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

Resource Management Techniques for Embedded Architectures executing Deep Neural Networks

Τεχνικές Διαχείρισης Πόρων για Βαθιά Νευρωνικά Δίκτυα που εκτελούνται σε Ενσωματωμένες Αρχιτεκτονικές

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

ΦΟΙΒΟΣ ΤΣΙΜΠΟΥΡΛΑΣ

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

Αθήνα, Ιούνιος 2018

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

Resource Management Techniques for Embedded Architectures executing Deep Neural Networks

Τεχνικές Διαχείρισης Πόρων για Βαθιά Νευρωνικά Δίκτυα που εκτελούνται σε Ενσωματωμένες Αρχιτεκτονικές

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

ΦΟΙΒΟΣ ΤΣΙΜΠΟΥΡΛΑΣ

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

Εγκρίθηκε από την τριμελή εξεταστική επιτροπή την 18η Ιουνίου 2018.

....	....	....
Δημήτριος Σούντρης	Κιαμάλ Πεκμεστζή	Γεώργιος Γκούμας
Αναπληρωτής Καθηγητής Ε.Μ.Π.	Καθηγητής Ε.Μ.Π.	Επίκουρος Καθηγητής Ε.Μ.Π.

Αθήνα, Ιούνιος 2018

....

Φοίβος Τσιμπουρλάς

Διπλωματούχος Ηλεκτρολόγος Μηχανικός και Μηχανικός Υπολογιστών Ε.Μ.Π.

Περίληψη

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

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

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

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

Λέξεις κλειδιά

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

Abstract

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

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

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

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

Key words

Machine learning, Convolutional neural networks, embedded systems, Multicore systems, Myriad 2

Ευχαριστίες

Η παρούσα διπλωματική εργασία πραγματοποιήθηκε στο Εργαστήριο Μικροϋπολογιστών και Ψηφιακών Συστημάτων της Σχολής Ηλεκτρολόγων Μηχανικών και Μηχανικών Υπολογιστών του Εθνικού Μετσόβιου Πολυτεχνείου από τον Ιούνιο του 2017 έως τον Ιούνιο του 2018.

Καταρχάς, θα ήθελα να ευχαριστήσω τον επιβλέποντα καθηγητή μου κ. Δημήτριο Σούντρη για την ευκαιρία που μου έδωσε να δουλέψω σε ένα ιδιαίτερα ενδιαφέρον και σύγχρονο θέμα. Επίσης θα ήθελα να ευχαριστήσω όλους τους ερευνητές του εργαστηρίου για τις ευχάριστες στιγμές και την υποστήριξη καθόλη τη διάρκεια αυτής της διπλωματικής.

Τέλος, ολοκληρώνοντας ένα περιπετειώδες κεφάλαιο της ζωής μου, θα ήθελα να αφιερώσω εξ ολοκλήρου τη διπλωματική αυτή, στα άτομα που σε αυτή την πορεία με στήριξαν με ανιδιοτέλεια, καθένας με δικό του τρόπο, διότι χωρίς εκείνα, όλα όσα σήμερα είναι εφικτά ή τετελεσμένα, θα έπλεαν στη σφαίρα της φαντασίας.

Φοίβος Τσιμπουρλάς,

Αθήνα, 18η Ιουνίου 2018

Contents

List of Tables

List of Figures

Κεφάλαιο 1

Τεχνικές Διαχείρισης Πόρων στην Εκτέλεση Βαθιών Νευρωνικών Δικτύων σε Ενσωματωμένες Αρχιτεκτονικές

1.1 Η μηχανική μάθηση

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

1.1.1 Τα είδη της μηχανικής μάθησης

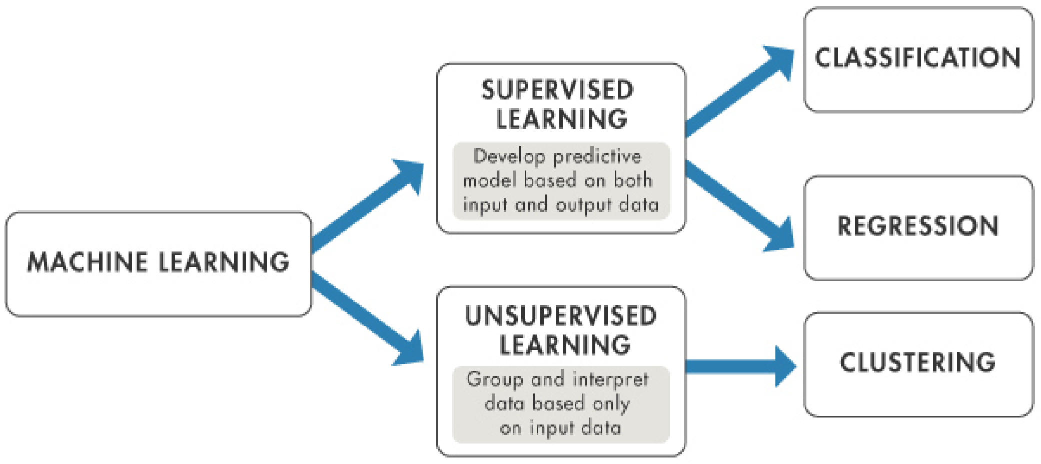
Αυτή η υποενότητα θα παρουσιάσει συνοπτικά τα τρία είδη μηχανικής μάθησης: Επιτηρούμενη μάθηση, Ενισχυτική μάθηση και Μη επιτηρούμενη μάθηση.

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

- *Ενισχυτική μάθηση*: Έχει ως στόχο την ανάπτυξη ενός συστήματος (πράκτορα) που βελτιώνει την απόδοσή του καθώς αλληλεπιδρά με το περιβάλλον. Στην ενισχυτική μάθηση, η πληροφορία σχετικά με την τρέχουσα κατάσταση του περιβάλλοντος περιλαμβάνει επιπλέον και ένα σήμα επιβράβευσης, που μοντελοποιεί πόσο καλή είναι η μια δράση του πράκτορα. Κατά συνέπεια, μέσα από την αλληλεπίδρασή του με το περιβάλλον, ο πράκτορας είναι σε θέση να μάθει μια σειρά από δράσεις που έχουν ως στόχο την μεγιστοποίηση της επιβράβευσής του.

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

Η εικόνα 1.1, παρακάτω, παρουσιάζει μία σύνοψη των δύο βασικών ειδών μηχανικής μάθησης.


MACHINE LEARNING
SUPERVISED LEARNING
Develop predictive model based on both input and output data
UNSUPERVISED LEARNING
Group and interpret data based only on input data
CLASSIFICATION
REGRESSION
CLUSTERING

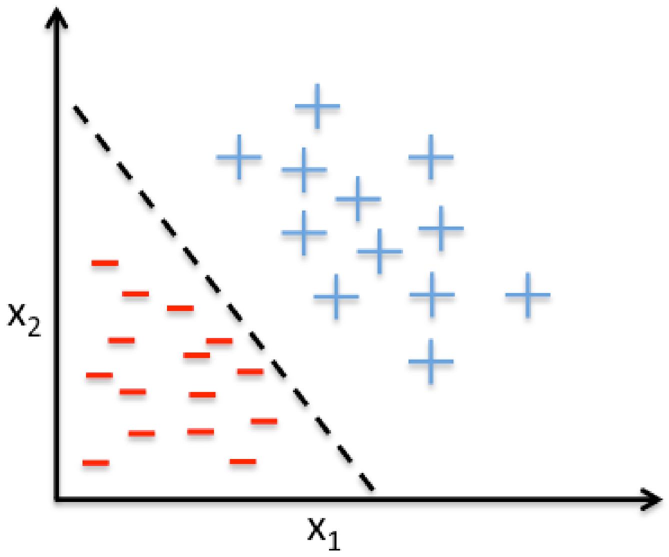
Σχήμα 1.1: Τα 2 βασικά είδη της μηχανικής μάθησης

Στην επόμενη υποενότητα τα θέματα που θα συζητηθούν αφορούν μονάχα το είδος της επιτηρούμενης μάθησης. Για παράδειγμα, στην περίπτωση του λογισμικού που φιλτράρει spam e-mail, ο τρόπος προσέγγισης του προβλήματος είναι να εκπαιδευτεί ένα νευρωνικό δίκτυο χρησιμοποιώντας έναν αλγόριθμο επιτηρούμενης μάθησης πάνω σε ένα σύνολο από κατηγοριοποιημένων e-mail. Τα mail αυτά θα είναι ορθά κατηγοριοποιημένα ως κανονικά ή ως κακόβουλα. Μέσω της μάθησης στο σύνολο αυτό, το δίκτυο θα μπορεί μετά να προβλέψει αν κάποιο άλλο mail είναι κανονικό ή όχι. Ένα τέτοιο πρόβλημα, με ξεχωριστές κατηγορίες πρόβλεψης, ονομάζεται πρόβλημα ταξινόμησης (classification task).

1.1.2 Ταξινόμηση στην επιτηρούμενη μάθηση

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

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

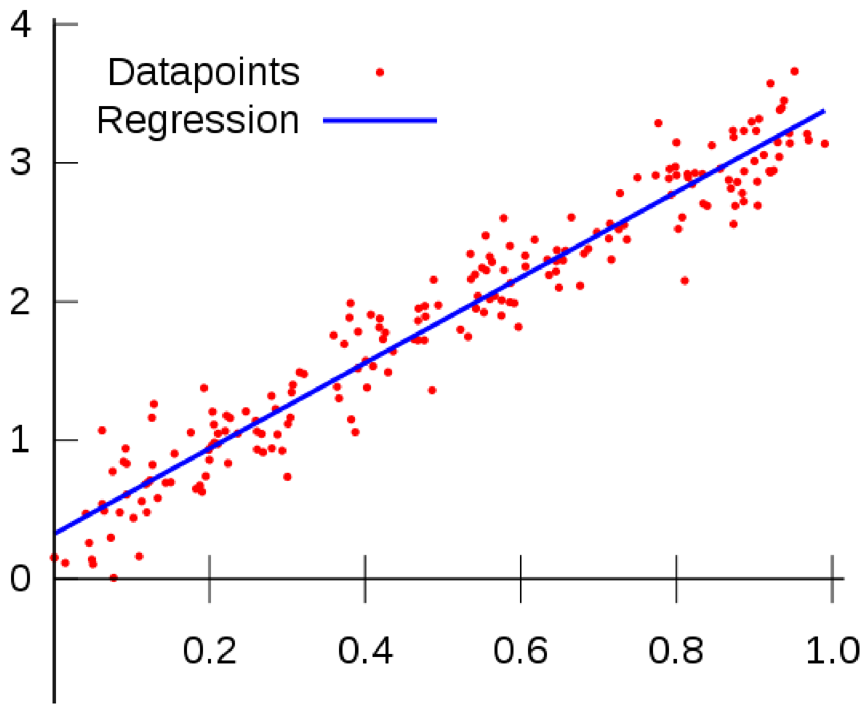


Σχήμα 1.2: Δυαδική ταξινόμηση στην επιτηρούμενη μάθηση

Το σχήμα 1.2 αποτυπώνει την ιδέα της δυαδικής ταξινόμησης, υποθέτοντας ότι έχουν δοθεί 27 δείγματα κατά το στάδιο της εκπαίδευσης: 15 δείγματα έχουν την ετικέτα της αρνητικής κατηγορίας (σύμβολο πλην) και 12 δείγματα έχουν την ετικέτα της θετικής κατηγορίας (σύμβολο συν). Σε αυτό το σενάριο, το σύνολο δεδομένων είναι δισδιάστατο, το οποίο σημαίνει ότι κάθε δείγμα εμπεριέχει δύο τιμές: x_1 και x_2. Ένας αλγόριθμος (επιτηρούμενης) μηχανικής μάθησης μπορεί να χρησιμοποιηθεί για να μάθει ένα κανόνα - το σύνορο της απόφασης που αναπαρίσταται με τη μαύρη διακεκομμένη γραμμή - που διαχωρίζει τις δύο κατηγορίες και ταξινομεί τα νέα δεδομένα σε μία από τις κατηγορίες δεδομένων των τιμών x_1 και x_2.

1.1.3 Παλινδρόμηση στην επιτηρούμενη μάθηση

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

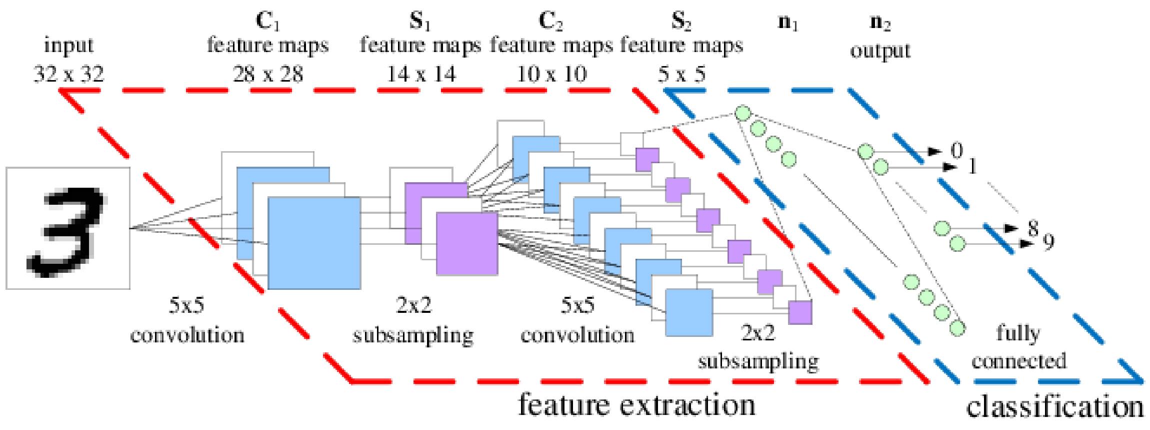


Σχήμα 1.3: Linear regression in supervised learning

Το σχήμα 1.3 αποτυπώνει την ιδέα της γραμμικής παλινδρόμησης. Δεδομένων μιας μεταβλητής πρόβλεψης x και μιας μεταβλητής απόκρισης y, σχεδιάζεται μια ευθεία γραμμή που "ταιριάζει" στα δεδομένα και ελαχιστοποιεί την απόσταση - που συνήθως είναι η μέση τιμή του τετραγώνου της απόστασης - μεταξύ των δειγμάτων και της γραμμής. Έπειτα, αυτή η γραμμή χρησιμοποιείται για να προβλέψει τη μεταβλητή απόκρισης σε νέα δεδομένα.

1.2 Συνελικτικά νευρωνικά δίκτυα

Ένα Συνελικτικό Νευρωνικό Δίκτυο (ΣΝΔ) αποτελεί μιας ειδική τοπολογία τεχνητού νευρωνικού δικτύου, η οποία είναι εμπνευσμένη από τον οπτικό εγκεφαλικό φλοιό των ζώων. Οι παράμετροι αυτών των τοπολογιών ρυθμίστηκαν κατάλληλα από τον Yann LeCun στις αρχές του 1990 [3], ώστε να μπορέσουν τα λύσουν προβλήματα στην όραση υπολογιστών. Επί της ουσίας, ένα ΣΝΝ είναι ένα μοντέλου ΤΝΔ που έχει σχεδιαστεί αποκλειστικά για την αναγνώριση δισδιάστατων αντικειμένων, παρουσιάζοντας υψηλό βαθμό αναλλοίωτης συμπεριφοράς κατά την μετάθεση, κλιμάκωση, στρέβλωση και άλλες παραμορφώσεις της εισόδου.

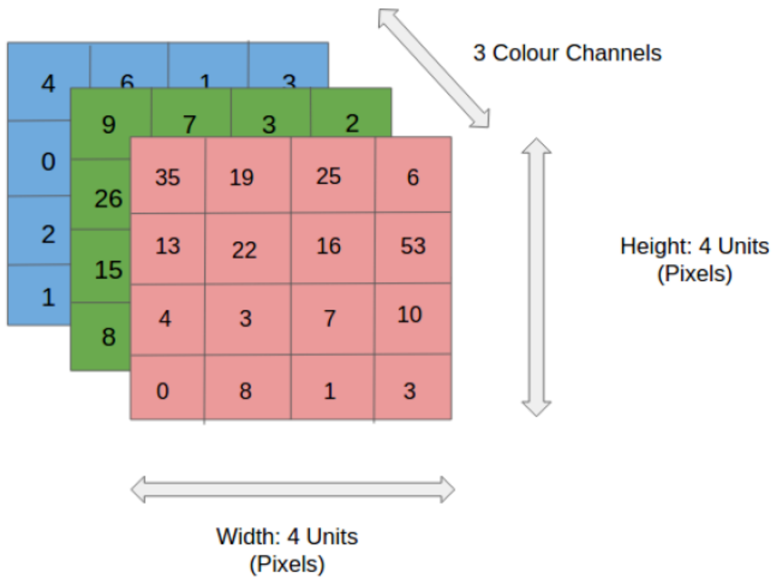


Σχήμα 1.4: Παράδειγμα τοπολογίας ένος ΣΝΝ κατάλληλου για αναγνώριση χειρόγραφων ψηφίων

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

1.2.1 Διάταξη των δεδομένων σε ένα ΣΝΔ

Τα ΣΝΔ εκμεταλλεύονται το γεγονός ότι η είσοδος αποτελείται από εικόνες και περιορίζουν την αρχιτεκτονική τους με τρόπο που να έχει περισσότερο νόημα [4]. Συγκεκριμένα, σε αντίθεση με ένα κανονικό ΤΝΔ, οι στρώσεις ενός ΣΝΔ οργανώνονται σε 3 διαστάσεις: πλάτος, ύψος, βάθος. Για παράδειγμα, 32×32 RGB εικόνες εισόδου αναπαριστούν ένα όγκο εισόδου που έχει διαστάσεις $32 \times 32 \times 3$ (πλάτος, ύψος, βάθος αντίστοιχα). Οι νευρώνες σε μία στρώση είναι συνδεδεμένοι μόνο με μια μικρή περιοχή της προηγούμενης στρώσης, σε αντίθεση με όλους τους νευρώνες της προηγούμενης στρώσης που συναντάται στις πλήρως συνδεδεμένες στρώσεις. Η περιορισμένη, τοπική συνδεσιμότητα των ΣΝΔ θα αποσαφηνιστεί σύντομα. Ένας 3Δ όγκος εισόδου απεικονίζεται στο σχήμα 1.5 για μια 4×4 RGB εικόνα. Σημειώστε ότι το εύρος των τιμών στον 3Δ όγκο εισόδου μπορεί να ρυθμιστεί, ώστε να βοηθήσει στην εκπαίδευση του δικτύου. Η διαδικασία μετασχηματισμού των δεδομένων εισόδου - προτού αυτά τροφοδοτηθούν στο ΣΝΔ - ώστε να έχουν την επιθυμητή μορφή, λέγεται προεπεξεργασία.



Σχήμα 1.5: Διατομή ενός όγκου δεδομένων εισόδου $4\times4\times3$. Αποτελείται από 3 μήτρες, που αντιστοιχούν στα 3 κανάλια της εικόνας

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

1.2.2 Συνήθεις στρώσεις που χρησιμοποιούνται στα ΣΝΔ

Υπάρχουν τρία είδη στρώσεων που χρησιμοποιούνται για τη δόμηση αρχιτεκτονικών ΣΝΔ: Συνελικτικές στρώσεις, Pooling Layer, and Πλήρως συνδεδεμένες στρώσεις [5]. Η τελευταία είναι ίδια με αυτή που συναντά κανείς στα κανονικά ΤΝΔ. Αυτές οι τρεις στρώσεις στοιβάζονται σε μια αλληλουχία, ώστε να παράξουν ενδιαφέρουσες αρχιτεκτονικές. Υπάρχει επίσης και η Στρώση Εισόδου, που δεν είναι τίποτε περισσότερο από τον ταυτοτικό μετασχηματισμό, δηλαδή η έξοδός της είναι η ίδια με την εισοδό της. Αυτές οι στρώσεις αναλύονται παρακάτω:

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

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

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

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

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

- *Στρώση LRN*: Εφαρμόζει ένα είδος ”πλευρικής συστολής” μέσω της κανονικοποιήσης πάνω σε γειτονικές περιοχές εικονοστοιχείων.

- *Στρώση Concat*: Συνδέει σειριακά πολλά διαφορετικά μπλοκ εισόδου σε ένα ενιαίο μπλοκ εξόδου.

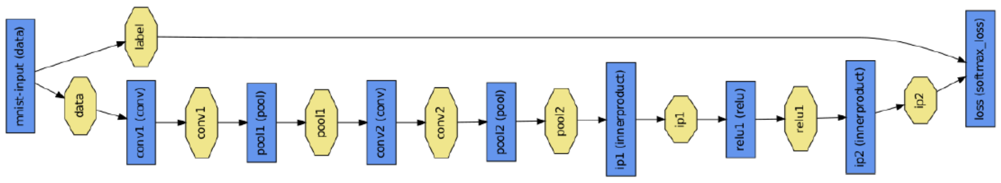
Κατ’ αυτό τον τρόπο, τα ΣΝΔ μετασχηματίζουν την εικόνα εισόδου καθώς αυτή διασχίζει τις στρώσεις του. Έτσι μετατρέπει την είσοδο από εικονοστοιχεία σε ετικέτες που αντιπροσωπεύουν την κατηγορία στην οποία εκτιμάται ότι ανήκουν. Επισημαίνεται ότι κάποιες στρώσεις περιέχουν/απαιτούν παραμέτρους, ενώ άλλες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

1.3 Caffe: Συνελικτική Αρχιτεκτονική για Γρήγορη Ενσωμάτωση

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

1.3.1 Εκπαίδευση ενός δικτύου

Το Caffe εκπαιδεύει τα μοντέλα χρησιμοποιώντας τον αλγόριθμο γρήγορης ή κανονικής στοχαστικής κατάβασης με τη μέθοδο της κλίσης. Το σχήμα 1.6 παρουσιάζει ένα τυπικό παράδειγμα δικτύου (για την ταξινόμηση ψηφίων από το σύνολο δεδομένων MNIST) κατά τη φάση της εκπαίδευσης caffe μια στρώση δεδομένων λαμβάνει τις εικόνες και τις ετικέτες τους από το σκληρό δίσκο, τροφοδοτεί τα δεδομένα διαμέσου πολλαπλών στρώσεων, όπως είναι η συνέλιξη, και τροφοδοτεί το τελικό αποτέ- λεσμα πρόβλεψης σε μια στρώση κατηγοριοποίησης. Αυτή η στρώση υπολογίζει το σφάλμα κατηγοριοποίησης και τις κλίσεις που εκπαιδεύουν όλο το δίκτυο, με σκοπό τη βελτιστοποίηση των παραμέτρων του. Το συγκεκριμένο παράδειγμα μπορεί να βρεθεί στον πηγαίο κώδικα του Caffe, στη διαδρομή examples/lenet/lenet_train.prototxt. Η επεξεργασία των δεδομένων γίνεται σε μικρές ομάδες που τροφοδοτούνται στο δίκτυο σειριακά. Κρίσιμο στη εκπαίδευση είναι η ρύθμιση του ρυθμού εξασθένησης της εκμάθησης, η ορμή και τα στιγμιότυπα. Τα τελευταία επιτρέπουν τη παύση και συνέχιση της εκπαίδευσης του δικτύου.



Σχήμα 1.6: Παράδειγμα κατηγοριοποίησης ψηφίων του συνόλου δεδομένων MNIST, όπου τα μπλε ορθογώνια αναπαριστούν στρώσεις και τα κίτρινα ορθογώνια αναπαριστούν τα δεδομένα που παράγονται και τροφοδοτούνται σε αυτές.

1.4 Πολυεπεξεργαστικό SoC Myriad 2

1.4.1 Γενικά χαρακτηριστικά

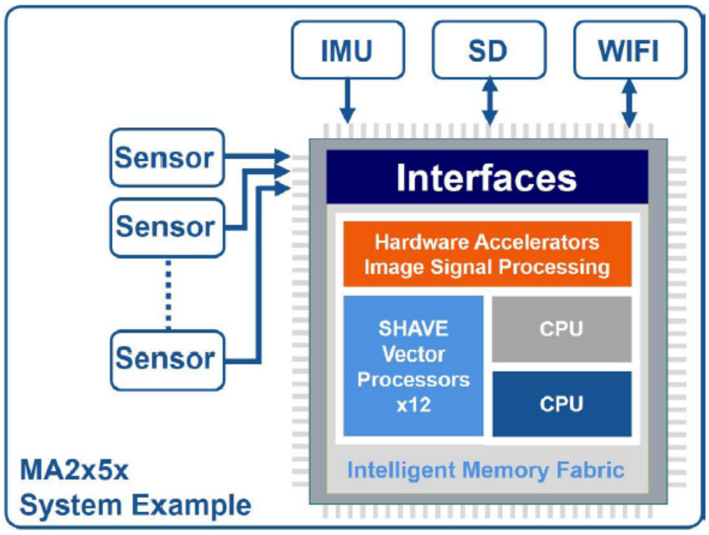
Για την υλοποίηση του συστήματος εκτέλεσης συνελικτικών δικτύων επιλέχθηκε η ενσωματωμένη πλατφόρμα Myriad2 [7]. Το συγκεκριμένο SoC αναπτύσσεται από τη Movidius Ltd, η οποία πρόσφατα έγινε μέλος της ομάδας Perceptual Computing Group της Intel, με στόχο την επίσπευση της δημιουργίας ευφυών συσκευών σε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

Τα κυριότερα χαρακτηριστικά της Myriad2 είναι:

- Σχεδιασμός πολύ χαμηλής ισχύος: Κάνοντάς τη κατάλληλη για χρήση σε φορητές συσκευές, που η αυτονομία της μπαταριάς είναι κυρίαρχη παράμετρος.
- Επεξεργαστής υψηλής απόδοσης: Δίνοντας τη δυνατότητα εκτέλεσης των υπολογιστικά απαιτητικών σύγχρονων εφαρμογών της όρασης υπολογιστών.
- Ευέλικτη αρχιτεκτονική: Παρέχοντας πρόσβαση στις λεπτομέρειες της αρχιτεκτονικής, οι προγραμματιστές είναι σε θέση να βελτιστοποιήσουν τις εφαρμογές τους ακόμα περισσότερο.
- Μικρές φυσικές διαστάσεις: Ώστε να είναι εφικτή η ενσωμάτωση της ψηφίδας σε οποιαδήποτε φορητή συσκευή.

Μια περιγραφή υψηλού επιπέδου του υλικού της Myriad2 δίνεται στο σχήμα 1.7. Από αυτό, είναι φανερό πως το συγκεκριμένο SoC διαθέτει 14 επεξεργαστές. Οι δύο επεξεργαστές στα δεξιά του σχήματος είναι θεμελιωδώς διαφορετικοί από τους 12 επεξεργαστές στα αριστερά. Πιο αναλυτικά, οι επεξεργαστές με το όνομα "CPU" υλοποιούν την 32-bit SPARC αρχιτεκτονική, που ανήκει στην οικογένεια επεξεργαστών RISC.

Μια λεπτομερέστερη περιγραφή ακολουθεί στη συνέχεια:



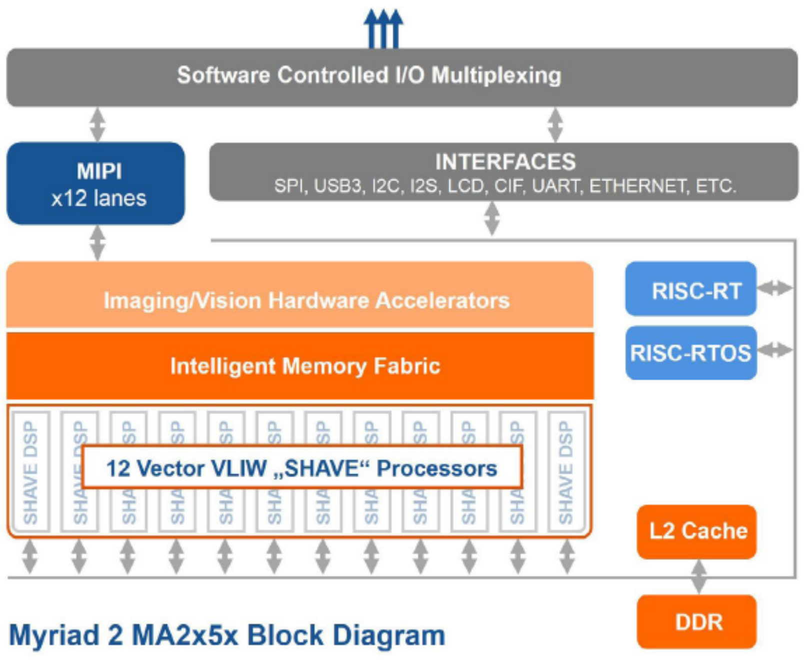
Σχήμα 1.7: Επισκόπηση του υλικού της Myriad2

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

- Leon RT: Αποτελεί το δεύτερο από τους επεξεργαστές SPARC. Ανήκει στο υποσύστημα Media sub-system (MSS), μια δομική μονάδα που επιτρέπει την διασύνδεση με συσκευές εικόνας, όπως αισθητήρες εικόνας, οθόνες LCD, ελεγκτές HDMI κ.λπ.. Ταυτόχρονα, το MSS είναι υπεύθυνο για το έλεγχο φίλτρων (όπως π.χ. το DeBayer) υλοποιημένων στο υλικό, που διατείθενται από την Myriad2.

- SIPP: Πρόκειται για ένα ιδιοταγή μηχανισμό υλικού/λογισμικού που χρησιμοποιείται απο τη Myriad2, με σκοπό την αποδοτική δρομολόγηση εργασιών ψηφιακής επεξεργασίας εικόνας. Αυτός ο μηχανισμός είναι βασισμένος σε επεξεργασία μορφής σωλήνωσης και χρησιμοποιεί τα φίλτρα υλικού που παρέχονται από την Myriad2, ώστε να επιτύχει την ταχύτερη δυνατή εκτέλεση. Το υποσύστημα SIPP απεικονίζεται στο σχήμα 1.7 με πορτοκαλί χρωμα.

- Microprocessor Array (UPA): Είναι η αρχιτεκτονική μονάδα της Myriad2 που συγκροτείεται από τους 12 Very Long Instruction Word (VLIW) διανυσματικούς επεξεργαστές SHAVE (βλ. σχήμα 1.8), τη CMX μνήμη SRAM μεγέθους 2 MB και μερικές ακόμα μονάδες, εκ των οποίων οι πιο σημαντικές είναι: Η εξειδικευμένη DMA


Software Controlled I/O Multiplexing
MIPI
x12 lanes
INTERFACES
SPI, USB3, I2C, I2S, LCD, CIF, UART, ETHERNET, ETC.
Imaging/Vision Hardware Accelerators
Intelligent Memory Fabric
SHAVE DSP
12 Vector VLIW „SHAVE“ Processors
RISC-RT
RISC-RTOS
L2 Cache
DDR
Myriad 2 MA2x5x Block Diagram

Σχήμα 1.8: Λεπτομερέστερη επισκόπηση του υλικού της Myriad2

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

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

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

1.5 Ελεγκτής DMA της μνήμης CMX

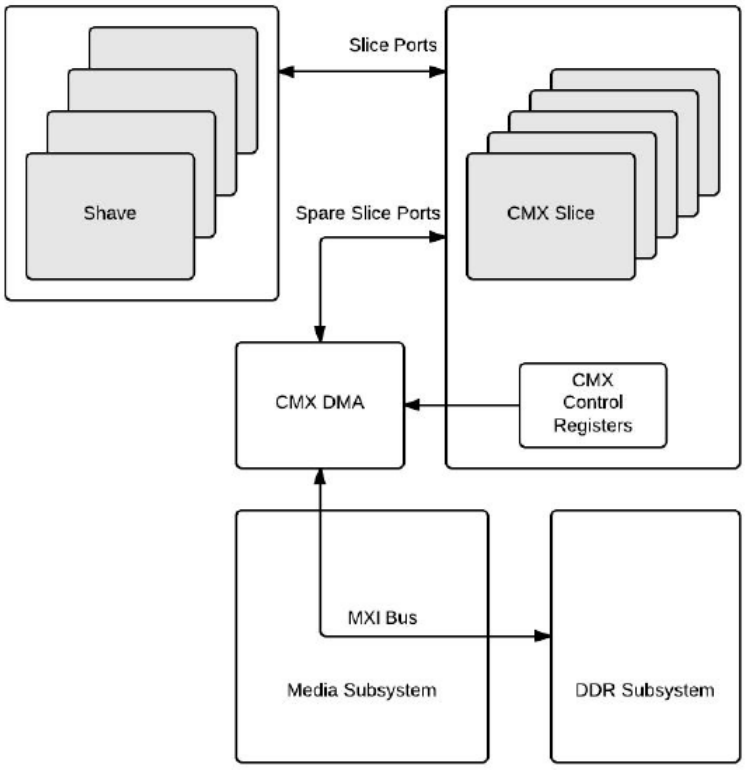
Αυτός ο ελεγκτής βρίσκεται ανάμεσα του διαύλου MXI των 128-bit και της μνήμης CMX [8]. Παρέχει μεταφορές δεδομένων υψηλού εύρους ζώνης μεταξύ της CMX και της DDR, προς οποιαδήποτε κατεύθυνση. Επιπλέον, υποστηρίζει μεταφορές δεδομένων από DDR σε DDR και από CMX σε CMX. Το σχήμα 1.9 παρουσιάζει ένα υψηλού επιπέδου διάγραμμα της DMA engine.

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

1.6 Μεθοδολογία και υλοποίηση

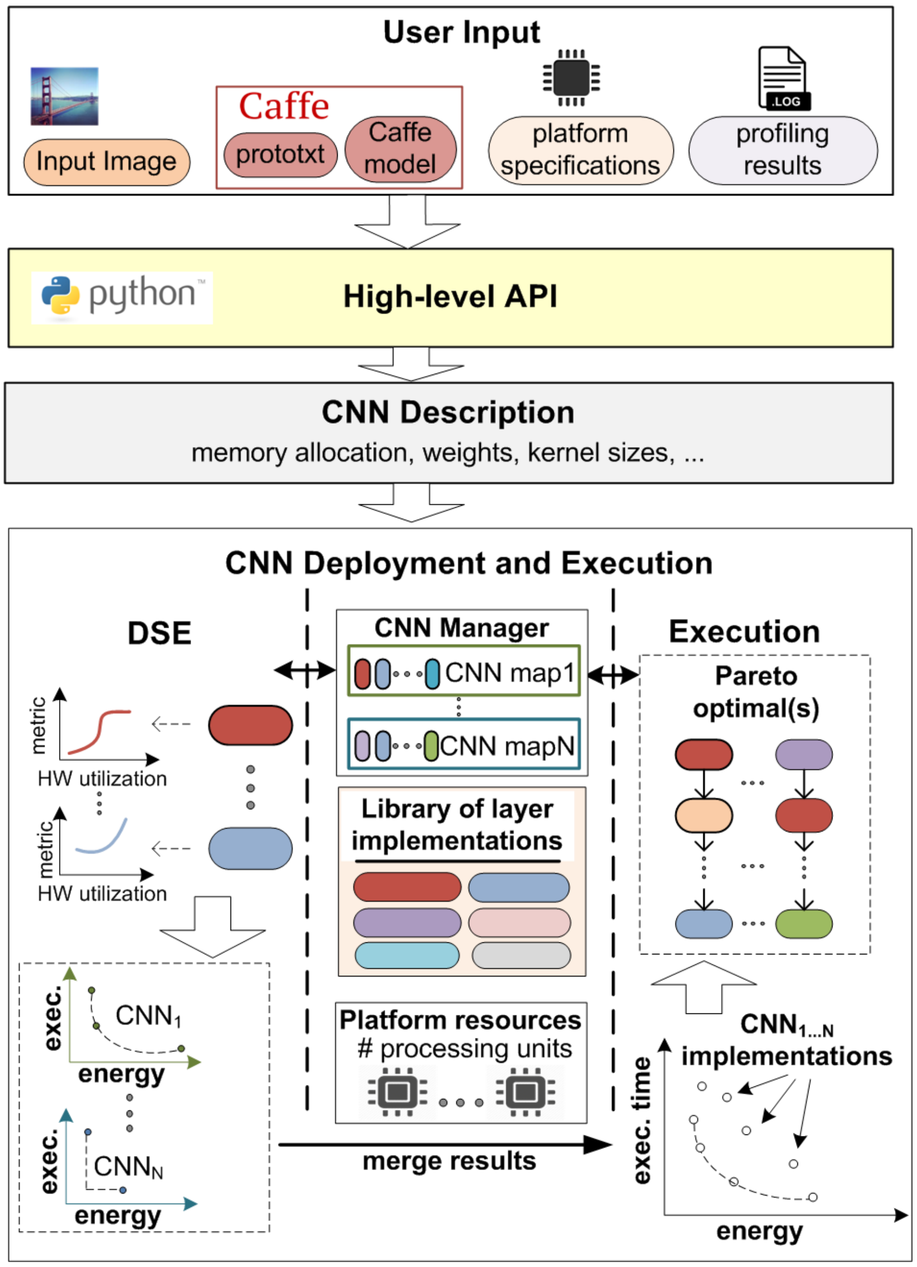
Η προτεινόμενη εφαρμογή που υλοποιήθηκε για την παρούσα διπλωματική στοχεύει στην δημιουργία ενός ενεργειακά αποδοτικού, και ταυτόχρονα υπολογιστικά ισχυρό μηχανισμό για συνελικτικά νευρωνικά δίκτυα και μπορεί να χρησιμοποιηθεί για την επίλυση προβλημάτων ταξινόμησης ή για να αξιολογεί την απόδοση δικτύων χρησιμοποιώντας ως χώρο σχεδιασμού το χρόνο εκτέλεσης, την κατανάλωση ενέργειας, την απαιτούμενη μνήμη ή το ποσοστό ευστοχίας ενός δικτύου στην ταξινόμηση. Για κάθε μία από τις παραπάνω λειτουργίες, διαφορετικές τεχνικές εκτέλεσης έχουν σχεδιαστεί και έγινε προσπάθεια για το σχεδιασμό ορθών και αποδοτικών αλγορίθμων για την επίλυση αυτών των εξειδικευμένων προβλημάτων. Με αυτόν τον τρόπο, η εφαρμογή καθίσταται, συνεπής, αποδοτική και ολοκληρωμένη.

Η υλοποίηση αποτελείται από τρία κυρίαρχα μέρη, καθένα από τα οποία εξυπηρετεί έναν τελείως διαφορετικό σκοπό: Μία διεπαφή προγραμματισμού εφαρμογών (ΔΠΕ) υψηλού επιπέδου, έναν διαχειριστή των νευρωνικών δικτύων που εκτελείται στον επεξεργαστή γενικού σκοπού του ενσωματωμένου, και τέλος απο τις χαμηλού επιπέδου υπολογιστικές βιβλιοθήκες.



Σχήμα 1.9: CMX DMA ελεγκτής της Myriad2

Στο σχήμα 1.10 φαίνεται μία διαγραμματική επισκόπηση ολόκληρης της υλοποίησης. Ο χρήστης δίνει ως είσοδο τα αρχεία που περιγράφουν το μοντέλο του νευρωνικού δικτύου (σε μορφή συμβατή με το Caffe), μαζί με μία εικόνα. Επίσης ο χρήστης μέσω των ορισμάτων της διεπαφής, ορίζει τη λειτουργία που θέλει να εκτελέσει καθώς και τα αρχιτεκτονικά χαρακτηριστικά του ενσωματωμένου που θα εκτελέσουν το δίκτυο.



