Εθνικο Μετσοβιο Πολυτεχνειο

Σχολη Ηλεκτρολογων Μηχανικων και Μηχανικων Υπολογιστων

Τομεας Τεχνολογιας Πληροφορικης και Υπολογιστων

Εργαστηριο Μικροϋπολογιστων και Ψηφιακων Συστηματων

Embedded Development of AI-based Computer Vision: Acceleration on Intel Myriad X VPU

Διπλωματικη Εργασια

Παναγιώτης Μηναΐδης

Επιβλέπων: Δημήτριος Σούντρης
Καθηγητής Ε.Μ.Π.

Εργαστηριο Μικροϋπολογιστων και Ψηφιακων Συστηματων

Αθήνα, Νοέμβριος 2021

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

Embedded Development of AI-based Computer Vision: Acceleration on Intel Myriad X VPU

Διπλωματικη Εργασια

Παναγιώτης Μηναΐδης

Επιβλέπων: Δημήτριος Σούντρης
Καθηγητής Ε.Μ.Π.

Εγκρίθηκε από την τριμελή εξεταστική επιτροπή την 8η Νοεμβρίου 2021.

(Υπογραφή)	*(Υπογραφή)*	*(Υπογραφή)*
.....	.....	.....
Δημήτριος Σούντρης	Γεώργιος Γκούμας	Παναγιώτης Τσανάκας
Καθηγητής Ε.Μ.Π.	Καθηγητής Ε.Μ.Π.	Καθηγητής Ε.Μ.Π

Αθήνα, Νοέμβριος 2021

(Υπογραφή)

.....

Παναγιωτης Μηναϊδης

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

© 2021 – All rights reserved

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

Copyright © –All rights reserved Παναγιώτης Μηναΐδης, 2021.
Με επιφύλαξη παντός δικαιώματος.

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

Ευχαριστίες

Με την παρούσα διπλωματική κλείνει ένας πενταετής κύκλος σπουδών, γεμάτος γνώση και μόχθο. Θα ήθελα να ευχαριστήσω, αρχικά, τον επιβλέποντα καθηγητή, κ. Δημήτριο Σούντρη, για την ώθηση του να ασχοληθώ με σύγχρονα ενσωματωμένα συστήματα, συντελλώντας καθοριστικά στην εξέλιξή μου από φοιτήτη σε μηχανικό. Στη συνέχεια, θα ήθελα να ευχαριστήσω τον υποψήφιο διδάκτορα, Βασίλειο Λέων, για τη συνεχή του βοήθεια, η οποία ήταν καταλυτική για την εκπόνηση αυτής της εργασίας, και τον μεταδιδακτορικό ερευνητή, Γεώργιο Λεντάρη, για τις κατατοπιστικές του παρατηρήσεις και προτάσεις, οι οποίες διαμόρφωσαν το θέμα και το ύφος της παρούσας διπλωματικής. Θα ήθελα ακόμη να ευχαριστήσω τους φίλους μου, Παναγιώτη, Ντέιβιντ, Ιωσήφ, Χρήστο, Γιώργο και Αφροδίτη για την ανιδιοτελή στήριξη τους. Τέλος, θα ήθελα να αφιερώσω την εργασία αυτή στους γονείς μου και να τους ευχαριστήσω για την ανεξάντλητη αγάπη τους, τη στήριξη και τις διδαχές που μου προσέφεραν, και εξακολουθούν να μου προσφέρουν, ο ένας επί της Γης, και ο άλλος από ψηλά.

Περίληψη

Υπολογίζεται ότι μέχρι το 2022, το 82% των πακέτων που διακινούνται στο διαδίκτυο θα αφορά δεδομένα βίντεο. Η εκμετάλλευση των δεδομένων αυτών σε πραγματικό χρόνο αποτελεί μία ελκυστική προοπτική, που μπορεί να οδηγήσει στη δημιουργία πολύ ενδιαφέροντων συστημάτων, εμπορικών ή μη. Σημαντικό εργαλείο σε αυτή την κατεύθυνση, καθίστανται τα συνελικτικά νευρωνικά δίκτυα (ΣΝΔ / CNNs), καθώς η ανάπτυξη που έχουν δει τα τελευταία χρόνια έχει οδηγήσει σε εντυπωσιακά αποτελέσματα σε κλασικά προβλήματα του πεδίου της όρασης υπολογιστών. Παρόλα αυτά, τα παραδοσιακά ενσωματωμένα συστήματα δεν μπορούν να υποστηρίξουν τις αυξημένες απαιτήσεις σε υπολογιστικούς πόρους και μνήμη των ΣΝΔ. Στο περιβάλλον αυτό αναπτύσσεται μία ανερχόμενη κλάση μικροεπεξεργαστών, τα υπολογιστικά συστήματα όρασης (VPUs).

Η Myriad X αποτελεί την πιο πρόσφατη έκδοση της οικογένειας VPUs που προσφέρει η Intel και πρόκειται για ένα ισχυρό, πολυπύρηνο, ετερογενές υπολογιστικό σύστημα, με ειδικό επιταχυντή για εφαρμογές βαθιάς μάθησης και υψηλές επιδόσεις ανά μονάδα ισχύος. Ωστόσο, τα περισσότερα σύγχρονα νευρωνικά δίκτυα αναπτύσσονται με γνώμονα τις επιδόσεις πολύ ισχυρότερων υπολογιστικών συστημάτων και εστιάζουν περισσότερο στην ακρίβεια, παρά στην αποδοτικότητα. Τέτοιου είδους νευρωνικά δίκτυα συναντάμε στην προσπάθειά μας να επιλύσουμε το πρόβλημα καθορισμού του προσανατολισμού ενός δορυφόρου και παρακολούθησής του, γνωστό και ως "Lost in Space".

Στην παρούσα διπλωματική, μελετήσαμε διαφορετικές τεχνικές δειγματοληψίας στα δεδομένα εισόδου, καθώς και την επίδραση που είχαν στην μείωση των υπολογισμών και των παραμέτρων, αλλά και την ακρίβεια ενός ΣΝΔ. Χρησιμοποιήθηκαν πολλαπλές μεθόδοι βελτιστοποίησης του τελικού κώδικα σε χαμηλό επίπεδο, συμπεριλαμβανομένης της χρήσης Scratchpad μνήμης και SIMD εντολών, με στόχο την αποφυγή ύπαρξης στενωπού σε αυτό το στάδιο προεπεξεργασίας της εισόδου. Το στάδιο αυτό τροφοδοτεί ένα ΣΝΔ, με το όνομα "UrsoNet", το οποίο εντοπίζει την θέση ενός δορυφόρου στην εικόνα εισόδου και υπολογίζει τον προσανατολισμό του. Για την καταγραφή των απαιτήσεων της εφαρμογής αυτής, ως προς την κατανάλωση ισχύος, προτείνεται ένα custom σύστημα, το οποίο υποστηρίζει και στατική διαχείριση ισχύος. Τέλος, προτείνεται ένα υβριδικό σύστημα, το οποίο βασίζεται στο ΣΝΔ για τον υπολογισμό της αρχικής ¨πόζας' ενός δορυφόρου, και ακολούθως χρησιμοποιεί έναν κλασικό αλγόριθμο όρασης, για να εξελίξει την ¨πόζα' αυτή σε πραγματικό χρόνο.

Τα αποτελέσματα που λαμβάνουμε είναι εξαιρετικά ενθαρρυντικά, καθώς με σωστή προεπεξεργασία των εικόνων εισόδου, ο χρόνος εκτέλεσης του συνελικτικού νευρωνικού δικτύου μειώνεται εως και 5 φορές, χωρίς κάποια ουσιαστική απώλεια στην ακρίβεια του. Αυτό μας επιτρέπει την ικανοποιητική εκτέλεση του στην Myriad X VPU σε πραγματικό χρόνο, και μάλιστα με πολύ χαμηλή κατανάλωση ισχύος. Συγκεκριμένα, επιτυγχάνεται εκτέλεση σε 2.12 - 2.22 FPS, αναλόγα με τον βαθμό προεπεξεργασίας της εισόδου, με μέση κατανάλωση ισχύος μικρότερη από 2 Watt. Το προτεινόμενο υβριδικό σύστημα λειτουργεί με ένα overhead της τάξης των 373.7 - 391.3 ms για την απόκτηση της αρχικής κατάστασης του δορυφόρου και, εν συνεχεία, η παρακολούθησή του απαιτεί 263 - 388 ms, ή 2.58 - 3.80 FPS.

Λέξεις Κλειδιά

Ετερογενείς Αρχιτεκτονικές, Ενσωματωμένα Συστήματα, Myriad X, Επεξεργασία εικόνας, Συνελικτικά Νευρωνικά Δίκτυα, Pose Estimation.

Abstract

It is estimated that, by 2022, 82% of the packets transferred through the Internet will contain video data. The real-time processing of these data is a rather attractive prospect, that can lead to the creation of very interesting systems, commercial or otherwise. Convolutional Neural Networks (CNNs) are an important tool in this direction, as their recent rapid growth has resulted in some impressive solutions to classic computer vision problems. On the other hand, traditional embedded systems cannot support the increased requirements of CNNs in computational or memory resources. In this environment, an upcoming class of microprocessors, the Vision Processing Units, are developed.

Myriad X is the latest installment in the family of VPUs offered by Intel/Movidius. It is a multicore, heterogeneous computing system, with a dedicated hardware accelerator for deep learning applications, and high performance per unit of power. However, most modern neural networks are developed, based on the performance of much more potent processing systems, and emphasize on accuracy rather than efficiency. This is the basis of many networks that attempt to solve the problem of estimating and tracking the pose of a satellite, more commonly known as the "Lost in Space" problem.

In this thesis, we studied several different resampling methods on the input data, in order to determine how they affect the total number of computations and parameters of a CNN, as well as its accuracy. Multiple optimization techniques were utilized, including the exploitation of the on-chip Scratchpad Memory and the SIMD utilities of the Myriad X VPU, so as to avoid creating a bottleneck during this preprocessing stage. The preprocessed data are fed into a CNN, named "UrsoNet", which locates the position of a satellite on the input image and estimates its pose. To measure the power requirements of this application, a custom Power Measurement system is introduced, which can also perform static power management. Finally, a hybrid system is proposed. This system utilizes the CNN for the estimation of the initial pose of the satellite and, consequently, runs a classic, pipelined CV algorithm, that evolves and refines this initial pose in real-time.

The results are highly encouraging, since the execution time required for a single inference is reduced up to 5 times, provided that proper preprocessing of the input frames is applied, with no noticeable degradation in accuracy. This allows for real-time execution on Myriad X, on a tight power envelope. Specifically, we achieve 2.12 - 2.22 FPS, depending on the scale on which the preprocessing takes place, with a mean power consumption of less than 2 Watts. The proposed hybrid system operates with an overhead of about 373.3 - 391.3 ms for the initial estimation and then requires approximately 263 - 388 ms to continue tracking the pose of the satellite, resulting in a throughput of 2.58 - 3.80 FPS.

Keywords

Heterogeneous Architectures, Embedded Systems, Myriad X, Computer Vision, Convolutional Neural Networks, Pose Estimation.

Contents

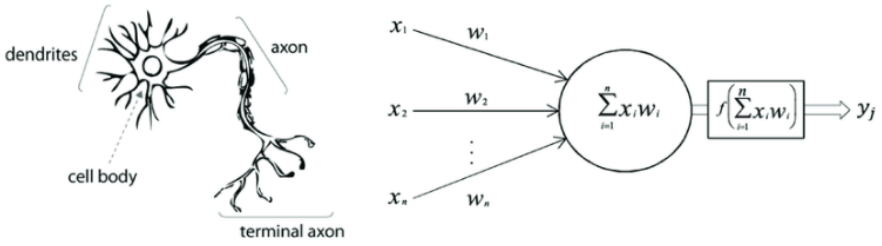
List of Figures

List of Tables

Εκτεταμένη Περίληψη

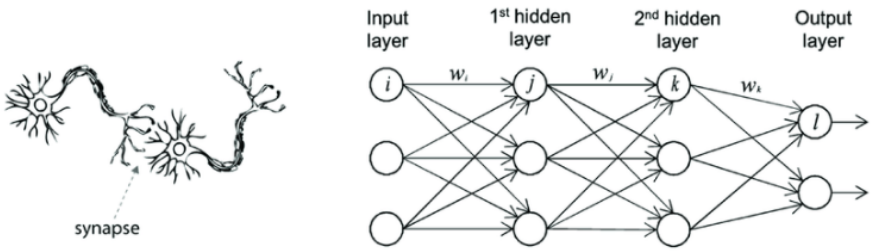
Τεχνητά Νευρωνικά Δίκτυα

Τα τεχνητά νευρωνικά δίκτυα (ΤΝΔ) μπορούν να γίνουν αντιληπτά ως επεξεργαστικά συστήματα κατανεμημένης αρχιτεκτονικής, τα οποία έχουν τη δυνατότητα να αποθηκεύουν εμπειρική γνώση, και να τη χρησιμοποιούν, όποτε χρειστεί, για να παράγουν κάποια απόκριση [12]. Μοιάζουν με τον ανθρώπινο εγκέφαλο, ως προς το γεγονός ότι αποκτούν γνώση από το περιβάλλον τους μεσω μιας διαδικασίας εκπαίδευσης και ότι αυτή η γνώση αποθηκεύεται στις συνδέσεις (συνάψεις) μεταξύ των δομικών στοιχείων τους (νευρώνων) στη μορφή ¨συναπτικών βαρών¨. Το σχήμα 1.1 αναπαριστά μία σύγκριση της δομής ενός βιολογικού και ενος τεχνητού νευρώνα.



Σχήμα 1: Ένας βιολογικός, ανθρώπινος νευρώνας (αριστερά) σε σχέση με έναν τεχνητό νευρώνα (δεξιά) [1].

Οι νευρώνες εντός του ανθρώπινου εγκεφάλου δέχονται ερεθίσματα από τους δενδρίτες και αποφασίζουν κατά πόσο θα παραχθεί κάποιο ερέθισμα μέσω του τερματικού άξονά τους ή όχι. Ο άξονας αυτός συνδέεται μέσω συνάψεων με τους δενδρίτες άλλων νευρώνων, δημιουργόντας ένα περίπλοκο δίκτυο διασυνδεδεμένων νευρώνων [12]. Οι τεχνητοί νευρώνες, είναι εμπνευσμένοι από τους βιολογικούς, και επιτελούν παρόμοιες λειτουργίες. Τα σήματα εισόδου $\{x_0, ..., x_n\}$ τροφοδοτούνται στον νευρώνα. Στη συνέχεια, υπολογίζεται το σταθμισμένο άθροισμα αυτών των δεδομένων εισόδου. Η βασική ιδέα είναι ότι η ¨δύναμη' των συνάψεων (τα βάρη w) είναι εκπαιδεύσιμα καιελέγχουν την επιρροή ενός νευρώνα σε ένα άλλο νευρώνα. Μία συνάρτηση ενεργοποίησης εξετάζει το τελικό άθροισμα και αποφασίζει αν ο νευρώνας θα πυροδοτήσει ή όχι. Το σήμα εξόδου του νευρώνα δίνεται ως είσοδος σε άλλους νευρώνες, παρόμοια με το βιολογικό μοντέλο, όπως φαίνεται και στο σχήμα 1.2.



Σχήμα 2: Συνάψεις μεταξύ βιολογικών νευρώνων (αριστερά) και διασυνδεδεμένοι τεχνητοί νευρώνες (δεξιά) [1].

Τα τεχνητά νευρωνικά δίκτυα εκπαιδεύονται ή χρησιμοποιούνται με τρεις διαφορετικούς τρόπους. Κάθε ένας από αυτούς συνιστά μια διαφορετική μέθοδο μηχανικής μάθησης.

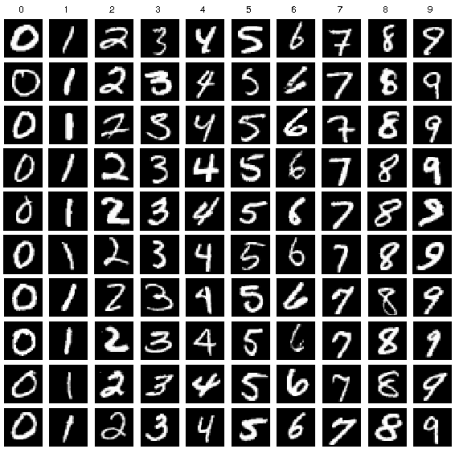
- Στην *Επιβλεπόμενη Μάθηση*, το δίκτυο έχει πρόσβαση σε δεδομένα με ετικέτες, όπου το επιθυμητό σήμα εξόδου για ένα συγκεκριμένο σήμα εισόδου είναι γνωστό. Το ΤΝΔ επιχειρεί να δημιουργήσει ένα μοντέλο, μαθαίνοντας από τα δεδομένα αυτά, ώστε να μπορεί μελλοντικά να κάνει προβλέψεις για άγνωστες εισόδους. Ο όρος ¨επιβλεπόμενη' εδώ, αναφέρεται στο γεγονός ότι οι επιθυμητές έξοδοι για τα δεδομένα εκπαίδευσης είναι διαθέσιμες [13].

- Στη *Μη-Επιβλεπόμενη Μάθηση*, από την άλλη πλευρά, το δίκτυο έχει να αντιμετωπίσει δεδομένα χωρίς ετικέτες ή άγνωστη δομή. Με τεχνικές μη επιτηρούμενη μάθησης, είναι εφικτό να εξερευνήσουμε την δομή των δεδομένων εισόδου, με στόχο την εξαγωγή χρήσιμων πληροφοριών, χωρίς την παροχή γνώσης που απαιτείται από την προηγούμενη κατηγορία μάθησης[14].

- Στην *Ενισχυτική Μάθηση*, το ΤΝΔ επιχειρεί να βελτιώσει τις επιδόσεις του μεσω των αλληλεπιδράσεών του με το περιβάλλον στο οποίο λειτουργεί. Εφόσον οι πληροφορίες για την τρέχουσα κατάσταση του περιβάλλοντος περιέχουν και ένα σήμα επιβράβευσης, η ενισχυτική μάθηση μπορεί να θεωρηθεί ως πεδίο, σχετική με την επιβλεπούμενη μάθηση[12]. Ωστόσο, στην ενισχυτική μάθηση, αυτού του είδους η επιβράβευση δεν έχει την μορφή μίας αναμενόμενης εξόδους, αλλά είναι μια εκτίμηση του πόσο καλά ανταποκρίθηκε μία επιλογή του δικτύου. Μέσω της αλληλεπίδρασης με το πέριβάλλον, ένα δίκτυο μπορει να χρησιμοποιήσει την ενισχυτική μάθηση για να μάθει μία σειρά από δράσεις που μεγιστοποιούν την επιβράβευση.

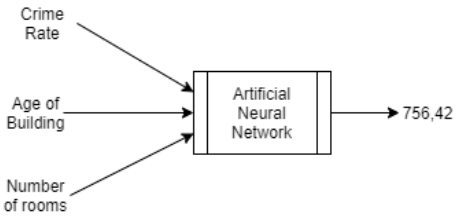
Στο πλαίσιο αυτής της διπλωματικής εργασίας, μελετήσαμε αποκλειστικά τα ΤΝΔ που χρησιμοποιούν τεχνικές επιβλεπούμενης μάθησης. Θα επικεντρωθούμε λοιπόν σε δύο λειτουργίες αυτών των ΤΝΔ, την ταξινόμηση και την παλινδρόμηση.

Η ταξινόμηση είναι μια υποκατηγορία της επιτηρούμενης μάθησης, στην οποία στόχος είναι η πρόβλεψη της κλάσης/κατηγορίας ενός νέου αντικειμένου, βάσει παλαιότερων παρατηρήσεων [12]. 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 σύνολο δεδομένων $MNIST$ είναι ένα παράδειγμα δειγμάτων, που μπορούν να χρησιμοποηθούν από ένα δίκτυο για την αποδοτική εκμάθηση ταξινόμησης χειρόγραφων ψηφίων. Μετά από επαρκή εκπαίδευση, αν ένας χρήστη εισάγει στο ΤΝΔ ένα νέο, χειρόγραφο ψηφίο μέσω κάποιας συσκευής εισόδου, το εκπαιδευμένο μοντέλο του ΤΝΔ θα μπορέσει να προβλέψει τον σωστό αριθμό με κάποια ακρίβεια. Ωστόσο, ο αλγόριθμος δε θα είναι σε θέση να αναγνωρίσει οποιοδήποτε από τους χαρακτήρες του αλφάβητου, εφόσον αυτοί δεν περιέχονται στο σύνολο εκπαίδευσής του.



