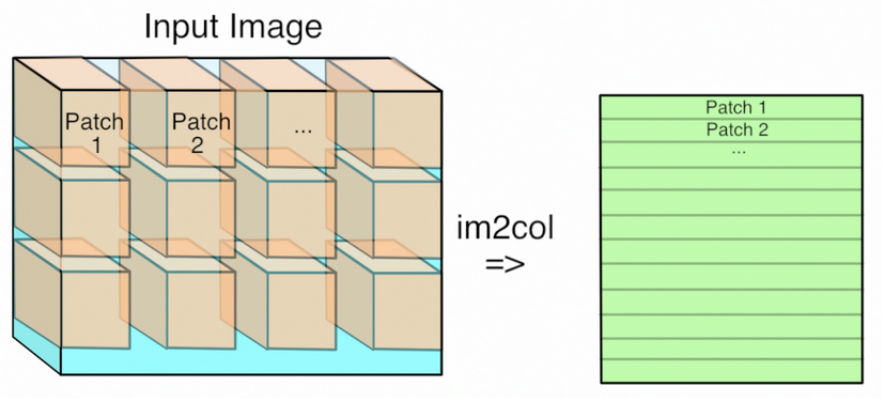


Figure 6.6: Conversion of the input to 2 Dimensions.

Now the expansion in memory size seems to happen with this conversion if the stride is less than the kernel size. This means that pixels that are included in overlapping kernel sizes will be duplicated in the matrix, which seems inefficient. Now the input image is in matrix form, the same for each kernel's weights has to be done, serializing the 3D cubes into rows as the second matrix for the multiplication. Here's what the final GEMM looks like:

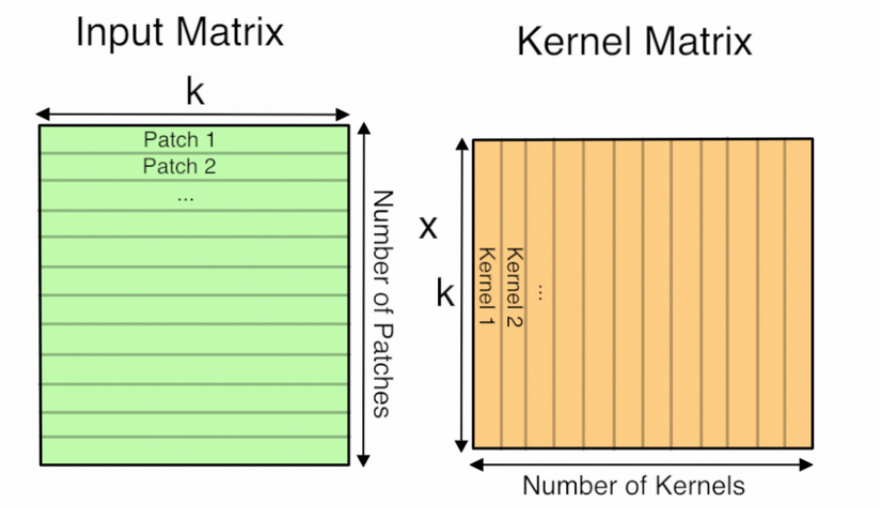


Figure 6.7: GEMM's Convolution ready.

Here 'k' is the number of values in each patch and kernel, so it's kernel width * kernel height * depth. The resulting matrix is 'Number of patches' columns high, by 'Number of kernel' rows wide. This matrix is actually treated as a 3D array by subsequent operations, by taking the number of kernels dimension as the depth, and then splitting the patches back into rows and columns based on their original position in the input image.

6.2 Im2Col's Convolution Implementation in Myriad2

Sections below show the way that Im2Col layer is implemented in Myriad2.

6.2.1 Import Input and Weight Data

First of all, the input and weight parameters are given from the CAFFE Framework with the same way. It is a fact that these data have to be edited and be reshaped in order that the GEMM's convolution happen but this is implemented in the code of the platform. This methodology gives the opprortunity to the programmer to choose with which way the convolution function will be imlpemented. In other words, again almost the same number of statical buffers placed in DDR for the network weights are used. The only difference is one additional buffer, which allocates as many bytes as the biggest Im2Col layer is going to need. This buffer is used for the Image to Column transformation.

6.2.2 Preprocess and start Layer's Execution

Below, the struct is provided with the variables, which are needed to be set in order to implement Im2Col approach.

Listing 6.1: Im2Col struct with the appropriate information

```
typedef struct {
    u8 *input;
    u8 inputBPP;

    u16 tiles;
    u8 *input_column;
    u8 *output;
    u8 outputBPP;

    u8 *conv_weights;

    u8 *conv_biases;
    u8 kernelBPP;

    u16 shaves;
    u16 channels;
```

```
    u8 ddr_function;
    u8 kernel_h;
    u8 kernel_w;
    u8 with_relu;

    u16 maps;
    u8 c_group;
    u8 in_stride;

    u16 offset;
    u16 in_col_height;
    u16 in_row_width;
    u16 weight_col_height;
    u16 weight_row_width;

} im2col_info;
```

The preprocess function here takes over to transform the 3 dimensional input volume into a 2 dimensional array and repeat the elements which are needed according to the values of the network parameters. In order to accomplish this, someone has to consider that for every input channel `kernel_size * kernel_size` elements have to been received, thus the size of the elements will be `input_channels * kernel_size * kernel_size` as shown in the figure 6.6. The other dimension of the array is going to be equal to `output_width * output_height`. The total dimensions and how they are going to increase related to the 3-Dimensional volume, has to do with the striding value as well. Below, preprocessing values are presented, which depend on the zero-padding value. In case that padding parameter is provided, first zero-padding elements are added and then the volume is transformed:

Listing 6.2: Import padding Elements into the initial 3D Volume

```
u8* NewMapWithPad(u8* previous, int width, int height, int kernel, int stride, int padding, int channels){

    int out_height = floor_int(height + 2*padding - kernel, stride) + 1;
    int out_width = floor_int(width + 2*padding - kernel, stride) + 1;

    int paddingbottom = (out_height - 1) * stride + kernel - height - padding;
    int paddingright = (out_width - 1) * stride + kernel - width - padding;

    int new_width = width + padding + paddingright;
    int new_height = height + padding + paddingbottom;

    for (int ch = 0; ch < channels; ch++){
        for (int y_column = 0; y_column < new_height; y_column++){
            for (int x_pad = 0; x_pad < padding; x_pad ++){
                data_pad[2 * ch * (new_width * new_height) + 2 * y_column * new_width + 2 * x_pad] = 0;
                data_pad[2 * ch * (new_width * new_height) + 2 * y_column * new_width + 2 * x_pad + 1] = 0;
            }
            for (int x_pad = (new_width - paddingright); x_pad < new_width; x_pad++){
                data_pad[2 * ch * (new_width * new_height) + 2 * y_column * new_width + 2 * x_pad] = 0;
                data_pad[2 * ch * (new_width * new_height) + 2 * y_column * new_width + 2 * x_pad + 1] = 0;
            }
        }
        for (int x_column = padding; x_column < new_width - padding; x_column++){
            for (int y_pad = 0; y_pad < padding; y_pad++){
                data_pad[2 * ch * (new_width * new_height) + 2 * y_pad * new_width + 2 * x_column] = 0;
                data_pad[2 * ch * (new_width * new_height) + 2 * y_pad * new_width + 2 * x_column + 1] = 0;
            }
            for (int y_pad = (new_height - paddingbottom); y_pad < new_height; y_pad++){
                data_pad[2 * ch * (new_width * new_height) + 2 * y_pad * new_width + 2 * x_column] = 0;
                data_pad[2 * ch * (new_width * new_height) + 2 * y_pad * new_width + 2 * x_column + 1] = 0;
            }
        }
        for (int y_pad = (padding); y_pad < (new_height - paddingbottom); y_pad++){
            for (int x_column = (padding); x_column < (new_width - paddingright); x_column++){
                data_pad[2 * ch * (new_width * new_height) + 2 * y_pad * new_width + 2 * x_column] =
                previous[2 * ch * (width*height) + 2 * (y_pad * width - padding * width) + 2 * (x_column - padding)];
                data_pad[2 * ch * (new_width * new_height) + 2 * y_pad * new_width + 2 * x_column + 1] =
               previous[2 * ch * (width*height) + 2 * (y_pad * width - padding * width) + 2 * (x_column - padding) + 1];
            }
        }
    }

}
```

For every input channel zero-padding elements are added into the boundary areas as shown above, before transform below:

Listing 6.3: Convert Input Volume into 2-dimensional array

```
u8* InputstoColumns(u8* pointer, int width, int height, int kernel, int stride, int padding, int channels){

    int out_height = floor_int(height + 2 * padding - kernel, stride) + 1;
    int out_width = floor_int(width + 2 * padding - kernel, stride) + 1;


    for (int ch = 0; ch < channels; ch++){
        for(int y_kernel = 0; y_kernel < kernel; y_kernel++){
            for(int x_kernel = 0; x_kernel < kernel; x_kernel++){
                for(int y_stride = 0; y_stride < out_height; y_stride++){
```

```
                    for(int x_stride=0; x_stride < out_width; x_stride++){
                        data[(2*ch*(kernel*kernel*out_width*out_height))+(2*y_kernel*(kernel*out_height*out_width))+
                        (2*x_kernel*(out_width*out_height))+(y_stride*2*out_width)+(2*x_stride)] =
                        pointer[(x_stride*2*stride+x_kernel*2)+(stride*2*y_stride*width)+(y_kernel*height*2)+
                    (ch*(width*height)*2)];
                        data[(2*ch*(kernel*kernel*out_width*out_height))+(2*y_kernel*(kernel*out_height*out_width))+
                        (2*x_kernel*(out_width*out_height))+(y_stride*2*out_width)+(2*x_stride)+1] =
                        pointer[(x_stride*2*stride+x_kernel*2)+(stride*2*y_stride*width)+(y_kernel*height*2)+
                        (ch*(width*height)*2)+1];
                    }
                }
            }
        }
    }
    return data;
}
```

As discussed above, after the Im2Col transform, the 2-dimensional arrays may have increased memory demands. For sure, this is translated into more memory transfers, but also a partition strategy has to be implemented in order to overcome the overflow problem. This strategy has been implemented with the following way, partition the 2-dimensional array into groups of columns. For every single group of them, seperate it into the number of SHAVEs in order that the final group of columns fit in local CMX memory. This means that the number of the partition depends to the number of SHAVEs too. An example is shown below, where the `input_width` is partitioned in 4 tiles and every tile assigns its columns into 3 SHAVEs.

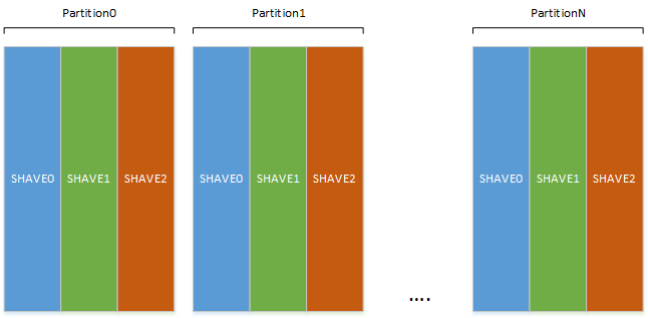


Figure 6.8: Parallization scheme of Im2Col Approach

6.2.3 SHAVE code and Computation

The entry code for Im2Col approach of convolution is stated again in CMX and is described below:

Listing 6.4: Boot code into CMX for Im2Col placed in "/shave/cmx/entry.c"

```
void shave_im2col(im2col_info **info, u32 firstMapNo, u32 lastMapNo, J_FUNCPTR_T jumpTable){
    int shaveId = swcWhoAmI() - PROCESS_SHAVE0;
    im2col_context context = {
        .dma = (dma_context){
            .dmaInitRequester = dmaInitRequester,
            .dmaCreateTransactionFullOptions = dmaCreateTransactionFullOptions,
            .dmaStartListTask = dmaStartListTask,
            .dmaWaitTask = dmaWaitTask,
            .task = task,
            .ref = ref
        },

        .com = (common_context){
            .shaveId = shaveId,
            .jumpTable = jumpTable
        },

        .mem = (memory_context){
            .setAlignedMem = setAlignedMem,
            .getAlignedMem = getAlignedMem
        },

        .info = info,
    };

    IM2COL_DDR_PTR im2col_ddr = (IM2COL_DDR_PTR) jumpTable(FT_im_col);
    im2col_ddr(firstMapNo, lastMapNo, &context);

    SHAVE_HALT;
}
```

Again the Jumptable is being used in order to follow the main idea and place the SHAVE code of the layer into DDR apart from the computation function. As someone can see from the entry code the ddr function "`im2col_ddr`" is called with parameters firstMapNo, lastMapNo and context. The 2 first parameters indicate the range of columns that every SHAVE is going to undertake in every partition. The third variable "context", which is passed by reference contains the following information as someone observes in "`im2col_context`":