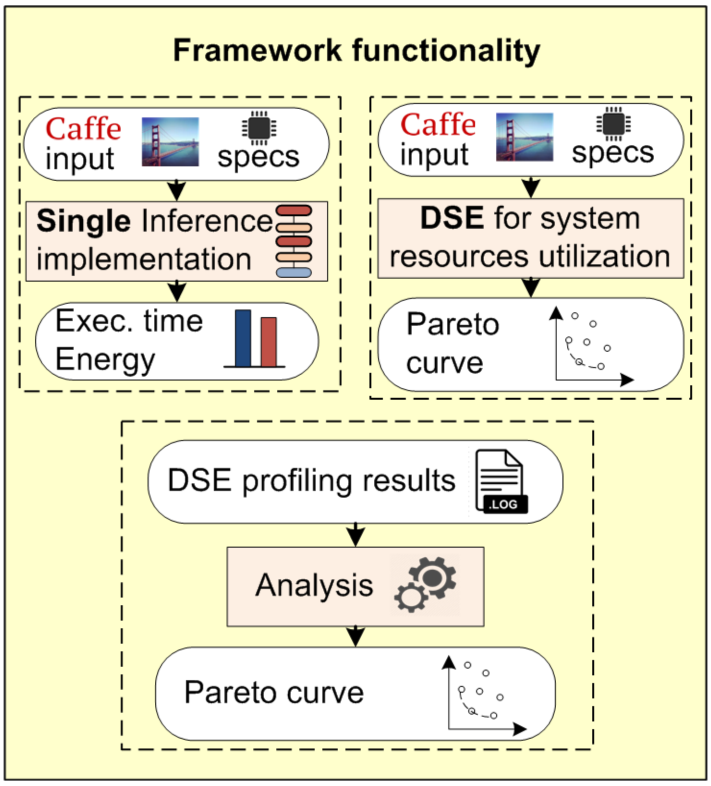
Σχήμα 1.10: Overview of the CNN implementation

1.6.1 Η Python διεπαφή της εφαρμογής

Η ενσωματωμένα πλατφόρμα Myriad2, ως ενσωματωμένο ειδικού σκοπού, δεν παρέχει στον χρήστη κάποια διεπαφή επικοινωνίας, ώστε εκείνος να περνάει ορίσματα ή αρχεία κατά την εκτέλεση μιας εφαρμογής. Αυτό σημαίνει, ότι χωρίς την κατασκευή κάποιας ιδιότυπης διεπαφής, είναι αδύνατον να χτιστεί μία γενικευμένη εφαρμογή όπου θα διαβάζει οποιοδήποτε μοντέλο νευρωνικού δικτύου ώστε να μπορέσει να δημιουργήσει το δίκτυο και να το εκτελέσει. Ως εκ τούτου, για να μπορέσει κάποιος να περιγράψει ένα δίκτυο μέσα στην Myriad, θα έπρεπε να γράψει χειροκίνητα δεκάδες γραμμές πηγαίου κώδικα ώστε να ανακατασκευάσει την πληροφορία του δικτύου και να το εκτελέσει. Αυτή η διαδικασία, σε μεγάλα δίκτυα, όπως για παράδειγμα το ”GoogleNet”, εκτός ότι απαιτεί χιλιάδες γραμμές κώδικα και πολλές ώρες τροποποίησης του κώδικα, θεωρεί επίσης ως δεδομένο ότι ο προγραμματιστής έχει πλήρη επίγνωση του low-level μηχανισμού που εκτελείται μέσα στο ενσωματωμένο.

Είναι προφανές, ότι αυτή η μέθοδος είναι εξαιρετικά μη-αποδοτική και η εύρεση μιας αποδοτικής και γενικευμένης λύσης είναι επιτακτική ανάγκη. Η ανάγκη αυτή, οδήγησε στη δημιουργία μιας υψηλού-επιπέδου διεπαφής, υλοποιημένη σε Python. Η διεπαφή αυτή προσφέρει ένα σύνολο απο προγραμματιστικά εργαλεία που αφορούν τις παραμέτρους της αρχιτεκτονικής που θα εκτελέσει το δίκτυο, η επιλογή των υπολογιστικών πόρων που θα χρησιμοποιηθούν και την επιλογή της λειτουργίας που θα εφαρμοστεί στο εκάστοτε δίκτυο. Τέλος, η διεπαφή αυτή παράγει ένα σύνολο από πηγαίους κώδικες και βιβλιοθήκες που περιέχουν όλες τις απαραίτητες πληροφορίες, σε αναγνώσιμη - από τη Myriad - μορφή, ώστε να κατασκευαστεί και να εκτελεστεί το δίκτυο.

Στο σχήμα 1.11, παρουσιάζεται η βασική λειτουργία της διεπαφής σε Python.



Σχήμα 1.11: Python Διεπαφή

1.6.2 Ο διαχειριστής της εκτέλεσης του δικτύου

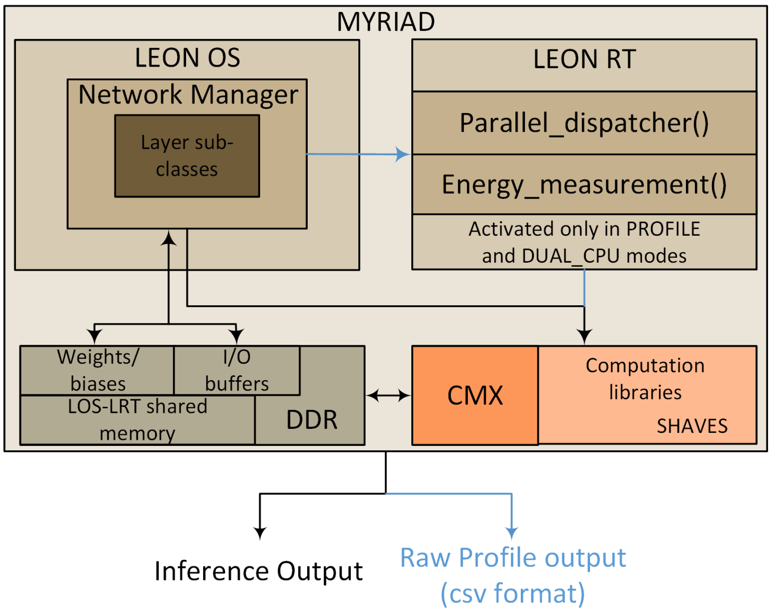
Το δεύτερο δομικό στοιχείο της CNN Engine που υλοποιήθηκε είναι ο διαχειριστής των νευρωνικών δικτύων εντός του ενσωματωμένου. Ένα κομμάτι του πηγαίου κώδικα που αφορά τον διαχειριστή αποτελείται από ένα σύστημα κλάσεων (ΑΔΤ), οι οποίες περιγράφουν όλα τα διαφορετικά είδη στρώσεων που χρησιμοποιούνται στα σύγχρονα συνελικτικά νευρωνικά δίκτυα, μαζί με τις εξειδικευμένες ιδιότητες, χαρακτηριστικά και λειτουργίες τους. Εκτός από την περιγραφή των στρώσεων, μία διαφορετική κλάση που αναπαριστά τον διαχειριστή του δικτύου ορίζεται και χρησιμοποιείται για να αποθηκεύει την πληροφορία του νευρωνικού δικτύου, να εφαρμόζει κάποια από τις διαθέσιμες τεχνικές εκτέλεσης και να τροποποιεί, αν χρειάζεται, τις παραμέτρους κάθε στρώσης (π.χ. το είδος της συνέλιξης που θα εκτελεστεί). Ο διαχειριστής γνωρίζει όλες τις απαραίτητες πληροφορίες του δικτύου που θα εκτελέσει, από τις αυτόματα παραχθήσες βιβλιοθήκες και πηγαίους κώδικες που προέκυψαν από την διεπαφή, όπως εξηγήθηκε νωρίτερα.

Δεδομένου ότι η διαχείριση ενός δικτύου, που αποτελείται από μία σειρά διαφορετικών κόμβων (στρώσεις) με κοινά χαρακτηριστικά, είναι η επιτομή μιας αντικειμενοστραφούς πρόκλησης, όλο το framework που αποτελεί την περιγραφή των στρώσεων και του διαχειριστή έχουν υλοποιηθεί σε C++ και εκτελούνται στους επεξεργαστές LEON OS και LEON RT. Ο master επεξεργαστής είναι ο LEON OS, ενώ σε περιπτώσεις που απαιτούνται και οι δύο (λειτουργία ”profiling” και ”dual processor”), τότε δίνεται εντολή και στον LEON RT να ενεργοποιηθεί για να συμβάλει στη διοχέτευση των κόμβων στο πολυεπεξεργαστικό σύστημα. Ο δεύτερος λόγος που διαλέχθηκε η C++ για την επίλυση αυτού του ζητήματος, είναι η κλιμακωσιμότητα που προσφέρει ένα αντικειμενοστραφές λογισμικό: Εκτός ότι οι υπο-κλάσεις είναι εύκολα επεκτάσιμες ώστε να προσφέρουν περισσότερες λειτουργίες ή πιο πολλές στρώσεις, αν κάποιος τις χρησιμοποιήσει ως αυτοτελείς δομικές μονάδες, μπορεί να προσθέσει πολλές ακόμα αλληλοεξαρτώμενες λειτουργίες, χωρίς να χρειαστεί να μάθει σε βάθος τον τρόπο που αυτές οι κλάσεις λειτουργούν.

Το σχήμα 1.12 δείχνει μία συνοπτική άποψη των δυνατών λειτουργιών εκτέλεσης και αποφάσεων του διαχειριστή που τρέχει στον LEON OS επεξεργαστή.

1.6.3 Υπολογιστικές μέθοδοι της εφαρμογής

Τέλος, η ουσία της CNN Engine κρύβεται πίσω από τις υπολογιστικές ρουτίνες οι οποίες τρέχουν στο πολυεπεξεργαστικό σύστημα του ενσωματωμένου. Για το σκοπό αυτό, έχουν αναπτυχθεί βέλτιστοι αλγόριθμοι μεταφοράς πινάκων μεταξύ των μνημών, καθώς και ρουτίνες στην εξειδικευμένη γλώσσα assembly του ενσωματωμένου. Οι παρακάτω στρώσεις, σύμφωνα με τα πρότυπα του ”Caffe”, είναι υλοποιημένες και υποστηρίζονται από την εφαρμογή:

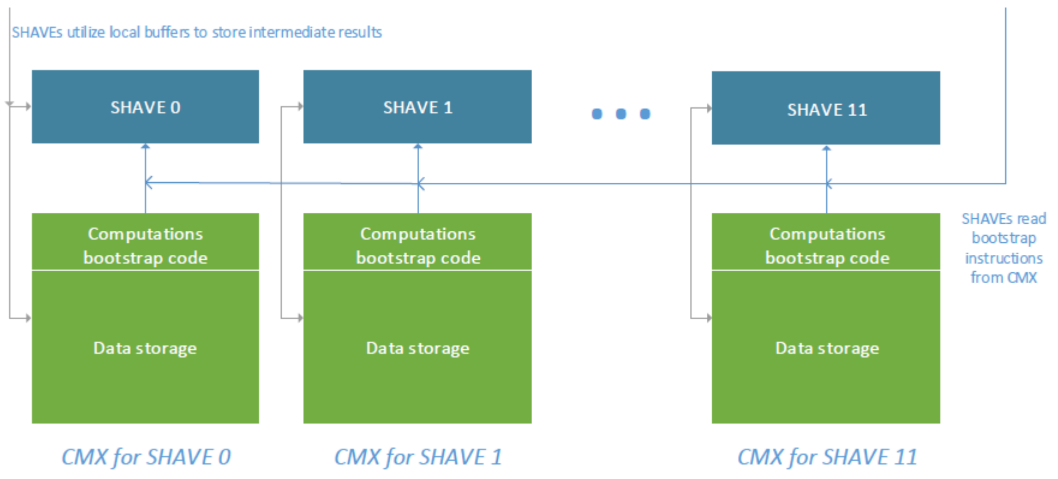


Σχήμα 1.12: Ο διαχειριστής των συνελικτικών νευρωνικών δικτύων

- Convolution (Direct και Im2Col τεχνικές)
- Pooling
- ReLU
- Fully Connected/Inner Product
- LRN-Across Channels
- Concat

Στο σχήμα 1.13 φαίνεται η βασική νοοτροπία που δομείται το πολυεπεξεργαστικό σύστημα:

Η CMX μνήμη, που χρησιμοποιείται από τα SHAVES (πυρήνες) για να αποθηκεύσει τα δεδομένα, είναι μικροσκοπική σε σχέση με το τεράστιο όγκο δεδομένων στα οποία πρέπει να εφαρμοστούν οι υπολογιστικοί αλγόριθμοι. Επομένως, η βασική αρχή της υλοποίησης είναι να φέρνει τμήματα από τους τεράστιους πίνακες, σε γύρους, και να τα υπολογίζει ώστε να τα επιστρέφει πίσω στην μνήμη DDR. Επιπλέον, το τμήμα κώδικα των συναρτήσεων των SHAVE έχει μεταφερθεί στην DDR, ούτως ώστε να μη χρειαστεί να δεσμεύσει πολύτιμο χώρο της CMX. Ο χώρος αυτός θα χρησιμοποιηθεί σχεδόν αποκλειστικά για την αποθήκευση πινάκων. Αυξάνοντας το διαθέσιμο μέγεθος για πίνακες, μειώνει τις επαναλήψεις που πρέπει να λάβουν χώρα ώστε να έρθουν όλα τα block, καθώς και τις μεταφορές από τη μία μνήμη στην άλλη.


SHAVEs utilize local buffers to store intermediate results
SHAVE 0
SHAVE 1
SHAVE 11
Computations bootstrap code
Data storage
SHAVEs read bootstrap instructions from CMX
CMX for SHAVE 0
CMX for SHAVE 1
CMX for SHAVE 11

Σχήμα 1.13: Εκτέλεση στα SHAVES

1.7 Αξιολόγηση της υλοποίησης

1.7.1 Αρχικοποίηση της διαδικασίας αξιολόγησης

Η υλοποίηση αξιολογήθηκε χρησιμοποιώντας 6 διαφορετικά συνελικτικά νευρωνικά δίκτυα, τα οποία παρουσιάζουν μεγάλη ποικιλομορφία μεταξύ τους, ως προς την πολυπλοκότητα: AlexNet, GoogleNet, NiN-imagenet, SqueezeNet, VGG and ZFnet. Τα δίκτυα αυτά επιλέχθηκαν με βάση 2 κριτήρια:

- Να απαιτείται ένα ευρύ φάσμα από υπολογιστικούς πόρους, έτσι ώστε να παρθούν αποτελέσματα τα οποία να διαφέρουν σημαντικά ως προς το χρόνο εκτέλεσης, την κατανάλωση ενέργειας και την ακρίβεια πρόβλεψης.
- Να θεωρούνται ευρέως διαδεδομένα, σύγχρονα νευρωνικά δίκτυα και να χρησιμοποιούνται στις τεχνολογικά εξελιγμένες εφαρμογές αναγνώρισης αντικειμένων και ταξινόμησης εικόνων.

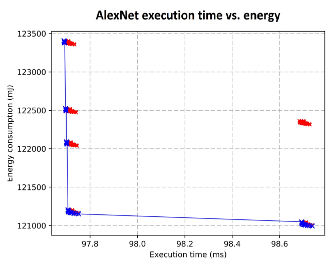
CNN	input image	output vector	#layers	#memory(MB)	Error rate
AlexNet	$[277 \times 277 \times 3]$	$[1 \times 1000]$	13	117	17
GoogleNet	$[277 \times 277 \times 3]$	$[1 \times 1000]$	83	16.6	7
NiN-imagenet	$[277 \times 277 \times 3]$	$[1 \times 1000]$	16	15.5	17.5
SqueezeNet	$[277 \times 277 \times 3]$	$[1 \times 1000]$	38	4.68	19.7
VGG	$[277 \times 277 \times 3]$	$[1 \times 1000]$	16	276	8
ZFnet	$[277 \times 277 \times 3]$	$[1 \times 1000]$	13	121	16.5

Πίνακας 1.1: Λεπτομέρειες των νευρωνικών δικτύων που χρησιμοποιήθηκαν για την αξιολόγηση

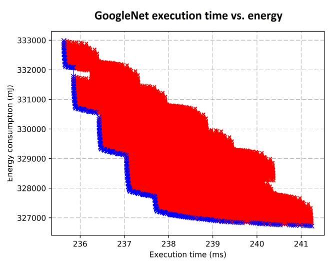
Στον πίνακα 1.1 φαίνονται τα χαρακτηριστικά των δικτύων που αναφέρθηκαν. Και τα 6 δίκτυα χρησιμοποιούν τις ίδιες διαστάσεις εισόδου και είναι εκπαιδευμένα στο σετ δεδομένων ”ImageNet” για τον ίδιο αριθμό κλάσεων εξόδου, επομένως μπορούν να χρησιμοποιηθούν εναλλάξ το ένα με το άλλο. Ο αριθμός των στρώσεων διαφέρει σημαντικά, από 13 στρώσεις (AlexNet and ZFnet) ως και 83 (GoogleNet). Προφανώς, το ίδιο ισχύει και για τις απαιτήσεις σε μνήμη, που αναφέρεται στο μέγεθος που απασχολούν, στην κεντρική μνήμη του συστήματος, τα βάρη και οι πολώσεις των δικτύων. Μία ακόμα σημαντική μετρική είναι το top 5 ποσοστό λάθους πρόβλεψης, που ποικίλλει από 19.7 (SqueezeNet) ως 8 (VGG). Η προτεινόμενη μεθοδολογία στην παρούσα διπλωματική θα χρησιμοποιηθεί για να i) προσφέρει βέλτιστα προσαρμοσμένες παραμέτρους για τα δίκτυα στην πλατφόρμα Myriad και ii) να επιδείξει τα trade-off που υπάρχουν μεταξύ του χρόνου εκτέλεσης και της κατανάλωσης ενέργειας μεταξύ αυτών των βέλτιστα προσαρμοσμένων συνδυασμών.

1.7.2 Εξερεύνηση του χώρου σχεδιασμού

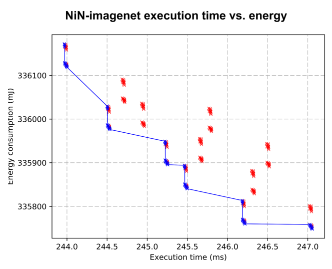
Η προτεινόμενη μεθοδολογία για την εκτέλεση των δικτύων εφαρμόστηκε στα 6 προαναφερθέντα δίκτυα του πίνακα 1.1. Η έξοδος του πρώτου τμήματος της μεθοδολογίας φαίνεται στο σχήμα 1.14. Τα διαγράμματα Pareto για κάθε νευρωνικό δίκτυο για τον χρόνο εκτέλεσης ως προς την κατανάλωση ενέργειας παρέχονται αυτομάτως από το framework που υλοποιεί την μεθοδολογία. Κάθε σημείο στα διαγράμματα είναι μία υλοποίηση της αρχιτεκτονικής του δικτύου που χρησιμοποιεί έναν διαφορετικό μέγεθος από υπολογιστικούς πόρους. Με άλλα λόγια, ένα σετ από ρυθμίσεις για κάποιο δίκτυο διαφέρει από ένα άλλο σετ, στους αριθμούς πυρήνων που θα επεξεργαστούν τουλάχιστον μία στρώση.



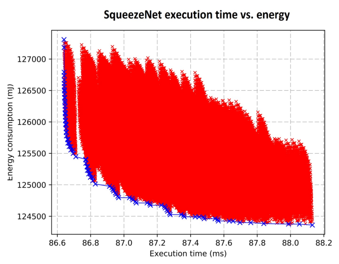
(a) AlexNet: Execution time vs. energy



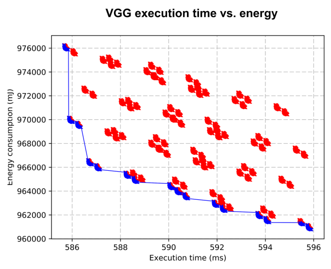
(b) GoogleNet: Execution time vs. energy



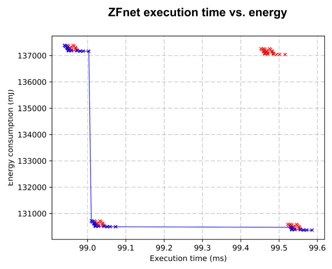
(c) NiN-imagenet: Execution time vs. energy



(d) SqueezeNet: Execution time vs. energy



(e) VGG: Execution time vs. energy



(f) ZFnet: Execution time vs. energy

Σχήμα 1.14: Βήμα 1 της μεθοδολογίας: Βέλτιστη προσαρμογή των νευρωνικών δικτύων στην Myriad2

Trade-offs μεταξύ του χρόνου εκτέλεσης και της κατανάλωσης ενέργειας παρατηρούνται σε όλες τις αρχιτεκτονικές νευρωνικών δικτύων. Με μία πιο προσεκτική ματιά, είναι εύκολα αντιληπτό ότι οι πιο υπολογιστικά έντονες στρώσεις, όπως προκύπτει από τα αποτελέσματα, είναι οι συνελικτικές στρώσεις, το οποίο είναι αναμενόμενο. Πράγματι, οι μετρήσεις δείχνουν ότι από το συνολικό χρόνο εκτέλεσης των δικτύων του πίνακα 1.1, το 68% ως 99% σπαταλάται σε συνελικτικές στρώσεις. Από τη στιγμή που η συνέλιξη είναι υπολογιστικά πολύπλοκη πράξη, συνήθως ο βέλτιστος αριθμός πυρήνων είναι 11 ή 12. Παρόλα αυτά, πολλές φορές αυτός ο αριθμός πυρήνων δεν είναι πάντα ο αποδοτικότερος ως προς την κατανάλωση ενέργειας.

Μία ενδιαφέρουσα παρατήρηση είναι το γεγονός ότι τα AlexNet, NiN-imagenet, VGG και ZFnet είναι συμπλεγμένα σε ομάδες. Τα αποτελέσματα δείχνουν ότι οι υλοποιήσεις που ανήκουν στο ίδιο σύμπλεγμα, οι συνελικτικές στρώσεις έχουν ακριβώς τις ίδιες παραμέτρους (στην περίπτωση μας τον ίδιο αριθμό πυρήνων). Εκεί που διαφέρουν, είναι στις υπόλοιπες στρώσεις, όπως για παράδειγμα στη στρώση pooling και τη στρώση fully connected, οι οποίες έχουν συσχετιστικά μικρότερο αντίκτυπο στο χρόνο εκτέλεσης ή την ενέργεια. Από την άλλη πλευρά, υλοποιήσεις που ανήκουν σε διαφορετικά συμπλέγματα, τείνουν να έχουν διαφορετικές παραμέτρους μεταξύ αντίστοιχων συνελικτικών στρώσεων.

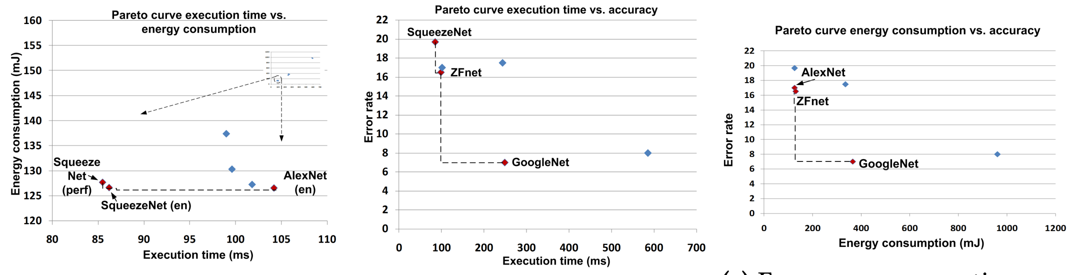
Πίνακας 1.2: Αθροιστικά αποτελέσματα του πρώτου βήματος της μεθοδολογίας

	AlexNet	GoogleNet	NiN-imagenet	SqueezeNet	VGG	ZFnet
Exec. time (ms)	97.69	203.05	243.96	78.08	585.66	98.93
% Exec. time gain	2.3	17.73	1.2	11.47	1.72	0.65
Energy consumption (J)	120.9	286.9	335.7	109.9	960.7	130.3
% Energy gain	1.2	13.8	0.12	13.6	1.53	5.11

Πίνακας 1.3: Σύγκριση μεταξύ τυχαίας παραμετροποίησης και βέλτιστα προσαρμοσμένης παραμετροποίησης στα ΣΝΔ.

	AlexNet	GoogleNet	NiN-imagenet	SqueezeNet	VGG	ZFnet
Exec. time 12 VPUs (ms)	98.16	238.6	244	87	587.2	99.1
Exec. time fine-tuned (ms)	97.6	203.5	243	78	585.6	98.9
% Exec. time gain	0.48	14.9	0.01	10.37	0.27	0.19
Energy 12 VPUs (J)	133.7	343.6	337	129.3	1004	141.2
Energy fine-tuned (J)	120.9	286.9	335.7	109.9	960.7	130.3
% Energy gain	9.56	16.61	0.6	18.34	10.83	8.46

Ο πίνακας 1.2 δείχνει τα μέγιστα κέρδη στο χρόνο εκτέλεσης και την κατανάλωση ενέργειας μεταξύ της γρηγορότερης και της αποδοτικότερης υλοποίησης για κάθε ΣΝΔ, το οποίο αγγίζει το 5.3%. Τα αποτελέσματα του δεύτερου βήματος της μεθοδολογίας παρουσιάζονται στην εικόνα 1.15. Διαγράμματα Pareto παρουσιάζουν τα trade-off μεταξύ του χρόνου εκτέλεσης, της κατανάλωσης ενέργειας και της ακρίβειας πρόβλεψης. Η εικόνα 1.15a δείχνει την καμπύλη Pareto ως προς το χρόνο εκτέλεσης και την κατανάλωση ενέργειας.



(a) Execution time vs. energy consumption **(b)** Execution time vs. accuracy **(c)** Energy consumption vs. accuracy

Σχήμα 1.15: Αποτελέσματα της μεθοδολογίας: Trade-offs μεταξύ χρόνου εκτέλεσης, κατανάλωσης ενέργειας και ακρίβειας πρόβλεψης μεταξύ των διαφόρων ΣΝΔ.

Chapter 2

Introduction to Machine Learning and Neural Networks

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

2.1 Machine learning

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

2.1.1 Types of machine learning

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

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

- *Reinforcement Learning*: The goal is to develop a system (agent) that improves its performance based on interactions with the environment. Since the information about the current state of the environment typically also includes a so-called reward signal, reinforcement learning can be thought of as a field related to supervised learning. However, in reinforcement learning this feedback is not the correct ground truth label or value, but a measure of how well the action was measured by a reward function. Through the interaction with the environment, an agent can then use reinforcement learning to learn a series of actions that maximizes this reward via an exploratory trial-and-error approach or deliberative planning

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

Figure 2.1 presents an overview of the two major types of machine learning.

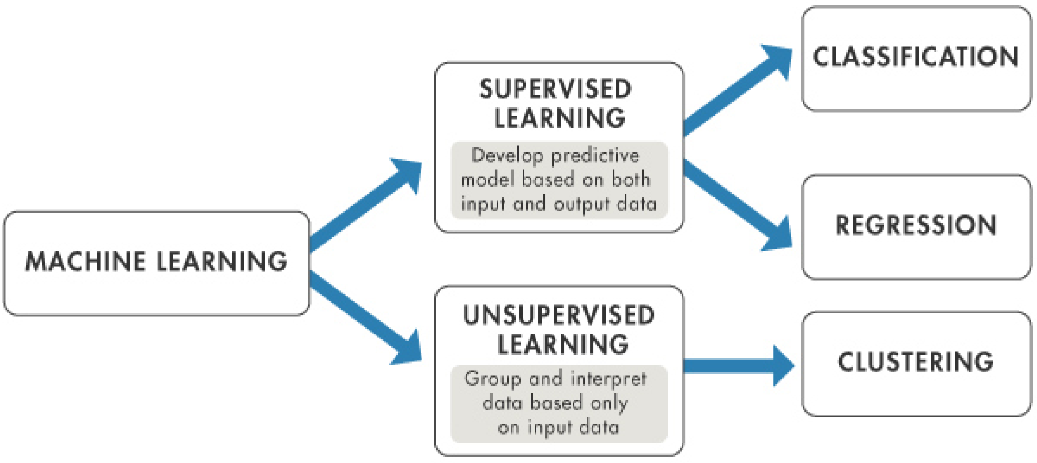

MACHINE LEARNING
SUPERVISED LEARNING
Develop predictive model based on both input and output data
UNSUPERVISED LEARNING
Group and interpret data based only on input data
CLASSIFICATION
REGRESSION
CLUSTERING

Figure 2.1: Machine learning basic types overview

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

2.1.2 Supervised learning classification

Classification is a subcategory of supervised learning where the goal is to predict the categorical class labels of new instances based on past observations [2]. Those class labels are discrete, unordered values that can be understood as the group memberships of the instances. The previously mentioned example of e-mail spam detection represents a typical case of a binary classification task, where the machine learning algorithm learns a set of rules in order to distinguish between two possible classes: spam and non-spam e-mail. However, the set of class labels does not have to be of a binary nature.

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

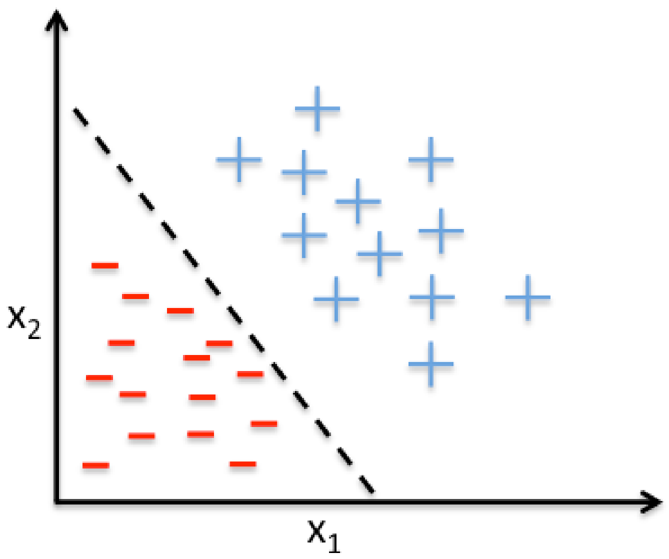


Figure 2.2: Binary classification in supervised learning

Figure 2.2 illustrates the concept of a binary classification task given 27 training samples: 15 training samples are labeled as negative class (minus signs) and 12 training samples are labeled as positive class (plus signs). In this scenario, the dataset is two-dimensional, which means that each sample has two values associated with it: x_1 and x_2 .Now, a supervised machine learning algorithm can be used to learn a rule - the decision boundary represented as a black dashed line - that can separate those two classes and classify new data into each of those two categories given its x_1 and x_2 values.

2.1.3 Supervised learning regression

The previous section showed that the task of classification is to assign categorical, unordered labels to instances [2]. A second type of supervised learning is the prediction of continuous outcomes, which is also called regression analysis. In regression analysis, a number of predictor (explanatory) variables and a continuous response variable (outcome) is given, and the goal is to find a relationship between those variables that allows the prediction of an outcome. For example, assume that there is interest in predicting the Math SAT scores of students. If there is a relationship between the time spent studying for the test and the final scores, it could be used as training data to learn a model that given the study time, predicts the test scores of future students who are planning to take this test.

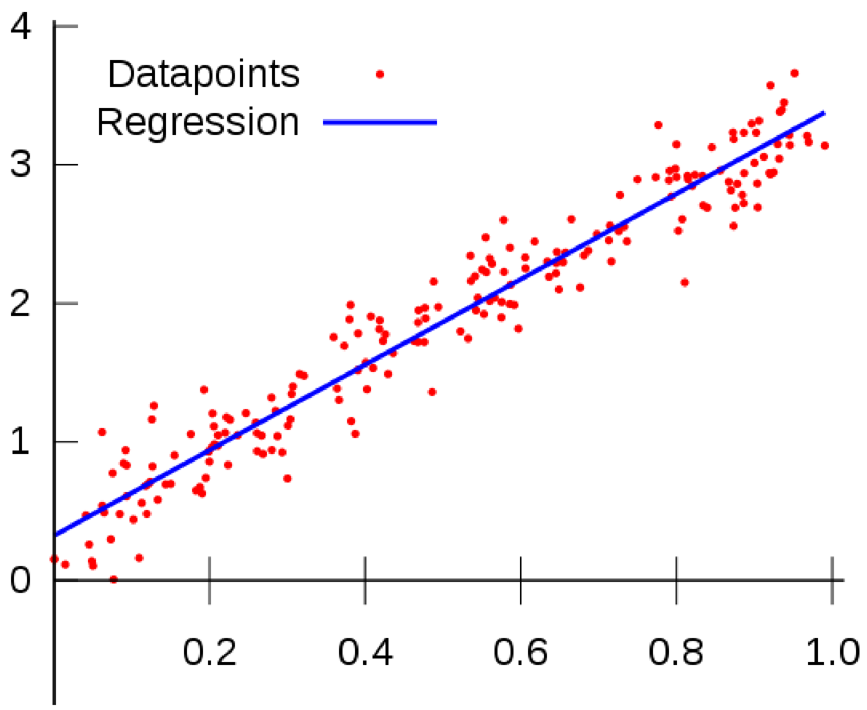


Figure 2.3: Linear regression in supervised learning

Figure 2.3 illustrates the concept of linear regression. Given a predictor variable x and a response variable y, a straight line is fitted to this data that minimizes the distance - most commonly the average squared distance - between the sample points and the fitted line. Then, the intercept and slope learned from this data is used to predict the outcome variable of new data.

2.2 Convolutional Neural Networks

A Convolutional Neural Network (CNN) is a special type of artificial neural network topology, that is inspired by the animal visual cortex and tuned for computer vision tasks by Yann LeCun in early 1990s [3]. It is a multi-layer perceptron, which is an artificial neural network model, specifically designed to recognize two-dimensional shapes. This type of network shows a high degree of invariance to translation, scaling, skewing, and other forms of distortion.

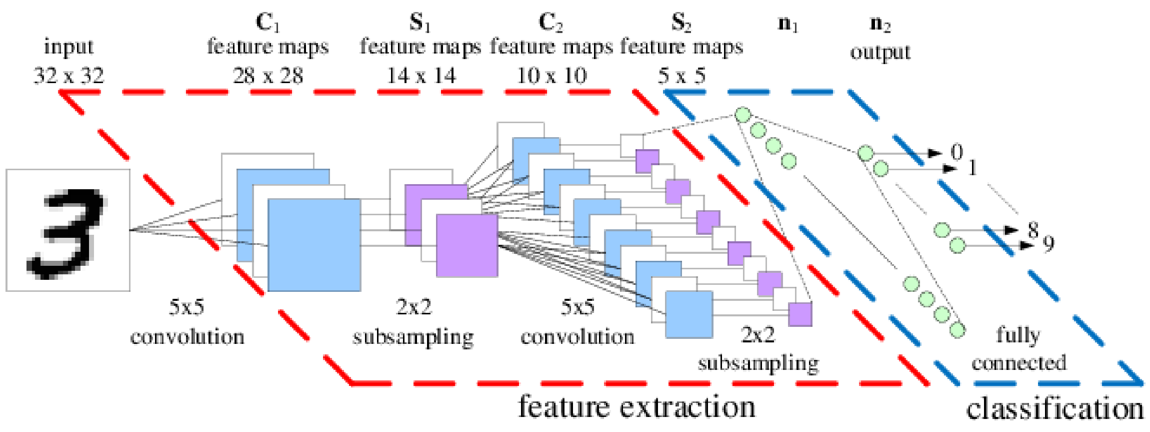


Figure 2.4: Example topology of a CNN suitable for handwritten digit recognition

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

2.2.1 Data arrangement in a CNN

CNNs take advantage of the fact that the input consists of images and they constrain the architecture in a more sensible way [4]. In particular, unlike a regular ANN, the layers of a CNN have neurons arranged in 3 dimensions: width, height, depth. Note that the word depth here refers to the third dimension of an activation volume, not to the depth of a regular ANN, which can refer to the total number of layers in the network. For example, 32 $\times$ 32 RGB input images represent an input volume of activations, that has dimensions 32 $\times$

32 × 3 (width, height, depth respectively). The neurons in a layer will only be connected to a small region of the layer before it, instead of all of the neurons in a fully-connected manner. The constrained, local connectivity of CNNs will be clarified shortly. An input 3D volume is shown in figure 2.5 for a 4 × 4 RGB image. It is noted that the range of values in the 3D volume can be adjusted to assist the training of the network. The process of manipulating the input data, prior to feeding them to the CNN, in order to bring them to the desired form is called preprocessing.

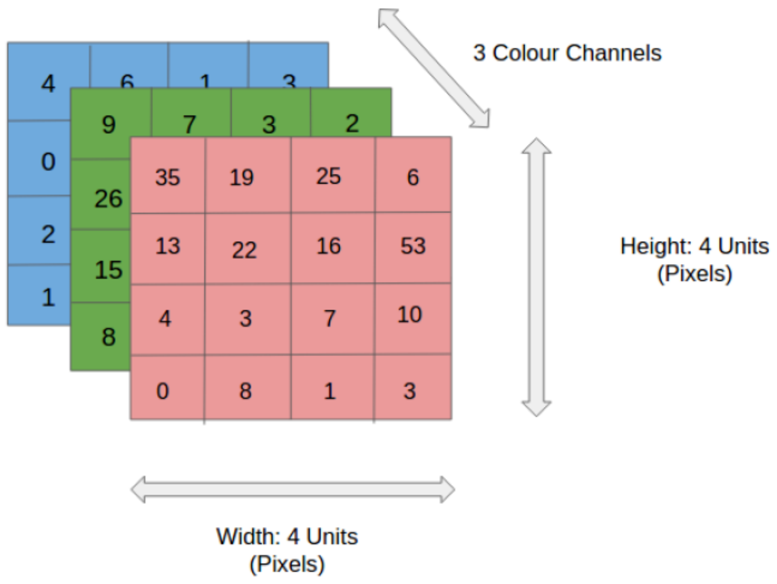


Figure 2.5: Cross-section of an input volume of size: 4 × 4 × 3. It comprises of the 3 color channel matrices of the input image.

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

2.2.2 Common layers used to build CNNs

There are three main types of layers to build CNN architectures: Convolutional Layer, Pooling Layer, and Fully-Connected Layer [5]. The latter is exactly as seen in regular ANNs. These three layers are stack interchangeably to produce interesting architectures. There is also the Input Layer, which is nothing more than the identity transform, i.e. its output is the same as its input. These layer are analyzed briefly below:

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

- *Convolutional Layer*: Will compute the output of neurons that are connected to local regions in the input. Each computation is a spatial (width, height) convolution between their weights and a small region they are connected to in the input volume. The depth of the output volume depends on the numbers of filters that is given to the layer as an extra parameter.
- *Pooling Layer*: Will perform a downsampling operation along the spatial dimensions (width, height).
- *Fully Connected (FC) Layer* As with ordinary ANNs and as the name implies, each neuron in this layer will be connected to all the elements in the input volume of the layer.
- *ReLU Layer*: Will apply an elementwise activation function, namely f(x) = max(0, x) thresholding at zero. This layer is used for helping the network training and leaves the size of the input volume unchanged.
- *LRN Layer*: Will perform a kind of "lateral inhibition" by normalizing over local input regions.
- *Concat Layer*: Will concatenate its multiple input blobs to one single output blob.