Σχήμα 3: Δείγματα του συνόλου δεδομένων $MNIST$ [2]

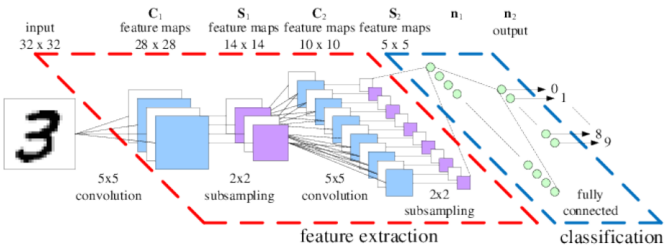
Η ανάλυση παλινδρόμησης είναι ένα άλλο είδος επιβλεπούμενης μάθησης. Στην ανάλυση παλινδρόμησης, υποθέτουμε ότι δίνονται ένα πλήθος από μεταβλητές πρόβλεψης και μια συνεχής μεταβλητή απόκρισης (αποτέλεσμα). Στόχος είναι η εύρεση μια σχέσης μεταξύ των μεταβλητών πρόβλεψης που επιτρέπει την πρόβλεψη ενός αποτελέσματος[15]. Για παράδειγμα, το σύνολο δεδομένων Boston House Price περιέχει 13 διαφορετικές (αριθμητικές) ιδιότητες των σπιτιών στα προάστια της Βοστώνης και ακολούθως σημειώνει την τιμή των σπιτιών αυτών σε χιλιάδες δολλαρίων. Μετά την διαδικασία εκπαίδευσης, αν ένας χρήστης παρέχει στο νευρωνικό δίκτυο δεδομένα σχετικά με τις τιμές των 13 αυτών ιδιοτήτων, αυτό θα μπορέσει να κάνει μία εκτίμηση για την τιμή του σπιτιού που περιγράφεται.



Σχήμα 4: Πρόβλεψη ενός ΤΝΔ, εκπαιδευμένου πάνω στο σύνολο δεδομένων Boston House Price

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, για εφαρμογές όρασης υπολογιστών, από τον Ψανν Λεῦν στις αρχές του 1990. Επί της ουσίας, ένα ΣΝΔ είναι ένα μοντέλου ΤΝΔ που έχει σχεδιαστεί αποκλειστικά για την αναγνώριση δισδιάστατων αντικειμένων, παρουσιάζοντας υψηλό βαθμό αναλλοίωτης συμπεριφοράς κατά την μετάθεση, κλιμάκωση, στρέβλωση και άλλες παραμορφώσεις της εισόδου. Ένα ΣΝΔ θα ήταν ιδανικό για την υλοποίηση ενός δικτύου για ταξινόμηση χειρόγραφων ψηφίων, δεδομένου ότι η είσοδος δίνεται σε μορφή εικόνας.



Σχήμα 5: Παράδειγμα τοπολογίας ενός ΣΝΔ για αναγνώριση χειρόγραφων ψηφίων

Δομή και ιδιότητες ΣΝΔ

Εφόσον τα Συνελικτικά Νευρωνικά Δίκτυα είναι μία υποκατηγορία των ΤΝΔ, αποτελούνται, όπως είναι αναμενόμενο, από νευρώνες με εκπαιδεύσιμα βάρη[16]. Κάθε νευρώνας δέχεται μία είσοδο, εκτελεί ένα εσωτερικό γινόμενο και εν συνεχεία μπορεί να εφαρμόσει κάποια μη-γραμμική συνάρτηση στα αποτελέσματα. Όλο το δίκτυο έχει ως στόχο να ταξινομήσει μια εικόνα σε κάποια κατηγορία βάσει των εικονοστοιχείων της. Όλη η γνώση που εφαρμόστηκε στα ΤΝΔ εξακολουθεί να ισχύει και να εφαρμόζεται στα ΣΝΔ. Ειδοποιός διαφορά των αρχιτεκτονικών ΣΝΔ είναι το γεγονός ότι κάνουν τη ρητή υπόθεση ότι η είσοδος είναι μια εικόνα, γεγονός που επιτρέπει την κωδικοποίηση ορισμένων ιδιοτήτων στην αρχιτεκτονική τους. Αυτές οι ιδιότητες μας επιτρέπουν να υπολογίζουμε την απόκριση του δικτύου πολύ πιο αποδοτικά, αφου μειώνουν σημαντικά των αριθμό των παραμέτρων του ΣΝΔ.

Ως παράδειγμα, ας εξετάσουμε το σύνολο δεδομένων CIFAR-10. Το συγκεκριμένο σύνολο δεδομένων περιέχει εικόνες μεγέθους 32x32x3 (μήκος x πλάτος x κανάλια χρώματος) και την κατηγορία αντικειμένου ή ζώου την οποία απεικονίζουν. Ένα απλό ΤΝΔ, όπως περιγράφηκε παραπάνωμ θα απαιτούσε 32*32*3 = 3072 εκπαιδεύσιμες παραμέτρους για κάθε νευρώνα του πρώτου κρυφού σταδίου. Όπως γίνεται αντιληπτο, αυτή η πλήρως διασυνδεδεμένη δομή δεν κλιμακώνει αποδοτικά για μεγαλύτερες εικόνες. Για παράδειγμα, μια εικόνα με διαστάσεις 200x200x3, θα απαιτούσε από κάθε νευρώνα 200*200*3 = 120,000 βάρη!

Τα ΣΝΔ, από την άλλη πλευρά, εκμεταλλεύονται το γεγονός ότι η είσοδός τους αποτελείται από εικόνες και περιορίζουν την αρχιτεκτονική τους με συνετό τρόπο [17]. 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 32×32×3 (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. Αυτή η τοπικά περιορισμένη διασύνδεση των νευρώνων των ΣΝΔ είναι εμπνευσμένη από βιολογικές λειτουργίες, όπου ένας νευρώνας αντιδρά στα ερεθίσματα που δέχεται μόνο από ένα περιορισμένο κομμάτι του πεδίου όρασης.

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

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

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

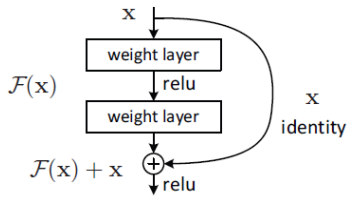
- **Pooling Στρώση:** Θα πραγματοποιήσει μια υποδειγματοληψία ή/και εξομάλυνση κατά μήκος του βάθους του όγκου εισόδου. Δεν είναι εκπαιδεύσιμη.

- **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 ενεργοποίησης 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. Δεν είναι εκπαιδεύσιμη.

Αρχιτεκτονική ResNet

Τα Υπολειπόμενα Νευρωνικά Δίκτυα (ResNets) [3] χρησιμοποιούνται εκτενώς σε αυτή τη διπλωματική εργασία, συνεπώς είναι απαραίτητο ο αναγνώστης να διαθέτει εξοικείωση σχετικά με τη λειτουργία τους. Τα ΥΝΔ δημιουργήθηκαν με σκοπό την αποφυγή του προβλήματος των εξαφανιζόμενων κλίσεων και την αντιμετώπιση της μείωσης που παρατηρούνταν στην ακρίβεια ενός νευρωνικού, όταν αυτό αποτελούταν από μεγάλο αριθμό στρώσεων. Το πρόβλημα των εξαφανιζόμενων κλίσεων εμφανίζεται κατά την εκπαίδευση νευρωνικών που χρησιμοποιούν μεθόδους που βασίζονται στην κλίση. Εφόσον η κλίση της συνάρτησης απώλειας του νευρωνικού υπολογίζεται με τον κανόνα της αλυσίδας, είναι σύνηθες, οι κλίσεις που αντιστοιχούν σε βάρη των νευρώνων των πρώτων σταδίων ενός νευρωνικού να είναι εξαιρετικά μικρές, αποτρέποντας την ανανέωση των τιμών των βαρών αυτών. Για να αντιμετωπίσει αυτό το πρόβλημα, ένα ΥΝΔ χρησιμοποιεί υπερπηδούσες συνδέσεις, για να αγνοήσει κάποιες στρώσεις. Οι υπερπηδούσες συνδέσεις αποτελούνται συνήθως από ένα ταυτοτικό μετασχηματισμό, αλλά μπορούν να περιέχουν και μία συνελικτική στρώση με πυρήνα μεγέθους 1x1. Συνεπώς, η αρχιτεκτονική της βασικής τους μονάδας είναι η παρακάτω:

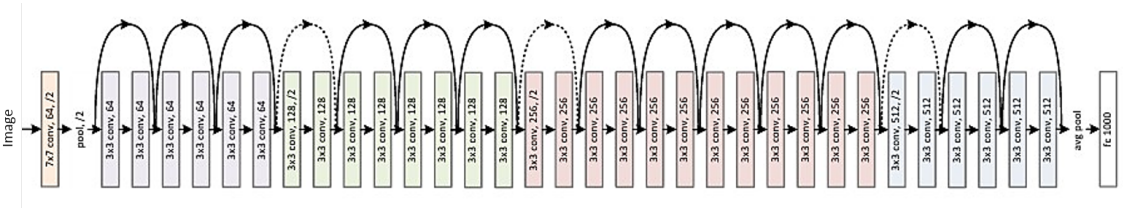


Σχήμα 6: Υπολειπόμενη Μάθηση: Η βασική δομική μονάδα [3]

Λόγω των ιδιαίτερων συνδέσεων των ΥΝΔ, είναι εφικτό να δημιουργηθούν πολύ βαθιά συνελικτικά νευρωνικά δίκτυα, με επαναληπτική προσθήκη στρώσεων. Στον παρακάτων πίνακα παρουσιάζονται οι αρχιτεκτονικές 4 διαφορετικών τοπολογιών ΥΝΔ. Το βάθος του δικτύου καθορίζεται από τον αριθμό των φίλτρων που θα χρησιμοποιηθούν στο τέταρτο συνελικτικό στάδιο, και χρησιμοποιείται για την ονοματοδοσία του δικτύου.

layer name	output size	18-layer	34-layer	50-layer	101-layer	152-layer
conv1	112×112	7×7, 64, stride 2				
conv2_x	56×56	3×3 max pool, stride 2				
		$\begin{bmatrix} 3\times3, 64 \\ 3\times3, 64 \end{bmatrix}\times2$	$\begin{bmatrix} 3\times3, 64 \\ 3\times3, 64 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 64 \\ 3\times3, 64 \\ 1\times1, 256 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 64 \\ 3\times3, 64 \\ 1\times1, 256 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 64 \\ 3\times3, 64 \\ 1\times1, 256 \end{bmatrix}\times3$
conv3_x	28×28	$\begin{bmatrix} 3\times3, 128 \\ 3\times3, 128 \end{bmatrix}\times2$	$\begin{bmatrix} 3\times3, 128 \\ 3\times3, 128 \end{bmatrix}\times4$	$\begin{bmatrix} 1\times1, 128 \\ 3\times3, 128 \\ 1\times1, 512 \end{bmatrix}\times4$	$\begin{bmatrix} 1\times1, 128 \\ 3\times3, 128 \\ 1\times1, 512 \end{bmatrix}\times4$	$\begin{bmatrix} 1\times1, 128 \\ 3\times3, 128 \\ 1\times1, 512 \end{bmatrix}\times8$
conv4_x	14×14	$\begin{bmatrix} 3\times3, 256 \\ 3\times3, 256 \end{bmatrix}\times2$	$\begin{bmatrix} 3\times3, 256 \\ 3\times3, 256 \end{bmatrix}\times6$	$\begin{bmatrix} 1\times1, 256 \\ 3\times3, 256 \\ 1\times1, 1024 \end{bmatrix}\times6$	$\begin{bmatrix} 1\times1, 256 \\ 3\times3, 256 \\ 1\times1, 1024 \end{bmatrix}\times23$	$\begin{bmatrix} 1\times1, 256 \\ 3\times3, 256 \\ 1\times1, 1024 \end{bmatrix}\times36$
conv5_x	7×7	$\begin{bmatrix} 3\times3, 512 \\ 3\times3, 512 \end{bmatrix}\times2$	$\begin{bmatrix} 3\times3, 512 \\ 3\times3, 512 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 512 \\ 3\times3, 512 \\ 1\times1, 2048 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 512 \\ 3\times3, 512 \\ 1\times1, 2048 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 512 \\ 3\times3, 512 \\ 1\times1, 2048 \end{bmatrix}\times3$
	1×1	average pool, 1000-d fc, softmax				
FLOPs		1.8×10^9	3.6×10^9	3.8×10^9	7.6×10^9	11.3×10^9

Μία ενδιαφέρουσα ιδιότητα των βαθύτερων υπολειπόμενων νευρωνικών δικτύων, δηλαδή όσων έχουν 50 ή παραπάνω στρώσεις, είναι, ότι παρά τον πολύ μεγάλο αριθμό στρώσεων, περιέχουν μονάχα δύο στρώσεις pooling. Αυτό συμβαίνει, επειδή οι στρώσεις pooling δεν είναι εκπαιδεύσιμες, και η υποδειγματοληψία που εφαρμόζουν στους ενδιάμεσους χάρτες χαρακτηριστικών οδηγεί στην απώλεια εντοπισμένων χαρακτηριστικών και, συνεπώς, στη μείωση της ακρίβειας. Τα ΥΝΔ, λοιπόν, αντικαθιστούν τις στρώσεις pooling με συνελικτικές στρώσεις με πυρήνες 1x1 πριν και μετά την εκτέλεση των συνελικτικών στρώσεων με πυρήνες 3x3. Η 1x1 συνέλιξη χρησιμοποιείται για την απεικόνιση της εισόδου σε ένα χώρο μικρότερων διαστάσεων, με στόχο τη μείωση των πράξεων της 3x3 συνέλιξης που ακολουθεί [18]. Ο χάρτης χαρακτηριστικών που προκύπτει, στην συνέχεια, απεικονίζεται σε ένα χώρο υψηλότερων διαστάσεων, ώστε να τονιστούν τα χαρακτηριστικά που εξάχθηκαν, εφόσον ο χάρτης χαρακτηριστικών αναμένεται να οδηγηθεί και πάλι σε μικρότερες διαστάσεις στο αμέσως επόμενο στάδιο, είτε συνελικτικό είτε pooling.



Σχήμα 7: ResNet-34 [3]

Το πρόβλημα 'Lost in Space'

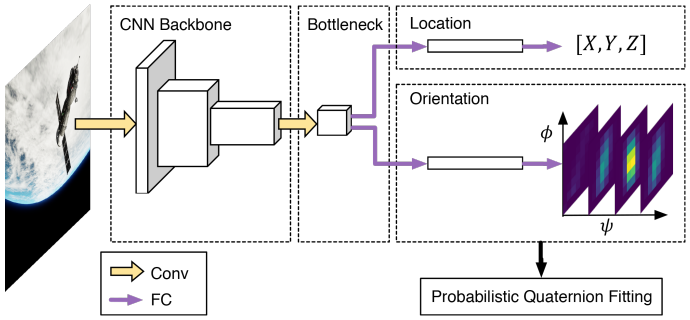
Στα πλαίσια της παρούσας διπλωματικής εργασίας, μελετήσαμε το πρόβλημα 'Lost in Space¨. Σε αυτό το πρόβλημα, θεωρούμε ότι εντός του πεδίου της κάμερας βρίσκεται ένας μη συνεργάσιμος στόχος. Η κάμερα είναι ενσωματωμένη σε ένα δορυφόρο σε τροχιά γύρω από τη Γη, και συνεπώς ο όρος ¨μη συνεργάσιμος' αναφέρεται στο γεγονός ότι το αντικείμενο-στόχος, δεν επιχειρεί να βοηθήσει στη διαδικασία εντοπισμού του μέσω επικοινωνίας. Το σύστημα που φέρει την κάμερα έχει γνώση του τρισδιάστατου μοντέλου του στόχου. Ένα τέτοιο σενάριο είναι ρεαλιστικό, και θα μπορούσε να συναντηθεί, για παράδειγμα, στη διαδικασία συλλογής διαστημικών σκουπιδιών.

Παραδοσιακά, η επίλυση αυτού του προβλήματος θα απαιτούσε τον σχεδιασμό ενός σύνθετου συστήματος επεξεργασίας ψηφιακών εικόνων, όπου το αντικείμενο θα εντοπιζόταν μέσω των άκρων του και ο προσανατολισμός του θα καθοριζόταν με σύγκρισή του με γνωστά τρισδιάστατα μοντέλα του. Στη συνέχεια, σε βάθος χρόνου, οι εκτιμήσεις της θέσης και του προσανατολισμού του στόχου

θα βελτιώνονταν μέσα από την εξέταση της εξέλιξης τους. Ωστόσο, μας δίνεται πλέον η δυνατότητα να εκπαιδεύσουμε ενα αξιόπιστο και αποδοτικό Συνελικτικό Νευρωνικό Δίκτυο, ώστε να υλοποιεί την ζητούμενη λειτουργία. Ένα τέτοιο Συνελικτικό Δίκτυο, βασιζόμενο σε τοπολογίες Υπολειπόμενων Νευρωνικών Δικτύων είναι το **UrsoNet**.

UrsoNet: Εκτίμηση Πόζας για Δορυφόρους

Το UrsoNet είναι ένα Συνελικτικό Νευρωνικό Δίκτυο, το οποίο προτάθηκε από τους Pedro Proenca και Yang Gao, κατά τη διαρκεία του διαγωνισμού της ESA, "Pose Estimation Challenge" το 2019, και κατέκτησε την τρίτη θέση του εν λόγω διαγωνισμού. Σε αντίθεση με τα μοντέλα των υπόλοιπων διαγονιζόμενων, το UrsoNet απαιτεί εκτεταμένους υπολογιστικούς πόρους και συνεπώς, υπάρχει η προοπτική να τρέξει σε ένα ενσωματωμένο σύστημα. Στο παρακάτω σχήμα, απεικονίζεται μία απλοποιημένη εκδοχή της αρχιτεκτονικής αυτού του δικτύου.



Σχήμα 8: Απλοποιημένη επισκόπηση της αρχιτεκτονικής του UrsoNet [4]

Το συγκεκριμένο νευρωνικό βασίζεται σε ένα Συνελικτικό Νευρωνικό Δίκτυο. Συγκεκριμένα, επιλέγεται ένα Υπολειπόμενο Νευρωνικό Δίκτυο, ανάμεσα από τα ResNet34, ResNet50, ResNet101. Ο βασικός λόγος που προτιμούνται τα ResNet είναι το γεγονός ότι διαθέτουν ελάχιστα στάδια pooling. Αυτού του είδους οι στρώσεις, δεν είναι επιθυμητές, αφού όπως εξηγήθηκε και νωρίτερα προκαλούν παραμόρφωση και απώλεια χαρακτηριστικών λόγω της λειτουργίας τους. Σε αυτή την κατεύθυνση, τα τελευταία δύο στάδια του δικτύου, δηλαδή ένα pooling και ένα πλήρως συνδεδεμένο στάδιο, έχουν αντικατασταθεί από μία συνελικτική στρώση. Αυτή με τη σειρά της, τροφοδοτεί δύο κεφαλές που αποτελούνται από δύο πλήρως συνδεδεμένες στρώσεις έκαστη, και αναλαμβάνουν τον υπολογισμό της τοποθεσίας και του προσανατολισμού του δορυφόρου αντιστοίχως.

Τα συνθετικά σύνολα δεδομένων, πάνω στα οποία βασίζεται η εκπαίδευση του νευρωνικού αυτού, περιέχουν έγχρωμες εικόνες με ανάλυση 1280x960 pixels. Γίνεται γρήγορα αντιληπτό, ότι για ένα τέτοιο μέγεθος εικόνας, η εκτέλεση σε πραγματικό χρόνο για κάποιο ενσωματωμένο σύστημα θα είναι αδύνατη. Για να επιταχυνθεί η εκτέλεση του νευρωνικού θα πρέπει να μειωθεί το μέγεθός του. Συνεπώς, θα χρειαστεί υποδειγματοληψία, ώστε να μειωθούν οι διαστάσεις των δεδομένων εκπαίδευσης. Με αυτό το τρόπο, το δίκτυο θα δέχεται ως είσοδο εικόνες μικρότερων διαστάσεων και άρα θα απαιτείται μικρότερος αριθμός υπολογισμών για την εξαγωγή αποτελέσματος.

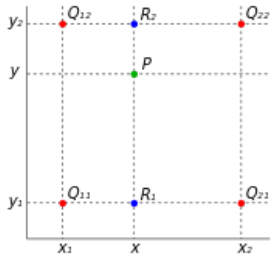
Μέθοδοι Υποδειγματοληψίας

Εξετάζουμε τρεις διαφορετικές τεχνικές υποδειγματοληψίας: διγραμμική παρεμβολή, δικυβική παρεμβολή και επαναδειγματοληψία Lanczos.

Διγραμμική Παρεμβολή

Η διγραμμική παρεμβολή πραγματοποιείται εφαρμόζοντας γραμμική παρεμβολή στον οριζόντιο άξονα και εν συνεχεία παρεμβάλλοντας τα αποτελέσματα γραμμικά στον κατακόρυφο άξονα, ή και αντίστροφα. Ένας πιο τυπικός ορισμός της μεθόδου είναι:

Έστω ότι θέλουμε να υπολογίσουμε την τιμή της άγνωστης συνάρτησης f στο σημείο (x, y), δεδομένου ότι η τιμή της συνάρτησης για τα σημεία $Q_{11} = (x_1, y_1)$, $Q_{12} = (x_1, y_2)$, $Q_{21} = (x_2, y_1)$ και $Q_{22} = (x_2, y_2)$ θεωρείται γνωστή.