Listing 6.5: Definition of the DDR's function parameters

```
#ifndef __DDR_CONV_H__
#define __DDR_CONV_H__

// 1: Includes
//--------------------------------------------------------------------------
#include "ddr_common.h"
#include <conv_api.h>
#include <im2col_api.h>

// 2:  Source Specific #defines and types  (typedef, enum, struct)
//--------------------------------------------------------------------------
typedef struct {
    dma_context dma;
    memory_context mem;
    common_context com;

    conv_info **info;
} conv_context;

typedef struct {
    dma_context dma;
    memory_context mem;
    common_context com;

    im2col_info **info;
} im2col_context;


typedef void (*CONV_DDR_PTR) (u32 firstMapNo, u32 lastMapNo, conv_context *context);
typedef void (*IM2COL_DDR_PTR) (u32 firstMapNo, u32 lastMapNo, im2col_context *context);

// 3: Static Local Data
//--------------------------------------------------------------------------

// 4:  Exported Functions (non-inline)
//--------------------------------------------------------------------------
void conv_ddr(u32 firstMapNo, u32 lastMapNo, conv_context *context);
void im2col_ddr(u32 firstMapNo, u32 lastMapNo, im2col_context *context);


#endif//__DDR_CONV_H__
```

The SHAVE code placed in DDR, is responsible to transfer the input matrix and compute the product with the weight matrix placed in the uncached memory of the DDR. This memory space can be accessed by the local memories as well, so both weights and biases are stored there. The assembly function, which is used for this operation is vector with vector product and is saved into the CMX memory. The computation function is being called many times. Below a pseudocode of the pre-Computation function in DDR is shown:

Listing 6.6: Im2Col pre-Computation Code

```
for every tile in the partitions of the width do
    for every column in the tile do
        Bring input columns with DMA
        for every row in the 2-dimensional weight array do
            Apply vector with vector multiplication in Assebmly
        end for
        //after this the whole vector will be ready
        Accumulate the vector result with the biases
        Send the output columns back with DMA
    end for
end for
```

Finally, the DMA transfering methodology is presented with two brief schemes as shown in 6.9.

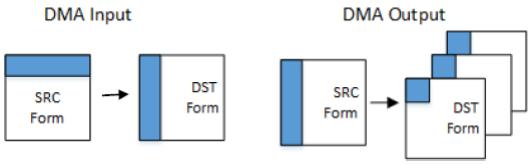


Figure 6.9: DMA transfering Methodology

The code, which is used for the Input/Output transfers is the following:

Listing 6.7: Input/Output DMA code

```
for(u8 gr = 0; gr < info[0]->c_group; gr++){
    for (u16 i = 0; i < info[0]->tiles ; i++){
        for (u8 mapNo = firstMapNo; mapNo < lastMapNo; mapNo++){
            // bring columns of every tile
            ref[0] = dmaCreateTransactionFullOptions(
                id,
                &task[0],
                (info[0]->input_column + (gr * info[0]->in_col_height * info[0]->in_row_width * info[0]->inputBPP))
                + i * (info[0]->in_row_width/info[0]->tiles) * info[0]->inputBPP + mapNo * info[0]->inputBPP,   // src
                input_column[mapNo],    // dst
                info[0]->in_col_height * info[0]->inputBPP,  // byte length
                info[0]->inputBPP,  // src line width
                info[0]->in_col_height * info[0]->inputBPP,  // dst line width
                info[0]->inputBPP * info[0]->in_row_width,      // src stride
                info[0]->inputBPP);   // dst stride
            dmaStartListTask(ref[0]);
            dmaWaitTask(ref[0]);

            ...
            //Computation routine
            //for every input vector column multiply it with the weight matrix

            ref[0] = dmaCreateTransactionFullOptions(
                id,
                &task[0],
                (u8 *)(localOutput[mapNo]), // src
                info[0]->output + mapNo * info[0]->outputBPP + i * (info[0]->in_row_width/info[0]->tiles)
                * info[0]->outputBPP + (gr * info[0]->in_row_width * info[0]->weight_col_height * info[0]->outputBPP),// dst
                info[0]->weight_col_height * info[0]->outputBPP,     // byte length
                info[0]->outputBPP * info[0]->weight_col_height,// src line width
                info[0]->outputBPP,// dst line width
                info[0]->outputBPP,                    // src stride
                info[0]->in_row_width * info[0]->outputBPP);                    // dst stride
            dmaStartListTask(ref[0]);
            dmaWaitTask(ref[0]);
        }
    }
}
```

Chapter 7

Evaluation of the Implementation

This chapter evaluates the implementation testing some CNNs from ILSVR challenge trained in ImageNet. Furthermore, conclusions are drawn for the rating of the software, which was produced. Finally, the explanation of the results give a first impression for the Myriad's hardware and architecture.

7.1 Evaluation of Direct's Convolution Parameters

It is a fact that every convolution layer has the parameter of **`kernel_size`**. For every convolution filter, the number of the operations is equal with the whole dimensions of the filter (e.g. 3x3 filter means 9 pixels and 9 multiplications). This means that the more the kernel size is, by raising the mutiplications, the more the execution time of the layer will be. Below 2 layers executed with direct convolution are shown with the same `input_dimensions`, `output_maps` and striding parameter, their only difference is their `kernel_size`.

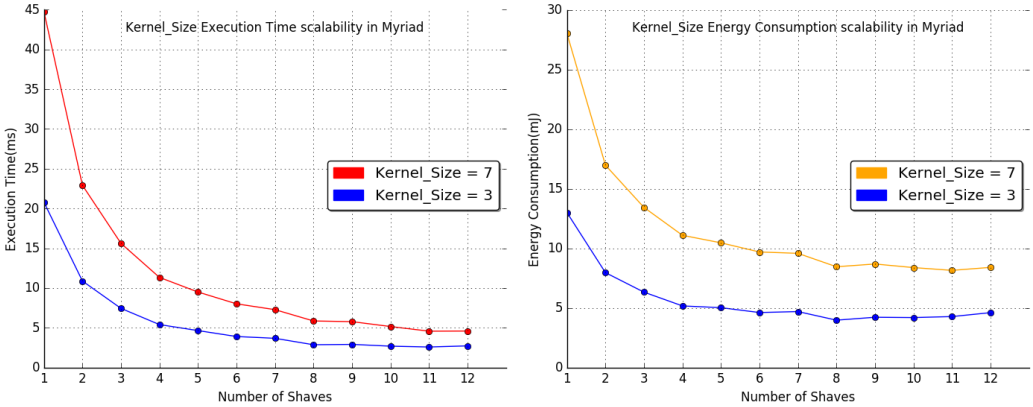


Figure 7.1: Increase of the `kernel_size` means increase of the Execution time and Energy consumption.