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

Chapter 3

Caffe Framework and Myriad2 Embedded platform

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

3.1 Caffe: Convolutional Architecture for Fast Feature Embedding

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

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, 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 Network training

Caffe trains models by the fast and standard stochastic gradient descent algorithm. Figure 3.1 shows a typical example of a Caffe network (for MNIST digit classification) during training [6]: a data layer fetches the images and labels from disk, passes it through multiple layers such as convolution, pooling and rectified linear transforms, and feeds the final prediction into a classification loss layer that produces the loss and gradients which train the whole network. This example is found in the Caffe source code at examples/lenet/lenet_-train.prototxt. Data are processed in mini-batches that pass through the network sequentially. Vital to training are learning rate decay schedules, momentum, and snapshots for stopping and resuming, all of which are implemented and documented.

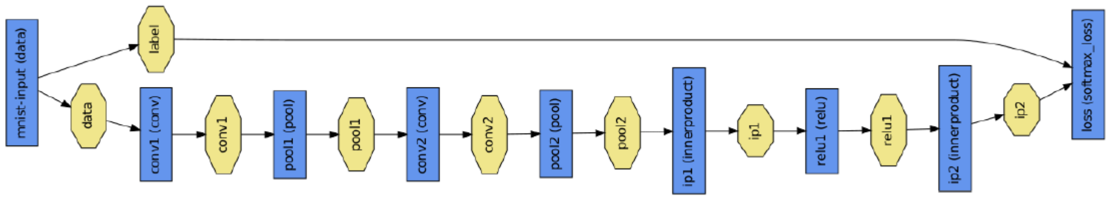


Figure 3.1: MNIST digit classification example of a Caffe network. Blue boxes represent layers and yellow octagons represent data blobs produced by or fed into the layers

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

3.2 Myriad 2 multicore SoC

The target platform of implementation is the Myriad2 System-on-Chip (SoC) processing unit [7]. It is developed by Movidius Ltd, that recently joined Intel's Perceptual Computing Group to accelerate adoption of visually intelligent devices. Myriad2 delivers high-performance machine vision and visual awareness in severely power-constrained environments. For that reason, it is the world's first Vision Processing Unit (VPU) that specifically targets embedded applications. The main characteristics of Myriad2 are:

- An ultra-low power design: For mobile and connected devices where battery life is critical, Myriad2 provides a way to combine advanced vision applications in a low power profile. This enables new vision applications in small form factors that could not exist before.
- A high-performance processor: Bringing vision technologies in connected devices closer to the capabilities of human vision is what Myriad2 is all about. It enables advanced vision applications that are impossible with conventional processors.
- A programmable architecture: The flexibility for developers to implement differentiated and proprietary applications is fundamental to Myriad2. The provided optimized software libraries give device manufacturers the ability to differentiate, not duplicate, at the core level.
- A small-area footprint: To conserve space inside mobile, wearable, and embedded devices, Myriad 2 was designed with a very small footprint that can easily be integrated into existing products.

A high level view of the hardware is shown in figure 3.2. From there, it is seen that Myriad2 SoC contains fourteen different processors. The two processors on the right are fundamentally different from the twelve vector processors on the left. In fact, the processors named "CPU" are of 32-bit SPARC architecture, which belongs to the RISC family of processors.

A more detailed view follows below [9]:

- Leon OS: Is one of the SPARC CPUs. It belongs to the CPU sub-system (CSS) that has been designed to be the main communication and control unit with the outside world via the external communication peripherals: I2C blocks, I2S blocks, SPI blocks, UART, GPIO, ETH and USB3.0. The control unit of this block is the Leon OS (LOS) RISC processor, but in this block the Leon owns much bigger L1 (32 KB) and L2 (256 KB) caches, which allows to put a modern RTOS on it. This block also offers an AHB

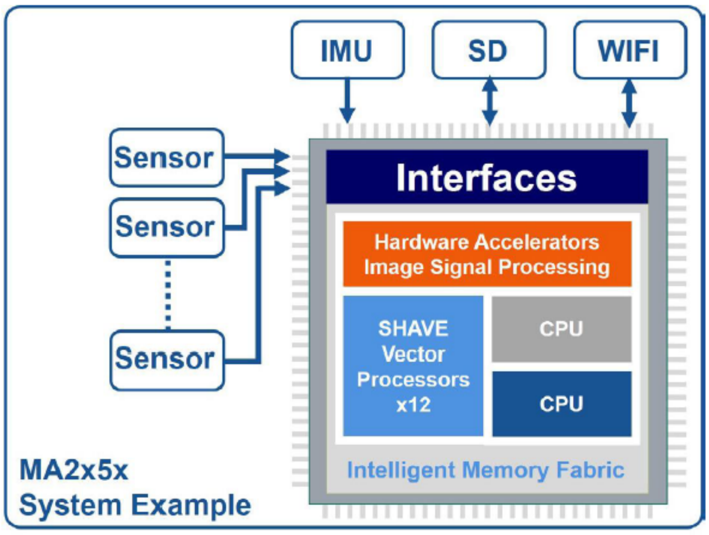


Figure 3.2: Overview of Myriad2 hardware

DMA engine for more optimal data transfer via the external peripherals. Beside handling the external interfaces and communication, Leon OS could also control SHAVE processors imaging algorithms

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

- SIPP: Is a proprietary software/hardware mechanism used by the Myriad2 processor to achieve highly optimized scheduling of Image Signal Processing (ISP) pipeline functionality. This mechanism is responsible for utilizing the HW filters provided by Myriad2 to achieve the best performance possible. This component is the orange block shown in figure 3.3.

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

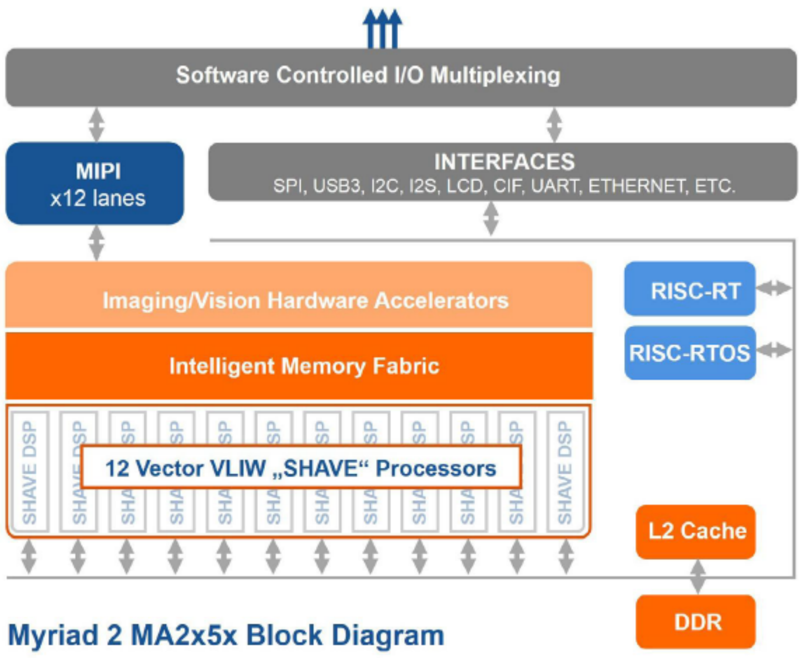

Software Controlled I/O Multiplexing
MIPI
x12 lanes
INTERFACES
SPI, USB3, I2C, I2S, LCD, CIF, UART, ETHERNET, ETC.
Imaging/Vision Hardware Accelerators
Intelligent Memory Fabric
12 Vector VLIW „SHAVE“ Processors
SHAVE DSP
RISC-RT
RISC-RTOS
L2 Cache
DDR
Myriad 2 MA2x5x Block Diagram

Figure 3.3: More detailed overview of Myriad2 hardware

- Microprocessor Array (UPA): Is the unit in Myriad 2 holding the 12 Very Long Instruction Word (VLIW) SHAVE vector processors (see figure 3.4), the 2 MB CMX SRAM memory and a few other blocks from which the most important are: the specialized DMA engine and the 256 KB L2 cache memory available to the SHAVE cores. This unit's main purpose is to provide support for customized code required by many computer vision and machine learning applications, as well as any other general computation intensive algorithms. Each VLIW processor controls multiple functional units which have SIMD capability for high parallelism and throughput at a functional unit and processor level. Each of these units can be launched in parallel in a single instruction. SHAVEs support SIMD instructions on multiple types, including but not limited to: 8 bits integers, 16 bits integer, 32 bits integer, 16 bits float, 32 bits float

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

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

3.3 CMX DMA Controller

The CMX DMA resides between the 128-bit MXI bus and CMX memory [8]. It provides high bandwidth data transfers between CMX and DDR in either direction. It also supports data transfers from DDR back to DDR or from CMX to CMX, allowing data to be relocated within the same physical location. Figure 3.4 shows a high level description of the DMA engine. The unit of work in the DMA engine is expressed though transaction tasks. Up to four linked lists of transactions are maintained in system memory, thus the DMA capability of serving transactions is not unlimited and can be easily flooded with requests if the programmer makes unregulated use of it.

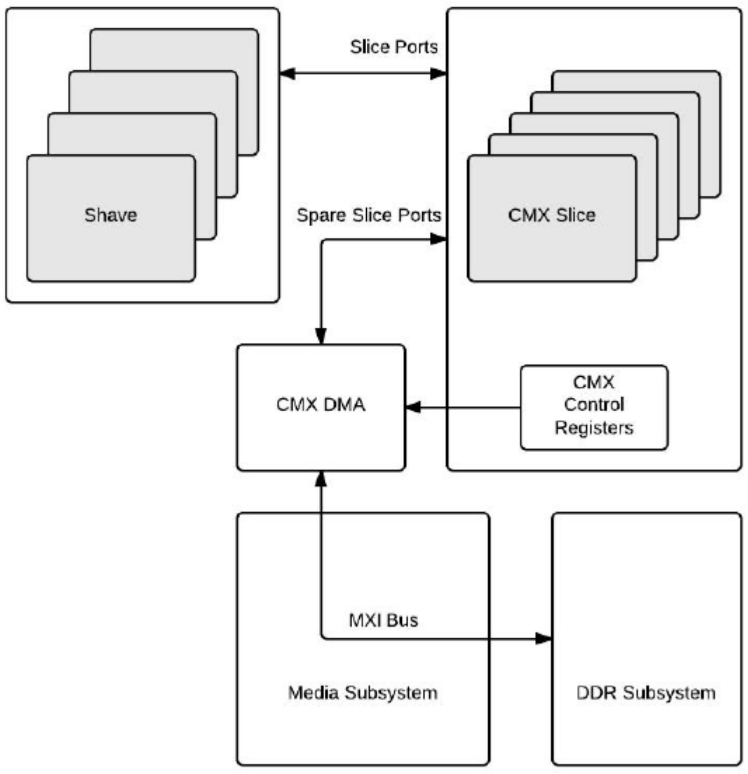


Figure 3.4: CMX DMA engine of Myriad2

3.4 Myriad2 Development Kit

The Myriad2 Development Kit (MDK) comprises common code which includes both drivers and components, and some example applications [9]. Also, the MDK provides an extensive build system - based on the GNU Makefile - that offers the means to build an application, the means to configure it and some functional targets, such as make, make run and make start_server.

3.4.1 MDK Components

This subsection provides a brief description of the reusable components included in the MDK [9]. Components are located under the mdk/common/components directory and selectively included in projects through the Makefile. A detailed description of each components can be found to the header file comments within each component. For the purposes of CNN implementation, an essential component is the KernelLib/MvCV, i.e. the Movidius Computer Vision kernel library. This library contains optimized assembly SHAVE routines for performing convolution, pooling and other related operations that are important to the implementation of the CNN.

Chapter 4

Overview of the CNN Engine

In this chapter, a brief and abstract explanation will be given regarding the functionality of the CNN Engine as well as the framework that supports it. The purpose of this section is to provide a reader with a good insight of how the different components of the engine fit together and get a good understanding of its functionalities before proceeding to the next chapters, in which each component is explicitely and thoroughly described.

4.1 An abstract view

The proposed CNN Engine and framework aims to provide an energy efficient, yet resource-powerful computational mechanism for Convolutional Neural Networks and can be used to execute networks for inference related problems or profile networks on different architectures and perform design space exploration on certain metrics like performance, energy consumption, memory footprint and prediction rate. For each of these subsets of the implementation, different execution techniques have been implemented and notable algorithms have been designed to solve challenges related to the specialized nature of the problem and provide a consistent, efficient and well-rounded application.

The implementation is divided into three major components, each one serving a totally different purpose: The high-level API (Application Programming Interface), the CNN Manager and the low-level DMA and Computational routines, each of which will be analyzed in depth in the next chapters.

Figure 4.1 shows a high-level view of the CNN implementation framework. The user input is the network architecture (in a format compatible with Caffe), the input image(s), platform specifications and/or profiling results.

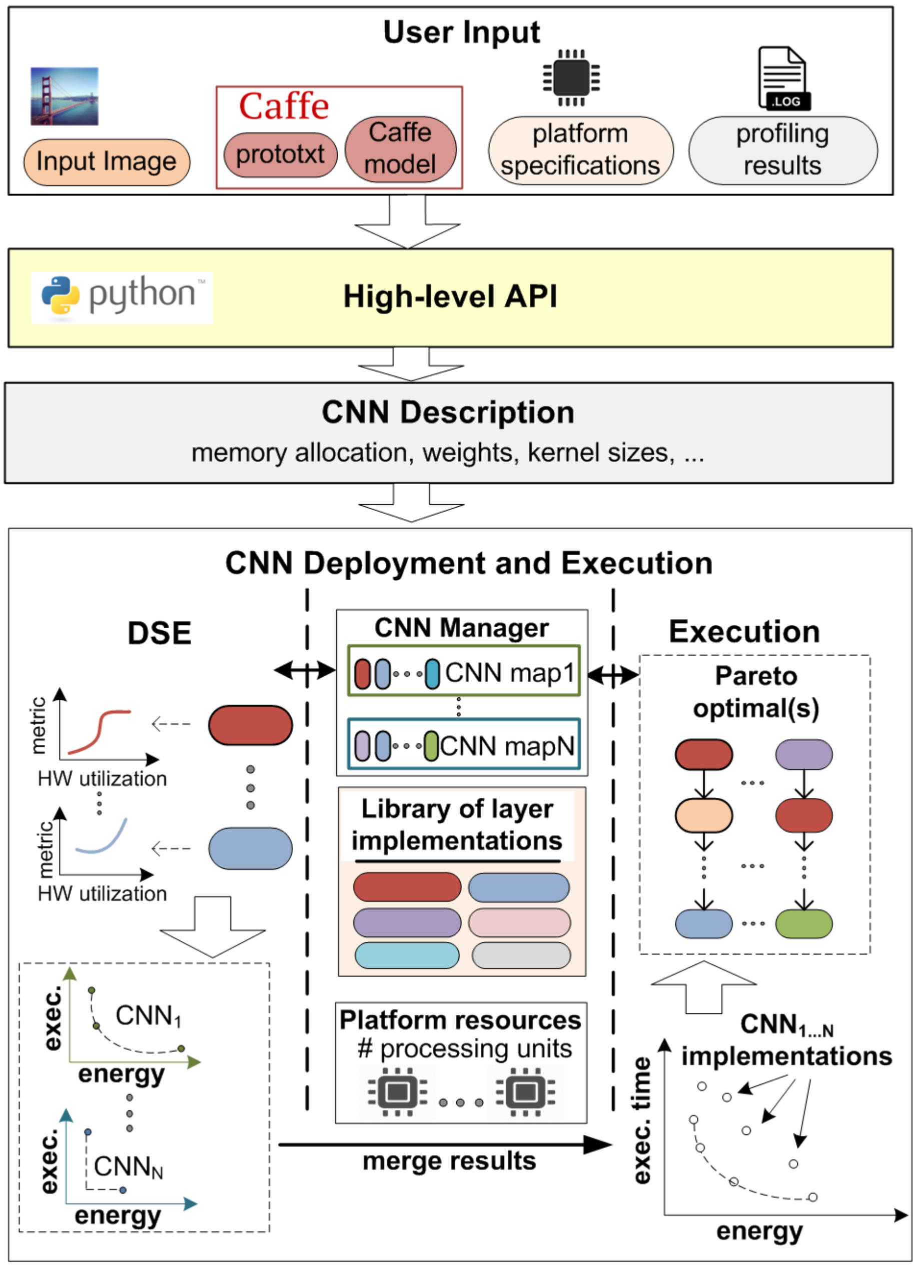


Figure 4.1: Overview of the CNN implementation

4.2 Introduction to the Python Interface

The Myriad embedded platform, being a purpose-specific chip does not provide to the user a communication interface which could be used to receive arguments and parse files on run-time. This means that it is not possible to provide the prototxt and caffemodel files (the ”Caffe” CNN description as explained in previous chapter) to the engine, for the network to be deployed. As a consequence, in order for the network description to be given inside Myriad, one should manually write hundreds of lines of code to reconstruct the network and execute it on this embedded platform: The execution of a different network, especially a large one like ”GoogleNet” would require hours of code modification, taking for granted that the user is fully aware of the code running under the hood. It is obvious that this method is extremely inefficient and it was vital to find a solution. The necessity to create a generalized engine, which would be able to deploy any possible network providing abstract features to the user, gave birth to the high-level API. This API, developed in Python, provides a set of functions accessible to developers for the configuration of CNN implementations, the utilization of hardware resources and the selection between different operation modes. The interface generates a set of CNN Description source files that contain information about the low-level CNN implementation on the underlying platform.

In figure 4.2, the basic principles of the python framework are presented.

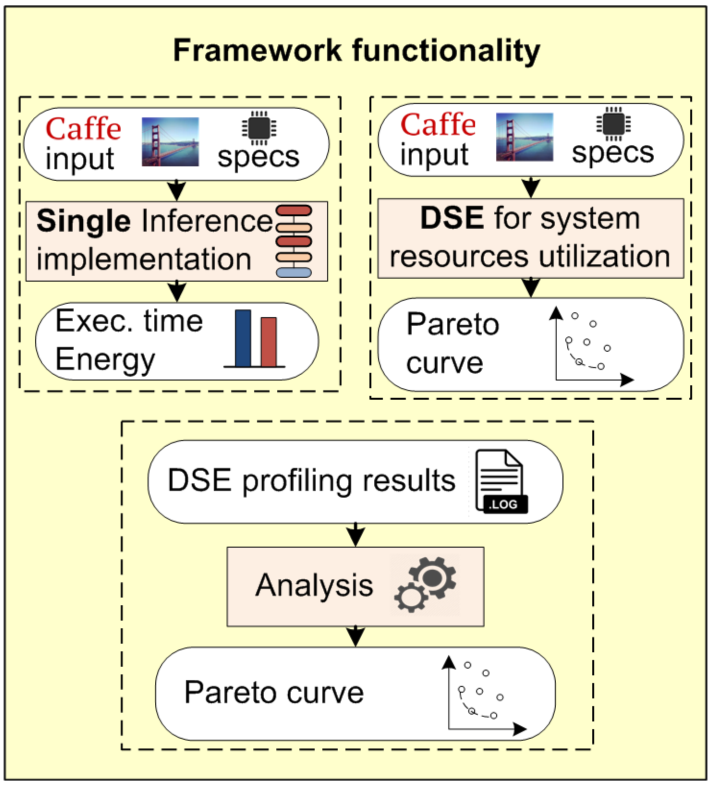


Figure 4.2: The High-level Python framework

4.3 Introduction to the Network Manager

Another major component of the engine is the CNN Manager. A part of the manager is consisted of abstract data types that describe the different layer types that are used in Convolutional Neural Networks, with their specific properties, characteristics and abstract methods. Apart from the layer description, a network manager class is defined and used to store the network, deploy an execution mode selected from a variety of modes and alter layer parameters (e.g. the convolution method used). The manager receives all appropriate network information from the auto-generated libraries produced by the Python interface, as explained above.

Since the handling of a network consisting of a set of nodes (layers) is the epitome of object oriented problems, the manager is purely written in C++ and is executed primarily on LEON OS. In ”profiling” and ”dual cpu” mode, both LEON OS and LEON RT are used to handle the network and dispatch it to the shaves. Another reason C++ was chosen for this sector of the application was the code scalability that it offers: Using these classes and data types as black boxes, it is very easy for someone to extend the application, without needing to understand in depth the implementation techniques of the existing structures.

Figure 4.3 presents an abstract view of the possible execution branches, actions and decisions taken by the Manager running on LEON processors.

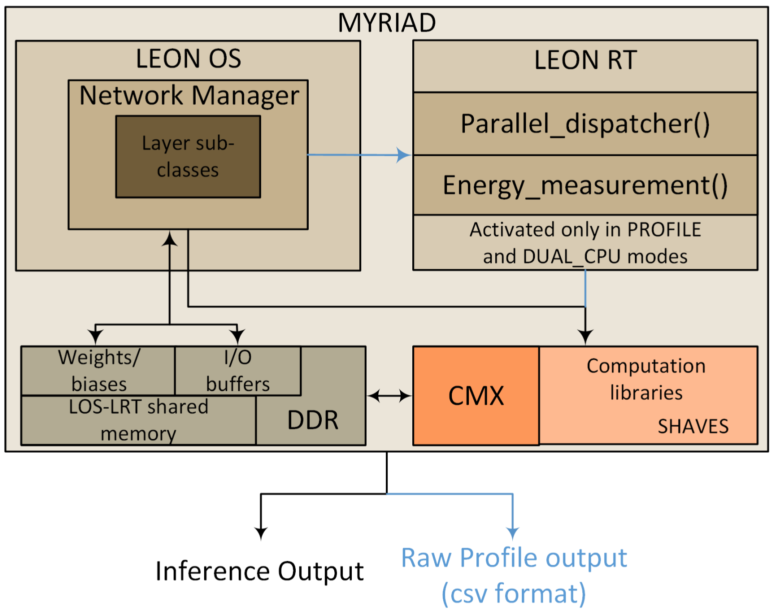


Figure 4.3: Abstract view of the Network Manager running on LEON processors

As shown in the figure above, the output of the application varies in respect to the execution mode selected by the user through the Interface. In ”profiling” mode the output is a file in csv format, which contains raw information about execution time and average power consumption, for each possible layer using each possible computational method and number of processing units. This csv file can be used by the user to obtain immediate information for specific layers or can be fed back to the Interface to conduct Design Space Exploration and produce optimal configurations and smart execution insights for that specific network. In all other modes, the application output is the network output maps, which consist the prediction of the network for a possible input image given.

4.4 Introduction to the Computational Engine

Finally, the essence of the CNN Engine is hidden behind its computational routines running on the VLIW processors, the shaves. Optimized DMA algorithms and heavy-duty routines written in Myriad Assembly are deployed, to provide the network inference of a specific image, or a set of images. The following ”Caffe” compatible layers are supported by the implementation:

- Convolution (Direct and Im2Col techniques)
- Pooling
- ReLU
- Fully Connected/Inner Product
- LRN-Across Channels
- Concat

In figure 4.4 the basic concept of the SHAVE configuration are shown:

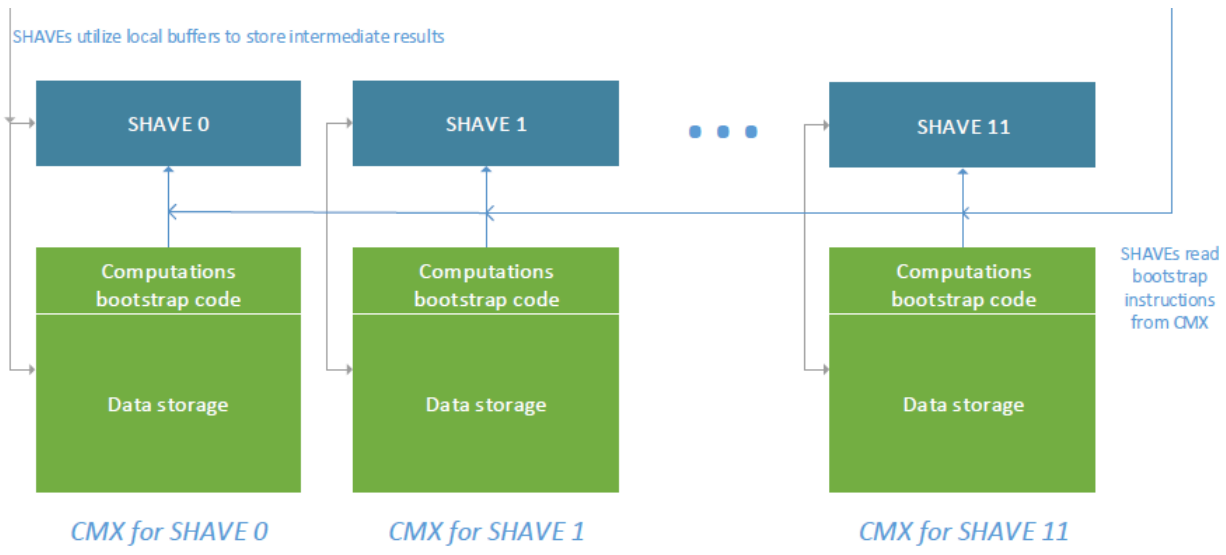

SHAVEs utilize local buffers to store intermediate results
SHAVE 0
SHAVE 1
SHAVE 11
Computations bootstrap code
Data storage
Computations bootstrap code
Data storage
Computations bootstrap code
Data storage
SHAVEs read bootstrap instructions from CMX
CMX for SHAVE 0
CMX for SHAVE 1
CMX for SHAVE 11

Figure 4.4: The Shave execution concept

The CMX memory used by the SHAVES to store data is tiny in comparison to the enormous amount of data they need to apply computation to. Therefore, the concept of the implementation is to bring chunks of the blobs to be calculated from the main memory (DDR) to the CMX in rounds. Moreover, the code segment of the SHAVE functions has been moved to DDR in order to spare valuable memory space inside the CMX for the blobs, increasing the capacity and reducing the loops needed for all the blocks to be transferred back and forth inside the DDR memory.

4.5 Evaluated Convolutional Neural Networks

4.5.1 Alexnet

Alexnet is a Deep Convolutional Neural Network [10] for image classification that won the ILSVRC-2012 competition and achieved a winning top-5 test error rate of 15.3%, compared to 26.2% achieved by the second-best entry. Alexnet has 8 layers. The first 5 are convolutional and the last 3 are fully connected layers. In between there are also some 'layers' called pooling and activation.

The highlights of this network are:

- Use of Relu instead of Tanh to add non-linearity, accelerating the speed by 6 times at the same accuracy
- Use of dropout instead of regularisation to deal with overfitting. However the training time is doubled with the dropout rate of 0.5

- Overlap pooling to reduce the size of network. It reduces the top-1 and top-5 error rates by 0.4% and 0.3%, repectively
- Local Response Normalization layer was firstly introduced in Alexnet
- ”Group” technique was implemented to reduce in half learnable parameters on memory greedy convolutional layers

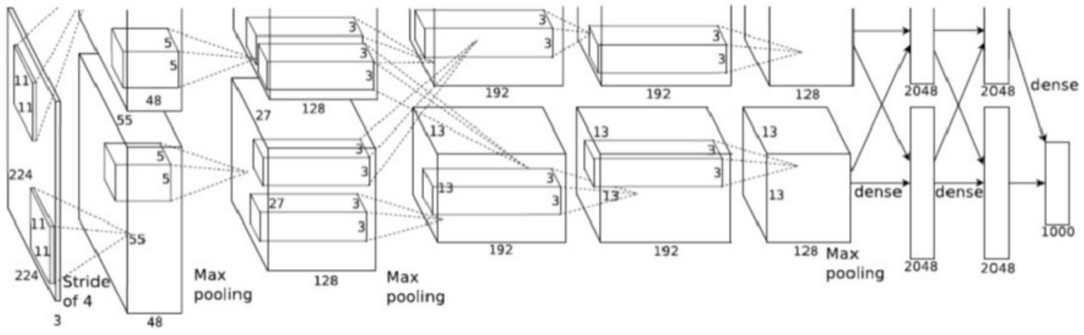


Figure 4.5: Alexnet architecture

The network has 62.3 million parameters, and needs 1.1 billion computation units in a forward pass. We can also see convolution layers, which accounts for 6% of all the parameters, consumes 95% of the computation. The output classifier of Alexnet has dimensions of 1000x1, receiving 3x227x227 images as input.

4.5.2 ZFNet

ZFNet is a CNN that won the ILSVRC 2013 and was implemented by Matthew Zeiler and Rob Fergus [11]. ZFNet is an It was an improvement on AlexNet by tweaking the architecture hyperparameters. More specifically, this mode was able to generalize well to other datasets except from Imagenet: when the softmax classifier is retrained, it convincingly beats the current state-of-the-art results on Caltech-101 and Caltech-256 datasets. The dimensions of the input image fed to ZFNet is 3x256x256, whereas the output classifier is 1000x1.

4.5.3 VGG

This architecture was created by VGG group, Oxford University [12]. It makes the improvement over AlexNet by replacing large kernel-sized filters (11 and 5 in the first and second convolutional layer, respectively) with multiple 3x3 kernel-sized filters one after another. With a given receptive field (the effective area size of input image on which output depends), multiple stacked smaller size kernel is better than the one with a larger size

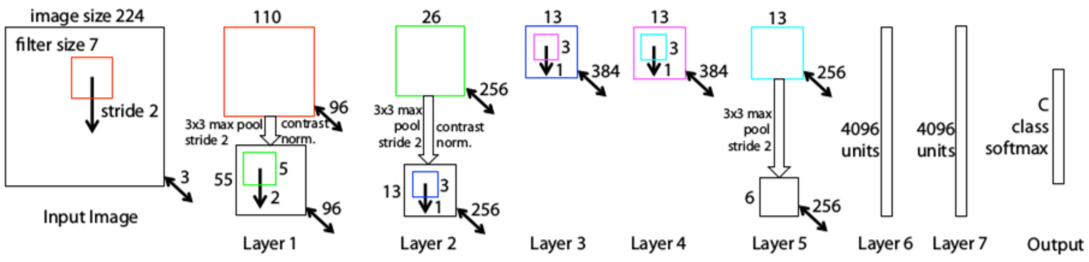
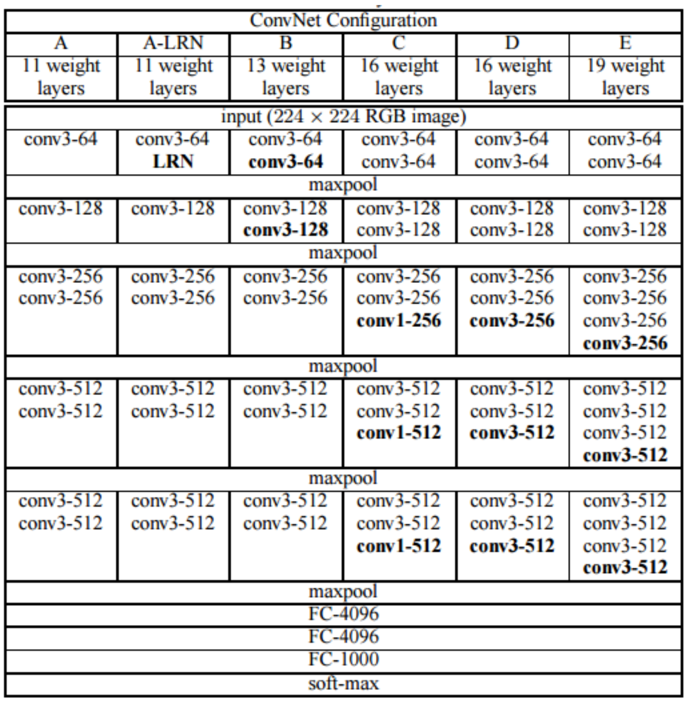


Figure 4.6: ZFNet layout

kernel because multiple non-linear layers increases the depth of the network which enables it to learn more complex features, and that too at a lower cost.

For example, three 3x3 filters on top of each other with stride 1 has a receptive size of 7, but the number of parameters involved is 3*9C^2 in comparison to 49C^2 parameters of kernels with a size of 7. Here, it is assumed that the number of input and output channel of layers is C.Also, 3x3 kernels help in retaining finer level properties of the image. The network architecture is given in figure 4.7.



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	conv3-64	conv3-64	conv3-64	conv3-64
	LRN	**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	conv3-256	conv3-256	conv3-256	conv3-256
			conv1-256	**conv3-256**	conv3-256
					conv3-256
maxpool					
conv3-512	conv3-512	conv3-512	conv3-512	conv3-512	conv3-512
conv3-512	conv3-512	conv3-512	conv3-512	conv3-512	conv3-512
			conv1-512	**conv3-512**	conv3-512
					conv3-512
maxpool					
conv3-512	conv3-512	conv3-512	conv3-512	conv3-512	conv3-512
conv3-512	conv3-512	conv3-512	conv3-512	conv3-512	conv3-512
			conv1-512	**conv3-512**	conv3-512
					conv3-512
maxpool					
FC-4096					
FC-4096					
FC-1000					
soft-max					

Figure 4.7: VGG architecture table

VGG is a memory intensive neural network as only the first out of three fully connected layers contain a weight array with dimensions 512x7x7x4096. The input image given to VGG is again 3x224x224 and the output classifier's dimension is 1000x1.

4.5.4 NiN-ImageNet

NiN (Network in Network) is a CNN implemented by Min Lin, Qiang Chen and Shuicheng Yan [13]. They proposed a novel deep network structure to enhance model discriminability for local patches within the receptive field. The conventional convolutional layer uses linear filters followed by a nonlinear activation function to scan the input. Instead, they built micro neural networks with more complex structures to abstract the data within the receptive field. They instantiated the micro neural network with a multilayer perceptron, which is a potent function approximator.

The feature maps are obtained by sliding the micro networks over the input in a similar manner as CNN; they are then fed into the next layer. Deep NIN can be implemented by stacking mutiple of the above described structure. With enhanced local modeling via the micro network, it is possible to utilize global average pooling over feature maps in the classification layer, which is easier to interpret and less prone to overfitting than traditional fully connected layers. They demonstrated the state-of-the-art classification performances with NIN on CIFAR-10 and CIFAR-100, and reasonable performances on SVHN and MNIST datasets.

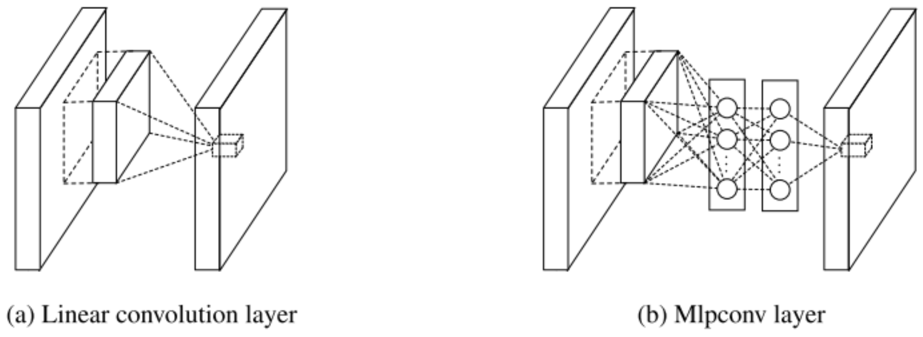


Figure 4.8: NiN convolutional layer concept

4.5.5 GoogLenet

While VGG achieves a phenomenal accuracy on ImageNet dataset, its deployment on even the most modest sized GPUs is a problem because of huge computational requirements, both in terms of memory and time. It becomes inefficient due to large width of convolutional layers.

For instance, a convolutional layer with 3X3 kernel size which takes 512 channels as input and outputs 512 channels, the order of calculations is 9x512x512.

In a convolutional operation at one location, every output channel (512 in the example above), is connected to every input channel, and this is called dense connection architecture. GoogLenet builds on the idea that most of the activations in deep network are either unnecessary (value of zero) or redundant because of correlations between them [14]. Therefore most efficient architecture of a deep network will have a sparse connection between the activations, which implies that all 512 output channels will not have a connection with all the 512 input channels. There are techniques to prune out such connections which would result in a sparse weight/connection. But kernels for sparse matrix multiplication are not optimized in BLAS or CuBlas(CUDA for GPU) packages which render them to be even slower than their dense counterparts.

So GoogLenet devised a module called inception module that approximates a sparse CNN with a normal dense construction(shown in the figure). Since only a small number of neurons are effective as mentioned earlier, width/number of the convolutional filters of a particular kernel size is kept small. Also, it uses convolutions of different sizes to capture details at varied scales(5x5, 3x3, 1x1).

Another salient point about the module is that it has a so-called bottleneck layer (1x1 convolutions in figure 4.9). It helps in massive reduction of the computation requirement as explained below.

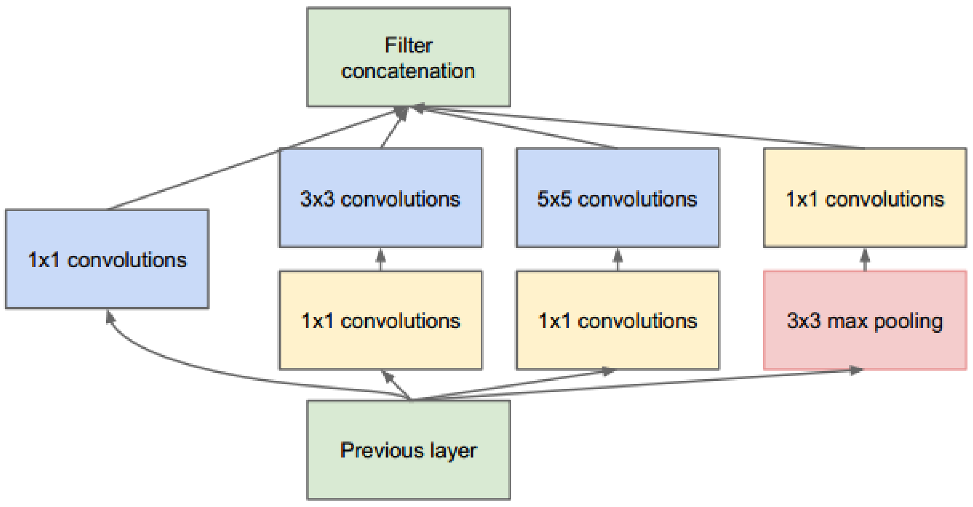


Figure 4.9: Architecture of GoogLenet's inception block

Let us take the first inception module of GoogLenet as an example which has 192 channels as input. It has just 128 filters of 3x3 kernel size and 32 filters of 5x5 size. The order of computation for 5x5 filters is 25x32x192 which can blow up as we go deeper into the network when the width of the network and the number of 5x5 filter further increases. In order to avoid this, the inception module uses 1x1 convolutions before applying larger sized kernels to reduce the dimension of the input channels, before feeding into those convolutions. So in first inception module, the input to the module is first fed into 1x1 convolutions with just 16 filters before it is fed into 5x5 convolutions. This reduces the computations to 16x192+25x32x16. All these changes allow the network to have a large width and depth.

Another change that GoogLenet made, was to replace the fully-connected layers at the end with a simple global average pooling which averages out the channel values across the 2D feature map, after the last convolutional layer. This drastically reduces the total number of parameters. This can be understood from Alexnet, where FC layers contain approx. 90% of parameters. Use of a large network width and depth allows GoogLenet to remove the Fully Connected layers without affecting the accuracy. It achieves 93.3% top-5 accuracy on ImageNet and is much faster than VGG.