Εφαρμόζουμε αρχικά γραμμική παρεμβολή στον οριζόντιο άξονα:

$$f(R_1) = f(x, y_1) = \frac{x_2 - x}{x_2 - x_1} \cdot f(Q_{11}) + \frac{x - x_1}{x_2 - x_1} \cdot f(Q_{21})$$

$$f(R_2) = f(x, y_2) = \frac{x_2 - x}{x_2 - x_1} \cdot f(Q_{12}) + \frac{x - x_1}{x_2 - x_1} \cdot f(Q_{22})$$

Στη συνέχεια, παρεμβάλουμε γραμμικά στον κάθετο άξονα για να λάβουμε την επιθυμητή εκτίμηση:

$$f(x, y) = \frac{y_2 - y}{y_2 - y_1} \cdot f(R_1) + \frac{y - y_1}{y_2 - y_1} \cdot f(R_2)$$

Η μέθοδος αυτή, στη συγκεκριμένη περίπτωση, θα χρησιμοποιηθεί για την υποδειγματοληψία μιας εικόνας. Μια γειτονιά τεσσάρων pixel θα χρησιμοποιείται κάθε φορά για τον υπολογισμό ενός pixel της τελικής εικόνας. Συνεπώς, θεωρούμε ότι τα τέσσερα pixel βρίσκονται στις θέσεις (0,0), (0,1), (1,0) ανδ (1,1) των αξόνων, και το επιθυμητό pixel στη θέση (0.5, 0.5). Ο προηγούμενος τύπος με αντικατάσταση γίνεται:

$$f(0.5, 0) = \frac{1}{2} \cdot f(0, 0) + \frac{1}{2} \cdot f(1, 0)$$

$$f(0.5, 1) = \frac{1}{2} \cdot f(0, 1) + \frac{1}{2} \cdot f(1, 1)$$

Και τελικά:

$$f(0.5, 0.5) = \frac{1}{2} \cdot f(0.5, 0) + \frac{1}{2} \cdot f(0.5, 1) = \frac{1}{4} \cdot (f(0, 0) + f(0, 1) + f(1, 0) + f(1, 1))$$

Συμπεραίνουμε ότι για την συγκεκριμένη λειτουργία, η διγραμμική παρεμβολή ισοδυναμεί με την εξαγωγή του μέσου όρου μιας γειτονιάς τεσσάρων pixel.

Δικυβική παρεμβολή

Η μέθοδος της δικυβικής παρεμβολής, όμοια με την διγραμμική, υλοποιείται με την εφαρμογή κυβικής παρεμβολής στους δύο άξονες διαδοχικά. Σε αντίθεση όμως με την προηγούμενη μέθοδο,

απαιτεί την εξέταση 16 στοιχείων, σε μία γειτονιά 4x4. Ως αποτέλεσμα, οι εικόνες που υποδειγματοληπτούνται με αυτή την τεχνική εμφανίζουν λιγότερες παρομορφώσεις σε σχέση με τη διγραμμική παρεμβολή, με αντίτιμο τον αυξημένο χρόνο εκτέλεσης του αλγορίθμου για μία εικόνα. Τα επιθυμητά pixel υπολογίζονται ως εξής:

$$p(x,y) = \sum_{i=0}^{3}\sum_{j=0}^{3} a_{\mathrm{ij}} \cdot x^i \cdot y^j$$

Το πρόβλημα της παρεμβολής, λοιπόν, ανάγεται στον καθορισμό των 16 συντελεστών a_{ij}. Ο απευθείας υπολογισμός των συντελεστών αυτών δεν είναι αποδοτικός, επειδή οι τιμές τους εξαρτώνται από τις τιμές των pixel που είναι υπό εξέταση. Αυτό σημαίνει, ότι για κάθε επανάληψη του αλγορίθμου, οι τιμές αυτές θα πρέπει να υπολογίζονται από την αρχή. Για αυτό το λόγο, αποφασίστηκε η χρήση του Δικύβικού Συνελικτικού Αλγορίθμου, όπως αυτός προτάθηκε από τον Keys [19].

Με βάση αυτό τον αλγόριθμο, η δικυβική παρεμβολή, όπως περιγράφηκε παραπάνω, είναι ισοδύναμη με την εφαρμογή μίας συνέλιξης με τον παρακάτω πυρήνα, και ως προς τους δύο άξονες:

$$W(x) = \begin{cases} (\alpha+2)\cdot|x|^3 - (\alpha+3)\cdot|x|^2 + 1 & |x| \le 1 \\ \alpha\cdot|x|^3 - 5\alpha\cdot|x|^2 + 8\alpha\cdot|x| - 4\alpha & 1 < |x| < 2 \\ 0 & otherwise \end{cases}$$

όπου α είναι μία παράμετρος, που τυπικά τίθεται ίση με -0.5 ή -0.75 και x είναι η απόσταση μεταξύ του εξεταζόμενου και του επιθυμητού pixel. Κατα μήκος του οριζόντιου άξονα, τα pixel αριθμούνται με ακέραιες τιμές που κυμαίνονται απο το -1 έως και το 2. Η ίδια αρίθμηση ισχύει και για τον κάθετο άξονα. Κάθε γραμμή της εξεταζόμενης γειτονιάς θα συνελιχθεί με τον παραπάνω πυρήνα. Ακολούθως, οι τιμές που προκύπτουν από αυτές τις συνελίξεις, συμμετέχουν σε μία νέα πράξη συνέλιξης με τον ίδιο πυρήνα, για την εξαγωγή της τελικής, επιθυμητής τιμής. Όπως ακριβώς και στην προηγούμενη μέθοδο, το επιθυμητό pixel θεωρείται ότι βρίσκεται στη θέση (0.5, 0.5). Εφόσον τόσο οι θέσεις όλων των pixel που συμμετέχουν στους υπολογισμούς είναι γνωστές, οι συντελεστές κάθε pixel μπορούν να υπολογιστούν και να ενσωματωθούν ως σταθερές εντός της εφαρμογής. Ακόμη, δεδομένου ότι $x = y = 0.5$, οι συντελεστές για τους δύο άξονες θα είναι ίδιοι. Συνεπώς, μόνο οι παρακάτω τέσσερεις τιμές είναι απαραίτητο να υπολογιστούν.

$$W(0.5-(-1)) = W(1.5) = -0.75\cdot|1.5|^3 + 5\cdot0.75\cdot|1.5|^2 - 8\cdot0.75\cdot|1.5| + 4\cdot0.75 = -0.09375$$

$$W(0.5) = 1.25\cdot|0.5|^3 - 2.25\cdot|0.5|^2 + 1 = 0.59375$$

$$W(1-0.5) = W(0.5) = 0.59375$$

$$W(2-0.5) = W(1.5) = -0.09375$$

όπου το α έχει τεθεί ίσο με -0.75. Η κυβική συνέλιξη σε μία διάσταση μπορεί να γραφτεί σε μορφή πινάκων:

$$f(P_{-1},P_0,P_1,P_2) = \begin{bmatrix} -0.09375 & 0.59375 & 0.59375 & -0.09375 \end{bmatrix} \cdot \begin{bmatrix} P_{-1} \\ P_0 \\ P_1 \\ P_2 \end{bmatrix}$$

Και για τον υπολογισμο του επιθυμητού pixel:

$$DesiredPixel = f\begin{pmatrix} f(P_{(-1,-1)},P_{(-1,0)},P_{(-1,1)},P_{(-1,2)}), \\ f(P_{(0,-1)},P_{(0,0)},P_{(0,1)},P_{(0,2)}), \\ f(P_{(1,-1)},P_{(1,0)},P_{(1,1)},P_{(1,2)}), \\ f(P_{(2,-1)},P_{(2,0)},P_{(2,1)},P_{(2,2)}) \end{pmatrix}$$

Επαναδειγματοληψία Lanczos

Η επαναειγματοληψία Lanczos είναι μία μέθοδος που βασίζεται στην συνέλιξη ενός σήματος εισόδου με τον πυρήνα Lanczos. Ο πυρήνας αυτός αποτελείται από μία κανονικοποιημένη συνάρτηση $sinc(x)$ εντός του εύρους τιμών [-α, α], όπου το α είναι μία θετική ακέραια παράμετρος και καθορίζει το πλήθος των δειγμάτων που καθορίζουν την τιμή μίας εξόδου. Εκτός του παραθύρου αυτού, ο πυρήνας έχει μηδενική τιμή.

$$L(x) = \begin{cases} sinc(x) \cdot sinc\left(\frac{x}{a}\right) & -a < x < a \\ 0 & otherwise \end{cases}$$

Στην παρούσα διπλωματική, επιλέχθηκε να τεθεί η παράμετρος α ίση με 4. Συνεπώς, ο πυρήνας Lanczos θα είναι:

$$L(x) = \begin{cases} 1 & x = 0 \\ \frac{sin(\pi x) \cdot sin\left(\frac{\pi x}{4}\right)}{4 \cdot \pi^2} & -a < x < a, x \neq 0 \\ 0 & otherwise \end{cases}$$

Και συνεπώς, για εξαγωγή ενός αποτελέσματος σε μία διάσταση, αρκεί η εκτέλεση της πράξης της συνέλιξης του σήματος εισόδου με τον πυρήνα Lanczos:

$$f(x) = \sum_{i=-a+1}^{a} f(\lfloor x \rfloor + i) \cdot L(i - x + \lfloor x \rfloor)$$

Στις δύο διαστάσεις, ο πυρήνας Lanczos είναι απλά το γινόμενο των δύο πυρήνων κάθε διάστασης. Ο πυρήνας που προκύπτει δεν είναι διαχωρίσιμος.

$$L(x, y) = L(x) \cdot L(y)$$

Για εξαγωγή, λοιπόν, ενός αποτελέσματος σε δισδιάστατο χώρο, αρκεί και πάλι η εκτέλεση της πράξης της συνέλιξης:

$$f(x, y) = \frac{1}{w} \cdot \sum_{i=-a+1}^{a} \sum_{j=-a+1}^{a} f(\lfloor x \rfloor + i, \lfloor y \rfloor + j) \cdot L(i - x + \lfloor x \rfloor) \cdot L(j - y + \lfloor y \rfloor)$$

$$w = \sum_{i=-a+1}^{a} \sum_{j=-a+1}^{a} L(i - x + \lfloor x \rfloor) \cdot L(j - y + \lfloor y \rfloor)$$

όπου το αποτέλεσμα κανονικοποιείται ως προς το συνολικό βάρος του φίλτρου. Όπως και στις προηγούμενες τεχνικές, το επιθυμητό pixel βρίσκεται στη θέση (0.5, 0.5). Εναλλάσοντας τους τελεστές άθροισης και θεωρώντας α = 4, η παραπάνω εξίσωση γράφεται:

$$f(x, y) = \sum_{j=-3}^{4} \frac{1}{w} \cdot L(j - y) \cdot \left(\sum_{i=-3}^{4} f(i, j) \cdot \frac{1}{w} \cdot L(i - x) \right)$$

Το πρόβλημα, όπως και στην περίπτωση του δικυβικού συνελικτικού αλγορίθμου, ανάγεται στον υπολογισμό των συντελεστών του κάθε pixel, ξεχωριστά για κάθε διάσταση. Οι τιμές των συντελεστών, ωστόσο, για τις δύο διαστάσεις είναι ίδιοι και άρα υπολογίζονται μόνο μία φορά:

$$L(-3 - 0.5) = L(-3.5) = -0.01266087782123867$$

$$L(-2.5) = 0.05990948337726289$$

$$L(-1.5) = -0.16641523160350802$$

$$L(-0.5) = 0.6203830132406946$$

$$L(0.5) = L(-0.5)$$

$$L(1.5) = L(-1.5)$$

$$L(2.5) = L(-2.5)$$

$$L(3.5) = L(-3.5)$$

$$sum(L_i) = 1.0024327743864216$$

Σε μορφή πίνακα, η επαναδειγματοληψία Lanczos που εφαρμόζουμε, περιγράφεται ως εξής:

$$DesiredPixel = \begin{bmatrix} P_{-3} & P_{-2} & P_{-1} & P_0 & P_1 & P_2 & P_3 & P_4 \end{bmatrix} \cdot \begin{bmatrix} -0.012630151512143303 \\ 0.05976409082786907 \\ -0.1660113634107474 \\ 0.6188774240950217 \\ 0.6188774240950217 \\ -0.1660113634107474 \\ 0.05976409082786907 \\ -0.012630151512143303 \end{bmatrix}$$

Intel/Movidius Myriad X VPU

Η Myriad X είναι η πιο πρόσφατη προσθήκη στην οικογένεια των Vision Processing Units (VPUs), που προσφέρει η Intel/Movidius. Πρόκειται για μία ανερχόμενη κατηγορία μίκροεπεξεργαστών, η οποία στοχεύει στην αποδοτική επιτάχυνση εφαρμογών όρασης υπολογιστών. Η συγκεκριμένη πλατφόρμα επιλέχθηκε για να φιλοξενήσει το σύστημα που πραγματεύεται η παρούσα διπλωματική, το οποίο θα λύνει σε πραγματικό χρόνο το πρόβλημα "Lost in Space". Η Myriad X προσφέρει:

- **Εξαιρετικά χαμηλή κατανάλωση ισχύος.** Η χαμηλή κατανάλωση ισχύος είναι απαραίτητη για ενσωματωμένες αρχιτεκτονικές, καθώς συνδέεται άμεσα με το χρόνο που μπορεί να υποστηριχθεί η τροφοδότηση της πλατφόρμας από κάποια ανεξάρτητη πηγή ενέργειας, για παράδειγμα μπαταρία. Η Myriad X όχι μόνο λειτουργεί με λιγότερα από 2.5 Watt, αλλά και παρέχει την δυνατότητα να απενεργοποιηθούν κομμάτια του συστήματος τα οποία δεν χρησιμοποιούνται και να μειωθεί η συχνότητα του ρολογιού, για περαιτέρω μείωση της καταναλισκώμενης ισχύος.

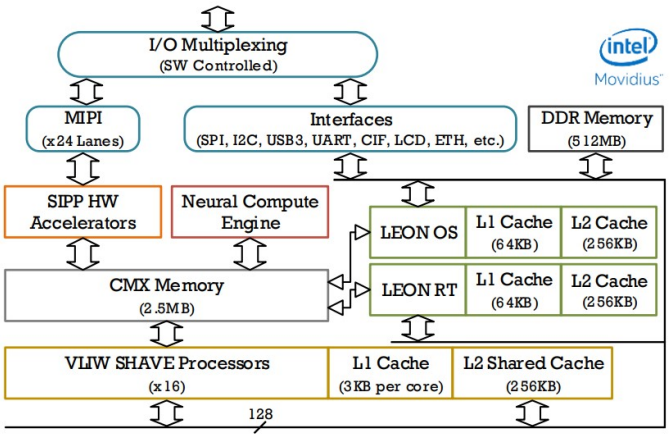
- **Μικρές φυσικές Διαστάσεις.** Οι διαστάσεις του επεξεργαστή αυτού είναι μόλις 8.8x8.0x1.0, επιτρέποντας την ενσωμάτωσή του σε άλλα, μεγαλύτερα ενσωματωμένα συστήματα. Η ενσωμάτωση αυτή, διευκολύνεται και από το γεγονός ότι η Myriad X δεν απαιτεί κάποιο εξωτερικό σύστημα ψύξης, χάρη στην χαμηλή της κατανάλωση.

- **Μία πολυπύρηνη, ετερογενή αρχιτεκτονική,** που βασίζεται σε:

  - 2x 32-bit SPARC V8 RISC επεξεργαστές (LEONs), ιδιαίτερα ανθεκτικούς στην ακτινοβολία
  - 16x VLIW 128-bit Vector επεξεργαστές (SHAVEs) βελτιστοποιημένους για εφαρμογές όρασης υπολογιστών
  - 1x Neural Compute Engine, έναν ειδικό επιταχυντή για βαθιά συνελικτικά νευρωνικά δίκτυα

- Επιταχυντές υλικού για επεξεργασία εικόνας και βίντεο
- 2.5MB SRAM μνήμη (CMX), που βρίσκεται πάνω στο ολοκληρωμένο κύκλωμα
- 512MB LPDDR4 μνήμη
- Κρυφές μνήμες πολλαπλών επιπέδων με επιλογές διαμόρφωσης κατά το run-time

- **Υποστήριξη για ένα μεγάλο πλήθος διεπαφών για I/O περιφερειακών συσκευών,** όπως MIPI, SPI, I2C, I2S, SDIO, UART, USB3, GbE, CIF.



Σχήμα 9: Διάγραμμα της Αρχιτεκτονικής της Myriad X [5]

Εργαλεία

Ο προγραμματισμός μίας ετερογενούς πλατφόρμας αποτελεί μία απαιτητική εργασία. Για την συγκεκριμένη VPU, προσφέρεται τόσο η δυνατότητα προγραμματισμού σε χαμηλό επίπεδο, μέσω του Myriad X Development Kit, όσο και σε υψηλό επίπεδο, μέσω του OpenVINO Toolkit. Στόχος αυτής της διπλωματικής, ήταν ο συνδυασμός των δύο μεθόδων παραγωγής κώδικα, με στόχο την μεγιστοποίηση της παραγωγικότητας αλλά και της αποδοτικότητας της εφαρμογής.

Myriad X Development Kit

Το Myriad X Development Kit (MDK) παρέχεται από τον κατασκευαστή της VPU, και πρόκειται για ένα σύνολο από βιβλιοθήκες, drivers, εγχειρίδια χρήσης και εργαλεία, τα οποία είναι απαραίτητα για την ανάπτυξη εφαρμογών για την πλατφόρμα αυτή. Ένα από αυτά τα εργαλεία, είναι και το σύνθετο σύστημα μεταγλωττισμού εφαρμογών για την πλατφόρμα, το οποίο βασίζεται στο GNU Makefile. Επίσης, αξίζει να σημειωθεί ότι στο φάκελο mdk/common/components περιέχεται πληθώρα επαναχρησιμοποιούμενων component της VPU. Στα πλαίσια της παρούσας διπλωματικής, έγινε χρήση των παρακάτω component:

- Για αποσφαλμάτωση και έλεγχο του κώδικα:
  - PipePrint
  - UnitTest
- Για μέτρηση και διαχείριση της καταναλισκώμενης ισχύος:

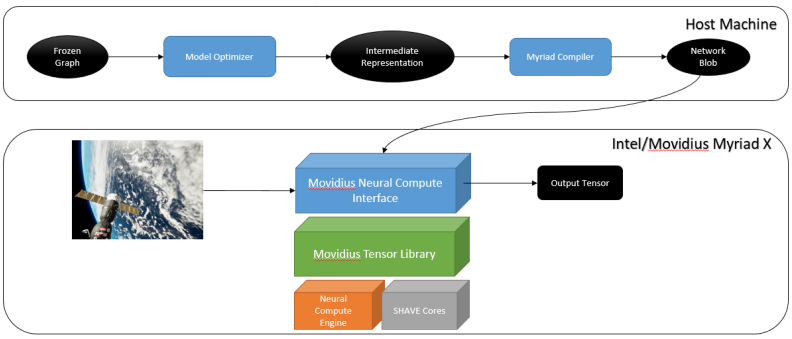
- MV0235
- PwrManager
- MVBoardsCommon

- Για δυναμική διαχείριση της μνήμης:
  - MyriadMemInit
  - MemoryManager
- Για την εκμετάλλευση του Neural Compute Engine:
  - MvNCI
  - MvTensor
- Για διαχείριση αρχείων:
  - VcsHooks

OpenVINO Toolkit

Το OpenVINO Toolkit είναι ένα εργαλείο που έχει σχεδιαστεί από την Intel, με στόχο την ουσιαστική ελάττωση του απαραίτητου χρόνου ανάπτυξης εφαρμογών, οι οποίες υλοποιούν διάφορες τυπικές λειτουργίες των ΤΝΔ, όπως η προσομοίωση της ανθρώπινης όρασης και η επεξεργασία φυσικής γλώσσας. Υποστηρίζει την ετερογενή εκτέλεση των εφαρμογών αυτών, καθώς είναι συμβατό με ένα μεγάλο εύρος πλατφόρμων της Intel, από επεξεργαστές και ενσωματωμένες μονάδες γραφικών, έως FPGAs και VPUs. Με αυτό τον τρόπο, η εξαγωγή αποτελεσμάτων μέσω νευρωνικων δικτύων επιταχύνεται και η διαδικασία ενσωμάτωσης της ετερογενούς εκτέλεσής τους απλοποιείται σημαντικά.

Η ροή εργασιών που ακολουθείται για την χρήση του OpenVINO Toolkit για την ενσωμάτωση νευρωνικών δικτύων στην αναπτυσσόμενη εφαρμογή απεικονίζεται στο παρακάτω σχήμα.