As far as the **partitioning methodology** is concerned, supposing that an input volume has dimensions octopus than an other one, the maximum execution time of this convolution should be octopus as well. This means that the best case scenario for the speedup is x8. In order to compute the speedup that the partitioning function offers, 2 layers with almost the same convolution's parameters apart from the input channels were executed. In fact the layer with the smaller dimensions had three input channels more, the results were the following:

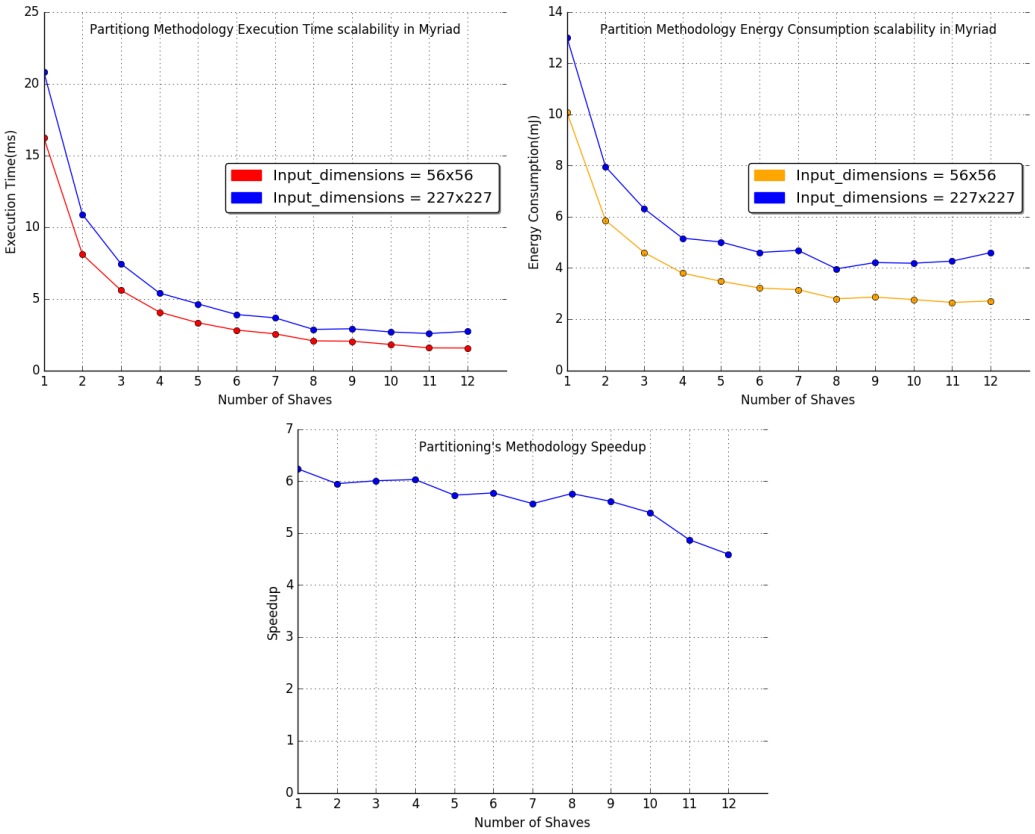


Figure 7.2: Partitioning's Parallization Scheme succeeds a notable Speedup.

In chapter 5 **group parameter** was explained, which is used for a reduction in the memory demands of the weights. As referred previously, group parameter means that the computation function may be executed twice for some combinations of VPUs number. Below are considered 2 different layers, which have exactly the same parameters apart from the group variable. These layers are layer number 3 and 4 in AlexNet.

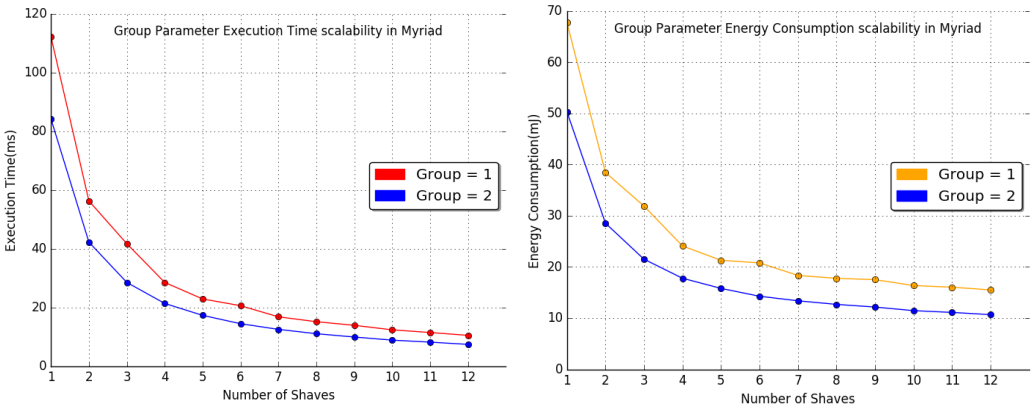


Figure 7.3: Group Parameter Dependance.

From the figure 7.3 is becoming clear that by reducing the weights both execution time and energy consumption are reduced. The reason is that the references to the main memory are less and this seems to be more important factor than times, that the computation routine is going to be executed. Finally, **striding** is a factor, which determines the output dimensions. Striding greater than 1 means that the output dimensions will be decreased and this parameter shows how many times the square of each weight kernel can be placed in the input. In other words, striding parameter greater than one is going to reduce operations and improve both execution time and energy consumption.

7.2 Comparison of the Different Convolutional Approaches

As far as the Im2Col approach is concerned, in the previous Chapter was shown, that the input has $width = output_{width} * output_{height}$ and $height = kernel_{size} * kernel_{size} * input_{channels}$. The weight matrix dimensions are $width = kernel_{size} * kernel_{size} * input_{channels}$ and $height = output_{maps}$. Description above shows that for every vector with vector product $kernel_{size} * kernel_{size} * input_{channels}$ elements are multiplied. Taking into account the number of vector to vector products $kernel_{size}$ seems to be an important factor as well as the $input_{channels}$. Below is shown how 2 layers with different $kernel_{size}$ scale for every convolutional approach.