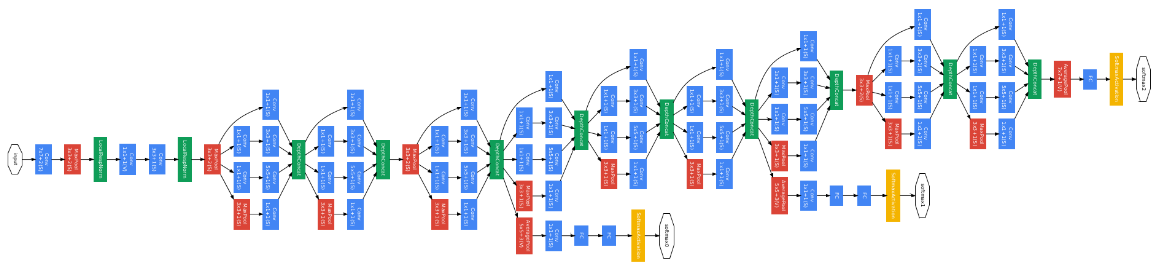


Figure 4.10: Overview of GoogLenet's structure

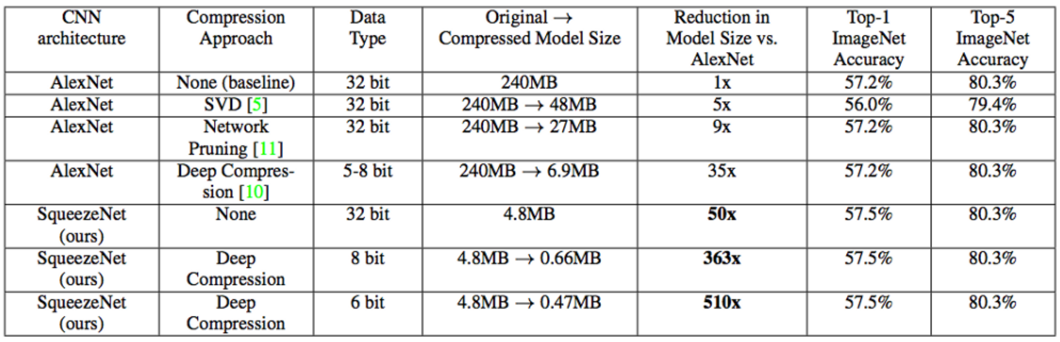
4.5.6 SqueezeNet

SqueezeNet is another Convolutional neural networks that was designed to reduce the number of learnable parameters in comparison to Alexnet, while keeping the accuracy in high standards [15]. The techniques deployed to the design of SqueezeNet might resemble up to a point GoogleNet's inception block, which was described above.

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.

Figure 4.12 shows the advantages of SqueezeNet against other famous CNN Architectures:



CNN architecture	Compression Approach	Data Type	Original → Compressed Model Size	Reduction in Model Size vs. AlexNet	Top-1 ImageNet Accuracy	Top-5 ImageNet Accuracy
AlexNet	None (baseline)	32 bit	240MB	1x	57.2%	80.3%
AlexNet	SVD [5]	32 bit	240MB → 48MB	5x	56.0%	79.4%
AlexNet	Network Pruning [11]	32 bit	240MB → 27MB	9x	57.2%	80.3%
AlexNet	Deep Compression [10]	5-8 bit	240MB → 6.9MB	35x	57.2%	80.3%
SqueezeNet (ours)	None	32 bit	4.8MB	**50x**	57.5%	80.3%
SqueezeNet (ours)	Deep Compression	8 bit	4.8MB → 0.66MB	**363x**	57.5%	80.3%
SqueezeNet (ours)	Deep Compression	6 bit	4.8MB → 0.47MB	**510x**	57.5%	80.3%

Figure 4.11: SqueezeNet benchmarking against other CNN architectures

The building brick of SqueezeNet is called fire module, which contains two layers: a squeeze layer and an expand layer. A SqueezeNet stacks a sum 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 latter increase it. The squeezing (bottoleneck layer) and expansion behavior is common in neural network architectures. Another common pattern is increasing depth while reducing feature map size to get high level abstract.

As shown in the above figure, 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 does not 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

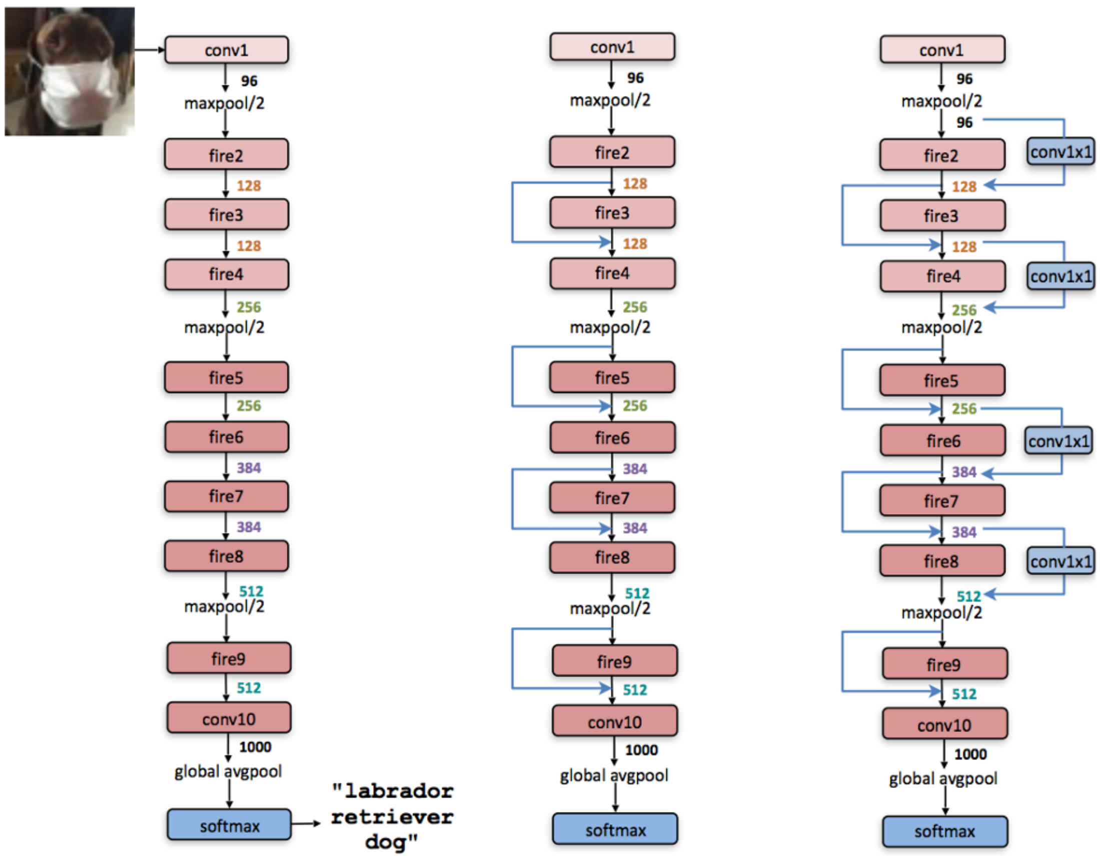


Figure 4.12: Macroarchitectural view from SqueezeNet architecture; Left: SqueezeNet; Middle: SqueezeNet with simple bypass; Right: SqueezeNet with complex bypass

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. By intuition, too much squeezing limits information flow; too few 3x3 filters limits space resolution. Finally, SqueezeNet's input image dimensions is 3x227x227 and output classifier's dimensions is 1000x1.

Chapter 5

Management and Configuration of CNNs

In this chapter the implementation of the Network Manager, including its interaction with the computational engine and the interface, will be carefully described. For brevity reasons, the implementation and any algorithms presented will be given as shortened pseudocodes in order to avoid unnecessary lines of code that have to do with the syntax specifics of C++.

5.1 Configuring the hardware for the application

This section is concerned about the proper way to configure the Myriad2 hardware as well as the Real Time Operating System (RTOS) that will be used as a manager for the memory and processor resources, like the heap size, the scheduling policy and the frequency of the system clocks. The operating system that will be used is RTEMS, which is specifically designed for embedded system applications.

Care must be taken, because there are multiple drivers involved in the process. The configuration process is complicated and any minor errors or deficiencies could result to increased chip power consumption, abated performance, unexpected memory behavior or errors initializing the chip components leading to stalls. Therefore the programmer is encouraged to delve into the details of the driver implementations for better understanding of the code provided by the MDK.

5.1.1 Setting up RTEMS

Time Executive for Multiprocessor Systems or RTEMS is an open source Real Time Operating System (RTOS) that supports open standard application programming interfaces (API) such as POSIX. It is used in space flight, medical, networking and many more embedded devices using processor architectures including ARM, PowerPC, Intel, Blackfin, MIPS, Microblaze and more.

RTEMS is designed for real-time, embedded systems and to support various open API standards including POSIX and μITRON. The API now known as the Classic RTEMS API was originally based upon the Real-Time Executive Interface Definition (RTEID) specification. It includes a port of the FreeBSD TCP/IP stack as well as support for various filesystems including NFS and the FAT filesystem.

In POSIX terminology, it implements a single process, multithreaded environment. This is reflected in the fact that RTEMS provides nearly all POSIX services other than those which are related to memory mapping, process forking, or shared memory. RTEMS closely corresponds to POSIX Profile 52 which is ”single process, threads, filesystem” [16].

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

```
MV_SOC_OS = rtems
RTEMS_BUILD_NAME = b_prebuilt
```

Listing 5.1: Makefile snippet

The next step is to write the code for setting up RTEMS itself, which is independent of the target platform (Myriad2 in this case). The schema proposed for accomplishing this involves the creation of a new directory leon/Configuration. Afterwards the code presented in source code 4.2 is required.

```
#include <rtems.h>
#include "rtems_config.h"

static void Fatal_extension(Internal_errors_Source the_source,
                            bool is_internal,
                            uint32_t the_error){
   switch (the_source){

      case RTEMS_FATAL_SOURCE_EXIT:
         if (the_error)
            printk("Exited with error code %lu\n", the_error);
               break;
         case RTEMS_FATAL_SOURCE_ASSERT:
            printk("%s : %d in %s \n",
                  ((rtems_assert_context * )the_error)->file,
                  ((rtems_assert_context * )the_error)->line,
                  ((rtems_assert_context * )the_error)->function);
            break;
         case RTEMS_FATAL_SOURCE_EXCEPTION:
            rtems_exception_frame_print((const rtems_exception_frame *)
   the_error);
               break;
         default:
            printk("\nSource %d Internal %d Error %lu  0x%lX:\n",
   the_source, is_internal, the_error, the_error);
```

```
            break;
        }
    return;
}
```

Listing 5.2: RTEMS configuration source file

This piece of code is mandatory and its form is suggested by the examples of the MDK. Its purpose is to define the behavior of RTEMS in case of failure. Such failure could be a driver malfunction or an unhandled hardware exception. In line 1 the main header file of RTEMS is included, which contains default configurations like the heap size or the thread scheduling policies.

Notice the include statement in line 2. This is an important line that defines a large part of the hardware configuration. The version of RTEMS shipped with the MDK comes with a board support package (BSP) that is capable of configuring Myriad2 with simple RTEMS directives. These directives are presented in the source code 4.3.

```
#ifndef LEON_RTEMS_CONFIG_H_
#define LEON_RTEMS_CONFIG_H_

#ifndef _RTEMS_CONFIG_H_
#define _RTEMS_CONFIG_H_
#include ”app_config.h”
#if defined(__RTEMS__)
#if !defined (__CONFIG__)
#define __CONFIG__
#define CLOCKS_NONE 0
#define CONFIGURE_INIT

#ifndef RTEMS_POSIX_API
#define RTEMS_POSIX_API
#endif

#define CONFIGURE_MICROSECONDS_PER_TICK              1000
#define CONFIGURE_TICKS_PER_TIMESLICE                10
#define CONFIGURE_APPLICATION_NEEDS_CONSOLE_DRIVER
#define CONFIGURE_APPLICATION_NEEDS_CLOCK_DRIVER
#define CONFIGURE_POSIX_INIT_THREAD_TABLE
#define CONFIGURE_MINIMUM_TASK_STACK_SIZE           (4*1024)
#define CONFIGURE_MAXIMUM_TASKS                      4
#define CONFIGURE_MAXIMUM_POSIX_THREADS              4
#define CONFIGURE_MAXIMUM_POSIX_MUTEXES              8
#define CONFIGURE_MAXIMUM_POSIX_KEYS                 8
#define CONFIGURE_MAXIMUM_POSIX_SEMAPHORES           8
#define CONFIGURE_MAXIMUM_SEMAPHORES                 4
```

```
29 #define CONFIGURE_MAXIMUM_POSIX_MESSAGE_QUEUES  8
30 #define CONFIGURE_MAXIMUM_POSIX_TIMERS          4
31 #define CONFIGURE_MAXIMUM_TIMERS                4
32 #define CONFIGURE_LIBIO_MAXIMUM_FILE_DESCRIPTORS  30
33 #define CONFIGURE_MAXIMUM_USER_EXTENSIONS     1
34 #define CONFIGURE_INITIAL_EXTENSIONS         { .fatal = Fatal_extension }
35 
36  static void Fatal_extension (
37    Internal_errors_Source  the_source,
38    bool                    is_internal,
39    uint32_t                the_error
40 );
41 void POSIX_Init (void *args);
42 
43 #include <rtems/confdefs.h>
44 
45 #endif // __CONFIG__
46 #endif // __RTEMS__
47 
48 BSP_SET_CLOCK(OSC_CLOCK_KHZ,  // Reference oscillator used
49         APP_CLOCK_KHZ,  // PLL0 Target Frequency
50         1,          // Master Divider Numerator
51         1,          // Master Divider Denominator
52         CSS_CLOCKS,    //CSS Clocks
53         MSS_CLOCKS,    // MSS Clocks
54         UPA_CLOCKS,    // UPA Clocks
55         CLOCKS_NONE,   // SIPP Clocks
56         CLOCKS_NONE    // AUX Clocks
57 );
58 BSP_SET_L2C_CONFIG( 1,           // Enable (1) / Disable (0)
59           L2C_REPL_LRU,      // Either L2C_REPL_LRU (default),
60                      //         L2C_REPL_PSEUDO_RANDOM,
61                      //         L2C_REPL_MASTER_INDEX_REP
62                      //      or L2C_REPL_MASTER_INDEX_MOD
63           0,            // Cache ways
64           L2C_MODE_COPY_BACK, // Either L2C_MODE_COPY_BACK
65                      //      or L2C_MODE_WRITE_TROUGH
66           0,            // Number of MTRR registers to program
67           NULL          // Array of MTRR configuration
68 );
69 #endif // _RTEMS_CONFIG_H_
70 #endif // LEON_RTEMS_CONFIG_H_
```

Listing 5.3: RTEMS configuration header file

The main feature of source code 4.3 is the configuration of the different clocks of the Myriad2 SoC. Lines 48-57 configure the various clocks. In particular, the

OSC_CLOCK_KHZ statement declares the main frequency that is commonly set to 480 or 600 MHz. The next three lines declare the frequency of the DDR RAM, which (the frequency) must be compatible with the main frequency. Currently the DDR is set up at the maximum possible frequency. The next lines enable or disable specific units inside the Myriad2 hardware.

In lines lines 58-68 configure the L2 cache of the Leon OS processor. The comments in the source code explain these lines in detail. However some reference to MTRR is needed. Memory Type Range Registers (MTRRs) are a set of processor supplementary capabilities control registers that provide system software with control of how accesses to memory ranges by the CPU are cached. It uses a set of programmable model-specific registers (MSRs) which are special registers provided by most modern CPUs.

The statements in these lines are actually macros, which are defined in the file app_-config.h included in line 6. Indeed this header file contains this information and is presented below

```
#define APP_CLOCK_KHZ       (600000)
#define OSC_CLOCK_KHZ       (12000)

#define CSS_CLOCKS (\
                    DEFAULT_CORE_CSS_DSS_CLOCKS       | \
                    DEV_CSS_GETH          | \
                    DEV_CSS_I2C0          )
#define UPA_CLOCKS (DEV_UPA_SH0         | \
                    DEV_UPA_SH1         | \
                    DEV_UPA_SH2         | \
                    DEV_UPA_SH3         | \
                    DEV_UPA_SH4         | \
                    DEV_UPA_SH5         | \
                    DEV_UPA_SH6         | \
                    DEV_UPA_SH7         | \
                    DEV_UPA_SH8         | \
                    DEV_UPA_SH9         | \
                    DEV_UPA_SH10        | \
                    DEV_UPA_SH11        | \
                    DEV_UPA_SHAVE_L2    | \
                    DEV_UPA_CDMA        | \
                    DEV_UPA_CTRL        )
#define MSS_CLOCKS     (DEV_MSS_APB_SLV       | \
                        DEV_MSS_APB2_CTRL     | \
                        DEV_MSS_RTBRIDGE      | \
                        DEV_MSS_RTAHB_CTRL    | \
                        DEV_MSS_LRT           | \
                        DEV_MSS_TIM           | \
                        DEV_MSS_LRT_DSU       | \
                        DEV_MSS_LRT_L2C       | \
                          DEV_MSS_LRT_ICB          |  \
                          DEV_MSS_AXI_BRIDGE       |  \
                          DEV_MSS_MXI_CTRL         )
```

Listing 5.4: app_config.h snippet

5.1.2 Setting up Myriad2 SoC

A very important configuration needed to be done on the Myriad2 chip is the setup of the Shaves Cache. The initialization of cache that is performed in source code 4.3 is only for the Leon OS processor. The CNN implementation also requires the initialization of the cache subsystem used by the SHAVE processors. For this reason, it provides a far more advanced software driver that is capable of separating the cache into several partitions. Source code 4.5 contains the necessary commands.

```
#define L2CACHE_CFG (SHAVEL2C_MODE_NORMAL)

int InitShaveL2C(void){

    s32 sc;
    sc = OsDrvShaveL2CacheInit(L2CACHE_CFG);
    if (sc) {
        puts("OsDrvShaveL2CacheInit failed");
        return sc;
    }

    sc = OsDrvShaveL2CResetPartitions();
    if (sc) {
        puts("OsDrvShaveL2CResetPartitions failed");
        return sc;
    }

    return OS_MYR_DRV_SUCCESS;
}
```

Listing 5.5: app_config.c shave cache initialization function

To understand the necessity of the source code 4.5, some clarification is required. The driver responsible for setting up the partitions works in the following manner: It keeps an internal structure that describes the partitioning schema. This structure is reset and then built as the programmer desires. Afterwards, the partition schema is instantiated into the hardware with another driver call that will be shown shortly. The macro L2CACHE_CFG configures the cache behavior. The available options are:

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

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

Finally, the actual set up of the partitions is shown in source code 4.6:

```
int ConfigShaveL2C(void){

    s32 sc;
    int last_part_id = -1;
    sc = OsDrvShaveL2CGetPartition(SHAVEPART128KB, &last_part_id);
    if (sc) {
        puts("OsDrvShaveL2CGetPartition failed");
        return sc;
    }
    for (int i = 1; i <= 6; i++) {
        sc = OsDrvShaveL2CGetPartition(SHAVEPART16KB, &last_part_id);
        if (sc) {
            puts("OsDrvShaveL2CGetPartition failed");
            return sc;
        }
    }
    for (int i = 0; i < 12; i++) {
        sc = OsDrvShaveL2CSetNonWindowedPartition(i, 0,
                                NON_WINDOWED_INSTRUCTIONS_PARTITION);
        if (sc) {
            puts("OsDrvShaveL2CSetNonWindowedPartition failed");
            return sc;
23          }
24      }
25      for (int i = 0, partId = 1; i < 6; i += 2, partId++) {
26          sc = OsDrvShaveL2CSetNonWindowedPartition(i, partId,
    NON_WINDOWED_DATA_PARTITION);
27          if (sc) {
28              puts("OsDrvShaveL2CSetNonWindowedPartition failed");
29              return sc;
30          }
31
32          sc = OsDrvShaveL2CSetNonWindowedPartition(i+1, partId,
    NON_WINDOWED_DATA_PARTITION);
33          if (sc) {
34              puts("OsDrvShaveL2CSetNonWindowedPartition failed");
35              return sc;
36          }
37      }
38      sc = OsDrvShaveL2CacheAllocateSetPartitions();
39      if (sc) {
40          puts("OsDrvShaveL2CacheAllocateSetPartitions failed");
41          return sc;
42      }
43      for (int i = 0; i <= last_part_id; i++) {
44          sc = OsDrvShaveL2CachePartitionInvalidate(i);
45          if (sc) {
46              puts("OsDrvShaveL2CachePartitionInvalidate failed");
47              return sc;
48          }
49      }
50      return OS_MYR_DRV_SUCCESS;
51  }
```

Listing 5.6: app_config.c snippet on shave cache partitions configuration

The source code 4.6 configures the L2 cache of the SHAVEs using 7 out of 8 partitions. The first partition is set to be 128KB, while the next 7 partitions are set to be 16KB each. This makes a total of 224KB of cache, while the total cache size is 256KB.

More precisely:

- Lines 17-22 assign the instruction port of each SHAVE to point to the first partition. This means that the first partition is going to be used as instruction cache and will be shared among all SHAVEs.

- Lines 25-37 assign the Load-Store Unit (LSU) (or data) port of each SHAVE to point to one of the six remaining partitions. In particular, every pair of consecutive SHAVEs will have their LSU port pointing at the same partition. SHAVE 0 and 1 will use the second partition, ..., SHAVE 10 and 11 with use the seventh partition.

- Lines 38-42 will instantiate the cache configuration defined in the previous lines. This means the internal data structure of the cache driver is stored to hardware registers.
- Finally, lines 43-49 invalidate the cache, in order to make sure there are no stale cache entries in the memory hierarchy of the SHAVEs

Another very important configuration needed to be done on the chip is the setup of Myriad2 Power islands on the system. There are 20 power islands in the Myriad2 chip. Power islands can be turned off to save dynamic and leakage power if not in use. For a CNN implementation this is a very handy feature and can be exploited to lower the power required to process an input image. Several operations are I/O bounded, meaning that there is not gain in parallelizing their execution across all the 12 SHAVEs. In such case, turning power islands off can be only a benefit. Source code 4.7 shows that the Myriad2 turns all SHAVEs off during the initialization process.

```
int InitClocksAndMemory(void){

    u32 sc;
    tyAuxClkDividerCfg appAuxClkCfg[] = {{AUX_CLK_MASK_UART,
   CLK_SRC_REFCLK0, 96, 625}, {0, 0, 0, 0},};

    sc = OsDrvCprInit();
    if (sc) {
        puts("OsDrvCprInit failed");
        return sc;
    }
    sc = OsDrvCprOpen();
    if (sc) {
        puts("OsDrvCprOpen failed");
        return sc;
    }
    sc = OsDrvCprAuxClockArrayConfig(appAuxClkCfg);
    if (sc) {
        puts("OsDrvCprAuxClockArrayConfig failed");
        return sc;
    }
    DrvDdrInitialise(NULL);

    sc = OsDrvCprSysDeviceAction(UPA_DOMAIN, DEASSERT_RESET, UPA_CLOCKS);
    if (sc) {
        puts("OsDrvCprSysDeviceAction failed");
        return sc;
27     }
28     sc = OsDrvCprSysDeviceAction(MSS_DOMAIN, DEASSERT_RESET, MSS_CLOCKS);
29     if (sc) {
30         puts("OsDrvCprSysDeviceAction failed");
31         return sc;
32     }
33     sc = OsDrvCprTurnOffShaveMask(-1);
34     if (sc) {
35         puts("OsDrvCprTurnOffShaveMask failed");
36         return sc;
37     }
38     OsDrvCprPowerTurnOffIsland(POWER_ISLAND_MSS_CPU);
39     OsDrvCprPowerTurnOffIsland(POWER_ISLAND_MSS_AMC);
40     OsDrvCprPowerTurnOffIsland(POWER_ISLAND_MSS_SIPP);
41     sc = OsDrvCprPowerTurnOffIsland(POWER_ISLAND_USB);
42     if (sc) {
43         puts("OsDrvCprPowerTurnOffIsland failed");
44         return sc;
45     }
46     return OS_MYR_DRV_SUCCESS;
47 }
```

Listing 5.7: app_config.c snippet regarding power management

Some explanations for source code 4.7 are:

- Line 6 initializes the Clock-Power-Reset (CPR) driver that controls the power islands of the Myriad2 chip.
- Line 21 initializes the DDR. The purpose of this line is to reset the DDR content. This helps during development time, since values from previous runs are not preserved, assisting the programmer to track bugs
- Finally, some power islands are turned off. More precisely, all the SHAVEs and the USB unit are turned off. However, there is a caveat. In order for the power islands to work properly, it is important to enable all required power islands beforehand. For example, since all the SHAVEs are required for the CNN implementation, the source code 4.3 in conjunction with the snippet 4.4 enables all SHAVEs in advance. However, afterwards, the CPR drivers turns them off, making it possible to turn them back on when necessary. If the programmer tried to enable the SHAVEs though the CPR driver without configuring them using the RTEMS BSP routine, an error would occur.

5.2 Configuring the dynamic memory map

In order to create the memory map described previously, the GNU Linker needs to be used [17]. The linker script only describes how each slice of CMX memory is split for data and code and defines the regions of memory used by Leon OS and Leon RT.

```
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  LENGTH = 2M



  DDR_DATA (wx) : ORIGIN = 0x80000000 + 4M, LENGTH = 500M

}

INCLUDE myriad2_leon_default_elf.ldscript
INCLUDE myriad2_shave_slices.ldscript
INCLUDE myriad2_default_general_purpose_sections.ldscript
```

Listing 5.8: Application memory map

With this script the following arrangement is made:

- 128KB of CMX are used explicitly by the DMA Engine.
- 128KB of CMX are used for other purposes. In particular, this space consists the uncached area of CMX and will be used for placing shared parameters used by the SHAVEs
- 2MB of DDR are used by the Leon OS and the RTEMS operating system to be handled accordingly.
- 2MB of DDR are used by the Leon RT for its code, stack segment and dynamic memory.
- Finally, 500MB, which are the vast majority of the DDR, are used for placing parameters/weights of the CNN.

5.3 How layers are described

In this section there will be a step by step explanation of how layers are implemented inside the Engine by LEON OS and what characteristics each type of layer must have for the application to process the information correctly.

5.3.1 The Layer Class definition

First of all, all types of computational layers have some common characteristics and behavior as their purpose in terms of network architecture is more or less the same. Therefore, the most efficient design that could be used for their implementation was the usage of a parent class holding all the common characteristics and a set of subclasses for each distinct functionality and differentiation on class members and methods. C++ is ideal for similar software designs and that is one of the reasons that this language was chosen for the implementation of LEON manager [18][19]. Finally, holding different layers in different subclasses under a parent class provides a major advantage: It is possible to store different layers in the same container (e.g. an array), whose type can be the type of the parent class. In that way, polymorphism is fully exploited and C++ provides significant performance benefits over the use of C.

The following source code provides the code used to implement the abstract layer types:

```
class Layers{
public:
  inline Layers(){}
  virtual ~Layers(){}
  virtual u64 execute(u8 *bottom_output_buffer, u16 &bottom_channels,
              u16 &bottom_input_height, u16 &bottom_input_width);

protected:
  ...
};

class Input : public Layers{
public:

  inline Input(Input arguments);
  ~Input();
  u64 execute(u8 *bottom_output_buffer, u16 &bottom_channels,
              u16 &bottom_input_height, u16 &bottom_input_width);
};

class Convolution : public Layers{
public:

  inline Convolution(Convolution arguments);
  ~Convolution();
  u64 execute(u8 *bottom_output_buffer, u16 &bottom_channels,
              u16 &bottom_input_height, u16 &bottom_input_width);

private:
  ...
};

class Pooling : public Layers{
public:
  inline Pooling(Pooling arguments);
  ~Pooling()
  u64 execute(u8 *bottom_output_buffer, u16 &bottom_channels,
              u16 &bottom_input_height, u16 &bottom_input_width);

private:
  ...
};

class InnerProduct : public Layers{
public:

  inline InnerProduct(Inner product arguments);
  ~InnerProduct(){}
  u64 execute(u8 *bottom_output_buffer, u16 &bottom_channels,
              u16 &bottom_input_height, u16 &bottom_input_width);

private:
  ...
};

class Lrn : public Layers{
public:

  inline Lrn(LRN arguments);
  ~Lrn();
  u64 execute(u8 *bottom_output_buffer, u16 &bottom_channels,
              u16 &bottom_input_height, u16 &bottom_input_width) override
    ;

private:
  ...
};

class Concat : public Layers{
```

```
public:

  inline Concat(Concat arguments);
  ~Concat() {}
  u64 execute(u8 *bottom_output_buffer, u16 &bottom_channels,
                   u16 &bottom_input_height, u16 &bottom_input_width);
};
```

Listing 5.9: Layer data types inside LEON OS and LEON RT

The code presented above provides an overview of the Class implementation. It should be noted that these classes are mainly used by LEON OS when creating and executing the network, but in "PROFILE" and "DUAL_PROCESSOR" modes they are used by LEON RT as well. Figure 5.1 offers an intuitive view of the implementation.

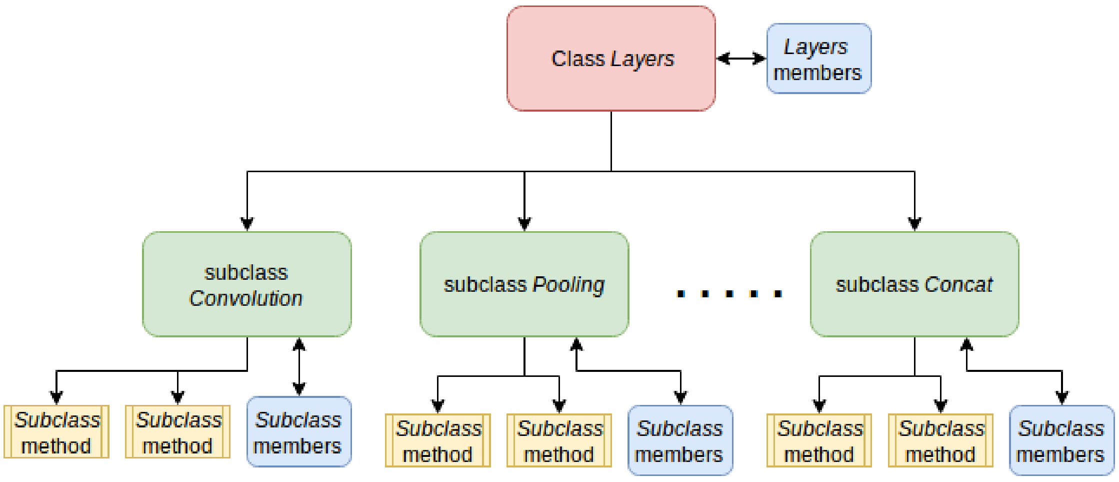


Figure 5.1: Overview of the Class Layers implementation

As shown above, apart from the constructors and destructors, the basic functionality of layers is to call a method called "execute". When this method is called from the caller, then the parent class overrides to the appropriate "execute" method, in respect to the type of layer. The execution methods in all layers, use their private members as well as three arguments passed by reference, which deal with information about the previous layer that is used as an input to the current layer. The execution method will be explained just below.

Before that, the private members, which are identical to the respective constructor arguments will be presented and explained below:

Class "Layer" parameters:

- Output buffer: A pointer holding the address in which the layer calculation output will be stored

- DDR function: An unsigned integer that represents the function that will be used from the computational library inside the SHAVES
- Shaves used: How many shaves will be used for the computation of this specific layer
- Bottom node: An unsigned integer indicating which node from the network array is the parent layer of the current layer executed
- input height, input width and channels: Dimensions of the input blob

These parameters are protected members of the parent class because they are common among all different subclasses and therefore it is more efficient to store them in the parent class. They are declared as protected, because private members of a parent class are not inherited to the child classes and declaring them public, would be a design mistake, as hiding class parameters ensures application abstractness.

Following up, the subclass-specific parameters are presented. Input and Concat layers are not mentioned as they are a special kind of layers: Input is used only as a reference to the starting point point of the network and its purpose is to provide to the next layer the appropriate dimensions of the blob. Therefore, the only arguments needed to its constructor is three integers regarding the blob dimensions and a pointer to its input buffer containing the input image. These members as explained are provided by the parent class and therefore Input's private member field is empty. Concat informs the following nodes of the network that a block with layers in parallel has finished and concatenates their blobs. However, this is done offline through the interface using a special algorithm which will be presented in following chapters.

Subclass ”Convolution” parameters:

- Weight pointer: Pointing to the address of the weights of this layer
- Bias pointer: Pointing to the address of the biases of this layer
- Kernel size, Stride, Pad and Group: Special convolutional hyper-parameters provided by the network designer
- ReLU flag: An unsigned integer providing information about whether an inline ReLU will be performed after the computation

Subclass ”Pooling” parameters:

- Pooling method: Variable indicating about the type of pooling performed: MAX, Average or Stochastic

- Kernel size, Stride and Pad: Pooling layer hyperparameters provided by the network designer

Subclass ”InnerProduct” parameters:

- Weight pointer: Pointing to the address of the weights of this layer
- Bias pointer: Pointing to the address of the biases of this layer
- ReLU flag: An unsigned integer providing information about whether an inline ReLU will be performed after the computation

Subclass ”LRN” parameters:

- Local size: Integer indicating the vertical kernel of the computation
- Alpha and Beta: Hyperparameters provided by the designer

5.3.2 The Layer Class execution method

A pseudocode of the ”execution” method of the layers is presented below. Note that it is not needed to present all execution methods of the layers, as they are using different data types and number of variables for their initialization, but they all follow the same basic principle: To initialize the SHAVES to perform the computation.

```

uint_64t Subclass::execute(
                  u8 *bottom_output_buffer, u16 &bottom_channels,
                  u16 &bottom_input_height, u16 &bottom_input_width){

  initialize variables(hyperparameters, pointers and buffers)
  place them in appropriate structs inside CMX Uncached area
  Initialize SHAVE masks
  Initialize timers
  Divide output maps to be computed by the number of SHAVES

  for(number of shaves used){

    Reset SHAVES and set the stack
    Start the SHAVES
  }
  Wait the SHAVES to finish execution
  Stop the SHAVES
  Turn of SHAVES mask to preserve energy

  return cpu_cycles;
}
```

Listing 5.10: Layer execution method algorithm

5.4 The network architecture inside Myriad2

In this section the format of the information inside Myriad2 about the network architecture will be presented. As explained in the previous chapter, the interface is fully automated and generalized: It can read Caffe compatible prototxt files and convert them to source code readable by the LEON OS structures. In this section, all source codes that consist the neural network in appropriate format, for the application, will be explained.

There are four types of information the Network manager needs to obtain in order to create and dispatch the network to the SHAVES:

- The number and type of layers that consist the Neural Network
- The way they are interconnected (Linear network, recursive network or network with parallel blocks)
- The layer specifics/hyper-parameters
- The weights and biases of Convolutional and Fully Connected filters which occured after training the network on a dataset

All this information is computed by the Python interface and provided in source code format to be compiled inside Myriad. There are five files that are produced by this process:

- ”network.cpp”: Contains a C++ method which creates a dynamic vector and pushes back layer objects one by one. Their parameters are computed by the interface using specific algorithms on the prototxt and caffemodel files
- ”network.h”: Header file for ”network.cpp”
- ”weight_data.c”: Source file containing static pre-initialized arrays containing large amounts of data regarding the weights and biases of the Neural Network
- ”weight_data.h”: Header file of ”weight_data.c”
- ”network_defines.h”: Contains conditional flags and architecture insights of the network. This file is part of the optimization process done on the manager and will be explained later in this chapter

5.4.1 The network dynamic array

The main idea behind the implementation of the LEON manager was to store all layers inside a linear array in order to efficiently iterate over each layer to execute the network. ”network.cpp” is the auto-generated code that makes this possible and is shown below:

```
create_network(){

    network.store(Input((u8*)(&data_input), 3, 224, 224));

    network.store(Convolution((u8*)(&branch_output_buffer_0_0), 0, 10, (
   u8*)(&conv1_7x7_s2_weights), (u8*)(&conv1_7x7_s2_biases), 64, 112,
   112, 7, 2, 3, 1, 12, 1));

    network.store(Pooling((u8*)(&branch_output_buffer_0_1), 1, 10, 64,
   56, 56, 3, 2, 0, 33, pooling_MAX));

    network.store(Lrn((u8*)(&branch_output_buffer_0_0), 2, 10));
    .
    .
    .
    network.store(InnerProduct((u8*)(&branch_output_buffer_0_1), 82, 10,
   (u8*)(&loss3_classifier_weights), (u8*)(&loss3_classifier_biases),
   1000, 0));

    return network;
}
```

Listing 5.11: network.cpp file for GoogleNet

This method presented above is over-simplified because providing too many technical details regarding the C++ syntax would make the understanding of how this file works too complex. The logic of this method is simple: A dynamic array resembling the neural network is initialized. Then, each layer, one by one, are placed dynamically at the end of the array. The method, then, returns the network array to the caller instance.