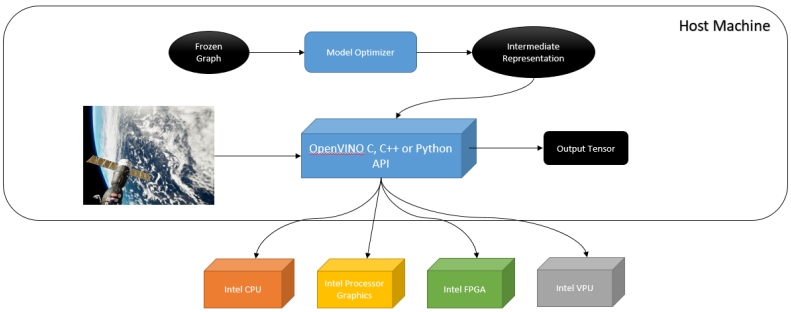
Σχήμα 10: OpenVINO Toolkit: Ροή εργασιών

Αρχικά, εξάγεται ένα Frozen Graph από το εκπαιδευμένο μοντελό του νευρωνικού δικτύου που θέλουμε να επιταχύνουμε. Στον γράφο αυτό εφαρμόζονται τεχνικές βελτιστοποίηση από το αντίστοιχο εργαλείο του OpenVINO. Παράδειγμα μίας τέτοιας βελτιστοποίησης αποτελεί η συγχώνευση μη εκπαιδεύσιμων σταδίων σε προηγούμενα ή επόμενα στάδια εντός του γράφου του νευρωνικού. Η ενδιάμεση αναπαράσταση που προκύπτει, τροφοδοτείται στον ειδικό μεταγλωττιστή για την VPU και παράγεται ένα δυαδικό αρχείο. Όλες αυτές οι λειτουργίες εκτελούνται εκτός της VPU.

Από την πλευρά της Myriad X, γίνεται χρήση της υπάρχουσας διεπαφής, για ανάγνωση του δυαδικού αρχείου, μέσω της οποίας φορτώνεται το προς εκτέλεση νευρωνικό στη μνήμη και δεσμεύονται οι

απαραίτητοι υπολογιστικοί πόροι. Στην συνέχεια, μπορούμε να αιτηθούμε εκτέλεση του νευρωνικού για κάποια εικόνα εισόδου και να λάβουμε κάποιοι αποτέλεσμα/πρόβλεψη. Η διεπαφή αυτή χτίζεται πάνω στην Tensor βιβλιοθήκη της Movidius, η οποία αναλαμβάνει την ανάθεση της εκτέλεσης των στρωμάτων που διαβάζονται είτε στον ειδικό επιταχυντή, είτε στους vector επεξεργαστές.

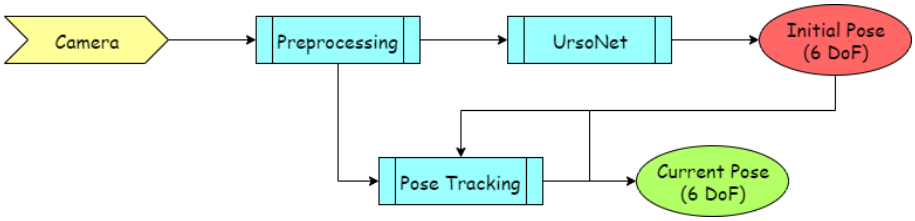
Η παραπάνω ροή εργασιών περιγράφει τον τρόπο με τον οποίο ενσωματώθηκε η εκτέλεση ενός νευρωνικού σε μια μεγαλύτερη εφαρμογή που εκτελείται στην πλακέτα MV0235. Ωστόσο, εάν επιθυμούμε να επιταχύνουμε μονάχα την εκτέλεση του νευρωνικού δικτύου, και οχι την εφαρμογή που το φιλοξενεί συνολικά, τότε μπορούμε να χρησιμοποιήσουμε την παρακάτω, εναλλακτική ροή εργασιών.



Σχήμα 11: OpenVINO Toolkit: Εναλλακτική Ροή εργασιών

Η κύρια διαφορά σε σχέση με την προηγούμενη ροή εργασιών, είναι ότι η εφαρμογή αναπτύσσεται εξ ολοκλήρου σε μία γλώσσα υψηλού επιπέδου και, με εξαίρεση την εκτέλεση του νευρωνικού, δεν εκτελείται πάνω στην VPU. Η δομή του νευρωνικού δικτύου διαβάζεται από την ενδιάμεση αναπαράστασή του και φορτώνεται στην επιλεγμένη πλατφόρμα. Στην συνέχεια, η εκτέλεσή του για κάποια εικόνα εισόδου εκφορτώνεται στην επιλεγμένη πλατφόρμα προς επιτάχυνση. Η πλατφόρμα αυτή επιστρέφει στο κυρίως σύστημα την έξοδο του νευρωνικού δικτύου. Με αυτή τη ροή εργασιών, προγραμματίζεται το Neural Compute Stick 2, ένα USB stick, το οποίο περιέχει πάνω μία Myriad X VPU.

Προτεινόμενη Εφαρμογή και Υλοποίηση



Σχήμα 12: Προτεινόμενο Σύστημα για Εκτίμηση και Παρακολούθηση Πόζας Δορυφόρων

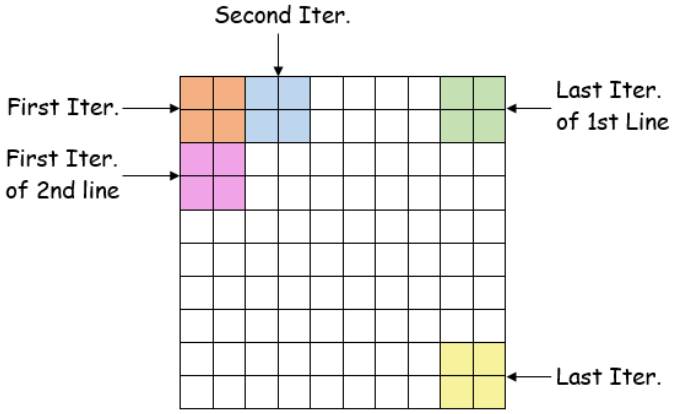
Στην παραπάνω εικόνα απεικονίζεται το προτεινόμενο σύστημα, μέσω του οποίου σχεδιάστηκε η επίλυση του προβλήματος "Lost in Space", σε πραγματικό χρόνο, στο ίδιο SoC. Η κάμερα τροφοδοτεί το στάδιο προεπεξεργασίας με έγχρωμες εικόνες ανάλυσης 1024x1024. Τα δεδομένα αυτα υφίστανται υποδειγματοληψία, ώστε οι διαστάσεις της εικόνας να μειωθούν στο μισό. Η τροποποιημένη εικόνα, διαστάσεων 512x512 χρησιμοποιείται, εν συνεχεία, για την εκτίμηση της αρχικής πόζας του δορυφόρου. Η αρχική αυτή πόζα εξελίσσεται σε βάθος χρόνου με χρήση κλασσικών τεχνικών όρασης

υπολογιστών. Ο αλγόριθμος που υλοποιεί την εν λόγω λειτουργία, βασίζεται σε προηγούμενη δουλειά του εργαστηρίου ([20]). Ο αλγόριθμος αυτός λειτουργεί πάνω σε ασπρόμαυρες φωτογραφίες διαστάσεων 1024x1024, συνεπώς το στάδιο της προεπεξεργασίας εκτελεί την λειτουργία του συνδυασμού των τριών χρωματικών καναλιών σε ένα.

Στην παρούσα διπλωματική, υλοποιούμε το σύστημα που πραγματοποιεί την αρχική εκτίμηση της πόζας του δορυφόρου, όπως έχει περιγραφεί παραπάνω. Το στάδιο της προεπεξεργασίας έχει προγραμματιστεί με χαμηλού επιπέδου κώδικα, και έχουν εφαρμοστεί διάφορες τεχνικές βελτιστοποίησης, ώστε να αποφευχθεί η δημιουργία στενωπού. Το στάδιο της εκτέλεσης του νευρωνικού δικτύου έχει ενσωματωθεί με χρήση του εργαλείου OpenVINO, με στόχο τη μείωση του προγραμματιστικού φόρτου που εμπεριέχει η ενσωμάτωση αυτή.

Στάδιο Προεπεξεργασίας: Βελτιστοποιήσεις

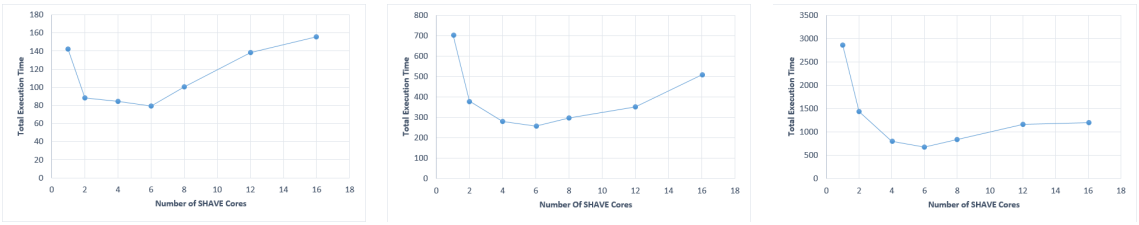
Οι τρεις μεθόδου υποδειγματοληψίας των δεδομένων βασίζονται στην πράξη της συνέλιξης. Συνεπώς, για την υλοποίησή τους αρκεί να υλοποιηθεί η πράξη αυτή με έναν υπολογισμένο πυρήνα. Ο κώδικας δομείται μέσω μίας επαναληπτικής εργασιάς, όπου γίνεται διαδοχικά ανάγνωση των απαραίτητων γραμμών εισόδου, εκτέλεση της πράξης της συνέλιξης, εγγραφή της παραγόμενης γραμμής εξόδου. Όπως γίνεται αντιληπτό, μεταξύ εκτελέσεων αυτής της επαναληπτικής εργασίας, δεν εντοπίζονται εξαρτήσεις οποιουδήποτε είδους. Αυτό μας επιτρέπει την εν-παραλλήλω εκτέλεση του αλγορίθμου, με την ανάθεση παραγωγής συγκεκριμένου αριθμού γραμμών εξόδου σε κάθε πυρήνα.



Σχήμα 13: Εκτέλεση Διγραμμικής Παρεμβολής για Υποδειγματοληψία Εικόνας

Παρόλα αυτά, κατά την μελέτη της επίδρασης της παραλληλοποίησης στο χρόνο εκτέλεσης των τριών αλγορίθμων, παρατηρείται ότι επέρχεται κορεσμός για χρήση 6 πυρήνων, και, για παραλληλοποίηση σε παραπάνω πυρήνες, παρατηρείται σταδιακή αύξηση του χρόνου εκτέλεσης σε σχέση με τον ελάχιστο. Μάλιστα, στην περίπτωση της διγραμμικής παρεμβολής, η σειριακή εκτέλεση απαιτεί λιγότερο χρόνο από την παράλληλη με χρήση του μέγιστου αριθμού πυρήνων!

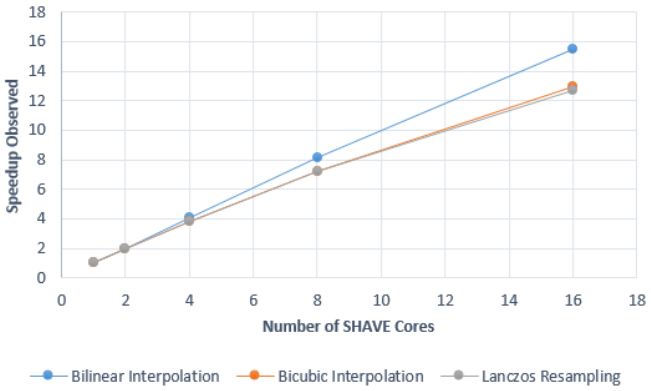
Η συμπεριφορά αυτή οφείλεται στο γεγονός ότι όλοι οι πυρήνες διαβάζουν τα απαραίτητα δεδομένα και γράφουν τα αποτελέσματά τους σε μία κοινή μνήμη. Όταν ο αριθμός των διαθέσιμων πυρήνων υπερβεί τον αριθμό των διαθέσιμων ports της μνήμης, αναπόφευκτα κάποιοι από αυτούς δεν θα εξασφαλίσου πρόσβαση σε πόρους της μνήμης. Συνεπώς, θα εισάγουν καθυστερήσεις στην εκτέλεσή τους, ώστε να περιμένουν την απελευθέρωση κάποιου port. Όσο αυξάνεται ο αριθμός των πυρήνων, τόσο μεγαλώνει και ο ανταγωνισμός μεταξύ τους για πόρους της μνήμης, και συνεπώς, παρατηρείται η σταδιακή αύξηση του απαιτούμενου χρόνου εκτέλεσης.



(α′) Διγραμμική Παρεμβολή (β′) Δικυβική Παρεμβολή (γ′) Επαναδειγματοληψία Lanczos

Σχήμα 14: Κλιμάκωση Χρόνου Εκτέλεσης στους SHAVE Πυρήνες

Η αποφυγή του ανταγωνισμού μεταξύ πυρήνων για πόρους της κοινής μνήμης επιτυγχάνεται αν ο κάθε πυρήνας ρυθμιστεί να διαβάζει και να γράφει δεδομένα από το κομμάτι της Scratchpad μνήμης, στο οποίο διαθέτει το ελάχιστο κοστος προσπέλασης. Για να μεταφέρουμε τα απαραίτητα δεδομένα από την DDR στην Scratchpad μνήμη, και το αντίστροφο, χρησιμοποιούνται DMA δοσοληψίες μεταξύ των δύο μνημών. Συγκεκριμένα, κάθε επανάληψη του αλγορίθμου τροποποιείται, ώστε αρχικά να περιέχει μία δοσοληψία για τη μεταφορά των γραμμών της αρχικής εικόνας που είναι απαραίτητες για τον υπολογισμό μίας γραμμής της τελικής εικόνας, ακολούθως να εκτελείται η επιλεγμένη μέθοδος υποδειγματοληψίας και τελικά, η παραγόμενη γραμμή της εικόνας εξόδου να μεταφέρεται πίσω στην DDR μνήμη μέσω δοσοληψίας. Με αυτό το τρόπο, όλες οι αναγνώσεις και εγγραφές δεδομένων κατά την εκτέλεση της μεθόδου υποδειγματοληψίας, που απαιτεί και το υπολογιστικά απαιτητικό κομμάτι αυτού του σταδίου, γίνονται στην γρήγορη Scratchpad, η οποία διαθέτει τον απαραίτητο αριθμο ports, για να εξυπηρετεί όλους τους SHAVE πυρήνες παράλληλα.

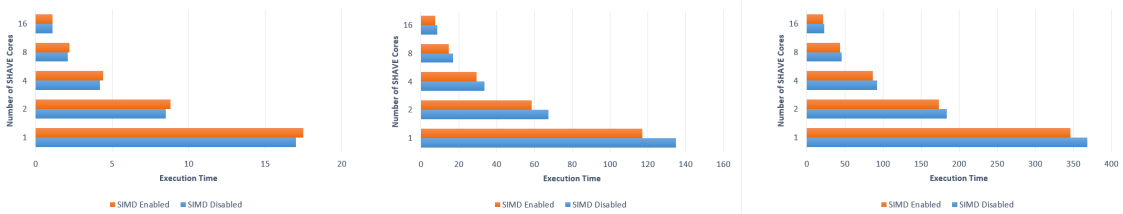


Σχήμα 15: Επιτάχυνση κατά την Παραλληλοποίηση με χρήση της Scratchpad μνήμης

Παρατηρείται γραμμική επιτάχυνση για την μέθοδο της διγραμμικής παρεμβολής. Ωστόσο, παρατηρούμε επίσης ότι υπάρχει χώρος για επιπλέον βελτιστοποιήσεις για τις άλλες δύο μεθόδους. Η ειδοποιός διαφορά μεταξύ της δικυβικής παρεμβολής και της επαναδειγματοληψίας Lanczos σε σχέση με την διγραμμική παρεμβολή είναι η χρήση padding. Τα αρχικά δεδομένα διαθέτουν padding με σταθερή τιμή, συγκεκριμένα το μηδέν. Για την εκτέλεση, ωστόσο, των αλγορίθμων αυτών απαιτείται padding με επανάληψη των τιμών των συνόρων της εικόνας (replication padding). Η προσθήκη αυτού του padding στα δεδομένα εισόδου γίνεται απο έναν LEON πυρήνα, με αναγνώσεις και εγγραφές στην αργή DDR μνήμη και απαιτεί 2.1 και 6.2 ms αντίστοιχα για τις δύο μεθόδους. Είναι προφανές, ότι αυτή η επαναληπτική διαδικασία δεν εμπεριέχει εξαρτήσεις μεταξύ επαναλήψεων και συνεπώς μπορεί να παραλληλοποιηθεί με τον ίδιο τρόπο που αναλύθηκε νωρίτερα. Με χρήση της Scratchpad μνήμης και αναθέτοντας σε κάθε SHAVE πυρήνα να δημιουργεί ο ίδιος το padding που θα χρειαστεί, στα δε-

δομένα εισόδου του, επιτυγχάνεται η πλήρης εξάλειψη της καθυστέρηση αυτής, αφού δεν παρατηρείται κάποια αύξηση στο χρόνο εκτέλεσης των SHAVE.

Η τελευταία βελτιστοποίηση που εφαρμόζεται, έγκειται στην χρησιμοποίηση των SIMD δυνατοτήτων των SHAVE πυρήνων, με στόχο την πλήρη εκμετάλλευση του παραλληλισμού σε επίπεδο δεδομένων. Οι SHAVE πυρήνες διαθέτουν αριθμητικές μονάδες, οι οποίες μπορούν να εκτελέσουν προσθέσεις και πολλαπλασιασμούς floating-point αριθμών με SIMD τρόπο. Για τις μεθόδους της δικυβικής παρεμβολής και της επαναδειγματοληψίας Lanczos, παρατηρείται βελτίωση 5-15% σε σχέση με την προηγούμενη βελτιστοποιημενη εκδοχή της υλοποίησης. Για την περίπτωση της διγραμμικής παρεμβολής, ωστόσο, παρατηρούμε συνολικά αύξηση του χρόνου εκτέλεσης. Αυτό συμβαίνει διότι οι SIMD εντολές εκτελούνται σε vector καταχωρητές των 128 bit.Για την εκτέλεση της διγραμμικής παρεμβολής, απαιτείται η άθροιση μόνο τεσσάρων 8-bit αριθμών. Συνεπώς, οι υπόλοιπες θέσεις του καταχωρητή συμπληρώνονται με μηδενικά, τα οποία ωστόσο συμμετέχουν στις προσθέσεις και εισάγουν μία επιπλέον καθυστέρηση, η οποία υπερκαλύπτει τα όποια οφέλη της SIMD εκτέλεσης.

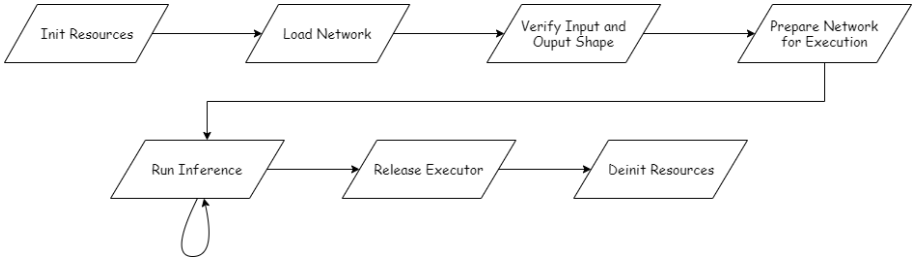


(α′) Διγραμμική Παρεμβολή (β′) Δικυβική Παρεμβολή (γ′) Επαναδειγματοληψία Lanczos

Σχήμα 16: Κλιμάκωση του Χρόνου Εκτέλεσης στους SHAVE Πυρήνες, με τις SIMD δυνατοτήτες ενεργοποιημένες/απενεργοποιημένες

Στάδιο CNN Inferencing

Για την ενσωμάτωση της εκτέλεσης του νευρωνικού δικτύου στην εφαρμογή μας, ακολουθούμε την τυπική ροή εργασιών του OpenVINO, όπως αυτή περιγράφηκε προηγουμένως. Η χρήση της προσφερόμενη διεπαφής για φόρτωση και εκτέλεση του νευρωνικού δικτύου στην VPU είναι εξαιρετικά απλή και αποτελείται από τα ακόλουθα βήματα:



Σχήμα 17: Ροή Εργασιών για χρήση του Neural Compute Interface

Παρατίθεται μία σύντομη εξήγηση της λειτουργίας κάθε σταδίου:

- **Αρχικοποίηση πόρων:** Αρχικά, η εφαρμογή πρέπει να αρχικοποιήσει τους απαραίτητους πόρους του ειδικού επιταχυντή συνελικτικών δικτύων.
- **Φόρτωση Δικτύου:** Φόρτωση της δομής του δικτύου στην μνήμη. Η δομή διαβάζεται από το δυαδικό αρχείο που δίνεται ως είσοδος.

- **Έλεγχος Διαστάσεων Εισόδου και Εξόδου:** Λήχη των διαστάσεων της εισόδου και της εξόδου του δικτύου και έλεγχος σχετικά με το αν είναι οι αναμενόμενες.