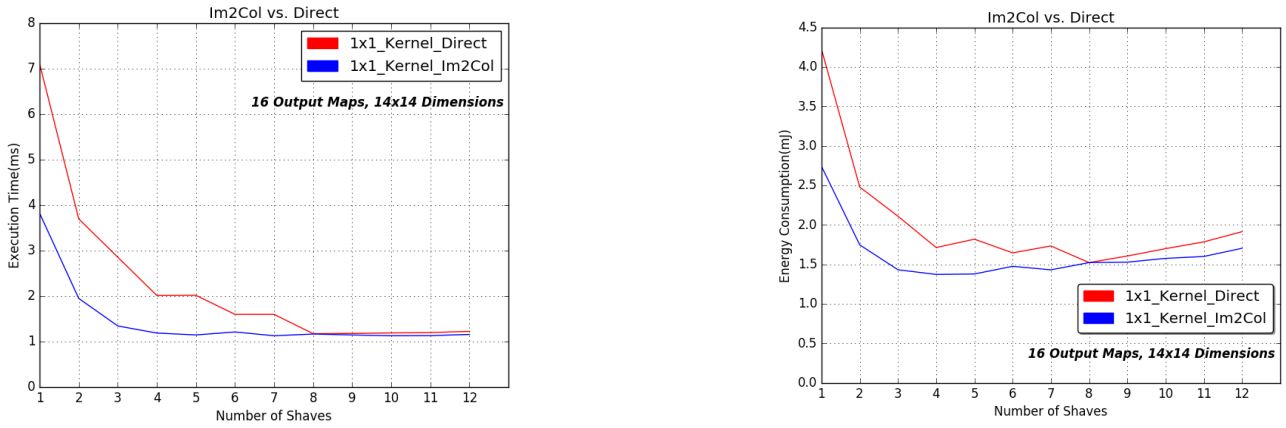


Figure 7.4: Im2Col for 1x1 kernel succeeds better execution time for restricted number of SHAVEs.

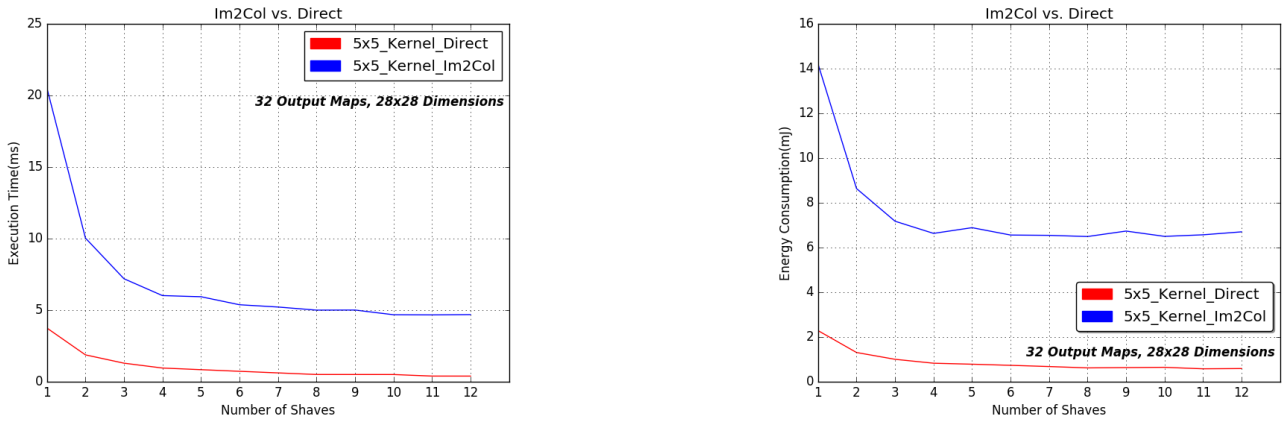


Figure 7.5: Direct is faster for bif kernels.

It is a fact that for small kernels the Im2Col scalability is saturated after 4 SHAVEs, because by partitioning the width into many tiles the number of columns, which correspond to every shave are decreased. So a lot of transfers happen for few computations. On the other hand for bigger kernels, where the output matrix width is a bigger number, it is observed that Im2Col approach continues to scale for bigger combinations of vector process units too.
Input channels are going to influence the implementation with the same way but without the square factor. As far as the output maps are concerned, below in 7.8 is seen that increasing output maps will lead to an increase in execution time, as expected. Furthermore, in direct approach the parallization scheme of SHAVEs is based on the depth as stated in previous chapters, which justifies the slope of the 2 lines in 7.6.

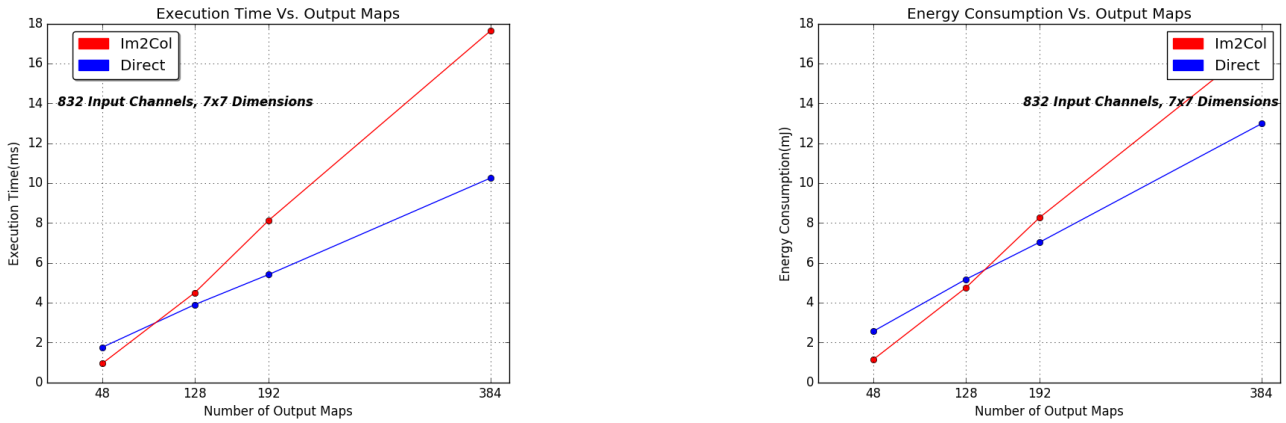


Figure 7.6: Influence of the number of Output Maps.

The last factor is $output_{dimensions}$, which are depending on the network's striding parameter. It is a fact that 1x1 convolution are used in the late layers of the networks, when the dimensions have been decreased (e.g GoogleNet). This means that the input width of the 1x1 layers is limited related to bigger kernels. Trying to partition a 2-dimensional array with small width does not offer a good scalability, as power and time is consumed pointless in tranfers. According to the above, bigger kernels have good scalability but not a notable execution time. Worth to mention is that in 1x1 kernel's execution time for a little VPUs number is better in Im2Col approach than direct.