5.4.2 The weights and biases of the network

As already explained, ”weigh_data.c” and ”weight_data.h” contain all appropriate information about the weight and bias arrrays of the Neural Network as shown below:

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

fp16 DDR_BUFFER input_data[1*1*50*50] = {
15636, 15521, 15681, . . .
};
fp16 DDR_BUFFER conv1_7x7_s2_weights[32*1*5*5] = {
13040, 13579, 13540, . . .
};
fp16 DDR_BUFFER conv1_7x7_s2_biases[32] = {
47182, 45891, 47157, . . .
};
fp16 DDR_BUFFER inception_3a_3x3_reduce_weights[200*1152] = {
11484, 40758, 9013, . . .
};
fp16 DDR_BUFFER inception_3a_3x3_reduce_biases[200] = {
43883, 39963 , 43263, . . .
};
```

Listing 5.12: Snippet from weight_data.c

The section .ddr_direct.data is placed at the region DDR_DATA by the MDK build system. Also, the access to the data is un cached, making sure no caches are used without explicitly stated by the programmer.

It is reiterated that the source code presented is compiled by the Leon compiler. As a result, the fp16 data type is actually an alias for uint16_t. That is why the data defined by the arrays are integer numbers. These integers are in fact the binary representation of 16-bit floating point numbers

5.5 The Network Manager

It is easily understood at this point, that the concepts described in the previous sections (configuration, memory map and layer class definition) are the underlying core of the LEON Framework. Though the structures explained are more than enough to dispatch the network to the SHAVES and execute it, a manager running on a higher level of abstraction was necessary, in order to render the LEON framework versatile on different execution modes and also more abstract and user friendly.

For the reasons explained above, another class called "Network_manager" was created. This class calls the "create_network" function through its constructor and holds the network array in its private members. It is responsible for calling each layer's constructors and destructors on creation and free of the network.

The class definition of the manager is presented below:

```
class Network_Manager{
public:
  inline Network_Manager();

  ~Network_Manager();

  void execute();
  void network_output();

  #ifdef PROFILE
  void shave_profile();
  void profile_output();
  #endif

private:
  u64 network_cycles = 0;
  #ifdef PROFILE
    double *power_consumption[12];
    u64 *cpu_cycles[12];
  #endif
};
```

Listing 5.13: The Network Manager class

The manager has two basic functionalities: Execution of the network (which actually means providing inference on an input image) and profiling. As presented, the choice of whether execution or profiling mode will be deployed, depends on the value of a conditional flag. This conditional flag, is declared inside the Makefile and is tweaked in respect to the user input arguments provided to the Python Interface.

A brief description of the class definition of Network Manager:

- The constructor initializes the network by calling the ”create_network” method. If PROFILE mode is chosen, then ”cpu_cycles” and ”power_consumption” arrays are initialized appropriately to store layer information occured from the profiling process
- The destructor is responsible for destructing each layer explicitely, as well as any pointers and dynamic arrays allocated to the heap, in order to ensure correct memory management
- ”execute” method iterates over all elements of network array, calling their virtual ”execute” method
- ”network_output” method outputs the results from the execution of the Neural Network on a specific image
- ”Shave_profile” executes the layers one-by-one for all available configurations, measures execution time and average power consumption and stores this information on the respective arrays
- ”profile_output” outputs the profile information and measurements and dumps a CSV file in readable format.

5.6 The Inference mode

In this section the algorithms that are responsible for executing the network and providing the inference to the output will be explained. There are two sub-modes in Inference mode: ”Single Processor” inference and ”Dual Processor” inference. The latter is concerned with the usage of both LEON OS and LEON RT dispatching layers simultaneously in the SHAVE array.

5.6.1 The Single Processor mode

The former CNN implementation, as well as the implementation demonstrated by Movidius, Intel followed the principle of using LEON OS processor to dispatch layers to the SHAVE array in a linear way: A layer is prepared, pre-processed and sent to the SHAVES for computation. The SHAVES return the output calculations to its output buffer, which will be fed as an input to the next layer of the network. This basic way is called in this CNN implementation ”Single Processor” execution mode.

The algorithm used in this mode is very simple and is provided below:

```
#if LINEAR
  for (each layer in network_map){
    network_cycles+=layer->execute((layer-1)->output_buffer,
                    (layer-1)->channels,
                    (layer-1)-input_height
                    (layer-1)->input_width)
  }
#else
  for (each layer in network_map){
    network_cycles+=layer->execute((layer->bottom_node)->output_buffer,
                    (layer->bottom_node)->channels,
                    (layer->bottom_node)-input_height
                    (layer->bottom_node)->input_width)
  }
#endif
```

Listing 5.14: Single Processor inference implementation

To begin with, network_cycles increments in each iteration, adding the execution cycles needed for each layer. Thus, the total execution time is computed in the end of the execution. Second, there are two conditional branches with similar for loops. In cases where a network is linear, then by default layers are organized inside the network array in successive order. But that is not the case with Neural networks with complex architecture, containing layers in parallel (e.g. GoogleNet).

The CNN implementation is fully generalized on any type of layers and this conditional flag takes care of both consistency and performance: If a network is linear then only the first iteration is chosen by the compiler on compile-time and at the same time it is faster iterating only over layers than searching for the ”bottom_node” layer member in each iteration. The second ”for” loop, which is compiled only when LINEAR flag is set to false, could also execute a linear network, at the minor performance drawback of searching the bottom node of each layer instead of selecting the previously iterated layer.

5.6.2 The Dual Processor mode

Throughout the processing of this thesis, a whole new idea was born about dispatching layers to the SHAVES, regarding networks with layers organized in parallel blocks. Since each parallel branch is independent from the others, a semi-stochastic algorithm was designed and implemented to run on the interface and explore possible configurations, using both LEON processors as managers dispatching layers simultaneously to the SHAVES by allocating complementary number of SHAVES, that could provide better performance than using maximum number of SHAVES and dispatching layers in a linear manner. This algorithm proved to be a major success of this thesis and will be thoroughly explained in a following chapter. This subsection is concerned about the way that, given that parallel configurations have been explored, how can they be deployed on Myriad and how both processors are co-ordinated.

This implementation relies on the use of custom semaphores. The only way to establish communication between these two processors in Myriad2 is through the declaration of a shared memory space. Therefore, standard semaphores or POSIX locks cannot be used. In this implementation, two variables, one for each processors, are exploited as multi-state semaphores.

In figure 5.2 a full flowchart of the algorithm is presented. Flowchart was chosen instead of providing code because it is more clear and easy to understand how the synchronization of the two processors is achieved.

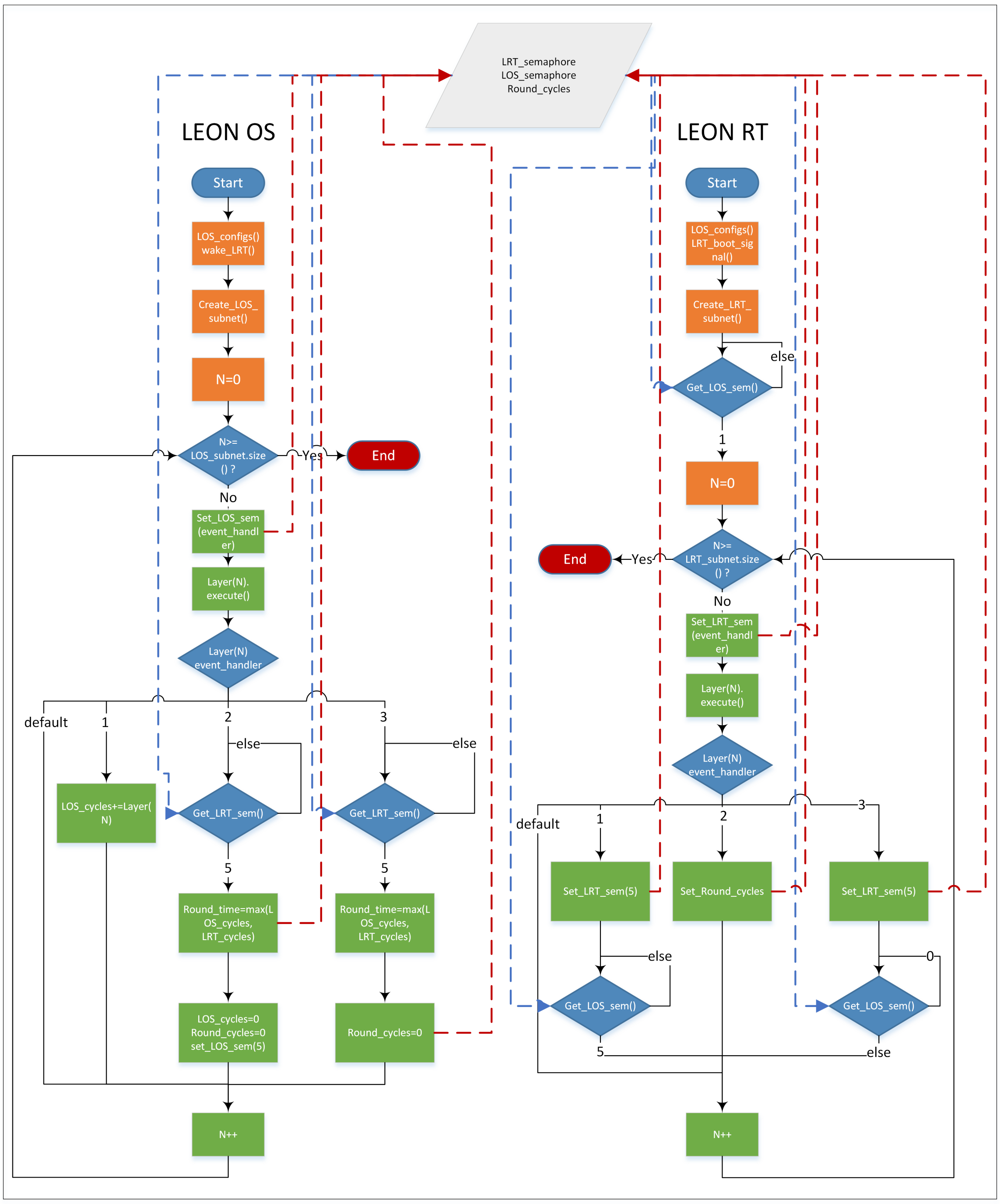


Figure 5.2: Dual processor mode algorithm flowchart

5.7 The Profiling mode

The most important part of this Thesis was to create a framework able to conduct profiling on Convolutional Neural Networks and perform Design Space Exploration on them, taking into consideration metrics like execution time and energy consumption. The purpose of this, would be to give information to embedded architecture designers or users using CNNs for inference applications, about the optimal configurations, in terms of number of processing units or way of executing layers, for each network. This is of major importance because state-of-the-art applications, using Neural networks, need to be as efficient as possible in order to be deployed to edge devices, like drones, smartphones and portable computers in order to save time and energy.

In this section a part of the algorithm which was designed to perform that operation will be described.

Profiling mode consists of two parts: The Myriad2 part and the Python interface part. The network manager deployed in the LEON OS framework is responsible for dispatching the layers to the SHAVES and extracting information about the execution time and average power consumption for each layer and possible configuration available. Then, a CSV file containing all this information, in raw format, is exposed by the manager to the Python interface. The interface uses this information and applies a set of algorithms to extract valuable information about the optimization of the network.

The following steps are being made in order to trigger the Profiling mode: The user provides the appropriate argument regarding this mode. Then, the Python Interface generates the Neural Network libraries and moves them inside the application directories. Then, the Interface compiles the application setting as true a conditional flag inside the Makefile, resulting in the compilation of a different branch of code of the Network Manager. The same conditional flag inside the LEON OS code, is used as a condition to call the appropriate methods that will commence the profiling of the network.

The manager calls the profiling method. Firstly, after the network is created, the manager iterates over each distinct layer and dispatches it for execution for all possible number of SHAVES, storing the execution time on an appropriate array. For the power consumption measurement, the process needed to be done is more complex. Myriad2 power measurement API exploits both LEON processors in the following manner: LEON OS commands LEON RT to execute a piece of instructions or activate the SHAVES, and at the same time LEON OS probes all power sampling rails of the board in order to measure the momentary power consumption. LEON OS samples all rails continuously, until a kill signal is received by LEON RT, having reached the end of execution. Then the sampling terminates and the average is calculated. This process is done for every SHAVE number combination of each layer and is illustrated in the figure 5.3 below.

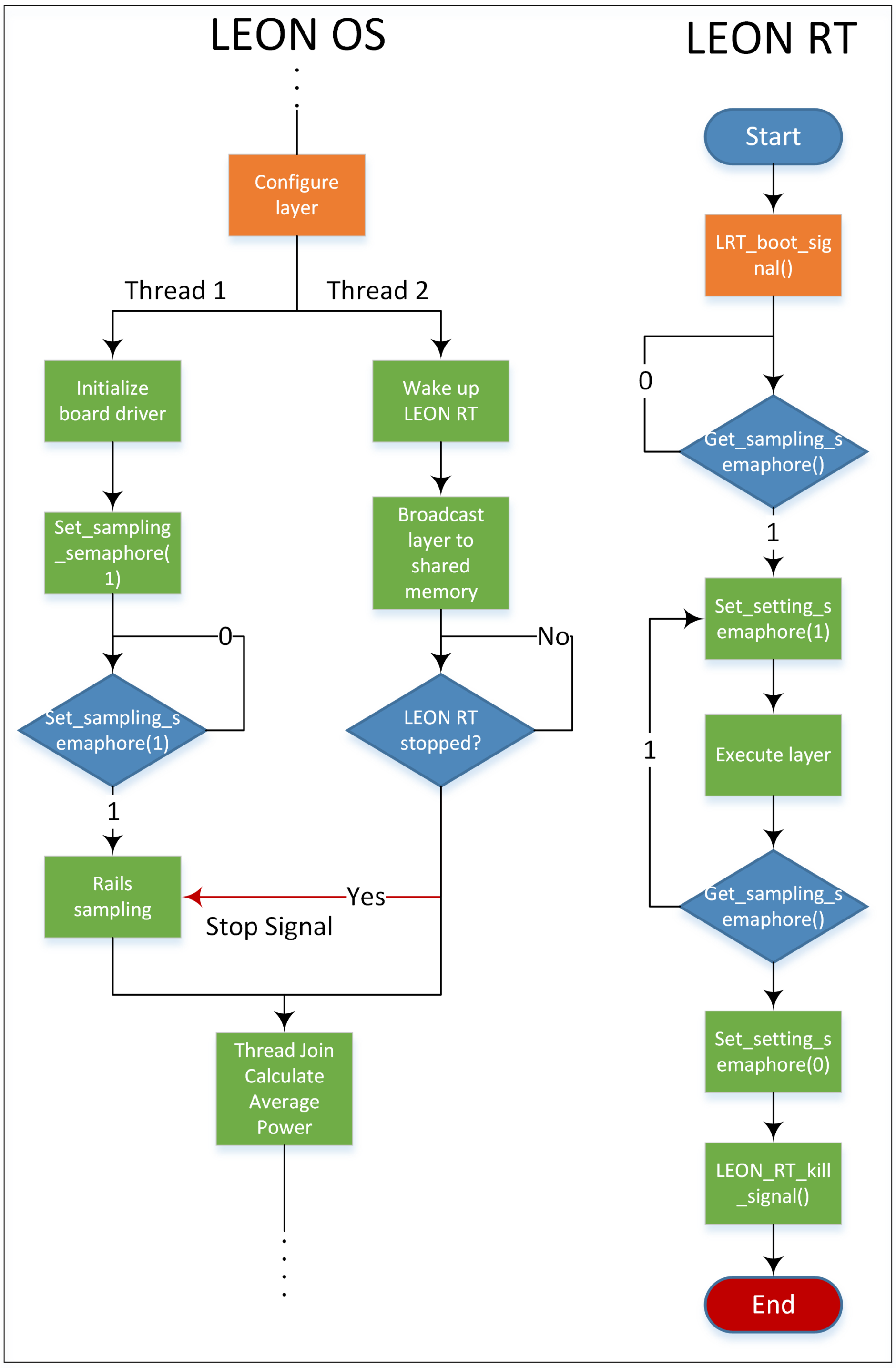


Figure 5.3: Profile mode algorithm flowchart

In tables 5.1 and 5.2 a sample output from profiling mode for Alexnet, regarding execution time and power consumption respectively, is provided.

Table 5.1: Alexnet profile output for execution time

ID-Type SHAVES	1	2	3	4	5	6	7	8	9	10	11	12
1-Convolution	188.347283	94.574138	69.006598	55.942502	48.558965	44.488375	40.832002	39.936248	37.654518	36.793235	35.816438	35.826485
2-LRN	7.770548	3.926377	2.667983	2.032187	1.680010	1.419407	1.268830	1.132187	1.039367	0.941543	0.871443	0.803340
3-Pooling	0.656877	0.387178	0.304605	0.239732	0.230412	0.223545	0.222852	0.218522	0.228772	0.222322	0.217302	0.218958
4-Convolution	86.144493	43.115018	28.989892	21.588535	17.552892	14.552535	12.548358	10.870782	9.842598	8.864388	8.164575	7.509935
5-LRN	5.105547	2.595163	1.792137	1.367487	1.140257	0.986750	0.835623	0.754367	0.707160	0.647047	0.597750	0.565680
6-Pooling	0.870422	0.437082	0.308862	0.258545	0.226432	0.203228	0.200292	0.189282	0.182745	0.192288	0.186782	0.180735
7-Convolution	122.607587	61.673092	41.913468	31.324632	25.370362	21.583155	18.701352	16.433805	15.065072	13.672425	12.568668	12.045382
8-Convolution	92.036237	46.312138	31.147538	23.532965	19.155398	16.077522	14.032698	12.361908	11.233668	10.263815	9.457838	8.926478
9-Convolution	61.539147	30.952795	20.932415	15.727958	12.836808	10.818032	9.426592	8.359202	7.486345	6.949828	6.347965	6.022222
10-Pooling	0.654788	0.329415	0.223862	0.169352	0.149875	0.132282	0.124925	0.117258	0.116332	0.117715	0.117252	0.117382
11-InnerProduct	45.983922	24.108270	19.128707	20.132467	19.137397	19.285393	19.540033	20.018530	19.179573	19.424837	19.263687	19.196363
12-InnerProduct	21.944573	11.309963	8.408787	9.094047	8.846503	8.477237	8.773227	9.065593	8.751287	8.838767	8.813887	8.567057
13-InnerProduct	5.365720	2.770780	2.062973	2.239040	2.303140	2.071863	2.179723	2.116227	2.128257	2.199463	2.130760	2.118223

Table 5.2: Alexnet profile output for average power consumption

ID-Type SHAVES	1	2	3	4	5	6	7	8	9	10	11	12
1-Convolution	570.29	642.13	706.15	759.92	811.32	851.95	895.64	915.56	953.67	985.08	1016.40	1040.79
2-LRN	570.93	634.48	696.64	757.94	817.56	880.48	940.94	991.2	1043.35	1092.72	1145.01	1198.91
3-Pooling	700.27	841.62	951.13	1064.93	1129.55	1177.7	1207.82	1260.93	1292.87	1323.68	1372.95	1419.58
4-Convolution	598.02	676.95	753.16	834.8	911.22	996.55	1070.51	1157.89	1223.63	1307.7	1369.61	1449.61
5-LRN	581.7	648.3	717.81	782.82	843.55	905.07	955.63	1016.75	1075.66	1136.29	1193.38	1242.35
6-Pooling	658.67	796.24	924.27	1017.97	1110.58	1193.12	1229.34	1291.07	1353.71	1348.51	1405.81	1465.43
7-Convolution	609.82	699.69	791.38	880.89	969.25	1056.26	1144.47	1221.65	1305.76	1383.21	1447.39	1522.64
8-Convolution	610.47	696.31	781.68	867.49	951.51	1036.14	1120.08	1195.26	1274.61	1349.39	1411.81	1486.29
9-Convolution	611.08	699.28	789.56	880.37	967.98	1054.26	1137.77	1225.98	1298.03	1379.81	1442.48	1513.78
10-Pooling	613.7	699.54	785.91	867.23	932.83	995.01	1040.07	1091.32	1126.23	1153.23	1183.1	1213.28
11-InnerProduct	811.9	1096.78	1244.69	1245.83	1312.58	1361.5	1378.06	1413.61	1475.87	1497.48	1549.25	1598.4
12-InnerProduct	803.68	1078.58	1255.6	1249.3	1308.94	1372.86	1382.81	1419.36	1484.68	1500.39	1543.34	1601.89
13-InnerProduct	804.92	1081.79	1260.23	1256.89	1299.79	1380.58	1382.74	1435.16	1491.46	1494.51	1551.1	1597.54

5.8 Optimization and Evaluation of the C++ LEON Framework

In this section the optimizations made on the LEON C++ Framework will be presented. These optimizations led to dramatic increase in performance of the LEON execution and provided augmented code flexibility and scalability. Then, a short evaluation of this framework will be provided in comparison with the older version of the CNN Engine.

5.8.1 Optimization techniques

The most important step done on the LEON OS code, given the object oriented nature of the challenge faced, was the development of the framework in C++. This, not only made the code more efficient due to the nature of the language, but also provided abstract libraries making future extensions of the engine a lot easier.

The following actions were done in order to make the application more efficient in terms of application memory footprint, execution time and application consistency:

- Inlining layer constructors and one-liner methods
- Usage of ”emplace_back” method instead of ”push_back” for constructing the network (See C++ documentation for method differences)
- Static DDR allocation on compile-time for weights, biases and output buffers instead of dynamic allocation
- Usage of STL iterators on loops instead of dereferenced pointers
- Usage of conditional flags to compile only the code semgnent regarding the mode selected and the layers included
- Reduced CMX uncached data segment by changing struct data types passed to SHAVES

5.8.2 Evaluation

For the evaluation process of this framework running on LEON OS, its metrics are compared with the former CNN framework.

- Network creation and layer dispatch time reduced by 40%

The comparison of the total LEON OS execution time between this and the former implementation was conducted using ”CIFAR-10” Neural Network, as it was one of the few small CNNs that were supported by the former Engine. The total time needed by the older version was 15 ms, whereas this time reduced to 9 ms on the newer version written in C++.

- The total memory segment needed by LEON OS was reduced by 97%

The total memory assigned to LEON OS was set to 64 MB, because the data segment was used dynamically to allocate space for the output buffers of each layer of the network. There was explicit allocation for the output buffer of each layer, leading to enormous memory allocation only for buffers, even on small layers. Deploying large networks, like ”GoogLenet”, would be impossible with this method.

In the newer Engine version, the memory segment used by LEON OS is 2MB, which offers a x32 reduction. The output buffers are statically allocated on DDR on the initiation of the application, leading to better performance. In linear networks only two output buffers are used, which are used as swap buffers. This means that, on each execution round, one buffer is used as input and the other as output, with their role switching layer after layer. In layers with blocks in parallel, there are more buffers used. The size and the count of the buffers is precomputed offline by the Interface, where a specific algorithm computes the exact size and count needed for the application. This is very critical when deploying state-of-the-art CNNs like ”Alexnet” or ”VGG” that need massive amount of memory to store their weight and bias arrays.

- Framework able to support large variety of networks

The current LEON OS implementation, in co-operation with the Python interface, is able to execute any possible CNN, just by providing a ”prototx”t and ”caffemodel” file through the user arguments. The C++ framework on its own, is easily generalized to as many types of layers a possible without modifying any existing code, due to the architecture of the ”Layers” abstract Class implemented. This makes it fully compatible with ”Caffe”.

Chapter 6

Vision Computational Libraries

One of the purposes of this thesis was to achieve generalization in multiple different CNNs. However, each network provides different features and exploits data in a different way with the others. Therefore, a wide variety of layer types are used, as well as a broad range of convolutional kernels. Also, the layer architecture of a network varies from one another. There are linear networks, in which each node is connected only with a previous layer and a next layer, and there are non-linear networks, where one layer can feed blobs to multiple different layers, creating branches that are trained to extract different features each. As a consequence, extensions had to be made to the computational engine to support a large variety of state of the art networks that are used today in computer vision tasks.

6.1 The Local Response Normalization layer (LRN)

6.1.1 Definition and usage of LRN

In neurobiology, there is a concept called "lateral inhibition". This refers to the capacity of an excited neuron to subdue its neighbors. A significant peak is definitely wanted, so that a form of local maxima is achieved. This tends to create a contrast in that area, hence increasing the sensory perception. Increasing the sensory perception is helpful in blobs of CNNs because sensitivity on desired pixels, that will be filtered afterwards, is increased, thus increasing output accuracy.

Local Response Normalization (LRN) layer implements the discussed lateral inhibition. This layer is useful when dealing with ReLU neurons. This is because ReLU neurons have unbounded activations and LRN is needed to normalize that. High frequency features with a large response need to be detected and if normalization occurs around the local neighborhood of the excited neuron, it becomes even more sensitive as compared to its neighbors [20].

At the same time, it will dampen the responses that are uniformly large in any given local neighborhood. If all the values are large, then normalizing those values will diminish all of them. So basically this inhibition is encouraged in order to boost the neurons with relatively larger activations [10].

There are two types of normalizations available in Caffe. You can either normalize within the same channel or you can normalize across channels. Both these methods tend to amplify the excited neuron while dampening the surrounding neurons. When the normalization occurs within the same channel, it is like considering a 2D neighborhood of dimension N x N, where N is the size of the normalization window. You normalize this window using the values in this neighborhood. If normalization happens across channels, a neighborhood will be considered along the third dimension but at a single location. You need to consider an area of shape N x 1 x 1. Here 1 x 1 refers to a single value in a 2D matrix and N refers to the normalization size.

”WITH_CHANNEL” mode, as is named in Caffe engine, is almost never used, because it provides insignificant normalization on input blobs. To be more specific, no contemporary and widely reputed CNN is currently using this kind of Local Response Normalization.

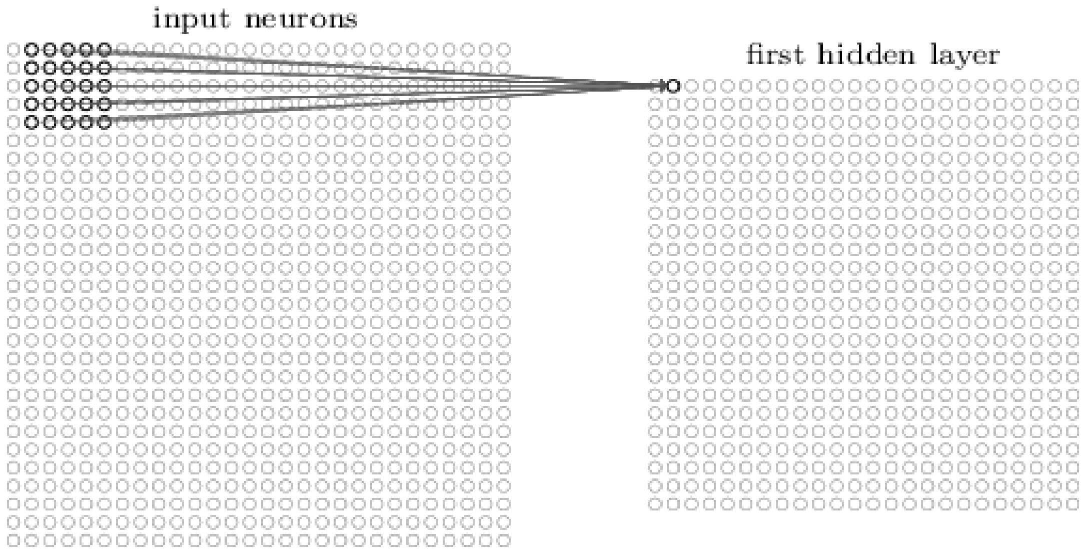


Figure 6.1: LRN Within Channel

On the other hand, ”ACROSS_CHANNELS” LRN is widely used among networks like, ”GoogLenet” and ”Alexnet”.

The formula for the computation of LRN is as follows:

$$out\_pix^i_{x,y} = \frac{in\_pix^i_{x,y}}{(1 + \frac{\alpha}{l\_r} \cdot \sum_{j=max(0,i-\frac{l\_r}{2})}^{j=min(N-1,i+\frac{l\_r}{2})} (in\_pix^i_{x,y})^2)^{\beta}}$$

Where:

- **in_pix(i,x,y)** represents the ith convolution kernel's output (after ReLU) at the position of (x,y) in the feature map
- **out_pix(i,x,y)** represents the output of local response normalization, and of course it's also the input for the next layer
- **out_pix(i,x,y)** is the number of channels of the input blob
- **l_r** is local ratio, hyperparameter of LRN layer and indicates the adjacent normalization kernel number. Most CNNs use local ratio equal to 5
- **α** and **β** are layer hyperparameters alpha and beta. Most CNNs use alpha equal to 0.0001 and beta equal to 0.75

Figure 6.2 illustrates thoroughly how LRN works, when applied after a convolutional filter paired with ReLU linear neuron:

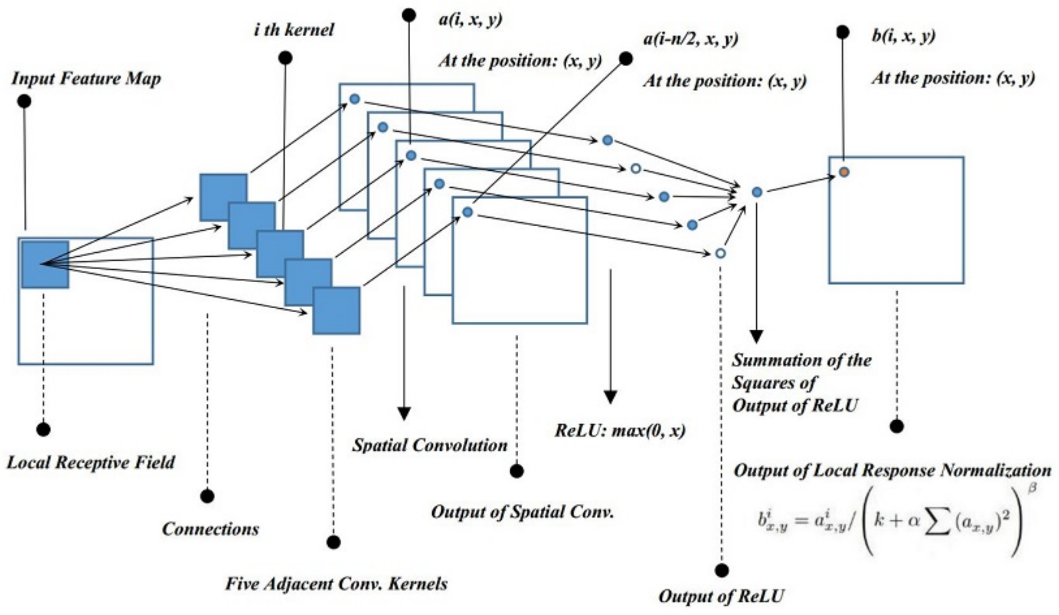


Figure 6.2: LRN Across Channels flowchart

Having explained what exactly is LRN, a graph representation (6.3) will be provided to give an intuitive understanding of the output and effect of LRN on input blobs. Keep in mind that, because the LRN happens after ReLU, the inputs should all be no less than 0.

Be noted that the x axis represents the summation of the squared output of ReLU, ranging from 0 to 1000, and the y axis represents out_pix(i, x, y) divides in_pix(i, x, y). The hyper-parameters are set default for this evaluation. So, the real out_pix(i, x, y)'s value should be the the y axis's value multiplied with the in_pix(i, x, y). Since the slope at the

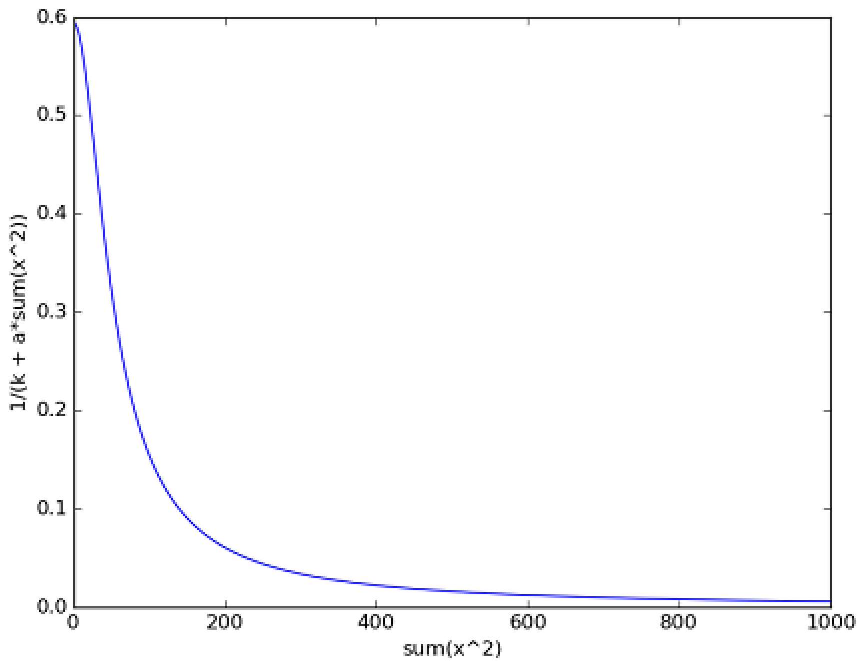


Figure 6.3: LRN Activation output

beginning is very steep, little difference among the inputs will be significantly enlarged and exactly this is where the competition happens.

```
>>> import numpy as np
>>> import matplotlib.pyplot as plt
>>> def lrn(x):
...     y = 1 / (2 + (10e-4) * x ** 2) ** 0.75
...     return y
>>> input = np.arange(0, 1000, 0.2)
>>> output = lrn(input)
>>> plt.plot(input, output)
>>> plt.xlabel('sum(x^2)')
>>> plt.ylabel('1 / (k + a * sum(x^2))')
>>> plt.show()
```

Listing 6.1: Python code used for generating LRN output plot

6.1.2 LRN DMA Algorithm

The LRN implementation is consisted of two code sectors: The parallelized DMA algorithm and the computational routine. While the assembly routine is critical to the performance of the layer, the DMA algorithm is equally important because a smart implementation can provide fruitful grounds for the computation to be efficiently parallelized and deployed.

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

There are several driver functions for declaring 2D transactions:

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

The latter function is used to create DMA transactions in LRN implementation because it provides the possibility to use striding techniques and isolate chunks of the image that need to be processed by each SHAVE.

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

```
ref[0]=dmaCreateTransactionFullOptions(
id,
&task[0] ,
input_address+image_offset,     //source
local_input,                    //destination
number_of_pixels*channels,      //byte length
number_of_pixels,               //source line width
number_of_pixels,               //dest line width
image_offset,                   //source stride
number_of_pixels);              //destination stride
dmaStartListTask(ref[0]);
```

Listing 6.2: DMA transaction example

Where:

- input_address+pixel_offset is the starting source address that each SHAVE will read
- local_input is the local SHAVE input buffer that will be used as input to the computation
- number_of_pixels*channels is the total pixels that will be processed by the SHAVE
- number_of_pixels is accordingly the number of pixels that will be processed by the SHAVE per channel
- image_offset is the stride that will be applied to the starting input address and increments as the SHAVE brings blocks in order to index the next unprocessed block

The LRN implementation is parallelized among any possible number of SHAVES used for any possible input image. When an input image needs to be normalized, the distribution of its blobs among the SHAVES is not done in regard of the input channels but rather transversely. Simply put, imagine an input blob of dimensions C x H x W (Channels x Height x Width) that is going to be processed by seven SHAVES. Then, each SHAVE is going to process the following amount of pixels:

$$Pixels\_per\_shave = C \cdot (\frac{H * W}{7})$$

Of course, each SHAVE pointer indexing to the starting address of the frame that is going to process, has an accordingly calculated offset, leading to each processor computing a different chunk of the blob.

To generalize for any input blob to be normalized, with a variable number of SHAVES, the pixel frame that each SHAVE is going to calculate is defined by the following formula:

$$Pixels\_per\_shave = C \cdot (\frac{H * W}{Number\_of\_shaves})$$

Whereas the offset of the input pointer of each SHAVE is defined as:

$$input\_offset = shave\_id\_per\_shave$$

Where shave_id ranges from 0 to 11, since the maximum number of SHAVES is 12. To better visualize this distribution, imagine a 2D matrix divided in N blocks with the same amount of pixels and expand this very same pattern in all the channels of the input image that will be normalized. In that perspective, these chunks can be seen as independent sub images with the same number of channels and smaller 2D dimensions. In figure 6.4 an example of 3 SHAVES processing different chunks of an N-channel input blob is shown.

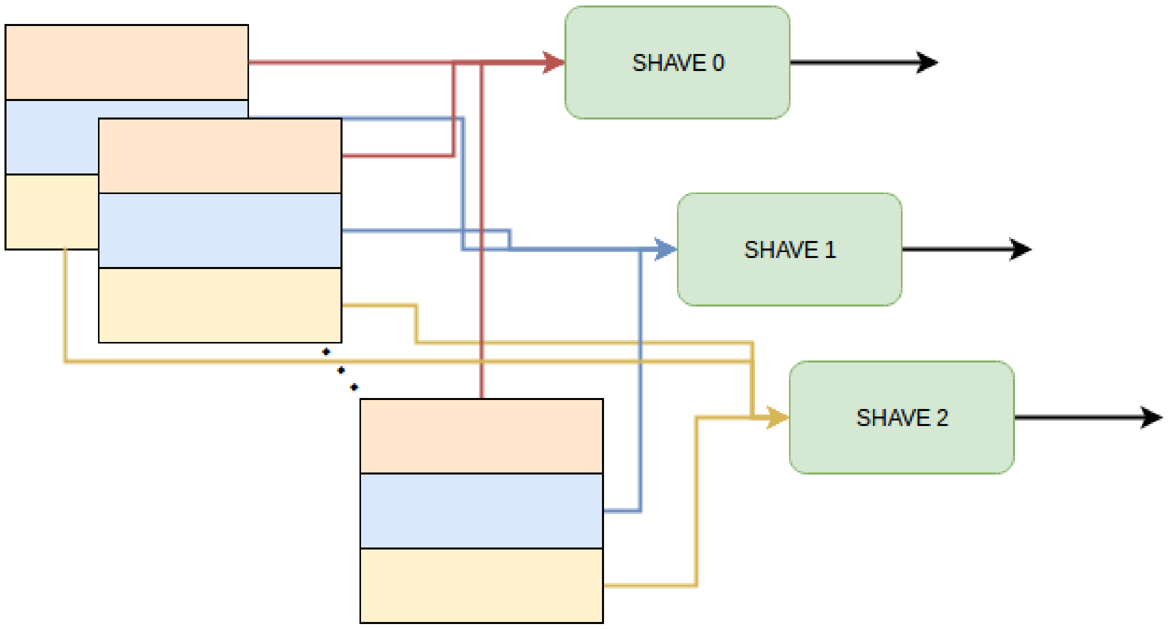


Figure 6.4: N input channels distributed among 3 SHAVES

The DMA Algorithm designed inside the SHAVES follows one basic principle, which proved to be useful on certain circumstances: There is a threshold which is dependent to the data segment available to each SHAVE, which the processor uses to decide if the sub image to be processed, will be brought all at once, or if it will be brought in blocks. This is a very crucial decision as the pattern of the computation hugely differs. In order to decide, the SHAVE examines the following condition:

$$\frac{memory\_pool - 4 \cdot 1024}{channels \cdot 4} < number\_of\_pixels$$

If this condition is false, then the SHAVE decides that the block that is assigned to compute, can be brought in one piece. In that case, only two DMA Transactions are created: One for bringing the block to CMX and another to return it to DDR after the computation has happened. This, although simple thought, kept low the code complexity in cases where the input image is small enough, keeping low the execution time of LRN.

On the other hand, if the above condition is true, then the processor adapts to a way different behaviour. The sub-image is divided into balanced pixel batches and three buffers are allocated: two as inputs and one as output. The idea of the algorithm is that the first pixel batch is loaded to the first input buffer. Then, while the first input buffer is processed by the assembly routine and written to output buffer, a second DMA transaction is called to bring the second block to the second input buffer. This described pattern is iterated over all batches of the block that need to be processed by each SHAVE, until the whole image is processed. The two input buffers alternate their roles in each iteration. This is achieved through a switching mechanism that is complemented after each iteration and defines which buffer will be loaded and which will be computed and written to the output.