- **Προετοιμασία για εκτέλεση:** Δέσμευση των απαραίτητων πόρων του επιταχυντή για την τρέχουσα εκτέλεση.

- **Εκτέλεση:** Τα δεδομένα εισόδου τροφοδοτούνται στο δίκτυο και λαμβάνεται η απόκρισή του. Η συγκεκριμένη συνάρτηση αναμένεται να καλείται πολύ συχνά και να καταλαμβάνει το μεγαλύτερο μέρος του συνολικού χρόνου εκτέλεσης.

- **Απελευθέρωση εκτέλεσης:** Απελευθερώνονται όλοι οι πόροι που είχαν δεσμευτεί για την τρέχουσα εκτέλεση του νευρωνικού.

- **Αποαρχικοποίηση πόρων:** Η αντίστροφη διαδικασία της αρχικοποίησης.

Η συγκεκριμένη διεπαφή έχει εσωτερικά πρόσβαση σε δύο διαφορετικά σύστήματα διαχείρισης: ένα σύστημα διαχείρισης πόρων και ένα σύστημα διαχείρισης ισχύος.

Το σύστημα διαχείρισης πόρων είναι υπεύθυνο για την καταγραφή των πόρων που χρησιμοποιούνται για κάποια εκτέλεση αλλά και των πόρων που είναι διαθέσιμοι. Αυτό συμβαίνει, διότι η διεπαφή δίνει τη δυνατότητα της ασύγχρονης εκτέλεσης. Αυτό σημαίνει ότι πολλαπλά αιτήματα μπορεί να δρομολογηθούν προς εκτέλεση εν παραλλήλω, και συνεπώς πολλαπλά αντίγραφα του νευρωνικού μπορεί να χρειαστεί να τρέξουν ταυτόχρονα. Απο την άλλη πλευρά, ένα σύστημα διαχείρισης πόρων είναι επίσης απαραίτητο για την εξακρίβωση της δυνατότητας εκτέλεσης ενός μοναδικού νευρωνικού. Δεδομένου ότι το συγκεκριμένο σύστημα είναι ενσωματωμένο στην προσφερόμενη διεπαφή, δεν απαιτείται κάποια ενέργεια από την πλευρά του προγραμματιστή για την χρησιμοποίησή του.

Ομοίως, το σύστημα διαχείρισης ισχύος χρησιμοποιείται εσωτερικά της διεπαφής. Σε αντίθεση ωστόσο με το σύστημα διαχείρισης πόρων, ο προγραμματιστής είναι σε θέση να επιλέξει τον τρόπο λειτουργίας του. Οι διαθέσιμες επιλογές είναι:

- Κανονική λειτουργία, όπου όλοι οι πόροι είναι ενεργοί

- Ενεργοποίηση ανά Αίτημα, όπου οι πόροι ενεργοποιούνται όταν χρειαστεί να εξυπηρετήσουν κάποιο αίτημα για εκτέλεση του νευρωνικού και απενεργοποιούνται ακολούθως

- Ενεργοποίηση ανά Στρώμα, όπου οι πόροι για την εκτέλεση ενός στρώματος του νευρωνικού δικτύου ενεργοποιούνται λίγο πριν την εκτέλεση και απενεργοποιούνται αμέσως μετά

- Ενεργοποίηση ανά Στρώμα, για στρώματα που εκτελούνται στους SHAVE πυρήνες

- Ενεργοποίηση ανά Στρώμα, για στρώματα που εκτελούνται στις μονάδες του ειδικού επιταχυντή

- Μεταβατικό στάδιο, όπου η λειτουργία του συστήματος απενεργοποιείται.

Ο τρόπος λειτουργίας που θα επιβάλει το σύστημα διαχείρισης ισχύος πρέπει να επιλεγεί προσεκτικά, καθώς μπορεί να επηρεάσει τις επιδόσεις της συνολικής εφαρμογής. Η ενεργοποίηση νησίδων ισχύος της πλατφόρμας, οι οποίες είχαν προηγουμένως απενεργοποιηθεί, εισάγει μία αξιοσημείωτη καθυστέρηση. Συνεπώς, η βελτιστες επιδόσεις από άποψη χρόνου εκτέλεσης επιτυγχάνονται, όταν επιλέγεται η κανονική λειτουργία και το σύστημα προθερμαίνεται με κάποιο αρχικό αίτημα εκτέλεσης, με κόστος φυσικά την αυξημένη κατανάλωση ισχύος. Αυτός είναι και ο τρόπος λειτουργίας που επιλέχθηκε στην συγκεκριμένη εφαρμογή.

Προτεινόμενο σύστημα για Μέτρηση και Διαχείριση Ισχύος

Σε αντίθεση με το στάδιο εκτέλεσης του επιλεγμένου νευρωνικού δικτύου, το οποίο διαθέτει, όπως επισημάνθηκε, ένα ιδιωτικό σύστημα διαχείρισης ισχύος, το στάδιο προεπεξεργασίας, δεν διαθέτει κάποιο σύστημα τέτοιου είδους. Ανεξαρτήτως του αριθμού των SHAVE πυρήνων που χρησιμοποιούνται, όλοι οι διαθέσιμοι πυρήνες παραμένουν ενεργοί κατά την εκτέλεση των αλγορίθμων, καταναλώνοντας άσκοπα ενέργεια. Παράλληλα, δεν υπάρχει κάποιο σύστημα το οποίο να καταγράφει την καταναλισκόμενη ισχύ της εφαρμογής.

Η μέτρηση της ισχύος μπορεί να επιτευχθεί με χρήση της βοηθητικής πλακέτας MV0257. Η πλακέτα αυτή, διαθέτει 7 μετατροπείς αναλογικού σήματος σε ψηφιακό (ADCs), οι οποίοι δειγματοληπτούν τέσσερεις γραμμές έκαστος, και καταγράφουν την ένταση ή την τάση του ρεύματος στις γραμμές αυτές. Η τροφοδοσία των διάφορων συνιστωσών της πλακέτας που φιλοξενεί την VPU, γίνεται μέσω αυτών των γραμμών, συνεπώς με τις ληφθείσες μετρήσεις μπορεί να υπολογιστεί η συνολική ισχύς. Υποστηρίζεται ένας αριθμός από διαφορετικές ακρίβειες για την μέτρηση που θα ληφθεί. Αναλόγως

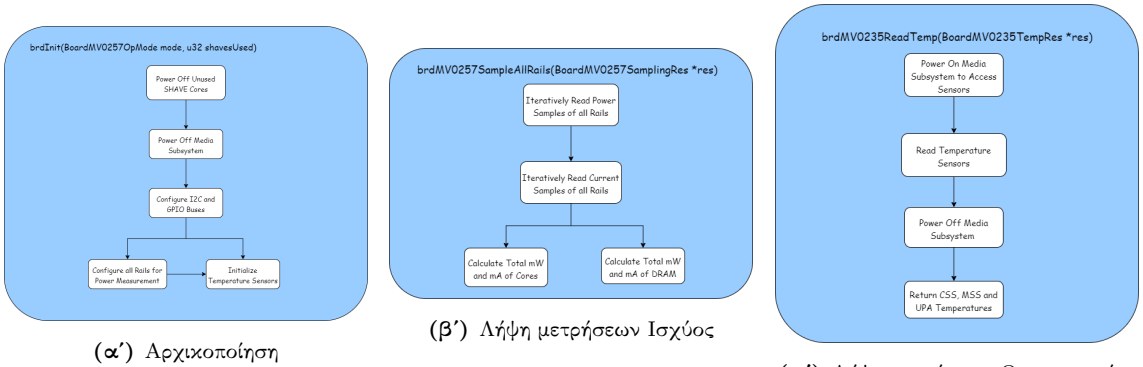
Σχήμα 18: Η πλακέτα μέτρησης ισχύος MV0257 [6]

την ακρίβεια που θα επιλεχτεί, καθορίζεται και ο χρόνος που πρέπει να διαρκέσει η δειγματοληψία για την εξαγωγή ενός δείγματος. Στην παρακάτω λίστα παρουσιάζονται οι χρόνοι αυτοί:

- Ακρίβεια 12 bits: Χρόνος δειγματοληψίας 5ms.
- Ακρίβεια 14 bits: Χρόνος δειγματοληψίας 17ms.
- Ακρίβεια 16 bits: Χρόνος δειγματοληψίας 67ms.
- Ακρίβεια 18 bits: Χρόνος δειγματοληψίας 267ms.

Η σωστή αρχικοποίηση όλων των απαραίτητων components για τη διαδικασία της λήψης μετρήσεων είναι μια κουραστική διαδικασία, η οποία απαιτεί από τον προγραμματιστή γνώση της υποκείμενης αρχιτεκτονικής. Στην παρούσα διπλωματική, σχεδιάσαμε μία διεπαφή, η οποία εσωτερικά υλοποιεί όλες τις απαραίτητες λειτουργίες για αρχικοποίηση και χρήση της πλακέτας MV0257, και η οποία εφαρμόζει μία απλή, στατική τεχνική διαχείρισης ισχύος. Στόχος της διεπαφής, είναι ο προγραμματιστής για την λήψη μίας μέτρησης, να χρειαστεί να καλέσει μόλις τρεις συναρτήσεις: μία για αρχικοποίηση, μία για λήψη μέτρησης και μία για τερματισμό. Εσωτερικά, ωστόσο, οι καλούμενες συναρτήσεις θα υλοποιούν μια πληθώρα εργασιών, οι οποίες σε διαφορετική περίπτωση θα έπρεπε να ενσωματωθούν στον κώδικα από τον προγραμματιστή:

Η συνάρτηση brdInit αρχικά απενεργοποιεί όλους τους SHAVE πυρήνες που δεν πρόκειται να χρησιμοποιηθούν από την εφαρμογή. Στη συνέχει απενεργοποιεί το υποσύστημα πολυμέσων. Εν συνεχεία, αρχικοποιείται ο driver της πλακέτας MV0257 και τίθεται ο κατάλληλος τρόπος λειτουργίας των διαύλων I2C και GPIO, μέσω των οποίων θα γίνει η μεταφορά εντολών προς την πλακέτα λήψης μετρήσεων και η μεταφορά των μετρήσεων από την πλακέτα πίσω στη VPU. Αυτό οδηγεί στην δημιουργία 7 διαφορετικών file descriptors, έναν για κάθε μετατροπέα αναλογικού σε ψηφιακό. Τέλος, τίθεται ο τρόπος δειγματοληψίας των γραμμών και ο τρόπος λειτουργίας των αισθητήρων θερμοκρασίας.



(α′) Αρχικοποίηση

(β′) Λήψη μετρήσεων Ισχύος

(γ′) Λήψη μετρήσεων Θερμοκρασίας

Σχήμα 19: Λειτουργίες του Συστήματος Μέτρησης και Διαχείρισης Ισχύος

Η συνάρτηση brdMV0257SampleAllRails διαβάζει επαναληπτικά ένα δείγμα ισχύος από κάθε γραμμή. Για να συμβεί αυτό, οι μετατροπείς αναλογικού σε ψηφιακό λαμβάνουν την εντολή να ξεκινήσουν την δειγματοληψία της επιθυμητής γραμμής. Μετά από 5ms, παράγεται μία μέτρηση, η οποία διαβάζεται αμέσως. Είναι δυνατή η παράλληλη δειγματοληψία γραμμών που έχουν ανατεθεί σε διαφορετικούς μετατροπείς. Συνεπώς, ο χρόνος που απαιτείται για την λήψη μιας μέτρησης που περιλαμβάνει την συνολική κατανάλωση ισχύος είναι $4 * 5 = 20$ms.

Η συνάρτηση brdMV0235ReadTemp αρχικά ενεργοποιεί δύο νησίδες ισχύος του υποσυστήματος πολυμέσων, για να είναι δυνατή η λειτουργία των αισθητήρων. Αυτή η λειτουργία απαιτεί περίπου 100ms, για να ολοκληρωθεί. Στη συνέχεια, διαβάζονται τα 4 αρχεία τα οποία αντιστοιχούν στους αισθητήρες θερμοτητας και επιστρέφεται η τιμή τους. Οι νησίδες ισχύος του συστήματος πολυμέσων απενεργοποιούνται και πάλι για εξοικονόμηση ενέργειας.

Η συνάρτηση brdUnInit απλώς τερματίζει την λειτουργία του driver της πλακέτας MV0257.

Αξιολόγηση των Προτεινόμενων Συστημάτων

Θα αξιολογήσουμε αρχικά τις υλοποιήσεις τους σταδίου προεπεξεργασίας δεδομένων και του σταδίου εκτέλεσης του νευρωνικού, χρησιμοποιώντας τις εικόνες του συνθετικού συνόλου δεδομένων "Soyuz Hard", το οποίο περιέχει έγχρωμες εικόνες διαστάσεων 1280x960, με στόχο την μείωση των διαστάσεών τους κατά το μισό. Στον παρακάτω πίνακα φαίνονται οι επιδόσεις των αρχικών υλοποιήσεων των αλγορίθμων, οι οποίες τρέχουν σε έναν LEON ή ένα SHAVE πυρήνα και χρησιμοποιούν μόνο την DDR μνήμη.

Πίνακας 1: Αρχική Υλοποίηση των Αλγορίθμων

Μέθοδος	**Χρόνος Εκτέλεσης (LEON/SPARC)**	**Χρόνος Εκτέλεσης (SHAVE)**	**Επιτάχυνση**
Διγραμμική Παρεμβολή	468.2 ms	141.9 ms	3.3x
Δικυβική Παρεμβολή	2112 ms	702.4 ms	3x
Επαναδειγματοληψία Lanczos	7925.3 ms	2859 ms	2.8x

Ενώ οι τελικές, βέλτιστες υλοποιήσεις φαίνονται παρακάτω:

Πίνακας 2: Βέλτιστη Υλοποίηση των Αλγορίθμων

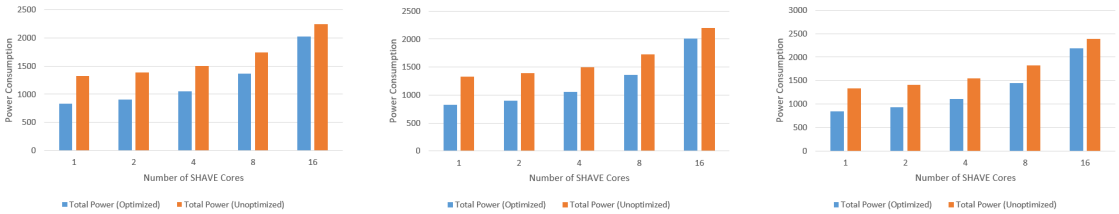
Μέθοδος	**Χρόνος Εκτέλεσης (LEON/SPARC)**	**Χρόνος Εκτέλεσης (SHAVEs)**	**Επιτάχυνση**
Διγραμμική Παρεμβολή	468.2 ms	1.1 ms	425.6x
Δικυβική Παρεμβολή	2112 ms	7.4 ms	285.4x
Επαναδειγματοληψία Lanczos	7925.3 ms	21.8 ms	363.5x

Οι τιμές αυτές δεν συνιστούν μονάχα εκτέλεση σε πραγματικό χρόνο (909, 135 και 46 FPS αντίστοιχα), αλλά και ικανοποιούν την απαίτηση να είναι σχεδόν αμελητέες μπροστά στο χρόνο εκτέλεσης του νευρωνικού δικτύου. Πράγματι, για το συγκεκριμένο μέγεθος εικόνας, το νευρωνικό μας δίκτυο απαιτεί 449.26 ms για την παραγωγή μίας εξόδου. Συνεπώς, ο χρόνος εκτέλεσης κάθε αλγορίθμου είναι τουλάχιστον μία τάξη μεγέθους μικρότερος από το χρόνο εκτέλεσης του νευρωνικού. Ο χρόνος εκτέλεσης του νευρωνικού μας δικτύου αποτιμάται για διάφορα μεγέθη εισόδου στον παρακάτω πίνακα:

Πίνακας 3: Καθυστέρηση σύγχρονης εξυπηρέτησης αιτημάτων

Μέγεθος Δεδομένων Εισόδου	**Καθυστέρηση (ms)**	**Frames Per Second**
1x192x256x3	67.40	14.84
1x512x640x3	449.26	2.23
1x960x1280x3	2297.62	0.44

Όσον αφορά την απόδοση του προτεινόμενου συστήματος στατικής διαχείρισης ισχύος, παρατηρείται ότι η εξοικονόμηση ισχύος κυμαίνεται μεταξύ 9% και 38%, όταν το σύστημα ενεργοποιείται. Στις παρακάτω γραφικές παραστάσεις, απεικονίζεται η επίδραση του συστήματος στην κατανάλωση ισχύος των τελικών υλοποιήσεων των αλγορίθμων:



(α′) Διγραμμική Παρεμβολή (β′) Δικυβική Παρεμβολή (γ′) Επαναδειγματοληψία Lanczos

Σχήμα 20: Κατανάλωση Ισχύος των Τελικών Υλοποιήσεων των Αλγορίθμων

Τέλος, αξιολογούμε πειραματικά το προτεινόμενο σύστημα για επίλυση του προβλήματος "Lost in Space" ως προς το χρόνο εκτέλεσης των επιμέρους σταδίων του. Οι χρόνοι εκτέλεσης των υλοποιήσεων μας είναι αισθητά μειωμένη, όπως είναι αναμενόμενο, λόγω των μικρότερων διαστάσεων των δεδομένων εισόδου. Η μείωση του χρόνου εκτέλεσης του σταδίου προεπεξεργασίας είναι σχετικά χαμηλή, εφόσον οι αλγόριθμοι ήδη τρέχουν πολύ αποδοτικά. Στο στάδιο της εκτέλεσης του νευρωνικού, ωστόσο, παρατηρούμε μία αξιοσημείωτη μείωση της τάξης των 80 ms. Με αυτό τον τρόπο, η συνολική καθυστέρηση των επιπλέον διεργασιών που προσθέτουμε στο ήδη υπάρχον σύστημα παρακολούθησης δορυφόρων, έχουν συγκρίσιμη καθυστέρηση με αυτό. Παραθέτουμε τις μετρήσεις μας:

Στάδιο	Καθυστέρηση		Μονάδες
Προεπεξεργασία	Διγραμμική	1,0ms	16 SHAVEs
	Δικυβική	6,3ms	
	Lanczos4	18,6ms	
Εκτέλεση ΣΝΔ	372,7ms		NCE + 7 SHAVEs
Παρακολούθηση	263-388ms		12 SHAVEs

Chapter 1

Introduction

1.1 Artificial Neural Networks

Artificial Neural Networks (ANNs) can be perceived as processing systems of distributed architecture, which have the ability to store empirical knowledge, and use it when needed to provide a response to some form of stimulus [12]. They are similar to the human brain, in that they acquire knowledge from their surroundings through a training procedure, and that this knowledge is stored in the connections (synapses) between their unit blocks (neurons) in the form of "synaptic weights". Figure 1.1 provides a comparison of the structures of a biological and an artificial unit.

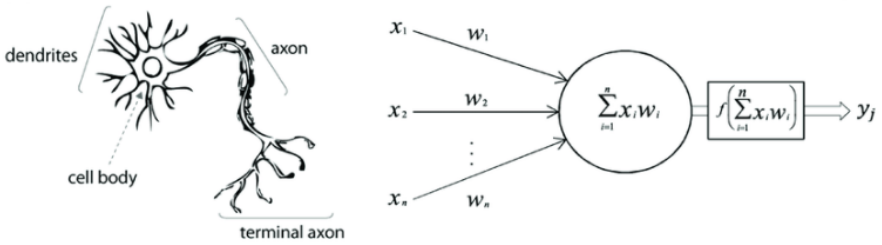


Figure 1.1: A biological, human neuron (left) in comparison to an artificial neuron (right) [1].

The neurons inside the human brain receive stimuli from the dendrites and determine whether to produce stimulus themselves through their terminal axon or not. The axon branches out and connects via synapses to dendrites of other neurons, thus creating a complex network of interconnected neuron cells [12]. Artificial neurons, inspired by the biological ones, operate on the same basis. The input signals $\{x_0, ..., x_n\}$ are fed into the neuron. Their weighted sum of these input data is then computed. The idea is that the synaptic strengths (the weights w) are learnable and control the strength of influence (and its direction: excitory (positive weight) or inhibitory (negative weight)) of one neuron on another. The final sum is considered by an activation function, which determines whether the neuron will fire or not. The output signal of the neuron is then redirected as input to other neurons, in a similar manner to the synapses formed between biological neurons, as depicted in figure 1.2.

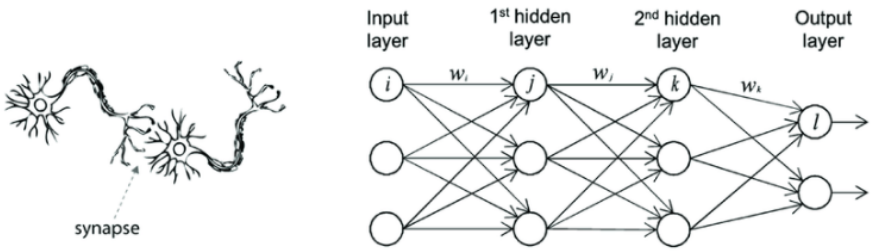


Figure 1.2: Synapses formed between biological neurons (left) and interconnected artificial neurons (right) [1].