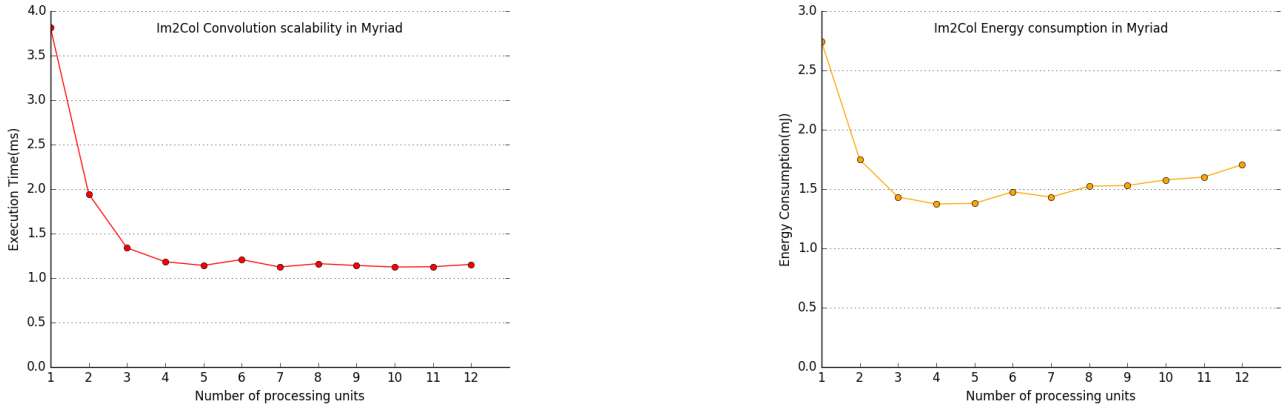


Figure 7.7: Scalability for 1x1 kernel.

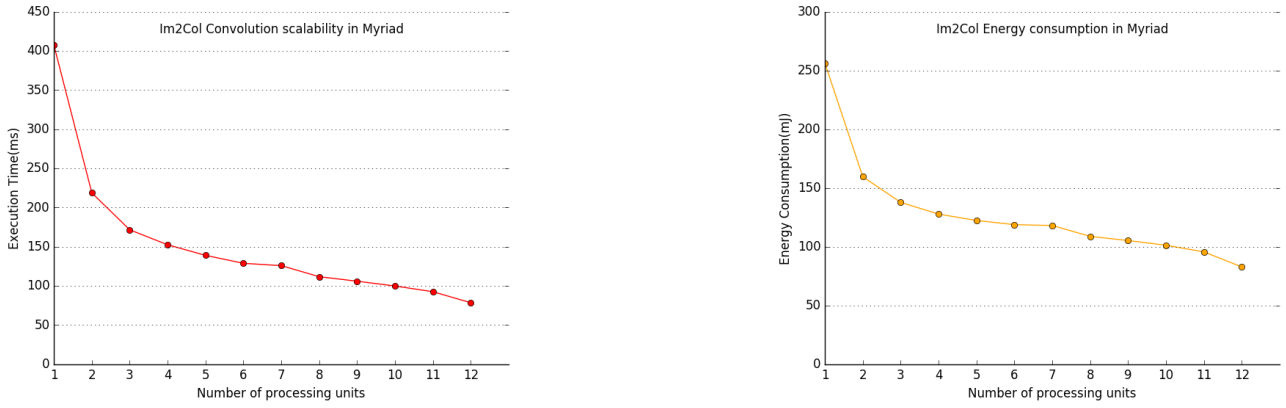


Figure 7.8: Improved scalability for 3x3 kernel.

In chapter 5 was explained that when the kernel size is greater than the striding parameter, the memory demands of Im2Col's layers are increased. The diagram below shows exactly this by providing

memory demands of the first layer of every CNN. Squeezenet is the only one, which has not so big difference between these 2 numbers, because kernel is equals to 3 and striding to 2.

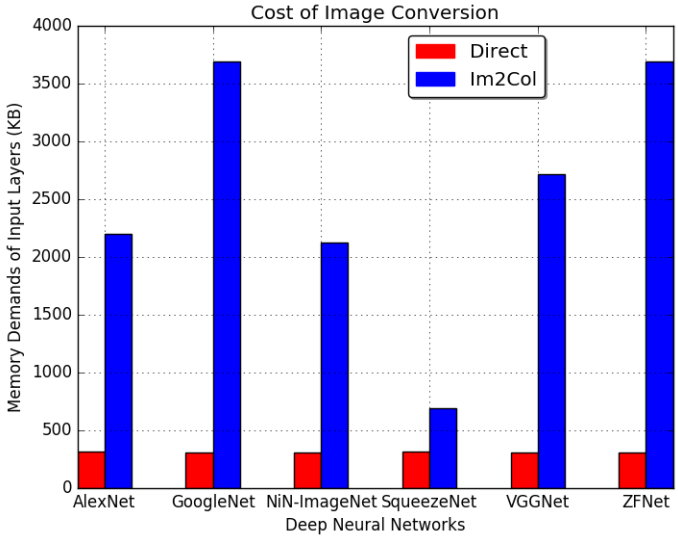


Figure 7.9: Increased memmory Demands of Im2Col approach.

Below is provided a comparison for every layer in Squeezenet between the 2 convolution ways. It is clear that the only values, which are competitive refer to 1x1 kernels. Finally, GoogleNet provides 7 layers with better execution time with Im2Col approach for every combination of SHAVEs, while with number of SHAVEs less than 4 Im2Col's times are better for many layers.