```
while (pixel_offset < last_pixel){

  dma(input_address, local_input[buffer_switch%2]

  compute(input_pointer[(buffer_switch+1)%2]

  dma(local_output, input_address)
   pixel_offset += pixels_per_batch;
   buffer_switch = (buffer_switch + 1) % 2;
}
```

Listing 6.3: Main loop of DMA routine

As it can be seen, pixel_offset, which is used for offsetting input and output and for controlling the loop, is incremented by a pixel batch after each iteration. On the last iteration the value of this variable will be equal to last_pixel, leading the loop to an end.

6.1.3 LRN Generic Computational Algorithm

The most important part of a layer implementation inside Myriad2, taking into account the performance and the energy consumption, is the computational routine used. For the scope of this implementation, two algorithms for the layer computation were designed: A generic one, written in C, able to perform computation on any type of local_ratio hyperparameter and a specialized Myriad2 assembly routine, executing with the local_ratio statically set to 5. This is because every state-of-the-art CNN that is used today, and was also used to evaluate this implementation, uses local_ratio equal to 5. Therefore, the C implementation is neither efficient in terms of execution time and power consumption, nor useful except only extremely rare circumstances. However, for the generity of the CNN compatibility, this function is also included in the computational library.

In the following page, the C algorithm that performs the generic LRN computation is provided and explained:

```c
void LRN_AcrossChannels(fp16** input, fp16** output,
                        u16 channels, u16 pixel_batch,
                        u8 local_ratio, fp16 alpha, fp16 beta){
    //variable declaration(...)
    for (u32 pix = 0; pix < pixel_batch; pix++){
        for (u32 ch = 0; ch < channels; ch++){
            sum = 0;
            low_margin = ch-(local_ratio/2);
            low_margin = ( low_margin < 0 ? 0 : low_margin);
            up_margin = ( ((channels - 1) < (ch + (local_ratio / 2)
                         + (local_ratio % 2) - 1)) ? (channels - 1)
                         : ch + (local_ratio / 2)
                         + (local_ratio % 2) - 1 );
            current_input_pixel = input[0][ch * pixel_batch + pix];
            for(int l = low_margin; l < (up_margin + 1); l++){
                squares = input[0][(pixel_batch)*(l) + pix ];
                squares *= squares;
                sum += squares;
            }
            current_output_pixel = current_input_pixel
                        / pow((1 + (alpha / local_ratio) * sum), beta);
            output[0][ch * pixel_batch + pix] = current_output_pixel;
        }
    }
    return;
}
```

Listing 6.4: LRN generic C implementation

Where:

- **fp16** input** is float-16 pointer to pointers to the input of the chunk to be processed
- **fp16** output** is float-16 pointer to pointers to the output where the computed chunk will be stored
- **u16 channels** is the number of channels that the normalization will be applied
- **u16 pixel_batch** is the number of pixels for which the computation is responsible
- **u8 local_ratio**, **fp16 alpha** and **fp16 beta** are the layer's hyperparameters affecting the value of the output computation

It should be noted that in Myriad2, there are custom data type abbreviations. ”fp16” type is a shortcut for ”half” and ”u16”, ”u8” are shortcuts for ”uint_16t” and ”uint_8t” respectively.

6.1.4 Hyperparameter specific assembly algorithm

The algorithmic section that defines, almost exclusively, the efficiency and the performance of the LRN implementation, is the Myriad assembly routine for implementing the computational method of the layer.

In order to save time and instructions, this implementation does not take the layer hyperparameters as arguments, but rather these are set to the default values used in all neural networks, including ”GoogLenet”, ”Alexnet” and ”ZFNet”. More specifically ”local_ratio” is equal to 5, ”alpha” το 0.0001 and beta to 0.75.

The assembly algorithm follows a very different structure from the generic implementation, using loop unrolling and loop tiling techniques, leading to better performance in overall. The SHAVES are VLIW (Very Long Instruction Word) processors, which means that they are able to process data in vectors and parallelize the fetching of many instructions at the same time. For that reason, the assembly routine of the same implementation, in comparison with the analogous C function, can be as much as 30 times faster. Comparison and evaluation of these two functions will be provided in the chapter regarding the experimental results of the CNN engine.

In the following listing, the algorithm of the assembly implementation for LRN-specific function is given. It is given as pseudocode, as the Myriad2 ISA and assembly commands are classified by Movidius, Intel.

```
arguments: fp16** input, fp16** output, u16 channels, u16 total_pixels)

start:
load input address
load output address
register local_ratio = 5
register alpha = 0.0001
register beta = 0.75

loop_on_whole_picture:

load input_address to temp_input_register
load output_address to temp_output_register

//------pixel loading and computation-----//

  load first pixel_batch to register_1
  temp_input_address += 4 elements

  load second pixel_batch to register_2
  temp_input_address += 4 elements

  compute first pixel_batch

  load third_pixel_batch to register_3
  temp_input_address += 4 elements

  compute second pixel_batch

  store first pixel_batch to output
  temp_output_address += 4

  load fourth pixel_batch to register_4
  temp_input_address += 4

  compute third pixel_batch

  store second pixel_batch to output
  temp_output_address += 4

  store third pixel_batch to output
  temp_output_address += 4

  channel_counter = 4

main_loop:

  load (K+1) pixel_batch to register_5
  temp_input_address += 4

  compute K pixel_batch

  store K pixel_batch to output
  temp_output_address += 4

  //slide registers holding input
  register_1 = register_2
  register_2 = register_3
  register_3 = register_4
  register_4 = register_5

  channel_counter += 1
  repeat if channel_counter < N channels

final_pixels:

  compute (N-1) channel
  store (N-1) channel to output
  temp_output_address += 4

  compute N channel
  store N channel to output
  temp_output_address += 4

pixel_counter += 4
if pixel_counter < total_pixels: repeat
else end
```

Listing 6.5: LRN specialized assembly routine

6.1.5 Iterative method design for float exponent calculation

As already shown in previous subsection, the LRN mathematical computation includes floating point numbers raised in the power of floats (e.g. $2.247^{1.47}$). This computation might be easily done using the C function "pow(base, exponent)" which is included in "math.h" library. However, this utility is obviously not available to use in Myriad2 bare-metal assembly language and even if it were, it would be nowhere near efficient.

Furthermore, the ISA of Myriad2 does not include any instructions available to floating point-16 numbers that are concerned with float powers. The only given instructions similar to raising to powers is the multiplication and divide instructions and the square root. Therefore, without designing a heuristic method to implement float-raised to-float mathematical operation, it is impossible to write a generalized, to layer hyperparameters, assembly routine for the LRN layer due to this ISA deficiency.

To tackle this challenge, a custom iterative method was designed from scratch that very efficiently converges, in floating point-16 accuracy, after three to five iterations, providing a very efficient way to perform this computation in assembly.

Below, the iterative formula is provided. Suppose that the base of the float power is a float number represented as "a.b" and the float power as "c.d". a and c represent the digits left of the decimal point whereas b and d represent the decimal point digits. For example in the number $2.24^{1.42}$: a=2, b=24, c=1, d=42.

$$f(a.b, c.d) = (a.b)^c \cdot f(\sqrt[8]{a.b}, 8 \cdot 0.d)$$

Where:

$$\sqrt[8]{a.b} = \sqrt{\sqrt{\sqrt{a.b}}}$$

And therefore can be computed by Myriad2 Instruction Set Architecture.

An example is provided in order to evaluate this iterative method. Suppose that, with the help of this iterative formula, $3.8791^{2.293}$ needs to be computed. Using a high precision calculator, this mathematical expression is equal to 22.38515608.

Using the iterative method:

$$3.8791^{2.293} = 3.8791^2 \cdot \left(\sqrt{\sqrt{\sqrt{3.8791}}}\right)^{0.293\cdot 8} = 3.8791 \cdot 3.8791 \cdot (1.184653)^{2.344} = 15.047416 \cdot (1.184653)^{2.344} = 15.047416 \cdot 1.184653^2 \cdot \left(\sqrt{\sqrt{\sqrt{1.184653}}}\right)^{0.344\cdot 8} = 21.11758584 \cdot (1.0214071)^{2.752} = 21.11758584 \cdot (1.0214071)^2 \cdot \left(\sqrt{\sqrt{\sqrt{1.0214071}}}\right)^{0.344\cdot 8} = 22.03139581 \cdot (1.002651156)^{2.752} = 22.148368 \cdot (1.00033)^{6.016}$$

iteration_output $= 22.148368$

Notice that after only 3 iterations the output number has converged effectively close to the originally computed number, reaching almost its 99%. Floating point-16 numbers have nowhere near the 32-bit or 64-bit precision and therefore the termination criteria of the iterative method could be set to a slightly bigger deviation than 1%.

For this numerical iteration to translated to algorithm and be successfully deployed to Myriad2, a recursive algorithm was design to represent this numerical procedure. The algorithm is presented below and uses only multiplication, divide and square root, as these are the only tools provided by the ISA for this specific functionality.

```
calculate_float_power(num, exponent):

  diff = 1-num;
  if diff < 0:
    diff *= -1

  if diff < 0.0005:
    return 1;
  else:
    for i=0 -> int(exponent):
      result *= num;
    exponent -= int(exponent);
    result *= calculate_float_power(sqrt8(num), 8*exponent);
    return result;
```

Listing 6.6: Recursive algorithm for calculating float number raised to float power

6.2 The Concat layer

6.2.1 Definition and usage of Concat

Another important layer used in state-of-the-art Convolutional Neural Networks is the ”Concat” layer. Concat is a utility layer that concatenates its multiple input blobs to one single output blob [21]. The need for using Concat occurs on networks with parallel branches of different layers that need to be merged at a specific point. Such example is ”GoogLenet”, which contains the inception blocks, and ”SqueezeNet”.

Usually, data in caffe is stored in 4D blobs: BxCxHxW (that is, batch size by channel by height by width). If there are two blobs B1xC1xH1xW1 and B2xC2xH2xW2 Concat is used to concatanate them along their channel dimension to form an output blob with C=C1+C2. This is only possible *iff* B1=B2=B and H1=H2=H and W1=W2=W, resulting with Bx(C1+C2)xHxW.

Figure 6.5 shows the way Concat concatanates input blobs on GoogleNet’s inception block structure.

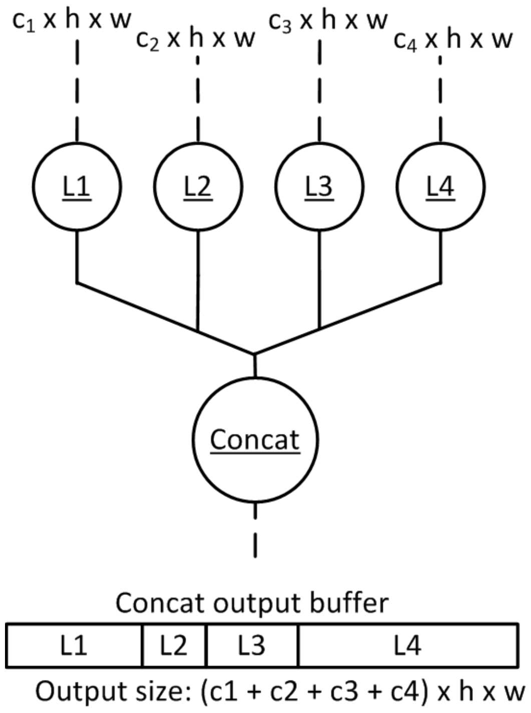


Figure 6.5: Concat usage overview on GoogleNet’s inception block

6.2.2 Implementation specifics

Since Concat is a utility layer, there is no need for intensive computation to happen and therefore the SHAVES are not needed to perform any operations. As it has already been stated, the Python Interface uses as few data buffers as possible for the blobs. In parallel layer branches, the interface defines different swap buffers for each distinct branch that it recognizes, as they have independent data flows. However, distinct swap buffers for each branch are necessary, because at the end of the parallel block, these buffers will be concatanated with a Concat layer.

The most obvious way of implementing the functionality of Concat layer inside the CNN engine would be to have LEON OS processor perform DMA transactions, one for each layer, storing in order, one by one, the input blobs of a Concat layer to its data buffer. This implementation sounds much less complex than the rest of vision computational layers and therefore it could be an accepted solution.

The concept of this solution is shown in figure 6.6 below:

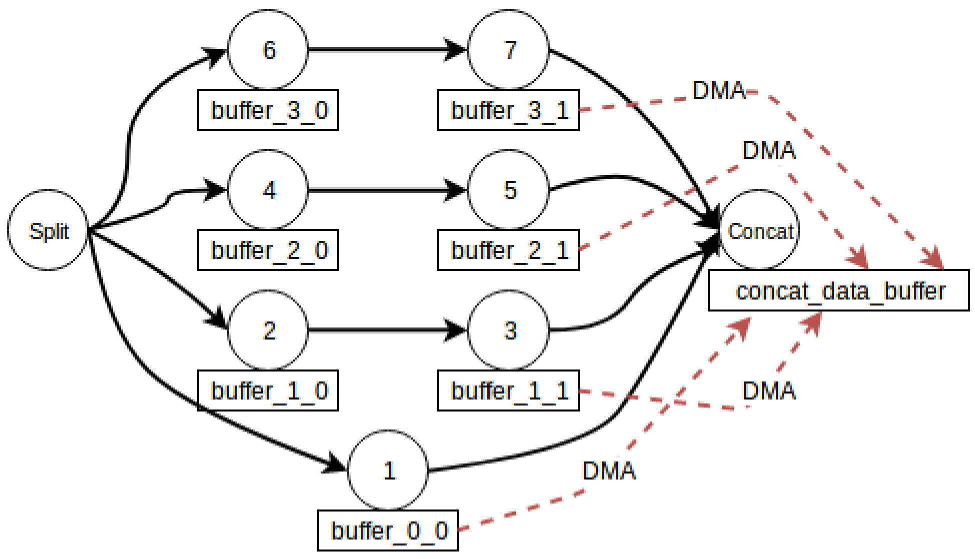


Figure 6.6: Implementation of Concat using LEON DMA Transactions on GoogleNet

However, an extremely efficient way of implementing Concat was designed, eliminating any possible overhead its computation could add to the engine. The implementation is included in the library generation function performed by the Python interface, avoiding unnecessary memory accesses and data transfers: The input layers of each Concat are offline identified and they write their output directly to the Concat buffer, instead of their owns. In figure 6.7 the current implementation of Concat can be seen:

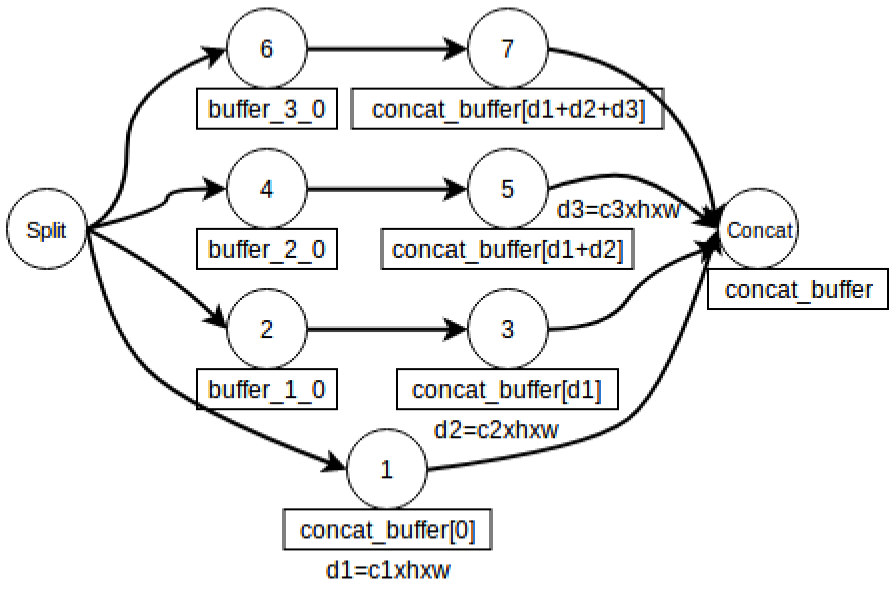


Figure 6.7: Concat implementation with offline Python precompute on GoogleNet

As it can be seen, not only does the execution overhead of Concat is eliminated on run-time, but also the number of the data buffers needed to store the blob information are reduced, thus reducing the total amount of memory needed on the embedded system.

6.3 Direct Convolution 1x1 implementation

Although the Convolution layer in the former CNN Engine was already implemented, there was no computational routine available neither by the engine nor by the Movidius MDK for Convolution with kernel equal to 1 (Convolution 1x1). This kernel is extremely important as it is lately used in most Deep CNNs, like GoogleNet and SqueezeNet.

1x1 convolution was first introduced in this paper titled Network in Network [13]. In this paper, the author's goal was to generate a deeper network without simply stacking more layers. It replaces few filters with a smaller perceptron layer with mixture of 1x1 and 3x3 convolutions. In a way, it can be seen as "going wide" instead of "deep", but it should be noted that in machine learning terminology, 'going wide' is often meant as adding more data to the training. Combination of 1x1 (x F) convolution is mathematically equivalent to a multi-layer perceptron.

Although 1x1 convolution is a 'feature pooling' technique, there is more to it than just sum pooling of features across various channels/feature-maps of a given layer. 1x1 convolution acts like coordinate-dependent transformation in the filter space. It is important to note here that this transformation is strictly linear, but in most of application of 1x1 convolution, it is succeeded by a non-linear activation layer like ReLU. This transformation is learned through the (stochastic) gradient descent. But an important distinction is that it

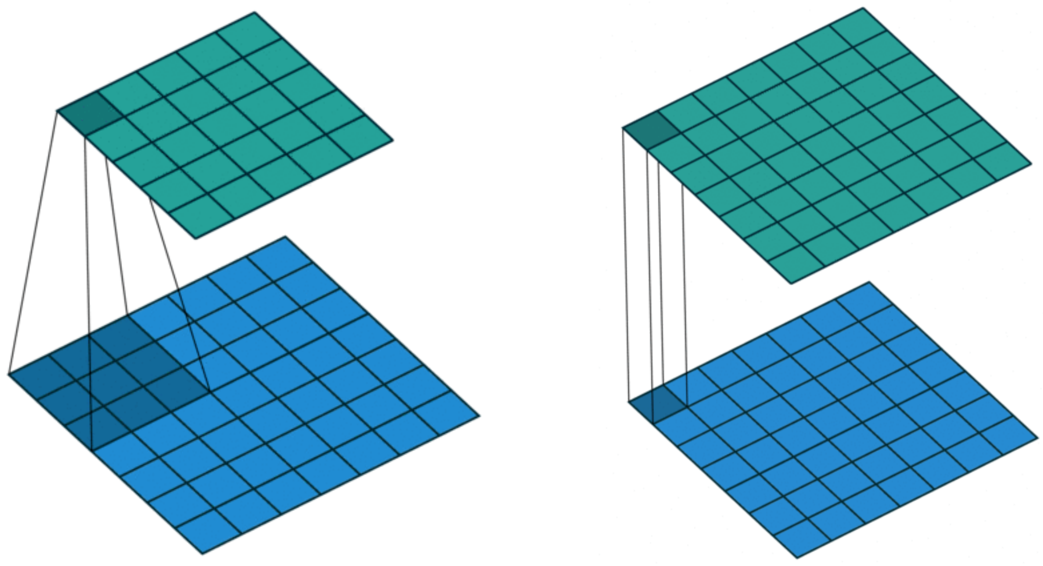

(a) Convolution with kernel of size 3x3 **(b)** Convolution with kernel of size 1x1

Figure 6.8: Dfference between kernel 3 and kernel 1

suffers with less over-fitting due to smaller kernel size (1x1).

Figure 6.8 above presents the difference on the output maps between 3x3 and 1x1 filtering.

The computational routine that implements 1x1 Convolution inside the engine was written in Myriad2 assembly language. The explanation of the routine arguments as well as the pseudo algorithm of the implementation are given below.

- **half** ** **in** is float-16 pointer to pointers to the input of the chunk to be processed
- **half** ** **out** is float-16 pointer to pointers to the output where the computed chunk will be stored
- **half conv[1]** represents the convolution kernel. This array parameter requires the same number of elements as the size of convolution. The 5×5 convolution would require 25 array elements. In this case, it is only one element (1x1).
- **inWidth** defines the width of the input line. Because the assembly routine utilizes SIMD instructions, it is important to bare in mind that inWidth needs to be a multiple of 8. If this is not the case, usually inWidth is rounded down to the closest multiple of 8, however this is not always the case (there is a discrepancy among the MDK routines)

```
//void Convolution1x1_asm(half** in,
                           half** out,
                           half conv[1],
                           u32 inWidth)
Convolution1x1_asm:

  load input address
  load output address
  load first input block
  load kernel address

  loop_counter = inWidth / 8

loop:

  if loop_counter = 0
    jump to the_end
  else
    load K+1 block from input_address
    input_address += 8

    multiply K block with kernel

    store block K to output_address
    output_address += 8

    loop_counter -= 1
    jump to loop

the_end:
multiply K+1 block with kernel
store block K to output_address
end
```

Listing 6.7: Convolution 1x1 algorithm

Chapter 7

Library generation interface

The final and equally important part of this thesis was the design and implementation of an interface capable of automating many different tasks into one package. To accomplish this task, Python was used as a programming language, because of the caffe module exposed by the caffe engine in this language. Therefore, it was the only language that could be used to exploit the caffe compatible model descriptions, meaning the -prototxt and -caffemodel files of a CNN.

The Python interface currently uses a wide set of different algorithms, but the tasks that it is used for can be divided into two categories:

- The generation and compile of network libraries
- The design space exploration of a network, in regard to execution time and energy consumption.

In this chapter, the first branch of the interface will be explained.

7.1 The library generation function

As already explained, this function is responsible for providing an abstract way to the user or the designer, to import any possible CNN into the Myriad2 engine either for inference or for profiling purposes. In this section, the algorithms used in this function will be analyzed.

The following snippet provided, contains the body of the library generator function as much abstract as possible, divided in sub-functions. Each sub-function will be explained in a different subsection in order for the function to be easier to understand.

```
def library_generator(prototxt, caffemodel, image):

  initialize_caffe()
  initialize_library_lists()
  initalize_variables()

  for index, one_layer in enumerate(network_layers):
    find_bottom_nodes_of_layer()
    check_if_parallel()
    append_layer_to_libs()

  write_lib_lists_to_files()

  return
```

Listing 7.1: Abstract overview of library_generator function

7.1.1 Caffe and library lists initialization

The caffe module needs to be initialized inside the body of the function in order for its members to be used. The following snippet of code is used to initalize the framework.

```
1    caffe.set_mode_cpu()
2    network = caffe.Net(prototxt, caffemodel, caffe.TEST)
3    ...
4    network_list = []
5    ...
```

Listing 7.2: Caffe module initialization

In line 1 the instruction initializes Caffe on cpu mode. There is the choice of ”gpu_mode” which is mainly used for network training and other computationally intensive tasks. Line 2 provides the network description as arguments to the ”Net” method of caffe, and assigns to the variable ”network” a list of dictionaries containing all the layers of the network with their attributes.

Before iterating over all layers of the network the library lists that will be used, as well as some variables, need to be initialized first. These variables are mainly lists that the information about the network will be appended to, and variables that will server their purposes on the different algorithms used inside the ”for” loop.

7.1.2 Bottom nodes extraction

After the initialization of the framework, a loop that iterates over all network layers follows. The layers iterate with the same order as in the prototxt file, which is created by the network designer. No special precautions and conditions are taken about the order of the layers, because the Caffe framework takes care of the layers being in the appropriate order. If the prototxt file contains layers that are not in the correct format (e.g. one after another in a linear network) then the Caffe initialization exits with an error code.

The first task that a layer does inside the loop is to register its own id inside a list, that will be read by the next layer, which will be searching its parent node's id. Then, as already implied, the layer is searching for the id of its parent. In case the layer is of type 'Input', then it skips this search, as Input layers are always the first nodes of the network.

```
bottom_list = []
top_to_id.append(my_id)
if layer_type != 'Input':
  for previous in layer_bottoms:
    for i, j in enumerate(top_to_id):
        if j == previous:
        bottom = i
    bottom_list.append(bottom)
```

Listing 7.3: Bottom node search

In each iteration the bottom list is re-initialized. The bottom id is necessary to be known for each layer, because it is provided as an argument to its class constructor, indicating the node of the network that feeds that specific layer's input.

7.1.3 Parallel networks configuration

The library generation function uses a swap buffering technique on linear networks in order to allocate inside Myriad2 as fewest and smallest blob buffers as possible. To be more specific, for linear networks the interface defines only two blob buffers which are used alternately. This, in practice, means that half of the layers read their input from
buffer_0 and write their output to buffer_1, whereas the rest read their input from buffer_1 and write their output to buffer_0.

The size of buffer_0 will be defined as equal to the size of the biggest blob, from layers of the subnet that this buffer is responsible for. The same applies for the complementary buffer. This technique ensures that not a byte more than those needed will be allocated for storing output blobs. In the following figure 7.1, it is obvious how this couple of buffers is used on linear networks or linear subnets of parallel networks.

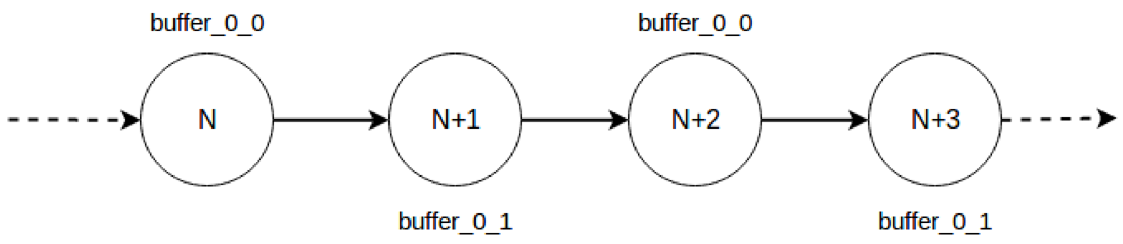


Figure 7.1: Swap buffering on linear networks or linear subnets

In cases where a network splits up on a specific point to multiple branches of layers, then the swap buffering technique, using two buffers, is ineffective. In such cases, swap buffering is again used, but on N couples of swap buffers, one for each parallel branch of the network. The split up of the network to parallel branches is recognized by the interface when a "Split" layer is met [22]. The coming of a Concat layer while iterating, indicates the end of a parallel block and the algorithm returns to the previous policy, using only a couple of buffers for the blobs, however starting from the complementary one in regard to the last used before the split layer (this is a very important note because otherwise unwanted data overwrite might occur). Figure 7.2 illustrates the extension on networks with parallel branches.

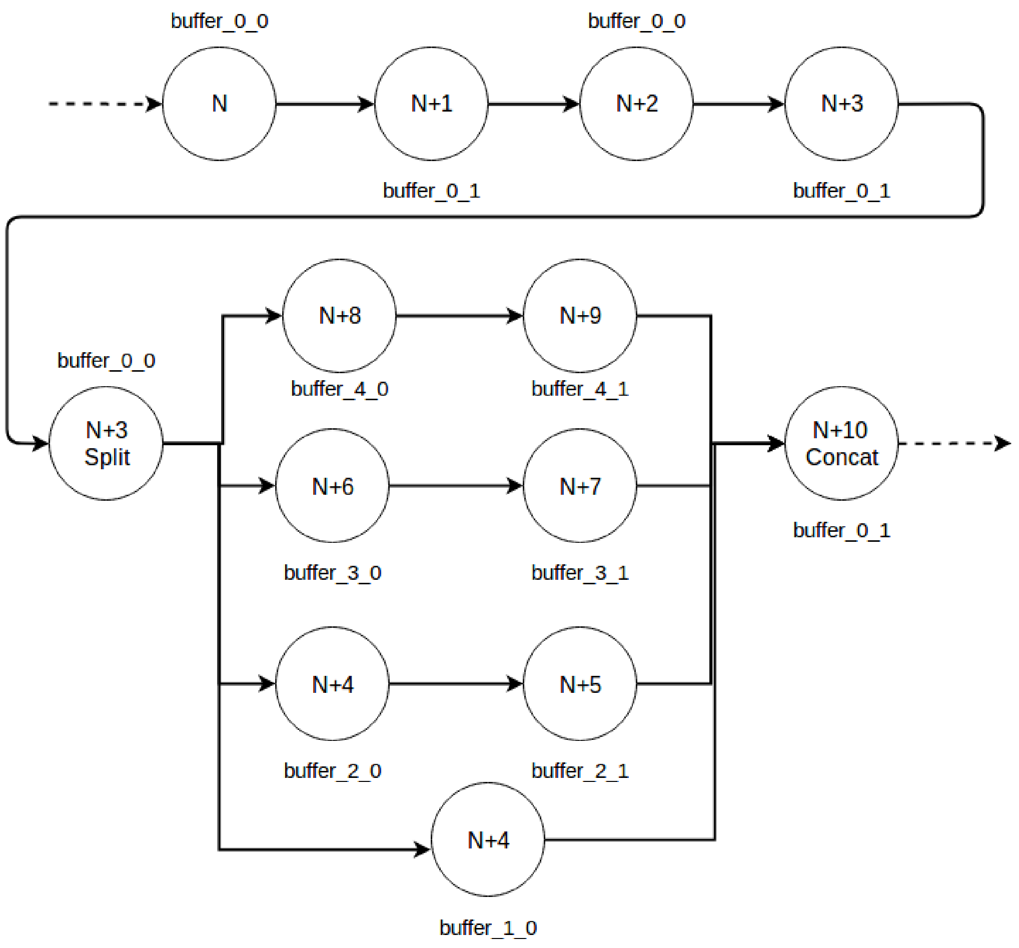


Figure 7.2: Swap buffering on parallel blocks

Do not forget, however, that except from the algorithm that is applied on parallel blocks to define the swap buffers, another branch of the code searches for parent nodes of Concat layer, in order to implement the pre-computation of this layer, as described in section 5.2 and figure 6.7, so keep that in mind.

The snippet of the code that implements the ideas described above is given:

```
if layer_type == 'Split':
  split_position = id - 1
  split_buffer_index = branch_buffer_index

  if layer_type != 'Concat':
    if bottom_vector[0] == split_position:
      branch_buffer_index = 0
      branch_counter += 1
      branch_list.append({'Id': id, 'Branch': branch_counter, 'Index':
   branch_buffer_index})
    else:
      for node in branch_list:
        if node['Id'] == bottom_vector[0]:
          node['Id'] = id
          branch_buffer_index = node['Index'] ^ 1
          node['Index'] = branch_buffer_index
          branch_counter = node['Branch']
  else:
    <reset all variables and lists>
.
.
.
if split_position >0:
  net_iterator=id
  while network.layer.layer_type!='Concat':
    net_iterator+=1
  for bottom in network.layer[net_iterator].bottom:
    if str(bottom)==layer_name:
      local_counter=0
      local_buffer_index=split_buffer_index
      buffer_offset_str='[' + str(2*buffer_offset) + ']'
      buffer_offset+=layer_data.shape[1]*
                     layer_data.shape[2]*
                     layer_data.shape[3]
```

Listing 7.4: Configuration on parallel blocks inside networks

7.1.4 Appending to libraries and writing files

In each iteration, the iterated layer appends all its attributes and characteristics to the library lists that will be written to the library source files at the end of the function. There is different kind of information that each layer need to append to the library lists or provide to the rest of the network:

- **Input** layer appends its constructor with the input dimensions to a list that will be written to ”network.cpp” (see Chapter 4). Also an array with the pixels of input image, if provided, in raw floating point-16 format is appended to the weight file
- **Convolution** and **Fully Connected** layers append a constructor with their attributes to the network list, as well as two arrays each to the weight file, containing the weights and the biases of the layers
- **Pooling** and **LRN** do not use weights and biases for their computation, therefore append only a constructor command. When the layer is LRN, its hyperparameters are checked in order to decide if the generic C computation will be used or the specialized assembly routine, as explained in previous chapter. The difference between these two possible routes, lies between the existence of two different constructors, using constructor overloading, that hold a different set of attributes for each case. The same applies to pooling layers for the different kinds of pooling (max, average or stochastic)
- **ReLU** is an in-line layer and therefore it is applied to the Convolution or the Fully Connected layer as a post-computation normalization. It is not implemented as a discrete layer. This is also how Caffe handles this layer. As a result, when a ReLU layer is met, the function updates the bottom node constructor’s variable regarding the in-line existence of ReLU, activating it.
- **Split** is a hidden layer produced by the Caffe computational engine, to indicate that a certain layer feeds multiple layers, initiating a block of layers in parallel branches. Split layer is used by our framework to identify parallel blocks and to perform computations required to calculate the number and the memory size of the layer data buffers required.
- **Concat** is used to determine the end of a parallel block and calculate the correct amount of buffers needed for this block, as explained before. In the end of the iteration, the Concat layer appends a pointer to its data buffer along with its blob dimensions to the network list, in order to be effectively constructed inside Myriad2.
- **Dropout** is a layer used only in training [23]. However, it is broadly used and therefore the library generator function identifies it and updates the id variables accordingly, in order for this layer to be effectively ignored.

7.2 Library compilation and execution

In this section, another part of the interface regarding the application initialization will be discussed. In order to provide high-level abstraction to the user and automate all tasks that have to do with the compile and execution of the engine, a function was designed to move the libraries to the correct directories and initialize the Movidius debug server.

The function uses process parallelization (fork, execve and SIGALARMS) to setup the application and also initialize the server. It is fairly easy to understand and is provided and explained below.

```
def compile_and_execute(clean_flag, server, network_name, profile_flag):
  ...
  move files to the appropriate directories
  ...
  if server:
    try:
      fork_process = os.fork()
      server_PID = 0
      if fork_process == 0:
        try:
          start server
          .
          .
          .
          exit
        except subprocess.CalledProcessError:
          try:
            get server PID
            kill server
          except subprocess.CalledProcessError:
            pass
          try:
            get compile process PIDs
            for pid in make_pid_list:
              kill every make process
          except subprocess.CalledProcessError:
            pass
          kill parent process
          exit
      else:
        ...
        check if clean compile given
        check the mode given (single processor, dual processor, profile)
        compile
        check if server is up
        run application
        wait until termination
        ...
        kill child process
    except KeyboardInterrupt:
      if server_PID != 0:
        kill server
      if fork_process > 0:
        kill child process
      exit
  else:
    check if server is up
    compile
    run app
    exit
```

Listing 7.5: Server setup and app compilation using forks and SIGALARMS

The above code works in the following manner:

- Library source files are moved to the appropriate directories
- If the user provides an argument instructing the interface to setup the server, then a fork is called. In any other case, the interface checks that the server is up and running (since the user has already set it up manually), compiles and runs the application
- The child process occured from the fork, initializes the MoviDebugServer. If the server is already initialized, then this process tries to kill it, as well as the compilation process of the parent process and the parent process itself, and exits with an error code
- The parent process compiles the application. If a ”clean” argument is given by the user, then cleans the object file directory of the application, in order to perform a ”clean compile”. The mode selected is also checked. Remember that there are three different modes: Single processor inference, dual processor inference and profile mode
- The parent process checks if the server is activated. If not then it kills all pending processes, as well as the child process and exits with an error code
- The application is compiled and then executed. The parent process waits until application termination and then kills the child process, checks if all processes are correctly terminated and then exits
- In the whole process there is a check for a Keyboard interrupt. In that case, it is ensured that no zombie processes are left. In the case of keyboard interrupt, the parent process takes care of killing the server (if initiated), the child process and any pending ”make” processes. Then the parent process exits.

7.3 Interface user level abstraction

In this section, the available user arguments and the functionalities of the interface will be explained. The creation of the arguments was done with the usage of ”argparse” package. The following arguments are available:

- **- -prototxt** / **-p**
- **- -caffemodel** / **-c**
- **- -image** / **-i**
- **- -shaves** / **-s** [default: 10]
- **- -profile** / **-pr** [choices: high_resolution / low_resolution]
- **- -clean** / **-cl**
- **- -server** / **-sv** [choices: keep_log / silent]
- **- -analyse_only** / **-an**
- **- -refeed** / **-re** [choices: performance / efficiency]

The **- -prototxt** and **- -caffemodel** arguments are accompanied with paths to the according files and are required by the interface for any mode selected. **- -image** specifies the path to an input image that will be infered by the engine. There is also an argument which sets the global number of SHAVES used for all layers of the network. It should be noted that optimal SHAVE configurations per layer are only accessible through profile logs that are refed to the engine using the **- -refeed** argument. The user can also choose the mode. By using the argument **- -profile**, the interface switches to profile mode, selecting either high resolution profiling or low resolution. By default, the inference mode is chosen.

The **- -clean** argument is used to instruct the interface to perform a clean compile and **- -server** is chosen to fork processes and setup the MoviDebugServer, as explained above. Finally, if a user has already profiled a network and holds its raw profile data and wants to perform a design space exploration on this data skipping the profiling part, he can choose the **- -analyse_only** option. The **- -refeed** argument is used to adapt an optimal configuration (fastest or most efficient) to the engine. It is important to note that if a network contains parallel blocks and it is found by the exploration algorithms that dual processor mode is faster, then by default this mode is deployed.

Chapter 8

Design Space Exploration interface

The other major task that the Python interface accomplishes is the manipulation of the raw profile data of networks in order to provide insights about optimal Myriad2 configurations. The metrics that are considered in these configurations are the execution time in regard to the energy consumption. Remember that by saying "raw profile data", tables like 5.1 and 5.2 are meant. These tables contain performance and power consumption information for each isolated layer and each amount of SHAVES used for it. The interface combines this data to export network configurations and suggestions about the order of execution of layers and plots to visualize these suggestions.

8.1 Pareto optimal points generation

8.1.1 Definition of Pareto points

Pareto optimal points are generated by the interface, in order to visualize the optimal network configurations on Myriad2, in terms of execution time and total energy consumption. According to the formal definition Pareto efficiency or Pareto optimality is a state of allocation of resources from which it is impossible to reallocate so as to make any one individual or preference criterion better off without making at least one individual or preference criterion worse off. The Pareto frontier is the set of all Pareto efficient allocations, conventionally shown graphically. It also is variously known as the Pareto front or Pareto set.

In other words, in multi-objective optimization, when the different objectives are contradictory, an optimal solution is said Pareto optimal when it is not possible to improve an objective without degrading the others. A Pareto optimal solution can then be seen as an optimal trade-off between the objectives. The set of all Pareto optimal solutions is called the Pareto front as it usually graphically forms a distinct front of points. Solutions which do not lay on the Pareto front are called Pareto dominated solutions. The figure 8.1 shows a typical convex Pareto front obtained when minimizing two objectives concurrently.

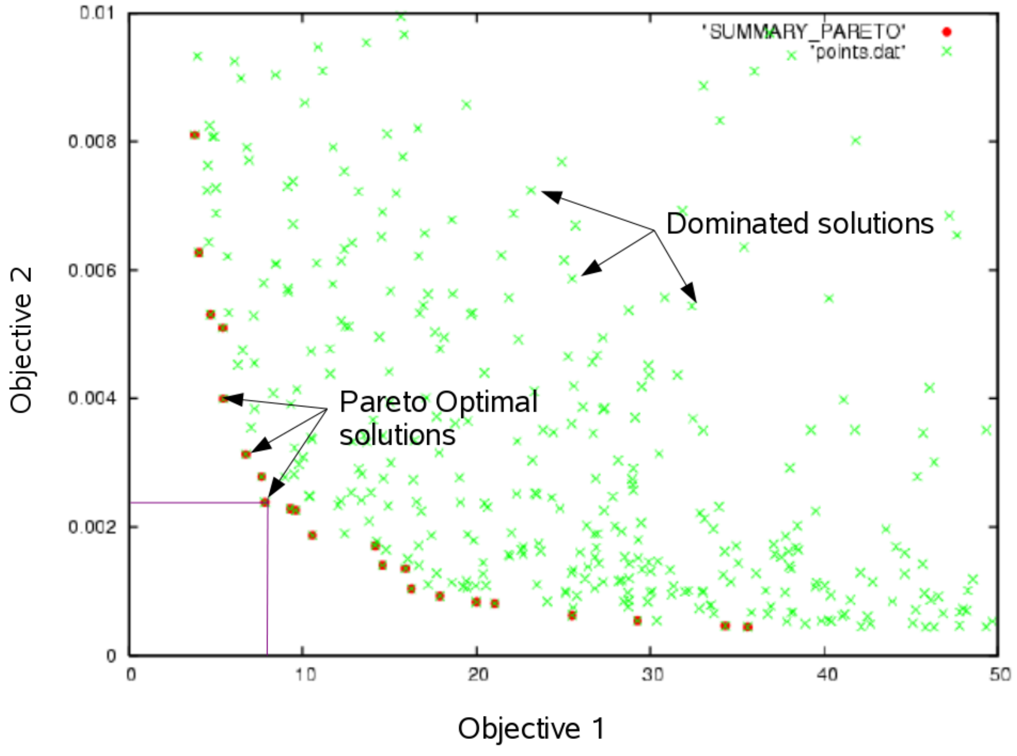


Figure 8.1: Pareto front sample

8.1.2 Pareto algorithm implementation

Some steps were made in order to implement the pareto criterion on the design space of a CNN. First of all, the raw data from the csv file need to be inserted to an appropriate Python structure.

```
network_profile_file = open(network_profile, ”r”)
profile_list = []

for line in network_profile_file:
  profile_list.append(line)
del profile_list[0:2]

network_profile_file.close()
```

Listing 8.1: Reading the raw data profile table

The snippet above, reads the CSV file that contains all raw information about the layers of a network and appends it to a list, called ”profile_list”. The first two elements contain junk information included in the CSV file (the axis names) and therefore they are deleted.

Afterwards, a list of dictionaries is created. Each element of that list is a dictionary that represents a layer, and holds 4 lists: A list for execution time, energy consumption, operation of the computation (e.g. Conv-Direct) and number of shaves used. These four lists are iterated concurrently.