Artificial neural networks can be trained or utilized in three different ways, with each one of them representing a different method of learning.

- In *Supervised Learning*, the network is given access to a set of labeled data, where the desired output signal for a specific input is provided. The ANN attempts to learn a model from these data in order to make predictions about unseen or future data. Here, the term supervised refers to the fact that the desired output signals of the training samples are already known [13].

- In *Unsupervised Learning*, on the other hand, the network has to deal with unlabeled data or data of unknown structure. With unsupervised learning techniques, it is possible to explore the structure of the data in order to extract meaningful information without the guidance of a known outcome variable or reward function [14].

- In *Reinforcement Learning*, the ANN attempts to improve its performance based on interactions with the environment it operates in. 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 [12]. 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.

In the scope of this thesis, we solely worked with ANNs operating in a supervised manner and, thus, we will, hereinafter, focus on the tasks an ANN can perform through Supervised Learning, i.e. classification and regression.

Classification is a subcategory of supervised learning where the goal is to predict the categorical class labels of new instances based on past observations [12]. Those class labels are discrete, unordered values that can be understood as the group memberships of the instances. The MNIST dataset is an example of sample data that can be used by a network, in order to efficiently learn to classify handwritten digits. After sufficient training, if a user provides a new handwritten digit via an input device, the predictive model will be able to predict the correct number with certain accuracy. However, the ANN would be unable to correctly recognize any of the characters of the alphabet were not part of the training dataset.

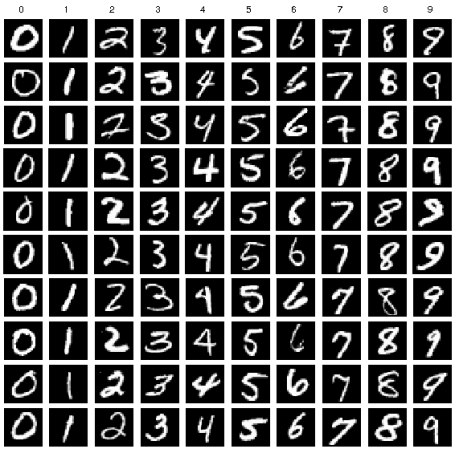


Figure 1.3: Samples taken from the MNIST dataset [2]

Regression analysis is another form of supervised learning. 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 [15]. For example, the Boston House Price dataset contains 13 numerical properties of houses in Boston suburbs and subsequently marks the price of these houses in thousands of dollars. After training, if a user provided input data containing the 13 properties required by the network, an estimation of the price of the house described could be made by the ANN.

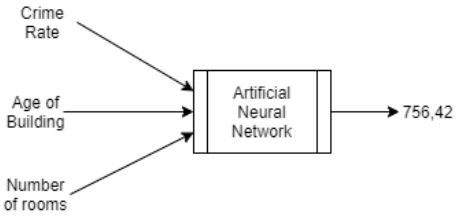


Figure 1.4: Prediction of an ANN trained on the Boston House Pricing dataset

Artificial Neural Networks have two modes of operation: Inference and Training. During inference, a forward-step computation is performed, to produce an output signal (prediction). Training is performed in two stages. Initially, given some training data, forward-step computation is performed to produce an output signal, similarly to inferencing. Then, a loss function is utilized to determine the error of the estimation compared to the expected result. Using this loss, backpropagation is performed, to modify all trainable parameters [12].

In forward-step computation, each neuron receives input signals either from neurons of earlier layers or from the input of the network itself. The weighted sum of these signals is then considered by an activation function. The output of this function is then either transferred to the neurons of the next layer, or directly presented as output of the network. This mode of operation is similar to the way biological neurons behave, as discussed earlier.

Backpropagation is an algorithm for training feedforward neural networks [12]. It functions by computing the gradient of the loss function with respect to the weights of the network for a single input-output sample, and does so efficiently, unlike a naive direct computation of the gradient with respect to each weight individually. This efficiency makes it feasible to use gradient methods for training multilayer networks, updating weights to minimize loss To this end, gradient descent, or

variants such as stochastic gradient descent, are commonly used. The backpropagation algorithm works by computing the gradient of the loss function with respect to each weight by the chain rule, computing the gradient one layer at a time, iterating backward from the last layer to avoid redundant calculations of intermediate terms in the chain rule.

1.2 Convolutional Neural Networks (CNNs)

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. 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. Considering the example of handwritten digit recognition, a CNN would be an ideal candidate to perform the task of classification of the input, since the data provided are images.

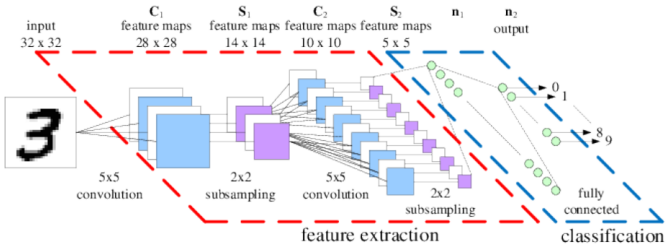


Figure 1.5: Example topology of a CNN suitable for handwritten digit recognition

1.2.1 Structure and Properties

Convolutional Neural Networks are a subcategory of Artificial Neural Networks. Therefore, they, too, are made up of neurons that have learnable weights and biases [16]. Each neuron receives some inputs, performs a dot product and optionally follows it with a non-linearity. The whole network still expresses a single differentiable score function: from the raw image pixels on one end to class scores or a continuous result at the other. And they still have a loss function on the last layer and all the methods, ie backpropagation, which were developed for learning ANNs still apply. ConvNet architectures make the explicit assumption that the inputs are images, which allows us to encode certain properties into the architecture. These then make the forward function more efficient to implement and vastly reduce the amount of parameters in the network.

To demonstrate this, we will consider the example of the CIFAR-10 dataset. This dataset contains images of size 32x32x3 (width x depth x colour channels), labelled according to the object or animal they depict. A regular ANN, as described in 1, would demand 32x32x3 = 3072 trainable parameters for a single neuron of the first hidden layer. Clearly, this fully-connected structure does not scale to larger images. For example, an image of more respectable size, e.g. 200x200x3, would lead to neurons that have 200x200x3 = 120,000 weights. Moreover, it is useful to have several such neurons, so the parameters would add up quickly! This full connectivity proves to be wasteful and the huge number of parameters would quickly lead to overfitting.

CNNs, on the other hand, take advantage of the fact that the input consists of images and they constrain the architecture in a more sensible way [17]. In particular, unlike a regular Neural Network, the layers of a ConvNet have neurons arranged in 3 dimensions: width, height, depth. Note that the word depth here refers to the third dimension of an activation volume, not to the depth of a full Neural Network, which can refer to the total number of layers in a network. For example, 32x32 RGB input images represent an input volume of activations, that has dimensions 32x32x3. 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. This constrained, local connectivity

of CNNs was inspired by biological processes in that the connectivity pattern between neurons resembles the organization of the animal visual cortex, where individual neurons respond to stimuli only in a restricted region of the visual field.

There are three main types of layers to build CNN architectures: Convolutional Layers, Pooling Layers, and Fully-Connected Layers. 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 (kernel) 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). This type of layer is not trainable.
- **Fully Connected/Dense 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 ltype of layer is not trainable and leaves the size of the input volume unchanged.

1.2.2 ResNet Architecture

Residual Neural Networks (ResNets) [3] will be used throughout this thesis and therefore, it is essential that the reader understands how these types of networks function. Residual networks were developed to avoid the problem of vanishing gradients and to deal with the degradation in accuracy, that was observed when building very deep networks. The vanishing gradient problem is encountered when training artificial neural networks with gradient-based learning methods and backpropagation. Since the gradient of the loss function is computed by the chain rule, it is common that gradients corresponding to weights in earlier layers will be vanishingly small, essentially preventing them from updating their values. To tackle this problem, ResNets utilize skip connections, on shortcuts, to jump over some layers. These shortcuts are most commonly identity layers, but can also contain a single 1x1 convolutional layer. Therefore, the architecture of the basic building block is the following:

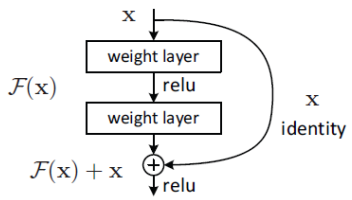


Figure 1.6: Residual Learning: a building block [3]

Due to these shortcuts, it is possible to create very deep networks by continuously stacking layers. Figure 1.8 describes the architectures of 4 different ResNets. The depth of the network is determined by the total number of filters used in the fourth convolutional stage, and is used to name the network, ie ResNet-50 contains 49 convolutional layers and one full connected layer (pooling and ReLU layers are excluded from the count).

layer name	output size	18-layer	34-layer	50-layer	101-layer	152-layer
conv1	112×112	7×7, 64, stride 2				
conv2_x	56×56	3×3 max pool, stride 2				
		$\begin{bmatrix} 3\times3, 64 \\ 3\times3, 64 \end{bmatrix}\times2$	$\begin{bmatrix} 3\times3, 64 \\ 3\times3, 64 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 64 \\ 3\times3, 64 \\ 1\times1, 256 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 64 \\ 3\times3, 64 \\ 1\times1, 256 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 64 \\ 3\times3, 64 \\ 1\times1, 256 \end{bmatrix}\times3$
conv3_x	28×28	$\begin{bmatrix} 3\times3, 128 \\ 3\times3, 128 \end{bmatrix}\times2$	$\begin{bmatrix} 3\times3, 128 \\ 3\times3, 128 \end{bmatrix}\times4$	$\begin{bmatrix} 1\times1, 128 \\ 3\times3, 128 \\ 1\times1, 512 \end{bmatrix}\times4$	$\begin{bmatrix} 1\times1, 128 \\ 3\times3, 128 \\ 1\times1, 512 \end{bmatrix}\times4$	$\begin{bmatrix} 1\times1, 128 \\ 3\times3, 128 \\ 1\times1, 512 \end{bmatrix}\times8$
conv4_x	14×14	$\begin{bmatrix} 3\times3, 256 \\ 3\times3, 256 \end{bmatrix}\times2$	$\begin{bmatrix} 3\times3, 256 \\ 3\times3, 256 \end{bmatrix}\times6$	$\begin{bmatrix} 1\times1, 256 \\ 3\times3, 256 \\ 1\times1, 1024 \end{bmatrix}\times6$	$\begin{bmatrix} 1\times1, 256 \\ 3\times3, 256 \\ 1\times1, 1024 \end{bmatrix}\times23$	$\begin{bmatrix} 1\times1, 256 \\ 3\times3, 256 \\ 1\times1, 1024 \end{bmatrix}\times36$
conv5_x	7×7	$\begin{bmatrix} 3\times3, 512 \\ 3\times3, 512 \end{bmatrix}\times2$	$\begin{bmatrix} 3\times3, 512 \\ 3\times3, 512 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 512 \\ 3\times3, 512 \\ 1\times1, 2048 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 512 \\ 3\times3, 512 \\ 1\times1, 2048 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 512 \\ 3\times3, 512 \\ 1\times1, 2048 \end{bmatrix}\times3$
	1×1	average pool, 1000-d fc, softmax				
FLOPs		1.8×10^9	3.6×10^9	3.8×10^9	7.6×10^9	11.3×10^9

Figure 1.7: ResNet Architectures [3]

An interesting property of the deeper ResNet networks, ie those with 50 or more layers, is that, despite the high amount of layers included in the network, only two pooling layers are utilized. This is because pooling layers are not trainable, and downsampling the intermediate feature maps in such a manner leads to the loss of learned features and, thus, a decrease in accuracy. Instead of pooling layers, ResNets use convolutional layers with a 1x1 kernel before and after the convolutional layers with 3x3 kernels. The 1x1 convolution is used as a projection of the input feature map onto a lower dimensional space, in order to reduce the amount of computations of the following 3x3 convolutional layer [18]. The produced feature map of that convolution is then projected to a higher dimensional space to enhance any features extracted, since the feature map is expected to be downsampled by the following layer, either convolutional or pooling.

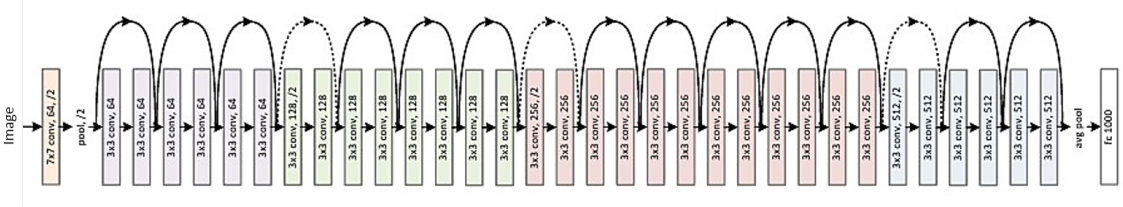


Figure 1.8: ResNet-34 [3]

1.2.3 Applications on Computer Vision

Convolutional Neural Networks have been widely adopted in the field of computer vision over the past few years, providing an interesting alternative to the classical computer vision algorithms. In fact, a single CNN architecture can be used in a plethora of different settings, to provide solutions to a wide variety of problems. All that is needed is sufficient data to train the model for the desired task... and if there is one resource that is abundant in the modern world, that has to be data. Instead of carefully crafting image processing pipelines, based on CV algorithms, such as edge detection or denoise filtering, developers can now work with a CNN, trained on a dataset that meets their needs.

Convolutional Neural Networks excel in a variety of different CV tasks, for example:

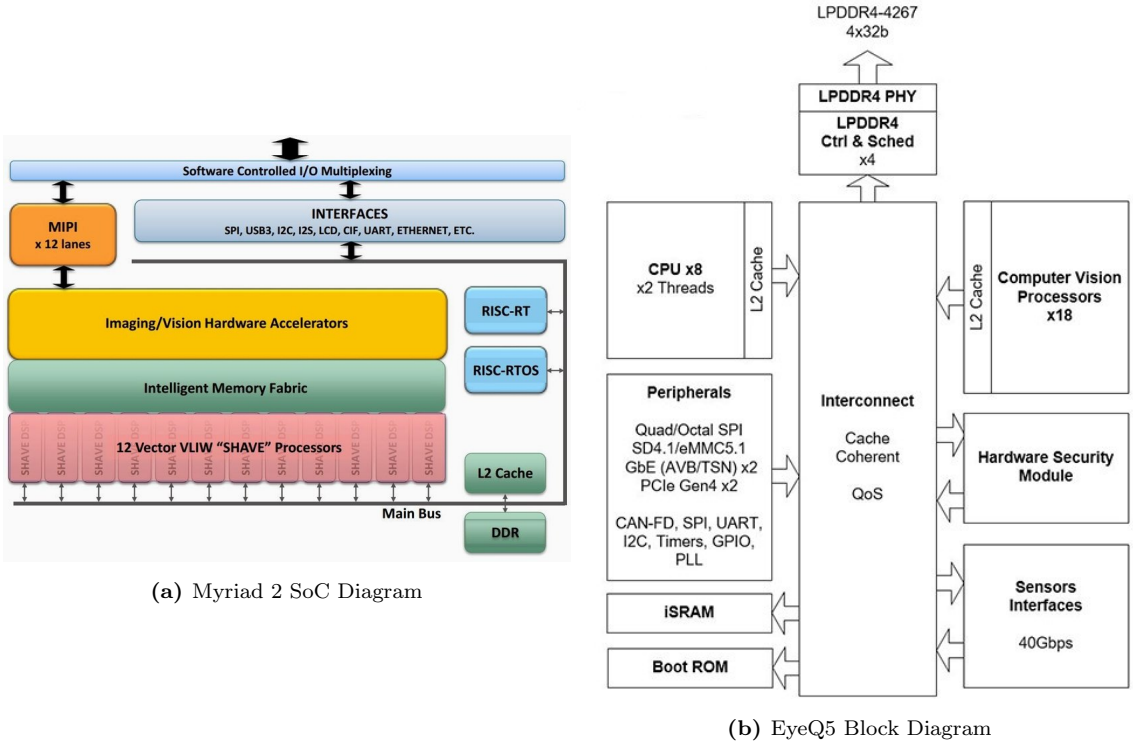
- Object Recognition/Detection
- Pose Estimation
- Facial Recognition
- Tracking
- Navigation
- Image Reconstruction

In the scope of this thesis, we studied the **"Lost in Space"** problem. The basic concept of this problem is that a spacecraft is approaching an object, but has no knowledge of its location or orientation. Such a scenario could exist, if, for example, there is memory corruption due to radiation and the data containing this information is lost, or simply, if the target object is uncooperative. Traditionally, one would create a complex pipeline of digital image processing, where the object would be detected through its edges, its pose would be estimated in comparison to known 3D models of it, and then, over time, pose refinement would be performed. On the other hand, pose estimation can be reliably and efficiently be performed by a CNN. An in-depth analysis of the architecture and the training procedure of such a CNN can be found in Section 3.2.

1.3 Vision Processing Units (VPUs)

On August 28, 2017, Intel introduced the world to their newest computing system, Myriad X, the company's first Vision Processing Unit. Granted, this may have been the first time Intel developed such a product, but such systems did already exist, i.e. Myriad 2 by Movidius, EyeQ by Mobileye, or even the "Holographic Processing Unit" found inside Microsoft's HoloLens. Thus, the need for dedicated accelerators for computer vision has existed for a long time, as were the platforms that promised to fulfill that need.

VPUs employ heterogeneity to efficiently provide acceleration [21]. General Purpose Cores are paired with SIMD and/or VLIW cores, and even dedicated Hardware Accelerators. Their memory system consists of caches and Low Power DDR memories, while some of them also include Scratchpad Memories. VPUs support multiple peripherals and emphasize on low latency transport of the data obtained through them to the corresponding memories. Figures 1.9a and 1.9b illustrate the architectures of Myriad 2 and EyeQ5 respectively.



(a) Myriad 2 SoC Diagram

(b) EyeQ5 Block Diagram

Figure 1.9: Examples of VPU Architectures

VPUs outperform both CPUs and GPUs in terms of performance per Watt [22], which renders them ideal candidates for executing computer vision tasks on the edge. For example, MyriadX can provide real-time inferencing of deep neural networks in under 1.5 or 2.5W, depending on the platform that hosts the chip. Therefore, it comes as no surprise that VPUs are considered a very promising, emerging class of microprocessors.

1.3.1 Myriad X Multicore SoC

Myriad X is the latest member of the Myriad family of VPUs developed by Movidius-Intel. This particular VPU was chosen as the target platform of the implementation of this thesis. Myriad X

provides:

- **An Ultra Low Power Design.** For devices meant to be operated on the edge, power consumption is critical, as it is directly correlated to battery life. Myriad X, not only operates on less than 2.5W, even on intensive applications, but is also capable of switching off power islands that are not being used and operating on lower frequencies than the default to further reduce power consumption. Thus, Myriad X 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 small-area footprint.** The chip has a mere 8.8mm width and 8.0mm height with 1mm depth, enabling its integration inside embedded, mobile or even wearable devices. Combined with the ultra low power consumption of the chip, Myriad X has no need for an external cooling system.
- **A high throughput, heterogeneous, multicore architecture** based on:
  - 2x radiation-tolerant 32-bit SPARC V8 RISC Processors (LEONs)
  - 16x VLIW 128-bit Vector Processors (SHAVEs) optimised for computer vision applications
  - 1x Neural Compute Engine, a dedicated accelerator for neural networks
  - Hardware Accelerators for Image and Vision Processing
  - 2.5MB of on-chip SRAM (CMX)
  - 512MB of LPDDR4 memory
  - Multi level run-time configurable Cache Infrastructure
- **Wide range of IO peripherals interfaces,** such as MIPI, SPI, I2C, I2S, SDIO, UART, USB3, GbE, CIF.

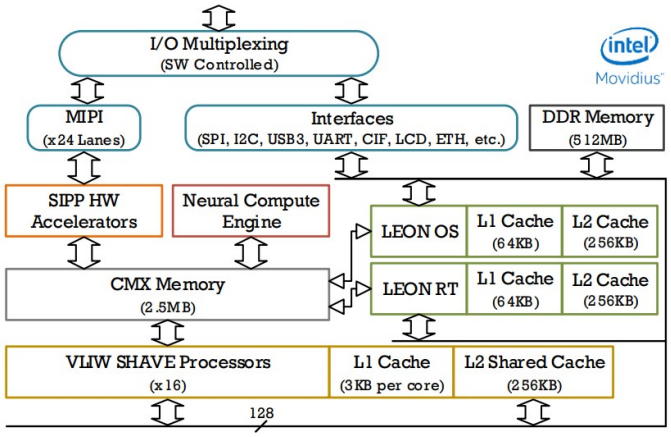


Figure 1.10: Myriad X Block Diagram [5]

Myriad X consists of three major architectural units: the CPU Sub system, the Media Sub System and the Microprocessor Array [6]. The following sections provide a detailed analysis of each one of them.