Kernel Size	Input Channels	Output Maps	Input Dimensions	Output Dimensions	Direct(ms)	Im2Col(ms)	Direct(mJ)	Im2Col(mJ)
3x3	3	64	51529	12769	2.56	27.52	4.33	34.64
1x1	64	16	3136	3136	1.50	4.67	2.47	6.62
1x1	16	64	3136	3136	1.14	4.66	1.87	6.45
3x3	16	64	3136	3136	1.55	14.17	2.69	19.31
1x1	128	16	3136	3136	2.98	8.39	4.84	12.28
1x1	16	64	3136	3136	1.14	4.67	1.85	6.48
3x3	16	64	3136	3136	1.55	14.18	2.70	19.33
1x1	128	32	784	784	1.22	1.90	1.96	2.66
1x1	32	128	784	784	1.07	3.03	1.56	3.60
3x3	32	128	784	784	1.50	13.97	2.44	15.17
1x1	256	32	784	784	2.43	3.27	3.84	4.57
1x1	32	128	784	784	1.07	3.00	1.55	3.64
3x3	32	128	784	784	1.50	14.50	2.42	15.76
1x1	256	48	196	196	1.08	1.23	1.68	1.51
1x1	48	192	196	196	0.73	1.17	1.02	1.38
3x3	48	192	196	196	1.07	11.20	1.67	11.58
1x1	384	48	196	196	1.61	1.72	2.5	2.1
1x1	48	192	196	196	0.73	1.17	1.02	1.38
3x3	48	192	196	196	1.07	11.11	1.65	11.50
1x1	384	64	196	196	2.29	2.28	3.24	2.62
1x1	64	256	196	196	1.31	2.13	1.78	2.36
3x3	64	256	196	196	1.94	28.57	2.89	29.02
1x1	512	64	196	196	3.04	2.90	4.43	3.38
1x1	64	256	196	196	1.31	2.14	1.79	2.37
3x3	64	256	196	196	1.94	29.61	2.89	30.17
1x1	512	1000	196	196	44.74	128.71	63.03	125.7

Figure 7.10: SqueezeNet's Layers executed with both convolution's ways.

7.3 Comparison with Other Implementations

This section will compare the execution time of the CNNs presented at the chapter 3, using different devices and/or implementations. The current CNN implementation will be compared with Movidius

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 Tegra TX1 with or without CuDNN device in order to get a filling of the execution times between multiple hardware devices. Figure 7.11 and table 7.1 below present the comparison.
First of all, notice that the current CNN implementation is executed on Myriad2, more precisely model MA2450. The Movidius NCS uses the same chip, which makes the comparison more fair, since the hardware is almost identical. On the other hand, Caffe is executed on Quad ARM A57, which has one to four cores, each with their L1 instruction and data caches, together with a single shared L2 unified cache. In general, this is an extremely powerful processor that is intended for server applications. In our case only one core is being used, because Tegra TX1 gives us the opportunity to use cuDnn with GPU 256-core Maxwell. Also one core only is used, in order to show the load of the computations. Caffe with cuDNN as well, performs better than the current CNN implementation in 2 out of 4 CNNs. If the lack in performance of the current CNN implementation (in comparison with the performance on Caffe) 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 GPU is designed high performance server applications.

Table 7.1: Execution times of different Implementations

CNNs	Current Implementation	Movidius NCS	Caffe in Quad ARM A57	Caffe with CuDNN
AlexNet	98.3	96.27	7518	22.2
GoogleNet	249.1	99.04	16836	180.26
SqueezeNet	85.5	50.26	8961	695.38
VGGNet	586	733.5	7587	85.8

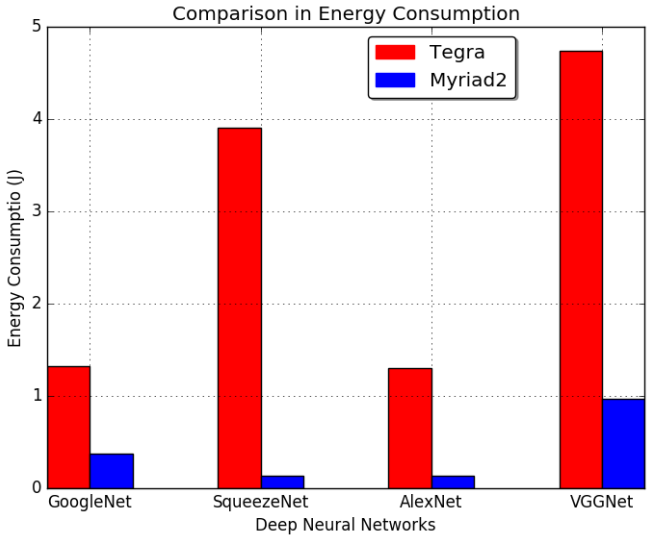


Figure 7.11: Low Energy Consumption in Myriad2 related to Caffe in TegraTX1.

Figures above show that linear CNNs execute very fast in Tegra. Obviously, a good parallization scheme for fully-connected node is achieved in Tegra, as VGG, AlexNet have huge weight matrices for these layers. Below, ImageNet's CNNs are presented with their execution time and some other attributes:

Table 7.2: Imagenet's CNNs execution's times

CNN	Execution Time(ms)	Energy Consumption(mJ)	layers	memory(MB)
AlexNet	98.3	125.6	13	117
GoogleNet	249.1	365.2	83	16.6
NiN-imagenet	244	335.7	16	15.5
SqueezeNet	85.5	126.7	38	4.68
VGG	586	961	16	276
ZFnet	99	130.3	13	121

7.4 Real-Time Application

This thesis, targeted to develop an efficient CNN engine for the Myriad2 embedded multiprocessor in order to test "deep" CNNs, which are trained in ImageNet dataset. After this, different approaches for the convolution layer were implemented. By achieving this, a real-time application was later an interesting challenge. This application was developed in order to classify objects by accepting images, which belonged in ImageNet. The accuracy results were the same with Caffe Framework. Also, it is observed that ImageNet's CNNs succeed in classifying objects in images, which do not belong in ImageNet.
In order to accomplish the real-time oject classification, a GUI with Python GTK was written. This GUI provides the user the ability to select the image, that wants to classify. After this, a selection to send the image to the embedded device is provided. This was implemented via socket-communication and with appropriate functions in order that GUI sends the preprocessed image and receives the results. A screenshot of the app is provided below.