```python
for iteration, line in enumerate(profile_list):
  line = line.split('\t')
  list_of_plane_points = []
  shaves_list = []
  time_list = []
  energy_list = []
  operation_list = []
  try:
    if int(line[0]) == layer_list[len(layer_list) - 1]['Id']:
      for line_offset in range(2, 14):
        shaves_list.append(line_offset - 1)
        operation_list.append(str(line[1]))
        time_list.append(float(line[line_offset]))
        energy_list.append(float(line[line_offset + 13]) * float(line[
  line_offset]))
      layer_list[len(layer_list) - 1]['shaves'] += shaves_list
      layer_list[len(layer_list) - 1]['Operation'] += operation_list
      layer_list[len(layer_list) - 1]['Time'] += time_list
      layer_list[len(layer_list) - 1]['Energy'] += energy_list
    else:
      for line_offset in range(2, 14):
        shaves_list.append(line_offset - 1)
        operation_list.append(str(line[1]))
        time_list.append(float(line[line_offset]))
        energy_list.append(float(line[line_offset + 13]) * float(line[
  line_offset]))
      layer_list.append({'Id': int(line[0]), 'shaves': shaves_list, '
  Operation': operation_list, 'Time': time_list, 'Energy': energy_list})
  except IndexError:
    for line_offset in range(2, 14):
      shaves_list.append(line_offset - 1)
      operation_list.append(str(line[1]))
      time_list.append(float(line[line_offset]))
      energy_list.append(float(line[line_offset + 13]) * float(line[
  line_offset]))
    layer_list.append({'Id': int(line[0]), 'shaves': shaves_list, '
  Operation': operation_list, 'Time': time_list, 'Energy': energy_list})

del profile_list[:]
```

Listing 8.2: Appending profile list to appropriate dictionaries

Notice that the code above is designed to recognize infinite number of different convolution methods (e.g. Direct and Im2Col and Winograd etc), and combine them appropriately to generate optimal configurations taking them all into account for each layer, meaning that this is a generalized implementation for any number of different computations per layer and can work as is for any future extension.

Now that the layer_list contains all information for each discrete layer in appropriate format and order, the only thing left before applying the Pareto optimality criterion, is to take all linear combinations between these 4 lists of each layer and produce network configurations, instead of layer configurations.

This is achieved with the following snippet:

```
for node in layer_list:
  aligned_time_list.append(node['Time'])
  aligned_energy_list.append(node['Energy'])
  aligned_operation_list.append(node['Operation'])
  aligned_shave_list.append(node['shaves'])

for time_network_config in product(*aligned_time_list):
  time_sum = 0
  for time_layer_config in time_network_config:
    time_sum += time_layer_config
  time_config_list.append(time_sum)

for energy_network_config in product(*aligned_energy_list):
  energy_sum = 0
  for energy_layer_config in energy_network_config:
  energy_sum += energy_layer_config
  energy_config_list.append(energy_sum)

for operation_network_config in product(*aligned_operation_list):
  operation_sum = ''
  for operation_layer_config in operation_network_config:
    operation_sum += '{0}-'.format(str(operation_layer_config))
  operation_config_list.append(operation_sum)

for shave_network_config in product(*aligned_shave_list):
  shave_sum = ''
  for shave_layer_config in shave_network_config:
  shave_sum += '{0}-'.format(str(shave_layer_config))
  shave_config_list.append(shave_sum)
```

Listing 8.3: Combining the layer data into network configurations

It is worth mentioning that combining the lists of each layer, with all possible combinations, leads to four lists that each one contains 12^N elements, where N is the number of layers combined. The base of the exponent is obviously twelve, since the size of each layer sub-list is a size of twelve (standing for 12 SHAVES).

Finally, the lists that hold all the possible configurations within the feasible region, can be fed to the criterion, in order for the optimal configurations to be produced.

```
energy_pareto_points.append(energy[0])
time_pareto_points.append(time[0])
operation_pareto_points.append(operation[0])
shave_pareto_points.append(shave[0])

x_reference = energy[0]
y_reference = time[0]

for x, y, op, shv in zip(energy, time, operation, shave):
  if x < x_reference:
    energy_pareto_points.append(x)
    time_pareto_points.append(y)
    operation_pareto_points.append(op)
    shave_pareto_points.append(shv)
    x_reference = x
    y_reference = y
```

Listing 8.4: Applying pareto criterion to the design space

Now there are distinct lists that contain the optimal configurations, and those that contain all the non-optimal configurations. These lists are exploited in order to plot this information and also append them to a config_log file.

```
fig_pareto = plt.figure()
plt.plot(layer_energy_sub_points, layer_time_sub_points, 'rx', mew = 0.5,
    ms = 4.0)
plt.plot(energy_pareto_points, time_pareto_points, 'b', linewidth =
    '0.8')
plt.plot(energy_pareto_points, time_pareto_points, 'bx', mew = 0.8, ms =
    4.0, mfc = 'none')
plt.grid(b = True, which = 'major', axis = 'both', linestyle = '--',
    linewidth = '0.6', animated = True)
fig_pareto.suptitle('{0} Pareto Plot Time/Energy'.format(str(network_name
    )))
plt.xlabel('Energy consumption (mJ)')
plt.ylabel('Execution time (ms)')
fig_pareto.savefig('./{0}_pareto_plot.png'.format(str(network_name)), dpi
    = 1200)
```

Listing 8.5: Applying pareto criterion to the design space

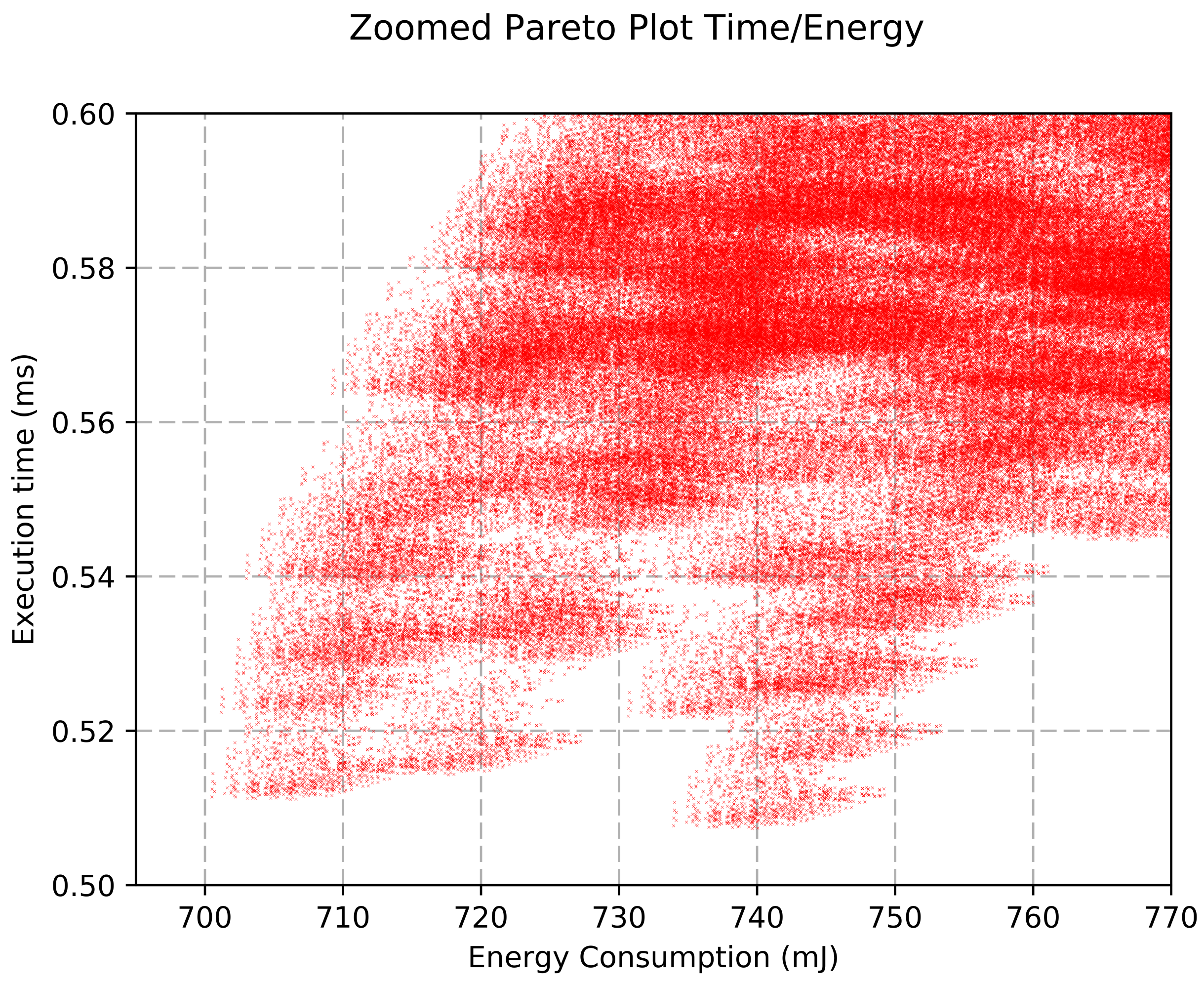


Figure 8.2: Pareto front of Lenet-MNIST

8.2 Pruning of the Design Space

As already mentioned, the size of the design space that occurs by calculating all linear combinations from N layers is 12^N elements. It is obvious that this size increases sharply, leading very soon to sizes that no modern computer or memory can withstand. For example, the design space of Lenet-MNIST, which is considered a tiny CNN consisted of only six layers, needs $4 \cdot 12^6$ 32-bit elements to be stored. This is equal to 45.56 MB of memory, which is already a considerable size, taking into account how small Lenet is.

However, this size escalates extremely quickly. Alexnet, a small to medium sized network, is consisted of thirteen layers. The design space of that network, as a result is equal to $4 \cdot 32 \cdot 12^{13}$ Bytes or 1.63×10^9 Gigabytes. It is obvious that this size is enormously huge and under no circumstances can it be either stored in any RAM, or calculated by any high-end processor. GoogleNet, which is considered the largest widely used CNN, is consisted of 74 computational layers (not including utility layers like Concat). With 74 layers, the design space has a size of $4 \cdot 32 \cdot 12^{74} = 9.26 \times 10^{81}$ Bytes, which is an unimaginable number!

This serious challenge is a heavy obstacle in the effort of finding optimal configurations of CNNs, as any widely used CNN with high accuracy uses at least 12 or 13 layers, therefore the necessity to tackle this problem was critical. In order to solve this problem, I came up with a lemma which is described and proved in the next subsection. Using this lemma, two different pruning algorithms were implemented and effectively succeeded to perform fast design space exploration, independable to the amount of layers a network was consisted of.

8.2.1 Pruning lemma

Case: Let a set named set A, which includes n N-subsets N_1, N_2, ... , N_n, so as $N_1 \cup N_2 \cup .... \cup N_n = A$. Each of these subsets contains k-dimensional points (d_1, d_2, ... , d_k). Suppose a different set B. Set B is consisted of the points that represent all the possible linear combinations that occur, if one point from each subset N_i is chosen and summed across dimensions with every possible point from the other N-subsets. For example, if a_1 is a k-dimensional point belonging to N_1, a_2 belongs to N_2 ... a_n belongs to N_n, then there is only one point b in set B which is k-dimensional and is equal to:

$$\begin{cases} b_{d1} = a_1^{d1} + a_2^{d1} + ... + a_n^{d1} \\ b_{d2} = a_1^{d2} + a_2^{d2} + ... + a_n^{d2} \\ \dots \\ b_{dk} = a_1^{dk} + a_2^{dk} + ... + a_n^{dk} \end{cases}$$

Since the points in set B are k-dimensional, the pareto optimality criterion can be applied onto them, regarding all k-dimensions.

Lemma: If j N-subsets, with $j < n$, are combined to produce a set of k-dimensional points, then, imperatively, a group of them will be pareto optimal against the others on the k-dimensional design space. Then, it is impossible for the combinations that led to non-optimal points of this subgroup (of the j N-subsets), to belong in a pareto optimal combination of set B.

Proof: Let C1, C2 two combinations of j N-subsets, $j < n$ and C1 being pareto optimal in k-dimensional space against C2. Then, according to the definition of pareto optimality criterion:

$$\begin{cases} C1_1^{d1} + C1_2^{d1} + ... + C1_j d1 < C2_1^{d1} + C2_2^{d1} + ... + C2_j^{d1} \\ C1_1^{d2} + C1_2^{d2} + ... + C1_j d2 < C2_1^{d2} + C2_2^{d2} + ... + C2_j^{d2} \\ ... \\ C1_1^{dk} + C1_2^{dk} + ... + C1_j dk < C2_1^{dk} + C2_2^{dk} + ... + C2_j^{dk} \end{cases}$$

Any possible sub-combination L from the other (n-j) N-subsets, if combined with either sub-combination C1 or C2, gives a valid set A combination contained in set B.

For every L sub-combination combined with C1 and C2, leading to L1 and L2 combinations included in set B, the following is obviously true because of the former inequality above:

$$\begin{cases} C1_1^{d1} + C1_2^{d1} + ... + C1_j d1 + L_{j+1}^{d1} + L_{j+2}^{d1} + ... + L_n^{d1} < \\ C2_1^{d1} + C2_2^{d1} + ... + C2_j^{d1} + L_{j+1}^{d1} + L_{j+2}^{d1} + ... + L_n^{d1} \\ C1_1^{d2} + C1_2^{d2} + ... + C1_j d2 + L_{j+1}^{d2} + L_{j+2}^{d2} + ... + L_n^{d2} < \\ C2_1^{d2} + C2_2^{d2} + ... + C2_j^{d2} + L_{j+1}^{d2} + L_{j+2}^{d2} + ... + L_n^{d2} \\ ... \\ C1_1^{dk} + C1_2^{dk} + ... + C1_j dk + L_{j+1}^{dk} + L_{j+2}^{dk} + ... + L_n^{dk} < \\ C2_1^{dk} + C2_2^{dk} + ... + C2_j^{dk} + L_{j+1}^{dk} + L_{j+2}^{dk} + ... + L_n^{dk} \end{cases}$$

Which is equivalent to:

$$\begin{cases} L1^{d1} < L2^{d1} \\ L1^{d2} < L2^{d2} \\ ... \\ L1^{dk} < L2^{dk} \end{cases}$$

Which, using the definition of pareto optimality, means that combination L1 is pareto optimal, in the k-dimensional design space, against L2.

In practice, this means to the design space exploration task of the interface, that it is possible to divide the network into sub-groups of layers and apply the pareto optimality criterion distinctively into these small sub-groups, in order to a-priori exclude a large amount of sub-combinations. Then, the pruned sub-groups can be combined together to provide the optimal configurations of the whole network.

8.2.2 Discrete layer pruning

The first algorithm designed to implement the pruning of sub-networks is the discrete layer pruning algorithm. This algorithm applies the pareto optimality criterion on each layer, which means that it splits the network into unity sub-groups, that contain only one layer. Among the twelve possible points of a layer, only the pareto optimals are kept, whereas the others are excluded. Then, the pruned layers are combined together for the computation of the network pareto optimals. In figure 8.3 the significant reduction in the design space is shown.

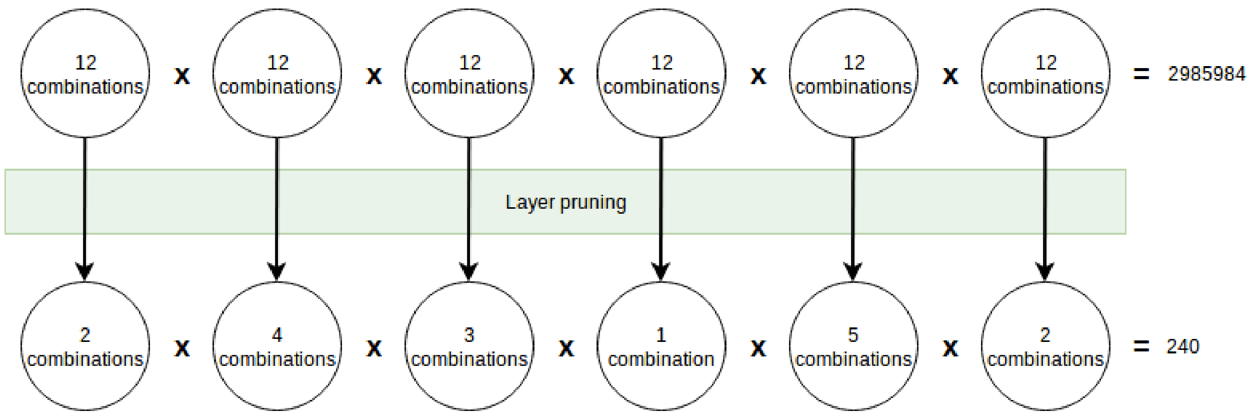


Figure 8.3: Discrete layer pruning example on 6 layers

This technique proved to be very effective on CNNs like VGG, AlexNet and medium sized networks. However, even this algorithm, when applied to large networks, like GoogLenet, could not reduce the design space up to the point on which its size could be computable and still remained prohibitive large.

Figure 8.4 provides the pareto plot of NiN-ImageNet, pruned with discrete layer pruning technique. Blue crosses represent the optimal points, whereas the red points represent the pruned non-optimal points.

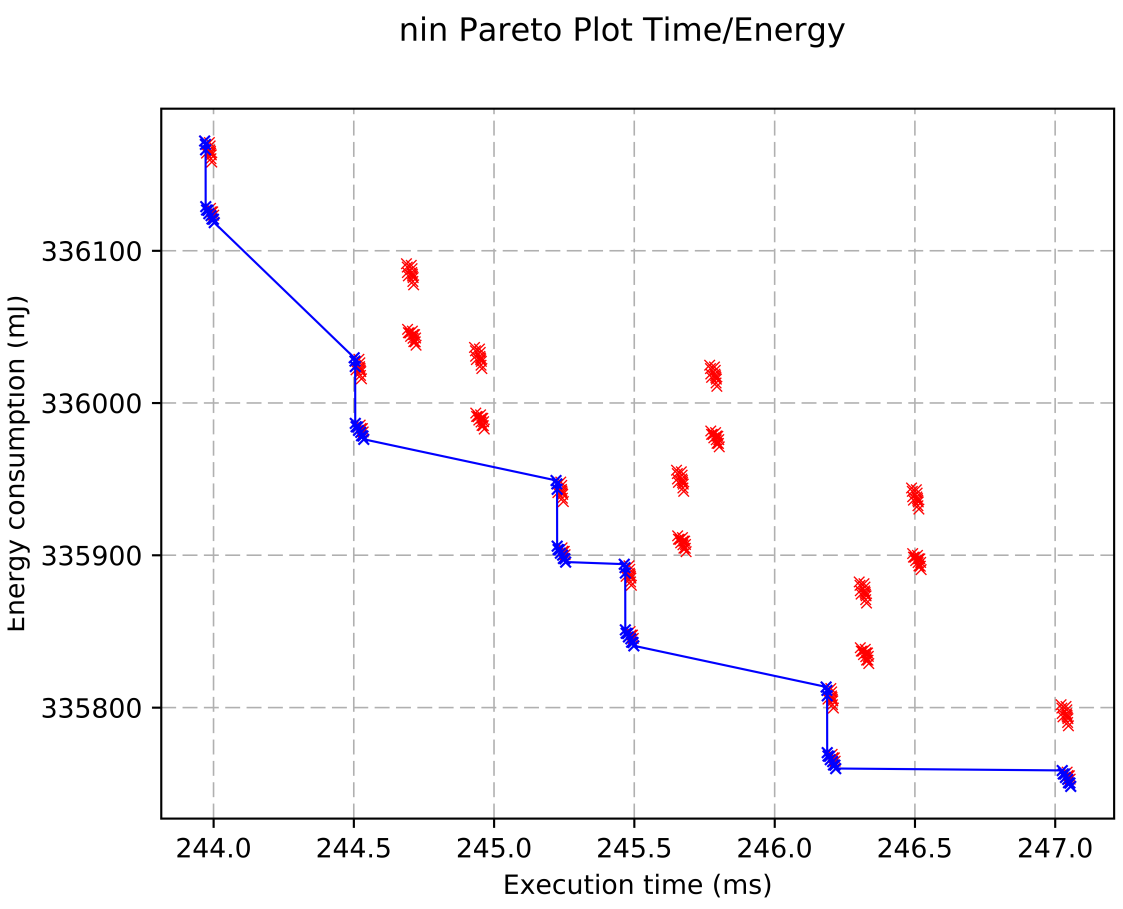


Figure 8.4: NiN-Imagenet network pruned with discrete layer pruning

8.2.3 Recursive pruning

A more sophisticated technique designed to approach globally the problem of the size of the design space is the recursive pruning algorithm. This algorithm implements binary search trees (BST) and a recursive function which instantiates new leaves. This algorithm eliminates fully non-optimal pareto points and outputs only optimals.

The main idea of this algorithm is that each node decides if the number of layers that it holds produce a design space whose size is computable. If not, the node divides the network that it holds into two chunks and provides through a recursion the left subnet to its left child-leaf and the right subnet to its right child-leaf. This recursion goes on until the leaves receive small enough chunks of network, capable to produce a computable size of design space. Then this space is produced and pruned, The leaf returns the computed pruned chunk to its parent. The idea of the algorithm is shown in figure 8.5 below:

In the code below, an abbreviated form of this described algorithm is provided.

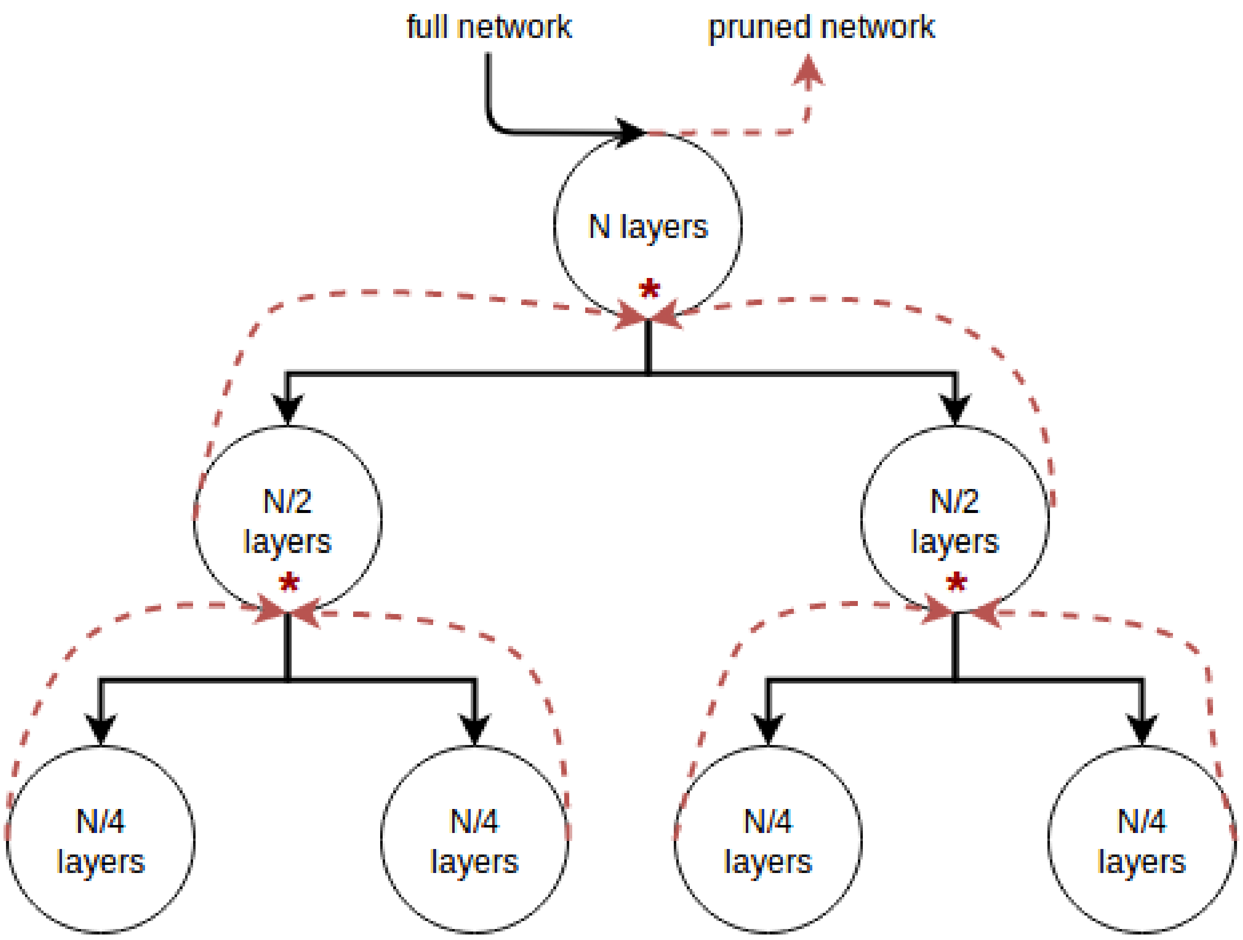


Figure 8.5: Illustration of the data flow of the pruned network with recursive pruning technique

```
def recursive_pareto_pruning(time_list, energy_list, operation_list,
    shave_list):
  pareto_points_length = 1
  initialize_lists()
  calculate_size_of_the_design_space
  if design_space_size < (12**4 + 1):
    generate_all_possible_combinations()
    calculate_pareto_optimal_points()
  else:
    call recursive_pareto_pruning(left_subnet)
    call recursive_pareto_pruning(right_subnet)
    combine_left_and_right_subnet()
    generate_all_possible_combinations()
    calculate_pareto_optimal_points()
  return time_pareto_points, energy_pareto_points,
    operation_pareto_points, shave_pareto_points
```

Listing 8.6: Recursive pareto algorithm illustration

In figure 8.6 the output of recursive pruning and design space exploration on ”SqueezeNet” is shown:

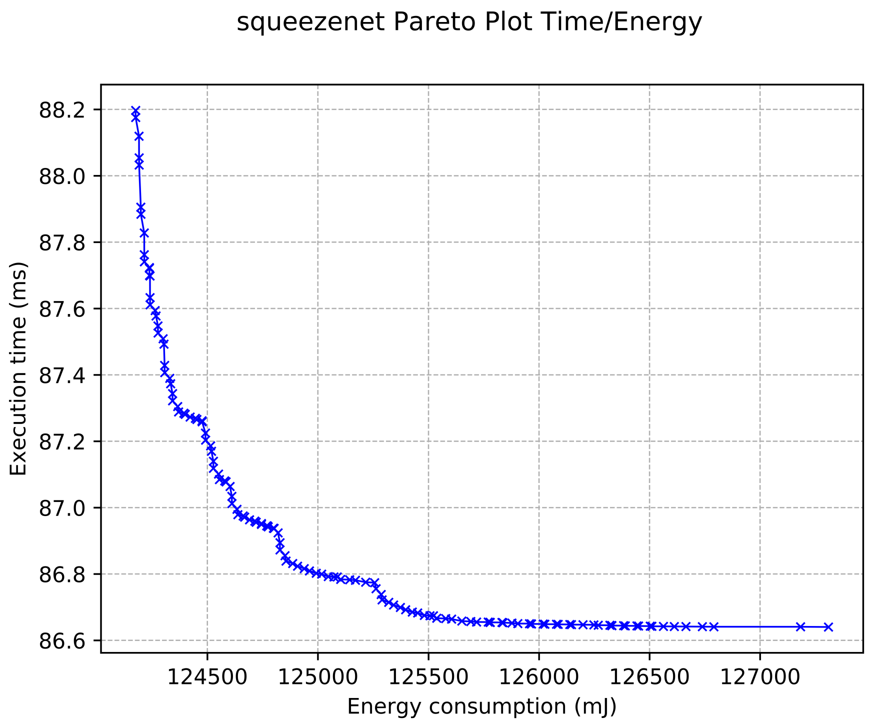


Figure 8.6: Pareto plot of SqueezeNet using recursive pruning

8.3 Exploration for concurrent layer execution

In Myriad2, the prevailed method of applying computations to the SHAVES, such as deploying deep neural networks, is to dispatch layers only from LEON OS to the SHAVES, one at a time. Most of the times, the number of SHAVES needed to achieve the best balance between execution time and energy consumption is the maximum available. This fact reinforces the attitude of using only LEON OS, since all SHAVES are utilized at once.

However, LEON RT was being unused in this dispatching design. In networks with parallel branches of layers, where the explicit data flows are independent from one another, this could be very inefficient. So, I came up with the following question: Instead of executing layers one-by-one using the maximum number of SHAVES, what if we used half of the SHAVES to execute one layer and the other half to execute another data-independent layer? The underlying premise of this question was that maybe it would be faster to execute concurrently two layers with lower number of SHAVES, despite that their personal time would be increased, rather than executing each with the maximum number of SHAVES and summing their times. This is because in the concurrent case, the total time would be t = max(t_1, t_2), which means that there could be a very efficient overlap in their executions, leading to this time being lower than $t_1^{min} + t_2^{min}$.

Table 8.1: Execution time of two GoogLenet's parallel layers (ms)

ID-Type SHAVES	1	2	3	4	5	6	7	8	9	10	11	12
46-Conv_Direct	54.28	27.18	18.18	13.67	10.97	9.17	8.03	6.99	6.28	5.56	5.19	4.68
46-Conv_Im2Col	361.24	176.97	153.02	154.32	142.56	140.72	136.67	131.49	116.5	99.26	99.36	82.15
50-Conv_Direct	32.27	16.08	11.25	8.22	6.53	5.63	5.19	4.33	4.28	3.5	3.05	3.05
50-Conv_Im2Col	11.41	5.69	3.87	3.71	3.34	3.32	3.13	3.14	3.01	2.9	2.9	2.93

Notice that this technique is rather concerned with re-ordering the execution queue than changing the computation itself. Applying this method to Googlenet and Squeezenet led to a reduction in both execution time and energy consumption of up to 14% and 7% respectively. A real concept of the idea is shown in figure 8.7 and table 8.1.

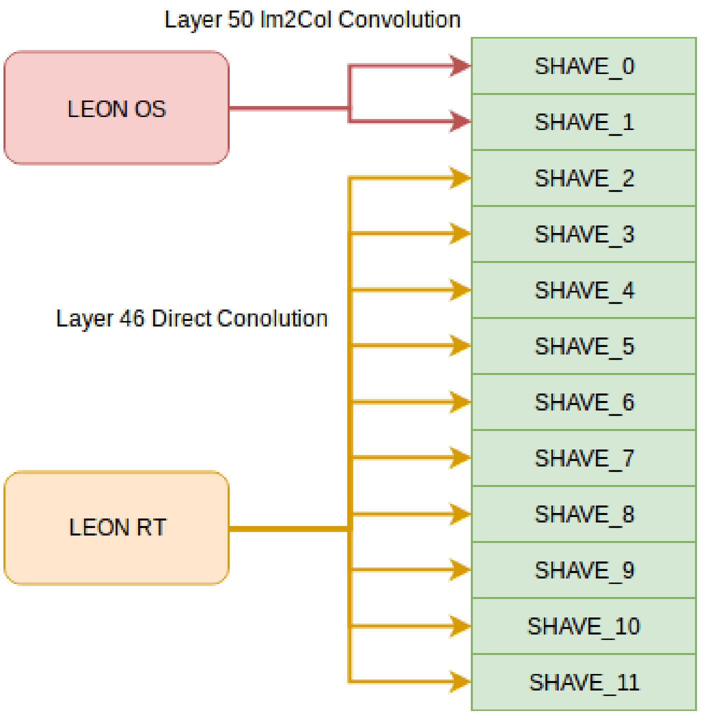


Figure 8.7: Concept of how two GoogLenet's layers were effectively executed concurrently

As seen from the example above, if one managing processor was used (LEON OS) to dispatch layers to the SHAVES, then layer 46 would be dispatched, allocating 12 SHAVES and "direct" method of Convolution. Then, layer 50 would be dispatched, using "Im2Col" method of Convolution, allocating again 11 SHAVES. This combination is the fastest and the total execution time would be:

$$t_{linear} = t_{46}^{12} + t_{50}^{12} = 4.68 + 2.9 = 7.58ms$$

On the other hand, on these specific layers, as an example, the concurrent exploration algorithm would produce a parallel configuration like the one shown in figure ??. If layer 46 with direct Convolution and 10 SHAVES, was dispatched at the same time with layer 50 with Im2Col Convolution and 2 SHAVES, then the total execution time would be:

$$t_{parallel} = max(t_{46}^{10}, t_{50}^{2}) = max(5.56, 5.69) = 5.69ms$$

It can be seen that, although the execution times of the individual layers were raised, they were utilized in a more efficient manner leading to an extraordinary reduction of 25% in their total execution time.

This algorithm takes advantage of any asymmetries between the layers that are in parallel branches and tries to find the best split-up among the RISC processors as well as the best order in which the layers must be organized and executed in order to achieve maximum performance. The concurrent dispatch is organized in rounds. Rounds do not necessarily contain only one layer per processor. It is a very common output of the algorithm, that while LEON OS executes one layer, LEON RT dispatches sequentially three different layers on the complementary SHAVES left.

The managing Python interface is responsible for reading the exported data and perform this special kind of exploration. Firstly, if the network is organized in blocks with layers in parallel, then these blocks are identified and stored exclusively, in order for this algorithm to be applied to each block separately. Then, each block is treated as a directed graph on which several graph-walking strategies are applied. These strategies are concerned with two parameters: A) How heavy or light must be the next node to be inserted and B) whether the last layers inserted in each traversal round must be kept or not to achieve the maximum execution overlap between the RISC processors. The algorithm includes an iteration feedback, which means that iteration by iteration, for each block, the algorithm converges to a local minima, providing the fastest graph traversal found.

An abstract pseudocode of the algorithm is provided:

```
For each parallel block in network:
  For each valid branch configuration among N RISC CPUs:
    For each graph traversal strategy:
      While layers inserted < total layers in block:
        Choose a reference layer. This layer belongs to one RISC CPU.
        Choose one coupling layer from every other RISC CPU.
        If there are not any other layers left:
          Execute reference layer alone
Else:
    For each processing unit number configuration among reference and
   coupling layers:
      While round times of all CPUs are not overlapping over a specific
   percentage:
        Add one more layer to the CPU with the lowest round time
    Among the latter iterations, choose the configuration with the lowest
```