The CPU Sub System (CSS)

The CSS 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 LEON OS (LOS) RISC is the control unit of this block. The LEON OS possesses a 32KB L1 and a 256KB L2 cache enabling it to run lightweight real-time operating systems (RTOS). An AHB DMA engine is also located inside the CSS for more optimal data transfer via the external peripherals. Besides handling the external interfaces, the LEON OS typically also controls software running on the SHAVE processors. This core is considered to be the main processor of the platform, as the entry point to most applications is designed to be executed by LOS. LEON OS is a LEON4 processor based on the SPARC V8 ISA. A block diagram of its inner architecture is provided below.

The Media Sub System (MSS)

The MSS is the architectural unit designed to allow connections with imaging devices (camera sensors, Display devices, HDMI controllers etc.) as well as allowing use of the Hardware Filters. The MSS processing flows are comprised of the MIPI, LCD, CIF IO interfaces, the SIPP and CV HW filters and the AMC block which enables connections between these and CMX (SRAM) memory.

The LEON RT (LRT) RISC coordinates frame input and the MSS processing pipelines. The LEON RT has access to a direct interface with the Hardware Filters, which allows for modification of any required parameters of the MSS HW filter blocks with the minimum amount of delay due to bus arbitration. As is the case with LEON OS, LEON RT has the capability to run RTOS due to the fact that it likewise has 32KB L1 and 256KB L2 caches.

The Microprocessor Array (UPA)

The Microprocessor Array (UPA) contains 16 VLIW SHAVE ("Streaming Hybrid Architecture Vector Engine") vector processors with shared 2.5MB CMX SRAM memory. In addition, it contains a specialized DMA engine, and 256KB of shared L2 cache memory available to the SHAVE processors. This UPA's main purpose is to execute the compute-intensive parts of an image or computer vision application, by providing support for VLIW optimized code.

Each SHAVE processor contains wide and deep register files coupled with a Very Long Instruction Word (VLIW) for code-size efficiency. The processor consists of multiple functional units which have SIMD capability for high parallelism and throughput. Each of these units can be launched in parallel in a single instruction packet. SHAVE processors support 8/16/32-bit integer operations and 16/32-bit floating point operations. They include:

- 1 Integer Arithmetic Unit
- 1 Scalar Arithmetic Unit
- 1 Vector Arithmetic Unit
- 1 Compare and Move Unit
- 2 Load Store Units
- 1 Branching Unit

- 1 Predicate Execution Unit

- 1 Integer Register File

- 1 Vector Register File

A block diagram of their inner architecture is provided below.

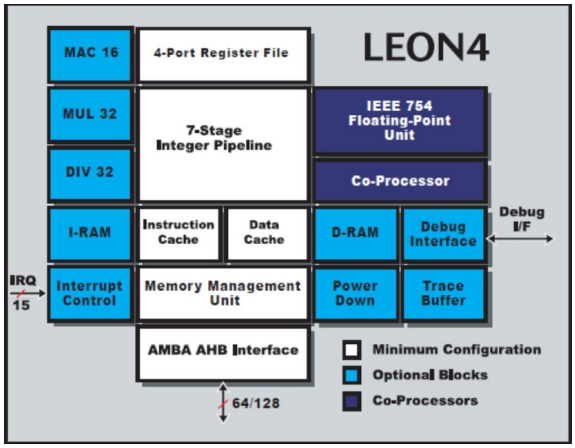


Figure 1.11: LEON4 Inner Architecture

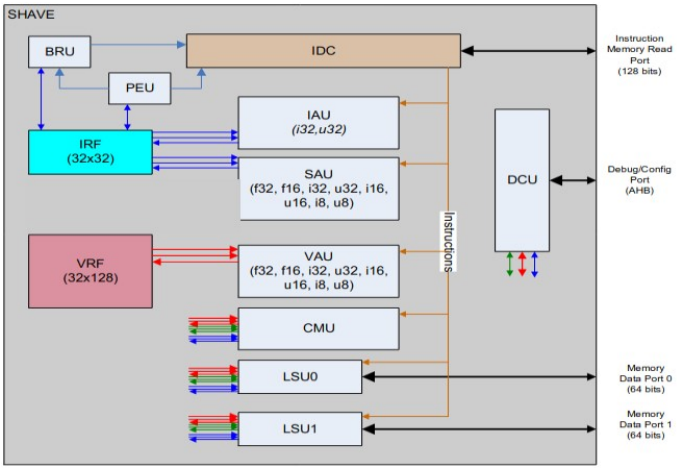


Figure 1.12: SHAVE Core Inner Architecture [6]

The following figure depicts the interconnection between the previously mentioned subsystems:

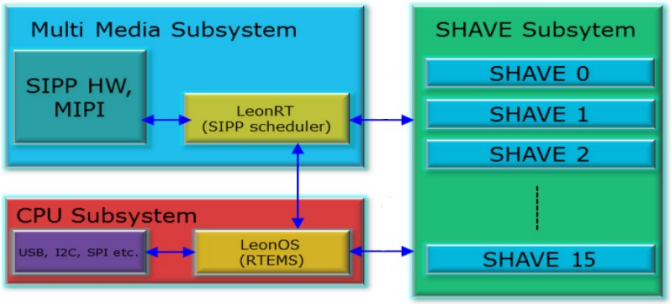


Figure 1.13: Interconnection of Subsystems [6]

The following sections are dedicated to an in-depth description of several other components of the platform, as depicted in figure 2.9.

On-Chip Memories and Cache Infrastructure

- **CMX Memory.** CMX is an abbreviation for Connection Matrix and refers to a 2.5MB Scratchpad Memory. Scratchpad is a fast, on-chip SRAM, that, unlike caches, neither flushes nor requests data to or from the main memory [23]. The CMX is comprised of 20 SRAM Blocks of 128KB each, known as "slices". It is a NUMA memory and each CMX has preferential ports to a specific slice, which offer higher bandwidth and lower access cost. Slices 16-19 are not tied to any SHAVE Core and can be freely used for other purposes i.e. storing critical parts of the operating system for better performance. However, one should keep in mind that slice locality is a weak concept, therefore each SHAVE can access any other slice in CMX at the same cost, but inter-slice routing resources are finite. Accessing data in the "local" slice is more energy-efficient though and should be exploited for optimal performance. Finally, it is worth noting that both LEON and SHAVE cores have a low access cost to the CMX Memory.

- **DDR Memory.** The main memory of the chip comprises of 512MB of volatile LPDDR4 memory. Both LEON and SHAVE cores can access data or execute code stored in the DDR. LEON Cores have access to uncached views of the DDR (as well as the CMX memory) which allows for easy sharing of control data between LEONs or between LEON and SHAVEs. On the contrary, SHAVE cores may only access the DDR memory through the cache infrastracture. For the LEON Cores, the access cost to the DDR is low when a data cache is hit, but high otherwise. Likewise, there is a high cost for random access to the DDR by the SHAVE cores, but moderate in case of an L2 cache hit, and low in case of an L1 cache hit.

- **Cache Infrastructure.** Myriad X has multiple different caches and cache hierarchies. A list of the available system caches is provided below:

Table 1.1: Myriad X System Caches

PU	Type	Size	Associativity	Cache line size	Policy
SHAVE[N]	L1 Instruction	2KB	2-way	16 bytes	read-only cache
SHAVE[N]	L1 Data	1KB	Directly Mapped	16 bytes	write-back or write-through cache
SHAVES	L2	256KB	2-way, 1-8 partitions	64 bytes	write-back cache
LeonOS	L1 Instruction	32KB	2-way	32 bytes	read-only cache
LeonOS	L1 Data	32KB	2-way	32 bytes	write-through cache
LeonOS	L2	256KB	4-way	64 bytes	write-through or copy-back cache
LeonRT	L1 Instruction	32KB	2-way	32 bytes	read-only cache
LeonRT	L1 Data	32KB	2-way	32 bytes	write-through cache
LeonRT	L2	256KB	4-way	64 bytes	write-through or copy-back cache

Image and Vision Hardware Accelerators

Myriad X employs 20 different Hardware Accelerators, targeted at speeding up Image Signal Processing or Vision Processing.The ISP accelerators include Sigma Denoise, Sharpen and Chroma Generation Filters, while the CV accelerators include Warp, Edge Operator and Harris Corner Detection Filters. Previous work, however, suggests that these filters are designed for reducing power, instead of increasing performance.

Neural Compute Engine

Myriad X is the first VPU to host Intel's Neural Compute Engine (NCE). The NCE is dedicated to accelerating Deep Neural Networks (DNNs). It is comprised of Hardware Accelerators specifically designed to execute common layers of DNNs, i.e. convolutional, pooling or dense layers, at high speed and low power without compromising accuracy. With the integration of the Neural Compute Engine, the Myriad X architecture achieves up to 1 TOPS of compute performance on DNN inferences. The engine is controlled by the MvNCI API which will be described in a later section.

Direct Memory Access Controller

Myriad X's DMA Controller resides between the 128-bit MXI bus and CMX memory. It provides high bandwidth data transfers between CMX and DRAM in either direction. It also supports data transfers from DRAM back to DRAM or from CMX to CMX, allowing data to be relocated within the same physical location.Figure XYZ shows a high level description of the DMA engine. The unit of work in the DMA engine is expressed though transaction tasks. Up to 128 linked lists of transactions are maintained in system memory, thus the DMA capability is high enough, that flooding the controller with requests would be highly unusual.

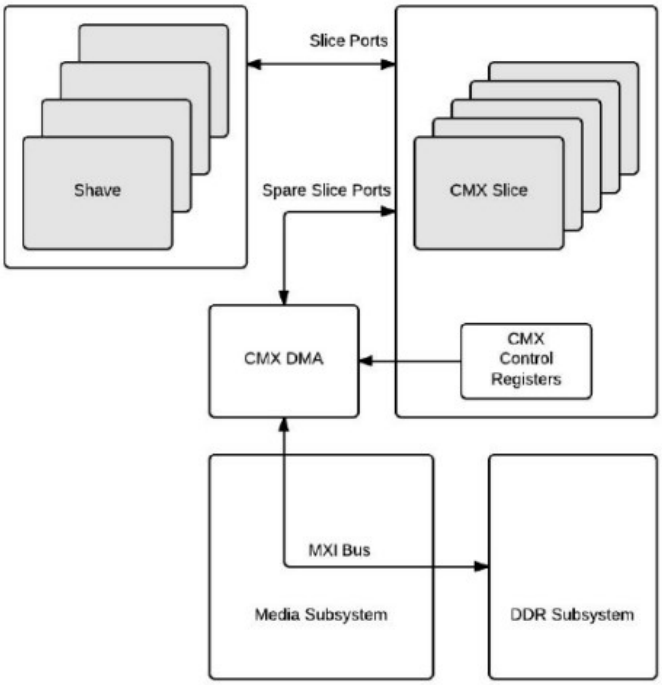


Figure 1.14: Direct Memory Access Controller Overview [5]

1.4 Related Work

1.4.1 CNN Inferencing on Embedded Devices

In recent years, multiple attempts have been made to perform operations on the edge, ie in near proximity to the sources that produce data, for example cameras, in real-time. Naturally, a lot of research has been made on how embedded devices can be utilized to efficiently perform CNN inferencing. To this end, methodologies for porting the compute and storage intensive CNNs to the limited resources embedded devices, as in [24], where the authors employ approximate computing techniques to reduce the computational load and memory occupation of the deep learning architecture by compromising accuracy with memory and computation. These methodologies prove to be invaluable, since they allow for off-the-shelf embedded platforms to be used in even intensive applications, for example non-mission-critical space applications from launch to orbit and potentially beyond [22]. On the other hand, a different school of thought does exist, where engineers attempt to design specialised processors or accelerators solely for CNN inferencing on embedded platforms, or on FPGAs [25, 26, 27] or ASICs [27, 28, 29]. An example of the former can be found in [30], where the authors propose a very compact embedded CNN processor design based on a modified logarithmic computing method, while the latter approach is encountered in [31], where the authors propose a framework for designing CNN accelerators on embedded FPGAs for image classification.

1.4.2 Implementations on VPUs

Vision Processing Units have been widely used to accelerate a variety of applications. In [20], a Myriad 2 VPU is employed to achieve low-power and high-performance on a Vision Based Navigation system, which consists of an ISP pipeline. In a similar manner, in [32], an algorithm for stereo correspondence was accelerated on the same VPU, with considerable less development effort than an FPGA-based implementation of it. Hybrid approaches have also been proposed, for example [33], where both FPGA and VPU are utilized to coopeatively accelerate space applications.

In addition to classical CV algorithms, common neural networks have been ported to VPUs. In [34], an efficient implementation of the Support Vector Machines is proposed, which achieves up to 40% energy savings against state-of-the-art relevant approaches. Convolutional Neural Networks have also benefited from the utilization of such processing systems. In [35], an efficient, Winograd-based implementation of kernel convolution is proposed, while [36] explores the possibility of a DSE framework, to be used to determine the optimal configuration for CNNs on edge devices, using the Myriad 2 VPU as a paradigm.

The Neural Compute Stick has gathered great interest, due to its high performance per Watt ratio, ability to accelerate complex Neural Networks, and user-friendly interface, and has been used to test models in academia. For example, in [37], the NCS was used to accelerate a network that performed object recognition on voxelized point-clouds, in order to evaluate a synthetic dataset generator. Object recognition on the NCS was also performed in [38], where the VPU was paired with a Raspberry Pi, and was used to analyze objects in the real time images and videos for vehicular edge computing.

The main reason that the NCS is used over a traditional board system containing a VPU, is that the inherent heterogeneity of the latter demands increased programming effort. To tackle this issue, the ParalOS framework [5] was developed to provide a much-needed layer of abstraction for developers, and integrates a dynamic task scheduler, a scratchpad memory management scheme,

I/O & inter-process communication techniques, as well as a visual profiler.

1.5 Thesis Scope and Contribution

As commercial-off-the-shelf embedded systems become readily available to the mass market, tools are developed to provide user-friendly interfaces for their quick and efficient programming. In the case of the Myriad X VPU, specifically, the process of CNN inferencing can be entirely automated by the OpenVINO Toolkit, which provides a higher-level of abstraction for programmers, and encourages them to utilize their VPUs in a plug-and-play manner. This approach streamlines the workflow of programming these complex multicore systems, and allows for increased productivity.

These tools do provide some room for customizability, but they are commonly platform-agnostic, and therefore they do not permit the exploitation of the heterogeneous architectures to their full potential. As a result, latency-critical applications suffer. In addition to this, the process of extending the operation sets of these tools may be confusing to developers, since multiple configuration files may need to be created and precise programming directives be followed. Therefore, the **main motivation** of this thesis was to exploit the best features of both worlds, by having our application partly consist of low-level code, highly-optimized for the heterogeneous VPU at hand, but also utilizing the available tools to effortlessly integrate the CNN inferencing into it. To this end, the aforementioned "Lost in Space" problem proved to be an ideal candidate for a proposed application.

More precisely, reliable pose estimation of satellites, especially those that are uncooperative, is a challenging task. This is evident, when observing the submissions of the contenders of ESA's 2019 Pose Estimation Challenge. In fact, no proposed solution to the problem can be satisfactorily run on a Myriad X VPU. For example, the winner of the competition employed a High-Resolution Network (HRNet), which consists of such a colossal number of layers and parameters, that it is impossible to even load the network on the device due to limited computational and memory resources. The lightest, but still fairly accurate, CNN developed by a team in this competition, "UrsoNet", was able to be executed on the VPU, albeit at such a high latency for a single inference, that the system could not provide estimations in real-time as desired. This means that the inferencing would need to be coupled with some form of preprocessing, in order for the network to operate on lower dimensions, and thus be accelerated. Since, we do not desire for a bottleneck to be created because of this preprocessing, the methods and functions employed need to be highly optimized on a low level.

The **main contribution** of this thesis is the implementation of an efficient application for CNN-based pose estimation of non-cooperative spacecrafts, that supports real-time execution. This is achieved by sufficiently preprocessing the input images, utilizing one of the three available resampling methods, namely bilinear interpolation, bicubic interpolation and Lanczos resampling. These algorithms were developed on a low level, in order to account for a small fraction of the total execution time of the system. Furthermore, a simple but efficient power measurement and management system, that monitors the power consumption of the board hosting the VPU and performs static power management, is proposed, and can be extracted and used in future projects. Moreover, we propose a system that performs both the pose estimation of a satellite and its subsequent tracking in a low-power high-performance manner, on a single SoC. Finally, since a plethora of the available components of the Myriad X VPU were utilized, the codes produced could be used as programming paradigms.

Chapter 2

Tools, Frameworks and Libraries

2.1 TensorFlow and Keras

TensorFlow

TensorFlow is a free, open-source platform, that is primarily used to develop, train and deploy Machine Learning models [7]. TensorFlow is capable of operating at high scale and in heterogeneous environments. Models are executed as stateful dataflow graphs. A single dataflow graph is used to represent all computations and states in a machine learning model. This graph expresses the communication between subcomputations explicitly, thus making it easy to execute independent computations in parallel and to partition computations across multiple devices. Figure 2.1 provides an example of a training pipeline.

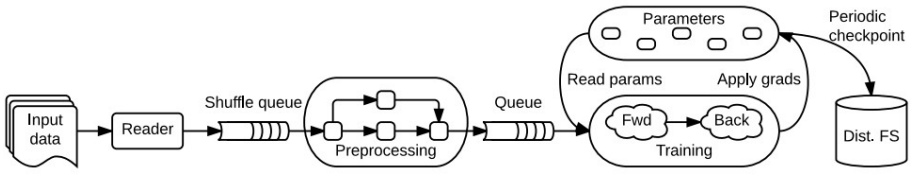


Figure 2.1: A schematic TensorFlow dataflow graph for a training pipeline [7]

TensorFlow is capable of keeping checkpoints of models during the training procedure, allowing users to stop the training and resume it later on, on the same or on a different platform. Trained models may also be saved in a SavedModel format, which contains both data describing the structure of the network as well as the weights of the different layers. Therefore, trained models can be loaded on any platform that supports TensorFlow and be used for inference.

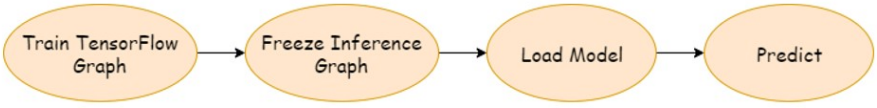


Figure 2.2: Inference Workflow in TensorFlow

Keras

Keras [39] is an open source library, built on top of TensorFlow. Its main purpose is to act as an interface for TensorFlow by providing a consistent and user-friendly API. This API offers access to numerous implementations of commonly used neural-network building blocks such as layers, objectives, activation functions, optimizers, as well as a host of tools to make working with image and text data easier and to simplify the coding necessary for writing deep neural network code. The following program demonstrates how a simple Convolutional Neural Network can be modeled with Keras. Multiple different layers have been used, such as 2-dimensional convolution, max pooling or even a dropout layer, that is only present during training. Thus, it is clear how Keras can be used to greatly reduce the complexity of developing ANNs.

```
from tensorflow import keras
from tensorflow.keras import layers

# Code adapted from https://keras.io/examples/vision/mnist_convnet/
input_shape = (28,28,1)
num_classes = 10
model = keras.Sequential(
    [
        keras.Input(shape=input_shape),
        layers.Conv2D(32, kernel_size=(3, 3), activation="relu"),
        layers.MaxPooling2D(pool_size=(2, 2)),
        layers.Conv2D(64, kernel_size=(3, 3), activation="relu"),
        layers.MaxPooling2D(pool_size=(2, 2)),
        layers.Flatten(),
        layers.Dropout(0.5),
        layers.Dense(num_classes, activation="softmax"),
    ]
)
```

Listing 2.1: Keras Example

Docker

Docker [40] is a set of platform as a service (PaaS) products that use OS-level virtualization to deliver software in packages called containers. In this thesis, a Docker image containing TensorFlow version 1.9.0 and Keras version 2.1.6 was used, to enable the training and evaluation of a CNN on a remote machine. This machine had access to an Nvidia Tesla V100 GPU, making it an ideal candidate for such a task.

2.2 OpenCV Library

As the name implies, OpenCV [41] is an open-source library for Computer Vision. It provides a variety of optimized functions that either support or perform real-time computer vision tasks. In this thesis, OpenCV was used, because it provides a variety of different options for resampling images. OpenCV uses NumPy Arrays to store and represent images, allowing us to easily integrate

it into our application. Furthermore, since the library is open-source, we managed to replicate its behaviour when porting the resampling algorithms used to Myriad X.

2.3 OpenVINO Toolkit

The OpenVINO toolkit [8] is a comprehensive toolkit developed by Intel, dedicated to providing reduced time-to-market of applications that solve a variety of typical ANN tasks including emulation of human vision, natural language processing and many others. Heterogeneous execution on different Intel Hardware (CPUs, Integrated Graphics Cards, FPGAs, VPUs) is supported by OpenVINO, maximizing performance, while providing tools and APIs to facilitate the deployment of such applications on the desired platform. Thus, the OpenVINO toolkit accelerates applications with high-performance, AI and deep learning inference deployed from edge to cloud.