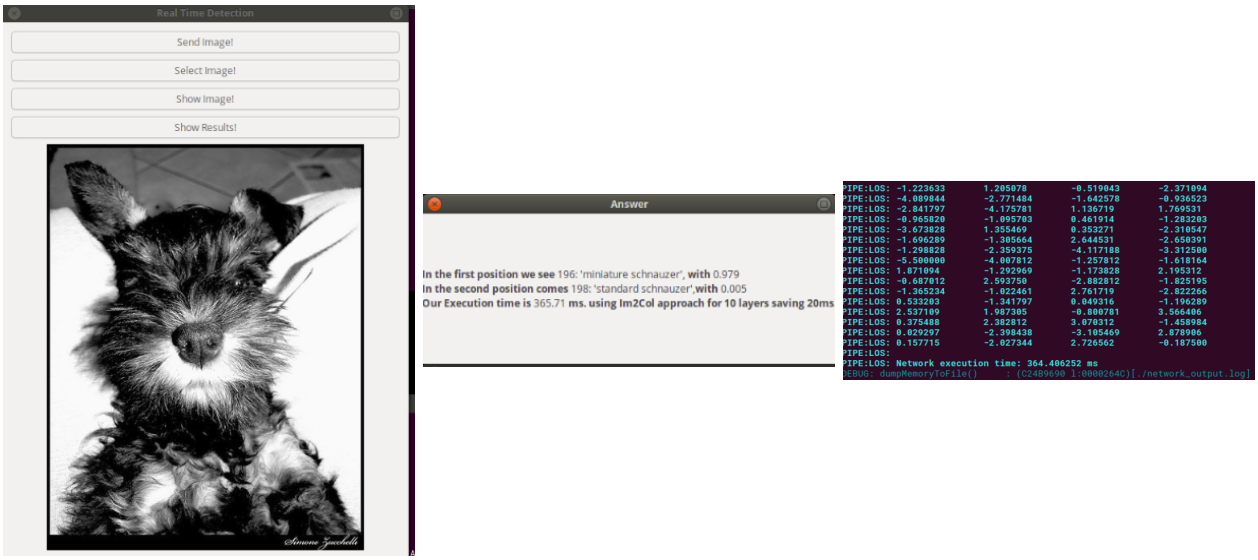


Figure 7.12: Screenshot of the real-time Application.

Chapter 8

Conclusion

8.1 Summary

Some basic ideas which were used in the current implementation could be applied to any embedded application which would have to suppport CNNs.

- It is a fact that in order to parallelize convolution, a procedure had to be implemented, which is going to **provide the corresponding input and output partitions**. After this, processing in these data can be done with every way (e.g. alignment, rebuilding of zero-padding elements).
- One of the major problems that needed to be solved early was the efficient management of given CMX memory. CMX can offer a very large performance boost if exploited correctly. In order to accomplish this boost, **code segment of the CMX had to be as restricted** as it is possible by writing efficient code residing in CMX. This contributed to provide more memory space for the data of ImageNet's CNNs.
- Myriad2 achieves low energy consumption in AI applications, like CNNs.

8.2 Future Work

After taking results of the implementation of the CNN engine, several ideas were grasped. Some of them refer to changes in the current implementation and some to additions. For a future extension, the following seem to be of the greatest importance:

- *Extend the CNN engine, in order to reduce DMA transfers.* The general idea of the implementation is to transfer the input from the main memory into local memories, make the computations and return the results back to the main memory for every layer. A great variation to this, would be to keep the data into the local memories after the first transfer, share them into the local memories and return them back once. This would need the SIPP engine.
- *The developement of CNNs and object classification is followed nowadays by object detection and R-CNNs.* The goal of a R-CNN is to take in an image, and correctly identify both where the main objects in the image are and classify them. R-CNN propose a bunch of boxes in the image and see if any of them actually correspond to an object, so it tests a CNN like AlexNet many times, taking as an input every box of this bunch.
- *Provide a matrix to matrix multiplication function with decreased complexity.* Im2Col convolution is a memory-bound problem, which transfers the whole input only for one multiplication per element. This means that not so many optimizations could happen apart from optimizing the computation routine with a new algorithm (e.g. Scharr's Algorithm).
- The biggest percent of the whole execution time is devoted in convolutional kernels bigger than 1x1. This means, that if these kernels are *segmented into 1x1 kernels*, then a noteable speedup may happen.

Bibliography

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

[2] Myriad 2 Ma2x5x Vision Processor. https://uploads.movidius.com/1463156689-2016-04-29_VPU_ProductBrief.pdf.

[3] NVIDIA Jetson TX1 Module. https://www.nvidia.com/en-us/autonomous-machines/embedded-systems-dev-kits-modules/.

[4] A Brief History of CNNs in Image Segmentation: From R-CNN to Mask R-CNN. https://blog.athelas.com/a-brief-history-of-cnns-in-image-segmentation-from-r-cnn.-to-mask-r-cnn-34ea83205de4.

[5] Adit Dashpande. The 9 deep Learning Papers You Need To Know About. http://adeshpande3.github.io/The-9-Deep-Learning-Papers-You-Need-To-Know-About.html.

[6] Matthew D. Zeiler, Rob Fergus. ZF-Net. https://arxiv.org/pdf/1311.2901.pdf.

[7] Alex Krizhevsky, Ilya Sutskever, Geoffrey E. Hinton. AlexNet. https://papers.nips.cc/paper/4824-imagenet-classification-with-deep-convolutional-neural-networks.pdf.

[8] Yani Ioannou. Group Parameter. https://blog.yani.io/filter-group-tutorial/.

[9] Yangqing Jia. Caffe. http://caffe.berkeleyvision.org/.

[10] Andrej Karpathy. Stanford University CS231n: Convolutional Neural Networks for Visual Recognition. http://cs231n.github.io/.

[11] Forrest N. Iandola, Song Han, Matthew W. Moskewicz, Khalid Ashraf, William J. Dally, Kurt Keutzer. SQUEEZENET: ALEXNET-LEVEL ACCURACY WITH 50X FEWER PARAMETERS AND <0.5MB MODEL SIZE. https://arxiv.org/pdf/1602.07360.pdf.

[12] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, Andrew Rabinovich. Going deeper with convolutions. https://arxiv.org/pdf/1409.4842.pdf.

[13] Sebastian Raschka. *Python Machine Learning*. 2015.

[14] Tariq Rashid. *Make Your Own Neural Network*. 2016.

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

[16] Thanasis Xiggis. Implementation of Convolutional Neural Networks on Embedded Architectures. http://artemis-new.cslab.ece.ntua.gr:8080/jspui/bitstream/123456789/8247/1/DT2017-0208.pdf.

[17] Min Lin, Qiang Chen, Shuicheng Yan. Network In Network. https://arxiv.org/pdf/1312.4400.pdf.

[18] Karen Simonyan, Andrew Zisserman. VERY DEEP CONVOLUTIONAL NETWORKS FOR LARGE-SCALE IMAGE RECOGNITION. https://arxiv.org/pdf/1409.1556.pdf.

[19] Βικιπαίδεια. Μηχανική Μάθηση. https://el.wikipedia.org/wiki/%CE%9C%CE%B7%CF%87%CE%B1%CE%BD%CE%B9%CE%BA%CE%AE_%CE%BC%CE%AC%CE%B8%CE%B7%CF%83%CE%B7.

[20] Βικιπαίδεια. Νευρωνικά Δίκτυα. https://el.wikipedia.org/wiki/%CE%9D%CE%B5%CF%85%CF%81%CF%89%CE%BD%CE%B9%CE%BA%CF%8C_%CE%B4%CE%AF%CE%BA%CF%84%CF%85%CE%BF.