```
     execution time
      Remove from the network the executed layers
        Store the block traversal rounds and the total execution time
  Among every branch configuration and graph traversal strategy
    iterations, choose the fastest
```

Listing 8.7: Concurrent exploration algorithm

Chapter 9

Evaluation and Experimental Results

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

9.1 LRN Evaluation

In this section, the LRN implementation performance will be analyzed, in order to examine its computational efficiency. Both the generic C function and the specific assembly routine will be benchmarked.

In tables 9.1 and 9.2 below, benchmarks, for the execution time and the average power consumption of the C and assembly implementation, for the LRN layers of ”GoogLenet” (G1 and G2) and ”AlexNet” (A1 and A2) are provided.

LRN layer SHAVES	1	2	3	4	5	6	7	8	9	10	11	12
LRN_G1(64x56x56)	159.9094	80.0401	53.382	40.082	32.156	26.897	23.98	21.966	20.501	19.189	18.409	17.702
LRN_G2(192x56x56)	480.591	240.387	160.4	120.36	96.541	80.741	71.201	65.82	61.21	57.473	55.04	53.89
LRN_A1(96x55x55)	231.583	115.857	77.336	58.084	46.65	39.06	34.463	31.277	28.902	27.792	26.567	24.889
LRN_A2(256x27x27)	149.017	74.639	49.762	37.493	30.013	25.059	22.031	20.435	19.039	17.845	17.059	16.325

LRN layer SHAVES	1	2	3	4	5	6	7	8	9	10	11	12
LRN_G1(64x56x56)	559.254	620.19	679.672	739.094	797.083	851.77	909.555	951.161	990.169	1029.976	1063.484	1087.498
LRN_G2(192x56x56)	559.352	620.407	680.558	739.52	794.679	858.315	909.749	942.764	989.155	1037.528	1051.573	1093.513
LRN_A1(96x55x55)	558.911	620.593	680.684	736.399	798.98	850.844	903.505	948.651	981.836	1029.645	1061.77	1093.083
LRN_A2(256x27x27)	558.774	619.791	680.265	737.771	798.56	859.768	912.331	955.426	990.337	1031.576	1065.101	1095.205

Table 9.1: Execution time (ms) and average power consumption (mW) of LRN C implementation

LRN layer SHAVES	1	2	3	4	5	6	7	8	9	10	11	12
LRN_G1(64x56x56)	5.382	2.725	1.863	1.421	1.177	1.015	0.872	0.771	0.706	0.646	0.603	0.565
LRN_G2(192x56x56)	16.374	8.228	5.557	4.202	3.435	2.925	2.515	2.29	2.053	1.945	1.768	1.659
LRN_A1(96x55x55)	7.77	3.926	2.667	2.033	1.679	1.418	1.268	1.133	1.037	0.942	0.871	0.803
LRN_A2(256x27x27)	5.105	2.596	1.791	1.367	1.141	0.987	0.835	0.754	0.71	0.647	0.597	0.565

LRN layer SHAVES	1	2	3	4	5	6	7	8	9	10	11	12
LRN_G1(64x56x56)	584.977	648.436	713.156	776.737	835.31	897.076	958.01	1015.175	1070.671	1123.518	1183.017	1236.038
LRN_G2(192x56x56)	582.417	647.702	714.395	779.622	844.132	908.222	976.85	1030.046	1095.75	1144.16	1207.782	1262.034
LRN_A1(96x55x55)	582.216	646.479	709.77	772.63	833.433	897.006	959.39	1012.004	1065.209	1116.59	1169.967	1224.901
LRN_A2(256x27x27)	587.957	657.748	729.161	796.095	856.176	919.854	971.253	1036.457	1096.091	1156.752	1216.643	1266.461

Table 9.2: Execution time (ms) and average power consumption (mW) of LRN Assembly implementation

As one can see, the difference in performance between the C and the assembly implementation is enormous, constituting the C function obsolete in any performance conscious application of the CNN engine. A chart comparison of the two computational methods for ”GoogLenet”, ”AlexNet” and ”ZFNet” is given in figures 9.1 and 9.2.

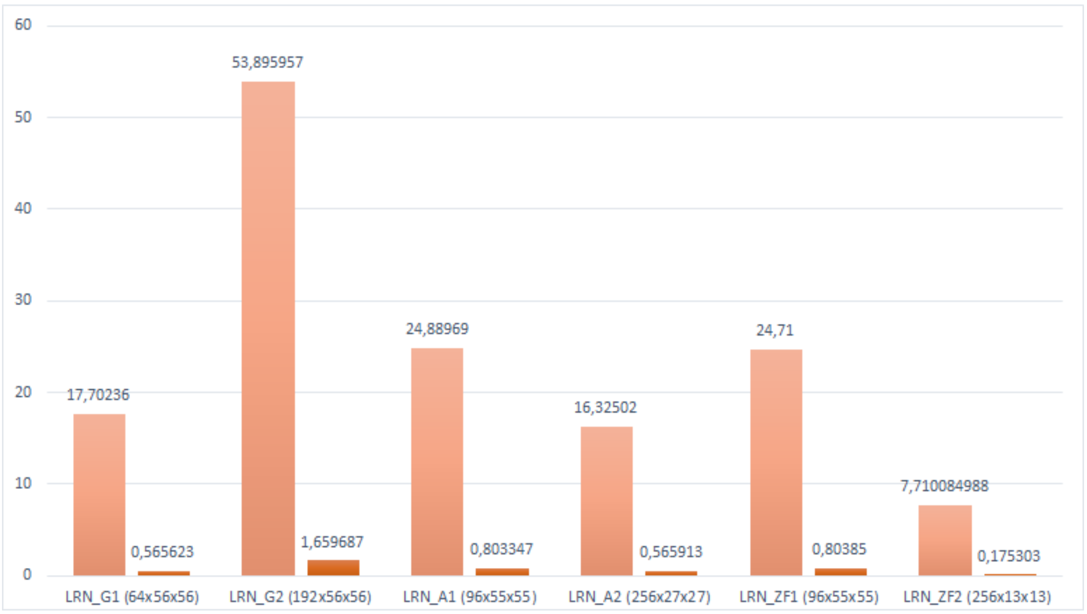


Figure 9.1: LRN layers C and Assembly time comparison

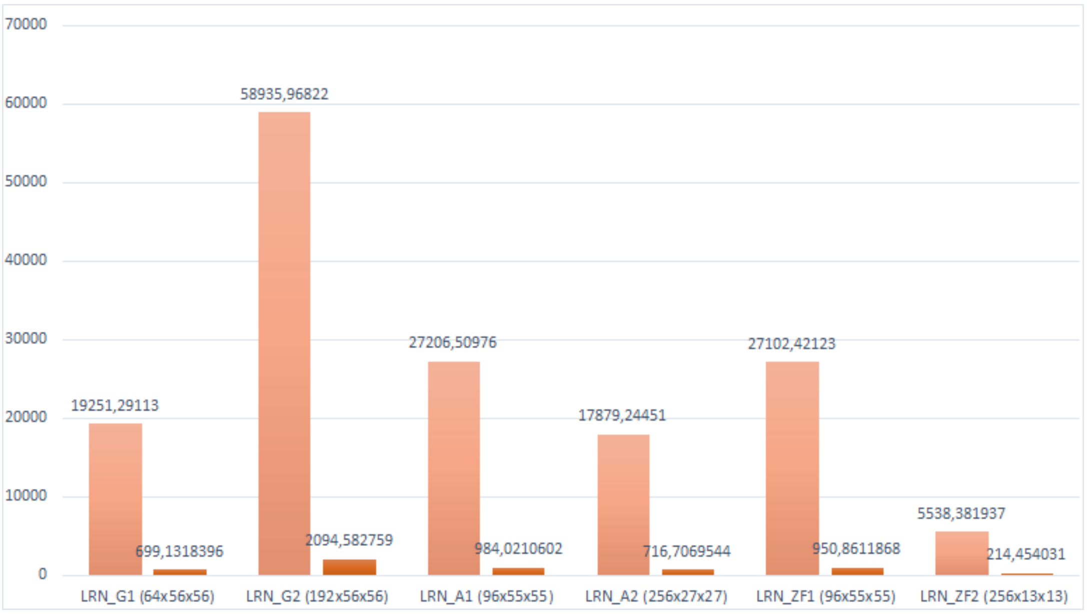


Figure 9.2: LRN layers C and Assembly energy comparison

Using the assembly implementation, the following table 9.3 shows the time needed to execute all LRN layers compared to the total network execution time, in ”GoogLenet”, ”AlexNet” and ”ZFNet”. It can be seen that the implementation is very efficient. Not only the time of the layer itself is low, but also it is insignificant compared with the total execution time of these networks.

	LRN layers	total layers	LRN time (ms)	total time (ms)	% LRN time
GoogLenet	2	83	2.23	203.053	1.09
AlexNet	2	13	1.369	97.69	1.4
ZFNet	2	13	0.976	98.94	0.98

Table 9.3: Time spent in LRN layers for GoogLenet, AlexNet, ZFNet

9.2 DMA Engine Evaluation

The CMX DMA controller resides between the 128-bit MXI bus and CMX memory[8]. It provides high bandwidth data transfers between CMX and DDR in either direction. It also supports data transfers from DDR back to DDR or from CMX to CMX, allowing data to be relocated within the same physical location.

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

In this section, the serving capacity of the DMA engine will be evaluated, in order to address its bottleneck. Two types of experiments were conducted to evaluate the DMA Engine:

- Increasing computational load as SHAVE number increases, keeping load per SHAVE constant
- Keeping computational load constant as SHAVE number increases, distributing it among the processors

9.2.1 Increasing computational load

In this case, it is obvious that if the DMA Engine could theoretically serve infinite requests and was independent for each distinct SHAVE, it would be expected for the total execution time to remain constant, as an increase in the SHAVE processors is accompanied with a proportionate increase in load. To evaluate this statement, three SHAVE algorithms were benchmarked:

- Only DMA transactions algorithm
- DMA transactions combined with SHAVE-independent calculations
- Only SHAVE calculations

These algorithms were selected, in order to evaluate how and in what extent does the DMA Engine affect the SHAVE computations. In figure 9.3, the performance for the algorithm using only DMA transactions is shown. The vertical axes represents the percentage of increase or decrease using as a reference the execution time for 1 SHAVE processor.

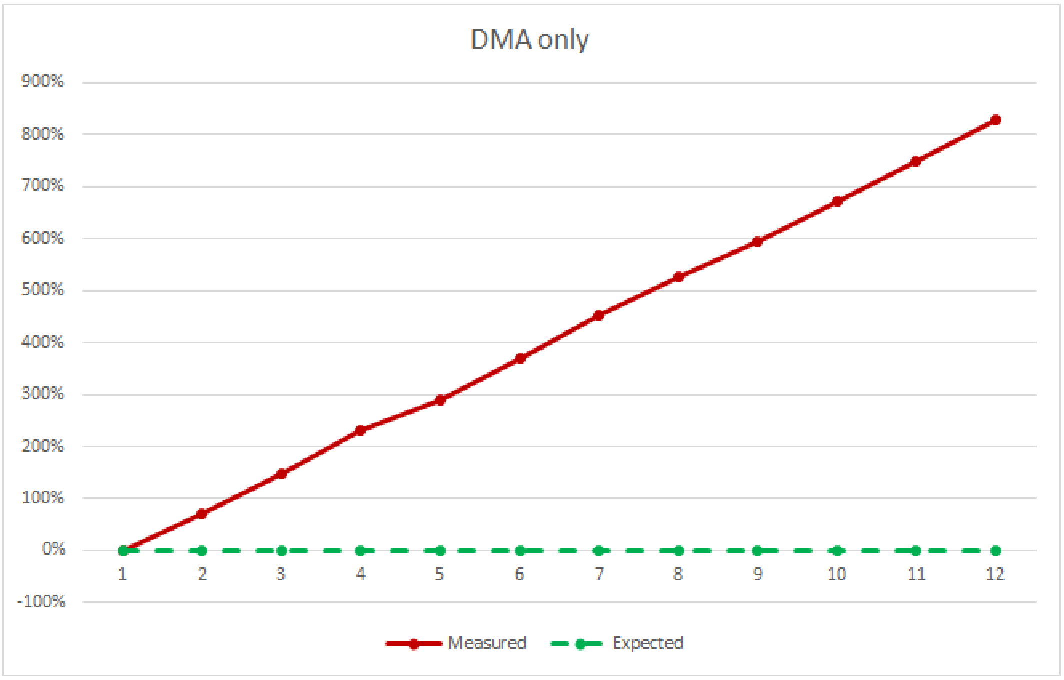


Figure 9.3: Only-DMA algorithm benchmark

It can be seen that the DMA Engine experiences a flooding in requests as the number of SHAVES rise. This graph, indicates the linear behavior of the controller. The rise is very sharp, however this is reasonable as the SHAVE algorithm consists only of DMA transactions, therefore any bottleneck in these transactions heavily affects the performance.

The next benchmark is conducted on an algorithm that performs both DMA transactions and pixel calculations on the blocks and is presented in figure 9.4.

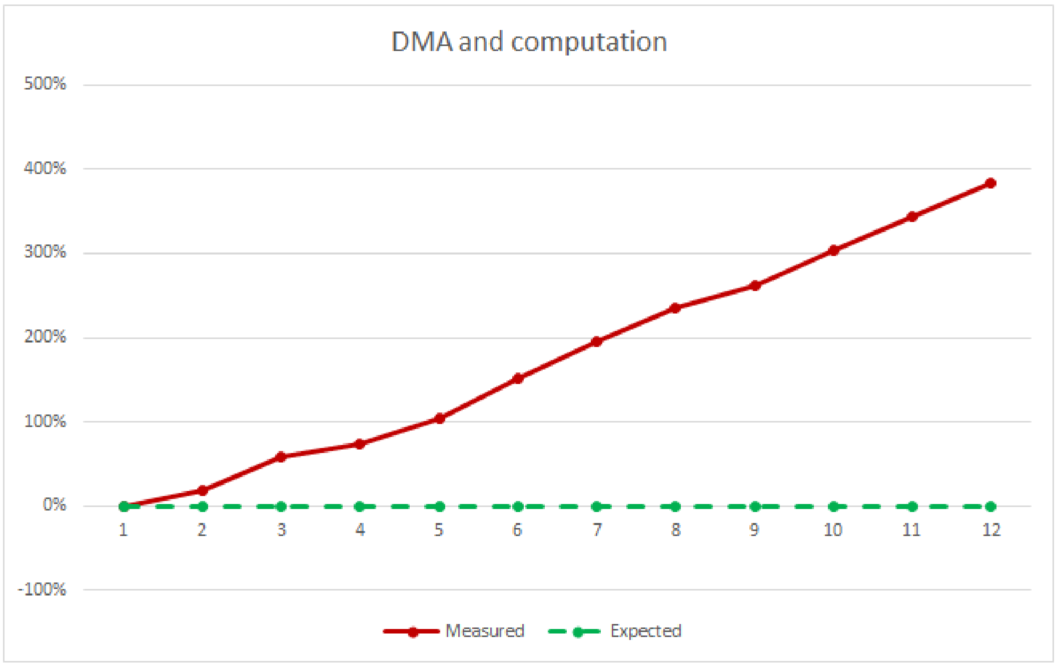


Figure 9.4: DMA transaction and pixel computation benchmark

As in the first case, again there is an increase in the execution time, despite the fact that it would be expected to remain constant. This increase is not as steep as it was before and this is because in this case, the DMA transactions affect the total algorithm by a smaller percentage, since each SHAVE does not only conduct DMA requests, but also performs matrix multiplications and additions.

Finally, an algorithm using only pixel calculations is benchmarked. In this case, the DMA Engine is not used at all and therefore, we expect the execution time to remain constant as the number of SHAVES increases. However, it can be noticed in figure 9.5 that, again, there is a smaller increase as the amount of SHAVES rises. This is very interesting, as it shows that despite the fact the SIMD processors are considered theoretically independent, there still is a resource that is used in common by all of them.

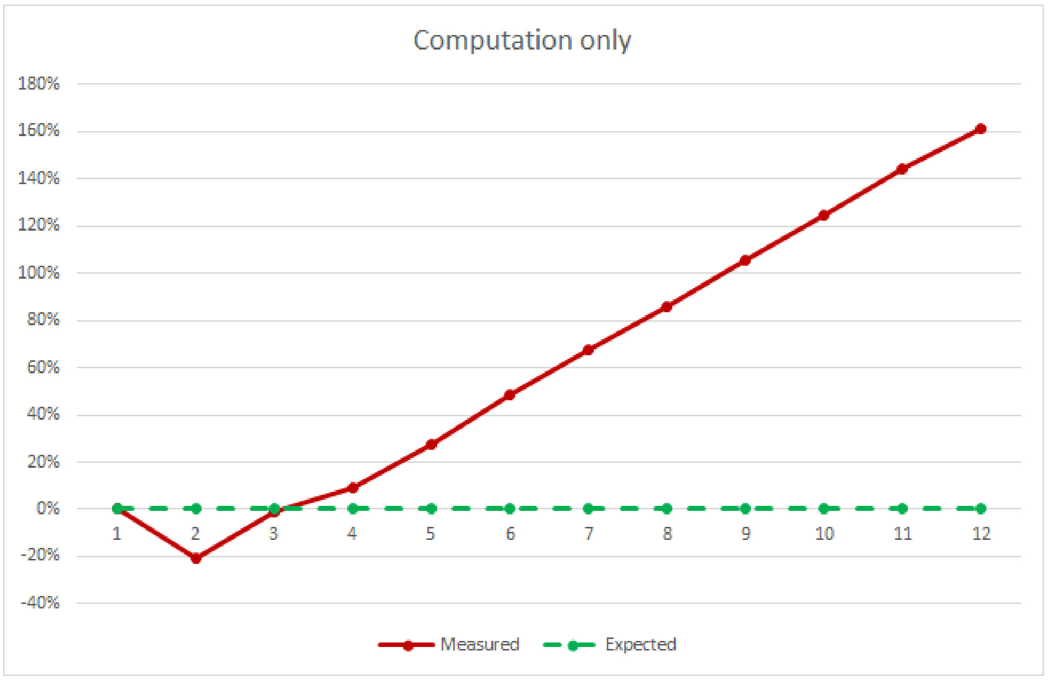


Figure 9.5: Pixel computation only benchmark

A comparison of the percentage increases for all three benchmarks is provided in figure 9.6, to give an insight of each benchmark's rate of increase.

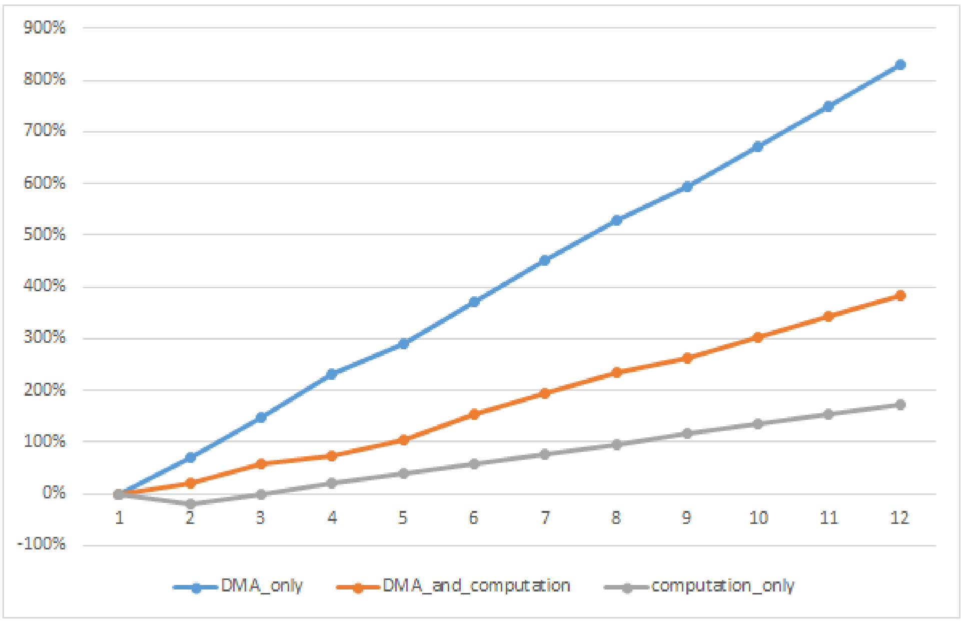


Figure 9.6: Increasing load benchmarks comparison

9.2.2 Constant computational load

In this case, the computational load remains constant, and each time, the number of SHAVES increases by one. This leads to a gradual reduction in the load per SHAVE, distributing the load more efficiently among the processors. Therefore, an analogous decrease in the execution time would be expected as the number of SHAVES increases. Although, the increasing number of DMA requests as the processing units increase, alters this expected behavior.

The algorithm that was benchmarked contains DMA requests the bring a block of memory to the CMX slices and computation on this block. As the number of SHAVES increases, the size of the block that each processor needs to compute, decreases. In figure 9.7 the performance of this algorithm is shown.

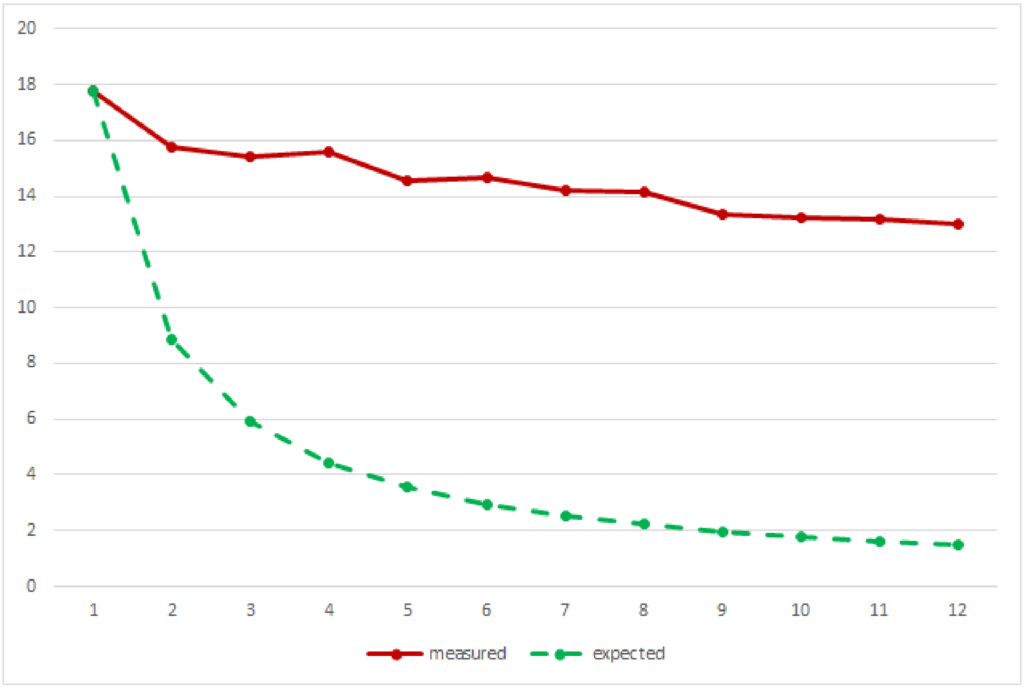


Figure 9.7: Distributed load benchmark

It can be seen that the measured curve strongly diverges from the expected. This deviation indicates the influence of the DMA engine bottleneck on the algorithm. Furthermore, the diminishing trend of the measured line shows that increasing the number of SHAVES on that particular algorithm, has a stronger influence on performance, than the flooding that occurs by the increasing DMA requests. Therefore, the conclusion of these benchmarks is that a designer should take care to write algorithms that contain as few DMA transactions as possible and also maximize the useful computations done on each block brought to the SHAVE local memory. In that way, one can fully exploit the advantages of the Myriad2 SIMD processors.

9.3 CNN Evaluation

9.3.1 Evaluation setup

The framework is evaluated using 6 CNNs of various complexity: AlexNet, GoogleNet, NiN-imagenet, SqueezeNet, VGG and ZFnet. The CNNs have been selected based on the following 2 criteria:

- To require various amount of computational resources and to provide significantly different execution time, energy consumption and accuracy results.
- To be considered state-of-the-art and be widely used for image recognition tasks.

CNN	input image	output vector	#layers	#memory(MB)	Error rate
AlexNet	$[277 \times 277 \times 3]$	$[1 \times 1000]$	13	117	17
GoogleNet	$[277 \times 277 \times 3]$	$[1 \times 1000]$	83	16.6	7
NiN-imagenet	$[277 \times 277 \times 3]$	$[1 \times 1000]$	16	15.5	17.5
SqueezeNet	$[277 \times 277 \times 3]$	$[1 \times 1000]$	38	4.68	19.7
VGG	$[277 \times 277 \times 3]$	$[1 \times 1000]$	16	276	8
ZFnet	$[277 \times 277 \times 3]$	$[1 \times 1000]$	13	121	16.5

Table 9.4: Details of CNNs used for evaluation

Table 9.4 shows the specifications of the above CNNs. All CNNs use the same input and are trained using ImageNet dataset for the same number of output classes, therefore they can be used interchangeably. The number of layers significantly ranges, from 13 (AlexNet and ZFnet) to 83 (GoogleNet). The same applies to the memory requirements, which refers the size that the weights and biases occupy in the global memory. Another important metrics is the top 5 error rate of each CNN, that ranges from 19.7 (SqueezeNet) to 8 (VGG). The proposed methodology will be used i) to provide fine-tuned implementations of the above CNNs in Intel/Movidius device and ii) to demonstrate trade-offs between the execution time, energy consumption and between the fine-tuned CNN implementations.

9.3.2 Computational scalability

Execution time was measured using recommended functions provided by the Myriad development kit. Energy consumption has been accurately calculated by using on-chip sensors that measure the current that flows through various power rails, provided by the Myriad2 evaluation board. Energy consumption measurement process is managed by the LEON RT, as shown in Figure 9.8. All experimental results refer to CNN inference execution time and operations are performed under the IEEE 754 fp16 standard. The hardware specification which is been explored in the context of this work is the number of Myriad VPUs in which each CNN layer is implemented.

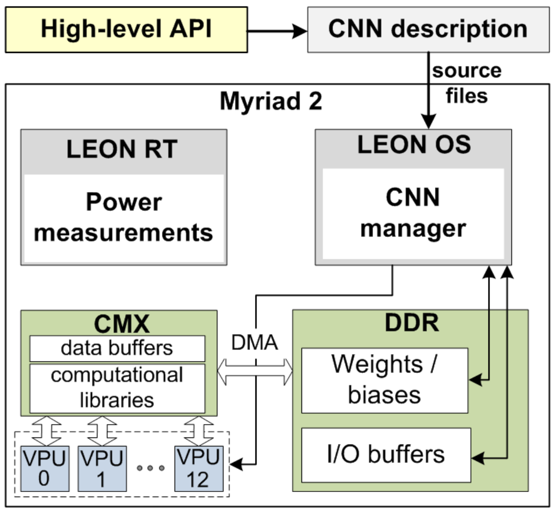
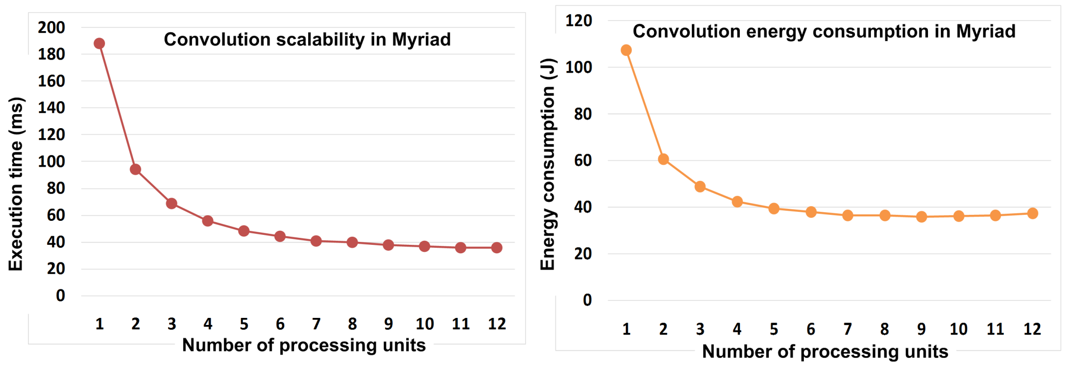


Figure 9.8: Framework instantiation in Intel/Movidius Myriad2

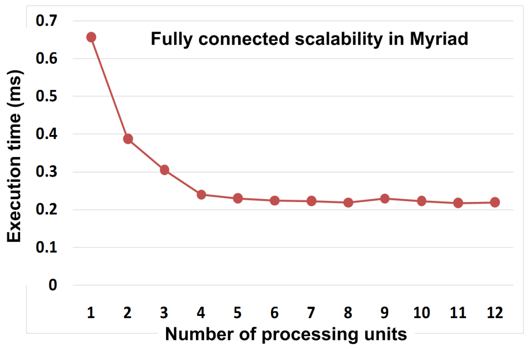


(a) Convolution execution time vs. number of processing units **(b)** Convolution energy consumption vs. number of processing units

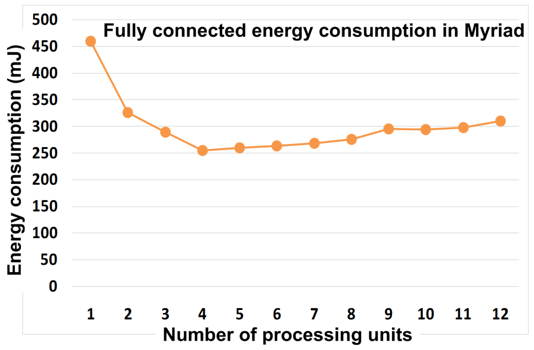
Figure 9.9: Scalability of convolution in Myriad2

Before applying the CNN deployment methodology in Myriad, we examine the scalability of the convolution and the fully connected layers. Figures 9.9 shows the execution time and the energy consumption of a 3x3 convolution, as the number of processing units increases gradually. Although the execution time continues to drop up to 12 cores, the energy consumption decreases up to 11 cores, while it slightly increases when using 12 cores.

With respect to the scalability of the fully connected layer (Figure 9.10), the execution time drops up to 4 VPUs. By utilizing more than 4 VPUs, the energy increases, without improvement in the execution time.



(a) Fully connected execution time vs. number of processing units



(b) Fully connected energy consumption vs. number of processing units

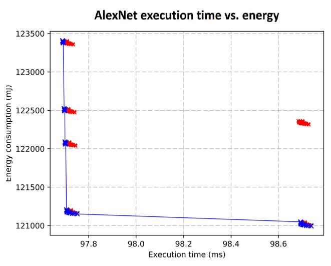
Figure 9.10: Scalability of fully connected in Myriad2

The execution time of convolutions is usually optimal on 11 or 12 VPUs. The reason, is the fact that since the convolution is computationally intensive, the execution time is dominated by the exploitation of parallelism and the communication overhead is only a small amount of the total execution time, even though a large number of VPUs is employed. On the other hand, in the fully connected, the computation part is trivial and the execution time is dominated by the communication overhead. Therefore, utilizing more than 3-4 VPUs it only adds communication overhead, with trivial benefits from the exploitation of parallelism.

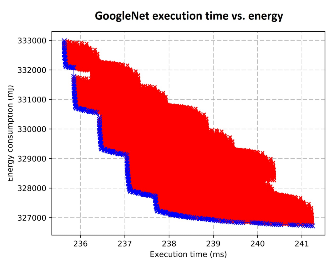
The above experiments underlying the need for exploration in order to identify the most efficient utilization of hardware resources for each layer. It is not possible for developers to be aware of the exact amount of VPUs in which each layer should be implemented in order to minimize execution time and energy consumption, without applying DSE techniques.

9.3.3 Design Space Exploration

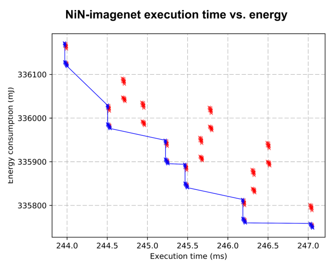
The proposed CNN deployment methodology has been applied to the implementation of the 6 CNNs of Table 9.4 on Intel/Movidius Myriad2. The output of the the first step of the methodology is shown in Figure 9.11. The Pareto plots for each CNN architecture for execution time vs. energy consumption are automatically provided by the framework that supports the methodology. Each point in each plot is a CNN implementation that utilizes a different amount of hardware resources. In other words, a CNN implementation differs from another one in the fact that at least one CNN layer utilizes a different number of VPUs.



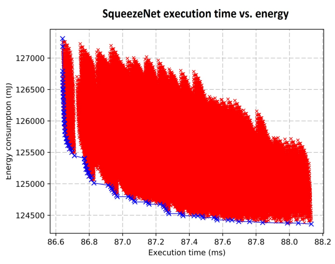
(a) AlexNet: Execution time vs. energy



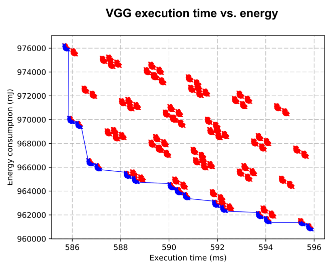
(b) GoogleNet: Execution time vs. energy



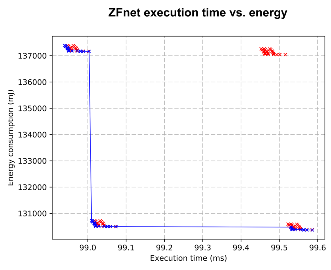
(c) NiN-imagenet: Execution time vs. energy



(d) SqueezeNet: Execution time vs. energy



(e) VGG: Execution time vs. energy



(f) ZFnet: Execution time vs. energy

Figure 9.11: Output of the 1st step of the methodology: Fine tuning of CNN implementations on an edge device

Trade-offs between execution time and energy consumption are identified for all CNN architectures. Detailed examination of results shows that the most computationally intensive layers are the convolution layers, as expected. Indeed, experiments shows that 68% up to 99% of the execution time of the CNNs of Table 9.4 is spent in convolution. Since convolution is a compute-bound operation, the corresponding layers tend to utilize 11 or 12 VPUs. However, as shown earlier in Figure 9.9b that is not always the most energy efficient solution.

An interesting observation is the fact that in AlexNet, NiN-imagenet, VGG and ZFnet, the implementations are clustered in groups. The results show that in the implementations that belong to the same cluster, the convolution layers have the exact same configuration (i.e. the corresponding convolution layers use the same number of VPUs). However, they differ in the number of VPUs used by the rest of the layers, such as the pooling and the fully connected, which have a relatively small impact in the execution time.

On the other hand, for the implementations that belong to different clusters, the convolution layers have different configurations. For example, AlexNet has 5 convolution layers. All the implementations of the bottom-right cluster (the most energy efficient) in the AlexNet Pareto curve in Figure 9.11a utilize 11 VPUs for the 1st layer and 12 for the rest ones. The implementations within this cluster differ in the number of VPUs utilized by a pooling layer. However, the implementations in the cluster above this one utilize 11 VPUs for the 3rd convolution, as well. Since the implementation of convolution has major impact both in the execution time and in the energy consumption, even a change in the implementation of a single convolutional layer only, affects the execution time and the energy consumption much more than the pooling and fully connected layers.

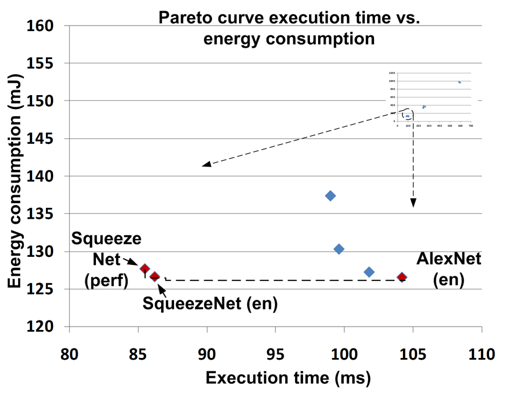
Table 9.5: Cumulative output results of the 1st step of the methodology

	AlexNet	GoogleNet	NiN-imagenet	SqueezeNet	VGG	ZFnet
Exec. time (ms)	97.69	203.05	243.96	78.08	585.66	98.93
% Exec. time gain	2.3	17.73	1.2	11.47	1.72	0.65
Energy consumption (J)	120.9	286.9	335.7	109.9	960.7	130.3
% Energy gain	1.2	13.8	0.12	13.6	1.53	5.11

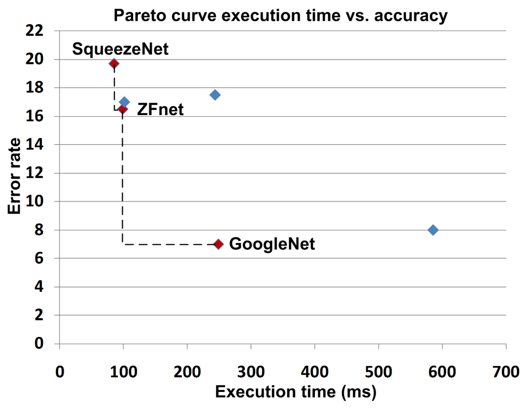
Table 9.5 shows the maximum gains in execution time and energy consumption between the most high performance and the most energy efficient implementations in each CNN, which reach 5.3%. The results of the second step of the methodology are presented in Figure 9.12. Pareto plots that demonstrate trade-offs between execution time, energy consumption and accuracy are presented. Figure 9.12a shows the Pareto curve for execution time vs. energy. The most efficient implementation in terms of execution time is the SqueezeNet, implemented for high performance (as obtained by the 1st step of the methodology). The rest of the Pareto points are the SqueezeNet and AlexNet, both tuned for energy efficiency. For instance, GoogleNet provides 258.6ms execution time and 7% error rate. However, by using SqueezeNet, error rate increases significantly, but execution time decreases by 65%. Similarly, switching from GoogleNet to AlexNet developers can trade accuracy (error rate increases from 7% to 17%) for energy consumption, which decreases by 63%.

Table 9.6: Comparison between straightforward and fine-tuned CNN implementations

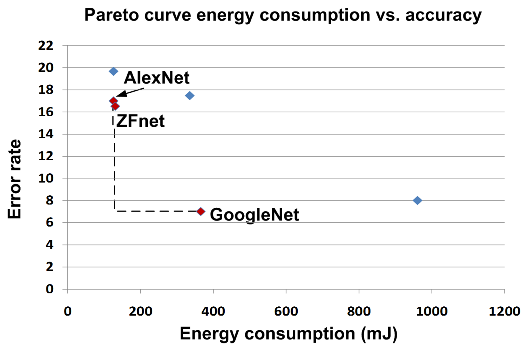
	AlexNet	GoogleNet	NiN-imagenet	SqueezeNet	VGG	ZFnet
Exec. time 12 VPUs (ms)	98.16	238.6	244	87	587.2	99.1
Exec. time fine-tuned (ms)	97.6	203.5	243	78	585.6	98.9
% Exec. time gain	0.48	14.9	0.01	10.37	0.27	0.19
Energy 12 VPUs (J)	133.7	343.6	337	129.3	1004	141.2
Energy fine-tuned (J)	120.9	286.9	335.7	109.9	960.7	130.3
% Energy gain	9.56	16.61	0.6	18.34	10.83	8.46



(a) Execution time vs. energy consumption



(b) Execution time vs. accuracy



(c) Energy consumption vs. accuracy

Figure 9.12: Output of the methodology: Trade-offs between execution time, energy consumption and accuracy between various CNNs

To further examine the benefits of DSE methodologies in comparison with straightforward solutions, Table 9.6 shows the execution time and energy consumption results of implementing all CNN layers in 12 VPUs. We consider the implementation on 12 VPUs as the baseline for comparison, since it is the one in which all available resources are fully utilized. The results show that by fine-tuning resource utilization through DSE, the execution time slightly improves in comparison with the straightforward approach. However, the energy consumption decreases significantly, up to 9%. This highlights the importance of fine-tuning CNN implementations, as performed by the 1st step of the methodology, before identifying trade-offs between execution time, energy and accuracy.

Chapter 10

Future Work

Throughout the course of the implementation of the CNN engine, several problems were confronted. The major goal of this thesis was to create general solutions as computationally efficient as possible and applicable to a wide range of CNN implementations and architectures. Another important purpose served by this implementation, was to be used as the foundation of new software engineering tasks, introducing new features on the engine. These features could be extensions of the current design principle (e.g. extending to new layers) or exploit more hardware units and sensors of Myriad2 to provide new run-time capabilities.

For a future extension, the following seem to be of the greatest importance:

- *Extend the supported Caffe layers.* Although all major layers needed to deploy state-of-the-art Convolutional Neural Networks have already been implemented, there are also some secondary layers that are used in a specialized CNN category, mainly used for speech recognition: R-CNNs (Recurrent CNNs or RNNs). The layers needed to be implemented for these networks are the ”Scale”, ”Crop” and ”Eltwise” (element-wise) layers [24].

- *Exploit Myriad2 sensors for run-time adaptation.* Myriad2 contains on-chip camera module, as well as battery level and proximity sensor. The camera could be used to extend the engine to a real-time object recognition application with a specific fps rate. The sensors could be used to create a real-time adaptation: Low-battery levels would lead the application to switch to more conservative configurations, or even switch to lighter networks, whereas a motion estimation algorithm could put the application to sleep if the camera's pictures on a specific time frame are alike, as this would mean that the camera is still.

- *Implement more convolutional techniques.* Designing new convolution methods (e.g. Winograd) would increase the design space and lead to a bigger variety of possible layer configurations. This could lead to major performance gains, since convolution layers are the major building block of CNNs.

Bibliography

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

[2] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. Deep convolutional network evaluation on the intel xeon phi: Where subword parallelism meets many-core. Master Thesis, Eindhoven University of Technology, 2016.

[3] Gaurav Raina. Deep learning. MIT Press, 2016. http://www.deeplearningbook.org.

[4] *Convolutional Neural Networks (CNNs): An Illustrated Explanation. http://xrds.acm.org/blog/2016/06/convolutional-neural-networks-cnns-illustrated-explanation/*, accessed June 5, 2018.

[5] Andrej Karpathy. Stanford university cs231n: Convolutional neural networks for visual recognition.

[6] Y. Jia, E. Shelhamer, J. Donahue, S. Karayev, J. Long, R. Girshick, S. Guadarrama, and T. Darrel. Caffe: Convolutional architecture for fast feature embedding. ArXiv e-prints, 2014.

[7] *Myriad 2 MA2x5x Vision Processor. https://uploads.movidius.com/1463156689-2016-04-29_VPU_ProductBrief.pdf*, accessed June 5, 2018.

[8] Movidius Ltd. In *Movidius Myriad2 MA245x Databook (under non-disclosure license).*

[9] Movidius Ltd. Movidius myriad2 development kit: Programmer's guide (under non-disclosure license).

[10] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems 25*, pages 1097–1105. Curran Associates, Inc., 2012.

[11] Matthew D Zeiler and Rob Fergus. Visualizing and understanding convolutional networks. In *Computer Vision and Pattern Recognition.* arXiv:1311.2901, 2013.

[12] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. arXiv:1409.1556, 2015.

[13] Min Lin, Qiang Chen, and Shuicheng Yan. Network in network. arXiv:1312.4400, 2014.

[14] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. Going deeper with convolutions. arXiv:1409.4842, 2014.

[15] Kaiming He and Jian Sun. Convolutional neural networks at constrained time cost. arXiv:1412.1710, 2014.

[16] Kaiming He and Jian Sun. Open group pilots embedded real-time posix conformance testing. 2005.

[17] Kaiming He and Jian Sun. Linkers and loaders. San Francisco, Calif. u.a, 2000.

[18] Bjarne Stroustrup. In *The C++ Programming Language*. Addison-Wesley, 1985.

[19] David Mayhew and Dov Bulka. In *Efficient C++: Performance Programming Techniques*. Addison-Wesley, 1999.

[20] *http://caffe.berkeleyvision.org/tutorial/layers/lrn.html*, accessed June 5, 2018.

[21] *http://caffe.berkeleyvision.org/tutorial/layers/concat.html*, accessed June 5, 2018.

[22] *http://caffe.berkeleyvision.org/tutorial/layers/split.html*, accessed June 5, 2018.

[23] *http://caffe.berkeleyvision.org/tutorial/layers/dropout.html*, accessed June 5, 2018.

[24] *http://caffe.berkeleyvision.org/tutorial/layers.html*, accessed June 5, 2018.

[25] S. Basu, S. Ganguly, S. Mukhopadhyay, R. DiBiano, M. Karki, and R. Nemani. Deepsat - a learning framework for satellite imagery. ArXiv e-prints, 2015.