2.3.1 Workflow

The following diagram illustrates the typical OpenVINO workflow:

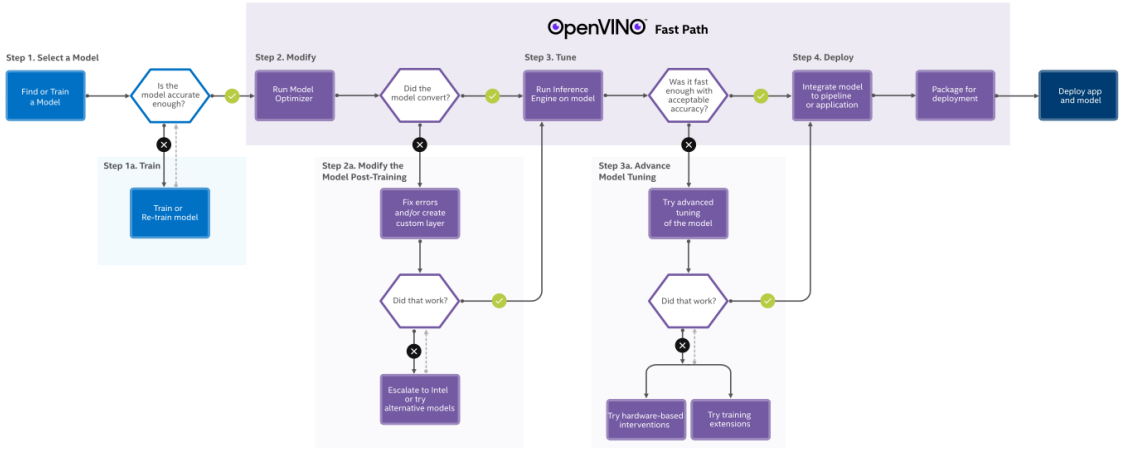


Figure 2.3: OpenVINO Toolkit Workflow [8]

The typical OpenVINO Toolkit workflow is divided into four stages.

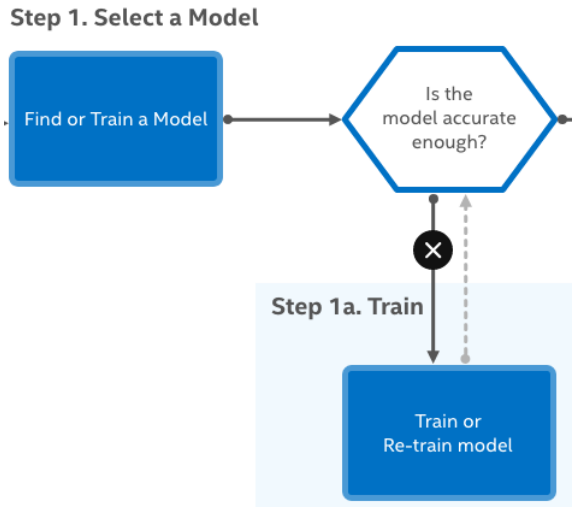


Figure 2.4: OpenVINO Toolkit Workflow Stage 1 [8]

On the first stage, a decision must be made on what model will be used. Developers can pick one of the pre-trained models provided by Intel's Open Model Zoo (OMZ) [42]. The OMZ includes deep learning solutions to a variety of vision problems, including object recognition, face recognition, pose estimation, text detection, and action recognition, at a range of measured complexities. Otherwise, if none of the existing OMZ models comply to the constraints that have been set, OpenVINO's Training Extensions provide a convenient environment to train deep learning models, by offering numerous pre-trained models, ideal for transfer learning. If none of the above solutions satisfy their needs, developers are free to train models using one of the following frameworks: Caffe, MXNet, TensorFlow, TensorFlow 2 Keras, Kaldi, ONNX.

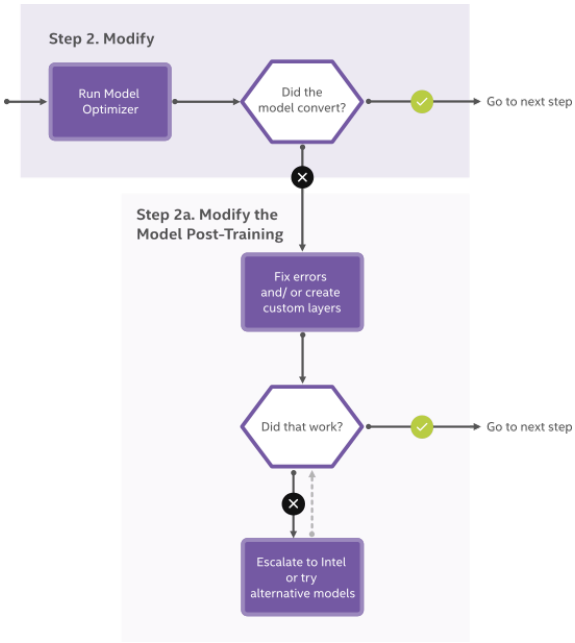


Figure 2.5: OpenVINO Toolkit Workflow Stage 2 [8]

The second stage is invoked by passing a frozen model to the Model Optimizer. An in-depth study of the Model Optimizer will be presented in a later section. In short, the Model Optimizer

performs a few optimizations, where possible, to create simpler, faster graphs. Any custom operations or unsupported layers that are present in the model, must be supported by extensions in the Model Optimizer, to enable recognition and parsing of them.

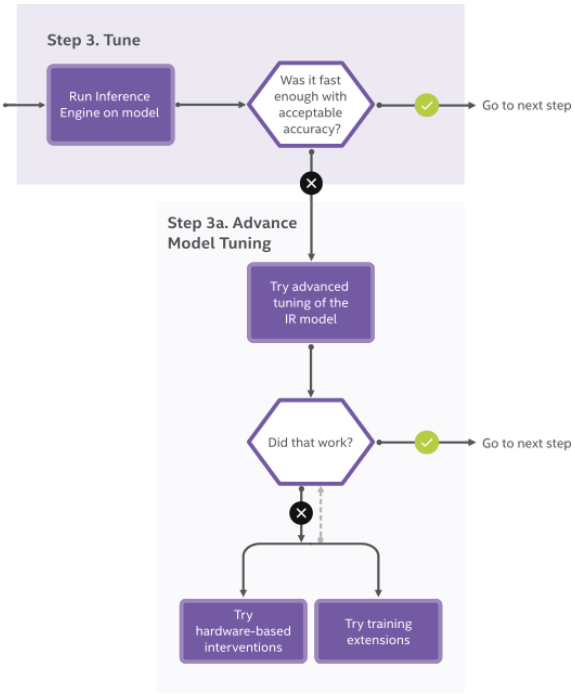


Figure 2.6: OpenVINO Toolkit Workflow Stage 3 [8]

The third stage manages the loading and compiling of the optimized neural network model, runs inference operations on input data, and outputs the results. This stage, based on the Inference Engine tool, can execute synchronously or asynchronously, on multiple Intel devices. Further optimizations can be performed during this stage. The Post-training Optimization Tool can accelerate the inference of a deep learning model by quantizing it to INT8, but is currently only available to OMZ models.Besides the Post-training Optimization Tool, the Neural Network Compression Framework (NNCF) can be used for model fine-tuning INT8 quantization or even for applying more aggressive compression methods to further speed up model inference and reduce the footprint. In that case the compression algorithms are integrated into the model training pipeline, and, thus, only TensorFlow models are supported for this utility. In addition, it should be noted that custom operations must be supported with the appropriate extensions to the Inference Engine.

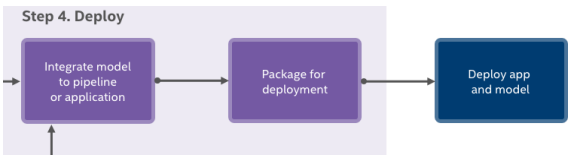


Figure 2.7: OpenVINO Toolkit Workflow Stage 4 [8]

The final stage includes the Deployment Manager. The Deployment Manager is a Python command-line tool that assembles the transformed model and the application files, as well as any required dependencies into a runtime package for the target device. It outputs packages for CPU, GPU, and VPU on Linux and Windows, and Neural Compute Stick-optimized packages with Linux.

2.3.2 The Model Optimizer

The Model Optimizer is a cross-platform command-line tool that facilitates the transition between the training and deployment environment by performing static model analysis, and adjusting deep learning models for optimal execution on end-point target devices [8].In order to use the Model Optimizer, the network model must be trained using one of the supported deep learning frameworks, as defined previously. The Model Optimizer produces an Intermediate Representation (IR) of the network, which consists of an XML file, that describes the network topology, and a binary file, that contains the weights and biases. This Intermediate Representation can be inferred with the Inference Engine.

Several optimization techniques are employed [9], to transform the input network models to simpler, faster graphs, and thus accelerating them:

Linear Operations Fusing is a process of fusing BatchNormalization or ScaleShift layers into previous or following Convolution or Fully Connected layers. These layers are present in many popular networks, such as ResNet, and can be easily represented as a sequence of linear operations, namely additions and multiplications. Therefore, they are first decomposed into sequences of Mul $\rightarrow$ Add operations. The Model Optimizer then searches backward and forward for available Convolution or Fully Connected layers, to fuse these operations into. The following figure illustrates the transformation of part of a ResNet269 network, using this optimization technique.

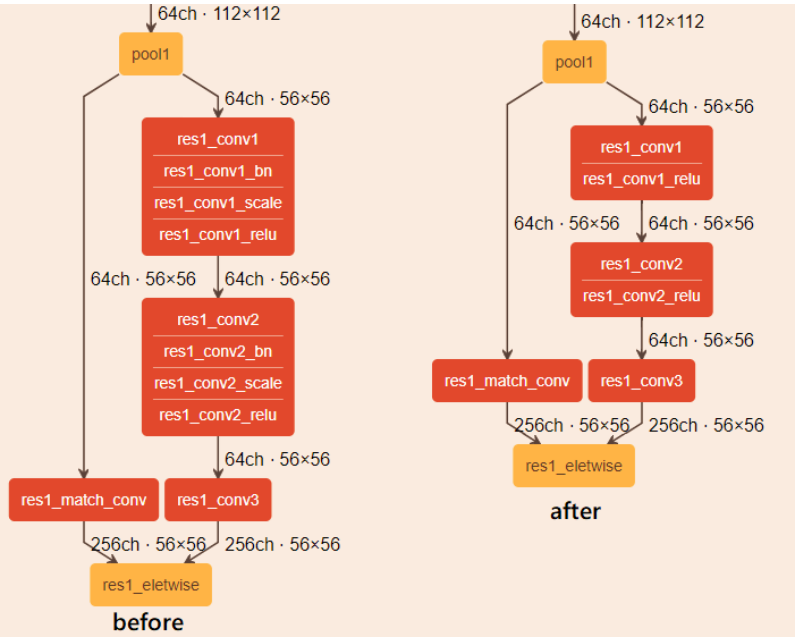


Figure 2.8: Usage Example of Linear Operations Fusing [9]

ResNet Stride Optimization is an optimization technique specifically designed to accelerate ResNet-based topologies. The main idea of this optimization is to move the stride that is greater than 1 from Convolution layers with kernel size equal to 1 to upper Convolution layers. In addition, the Model Optimizer adds a Pooling layer to align the input shape for a Eltwise layer, if it was changed during the optimization. This optimization results in downscaling the size of the produced feature maps earlier inside the network, which in turn reduces the number of operations performed

on late layers. For example, the third branch operates on 28x28 instead of 56x56 feature maps after the transformation, as illustrated in figure XYZ.

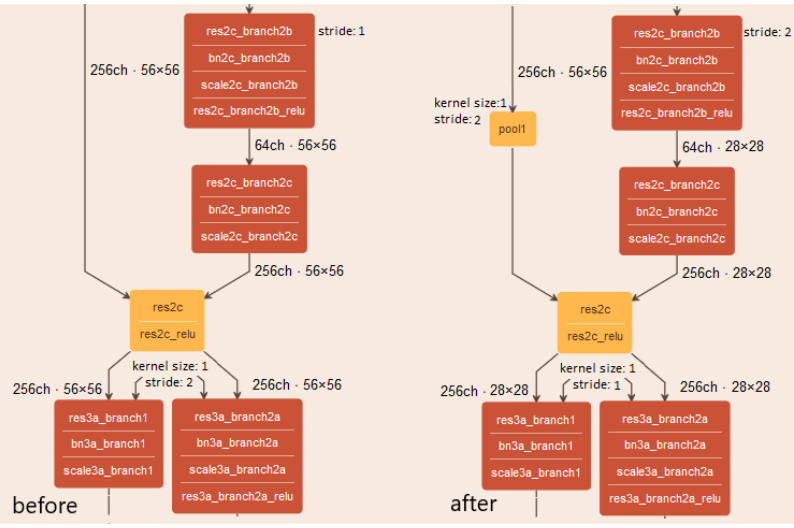


Figure 2.9: Usage Example of ResNet Stride Optimization [9]

Grouped Convolution Fusing is an optimization designed only for TensorFlow models. The Split Layer preceding the parallel convolutions is modified to feed multiple tensors into the groups of convolutions. The resulting feature maps are then concatenated appropriately. Due to incomplete documentation, it is yet unclear how this modification of the network affects performance.

Network Pruning is a technique that aims to increase performance of networks by removing redundant nodes of the model. Under the appropriate conditions, the model optimizer can prune several types of layers, such as Crop and Reshape Layers. Therefore, it reduces the overall amount of operations performed.

2.3.3 Benchmarking on VPU Neural Compute Stick 2

OpenVINO provides a Benchmark Tool, in the form of a C++ file, to estimate deep learning inference performance on supported devices, including the Neural Compute Stick 2. This stick contains an ma2480 chip (Myriad X). Performance can be measured for two inference modes: synchronous (latency-oriented) and asynchronous (throughput-oriented). Upon start-up, the tool loads a network and images/binary files to the Inference Engine plugin for the Myriad X VPU. The number of infer requests and execution approach depend on the mode defined with a command-line parameter provided by the user. In synchronous mode, one infer request is created and executed per iteration, while in asynchronous mode, multiple infer requests are created and executed in parallel. During the execution, the application collects latency for each executed infer request. Reported latency value is calculated as a median value of all collected latencies. On the other hand, reported throughput value is reported in frames per second (FPS) and calculated as a derivative from the reported latency, in case of synchronous execution, or from the total execution time elsewise.

2.4 Myriad X Development Kit

The Myriad X Development Kit (MDK) comprises of common code, which includes drivers and components, documentation support and toolchains that are required to develop applications for the Myriad Family products. Part of the toolchain is a quite extensive and complex build system based on GNU Makefile. The build system is responsible for cross-compiling the object code for the various heterogeneous processors. Afterwards, the linker generates the memory map, as per the configuration file provided by the developer.

Located in the mdk/common/components directory, the MDK includes a variety of reusable components. In the scope of this thesis, the following components were utilized:

- For Debugging and Testing purposes:
  - PipePrint
  - UnitTest
- For Power Management and Measurement:
  - MV0235
  - PwrManager
  - MVBoardsCommon
- For Memory Management:
  - MyriadMemInit
  - MemoryManager
- For the utilization of the Neural Compute Engine:
  - MvNCI
  - MvTensor
- For file management:
  - VcsHooks

Chapter 3

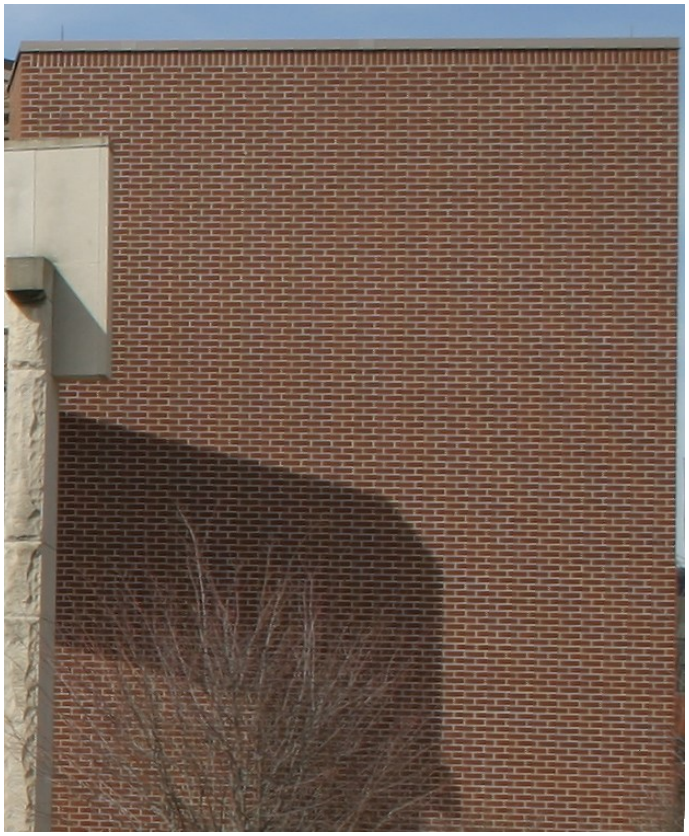
Theoretical Background

3.1 Image Scaling

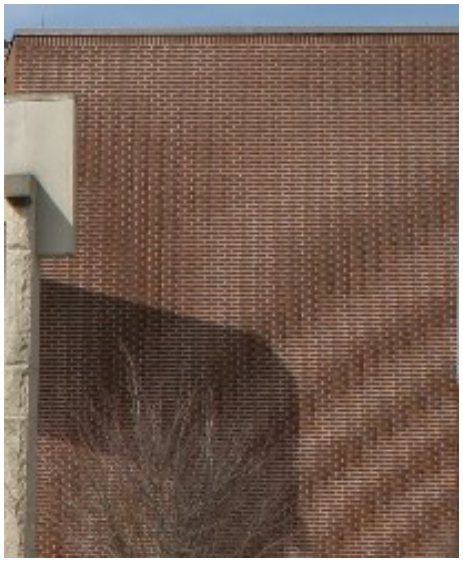
Image Scaling is the process of resizing images, either by downsampling the original image or upsampling it. It is an integral part of image processing pipelines. For example, input images may be rescaled to a lower resolution during earlier stages of the pipeline, to reduce the number of computations performed during the compute-intensive stages. On the other hand, upsampling the images is a very common practice as well, since the input image may be of lower resolution and/or quality than needed by the application.

In this thesis, image scaling was utilized to reduce dimension sizes, as a means of accelerating the convolutional neural network that is to operate on these specific images. More specifically, we examined 3 different resampling methods to determine the effect on the final accuracy, that aliasing and ringing artifacts have. Artifacts are anomalies introduced to digital signals as a result of the Digital Signal Processing applied to them. Figures 3.1 and 3.2 provide an example of how artifacts may affect the quality of a resized image.

The dataset on which our CNN is to be trained on, consists of images of satellites on orbit around the earth. Therefore, aliasing may appear when the Earth is on the background. Ringing artifacts are also highly likely to appear, as the sharp edges of the satellite are often contrasted by the black background. To resize the image and suppress these artifacts, Bilinear Interpolation, Bicubic Interpolation and Lanczos Resampling were employed. An in-depth analysis of these methods is included in the following sections.



(a) Original Image



(b) Aliasing appears in the resized image

Figure 3.1: Example of Aliasing when Resizing an Image [10]

(a) Original Image

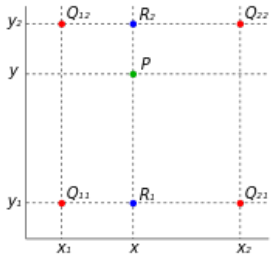
(b) Resized image has visible ringing artifacts around sharp edges

Figure 3.2: Example of Aliasing when Resizing an Image [11]

3.1.1 Bilinear Interpolation

Bilinear Interpolation is performed by applying linear interpolation on the x-axis and then interpolating the results linearly on the y-axis, or vice versa. A more precise definition is the following:

Suppose that we want to compute the value of the unknown function f at the point (x,y). The value of f at the points $Q_{11} = (x_1, y_1)$, $Q_{12} = (x_1, y_2)$, $Q_{21} = (x_2, y_1)$ *and* $Q_{22} = (x_2, y_2)$ *is assumed to be known.*



We first apply linear interpolation in the x-axis. This yields:

$$f(R_1) = f(x, y_1) = \frac{x_2 - x}{x_2 - x_1} \cdot f(Q_{11}) + \frac{x - x_1}{x_2 - x_1} \cdot f(Q_{21})$$

$$f(R_2) = f(x, y_2) = \frac{x_2 - x}{x_2 - x_1} \cdot f(Q_{12}) + \frac{x - x_1}{x_2 - x_1} \cdot f(Q_{22})$$

We proceed by interpolating in the y-axis to obtain the desired estimate:

$$f(x, y) = \frac{y_2 - y}{y_2 - y_1} \cdot f(R_1) + \frac{y - y_1}{y_2 - y_1} \cdot f(R_2)$$

In the scope of this thesis, we desire to use this method to downscale an image. Bilinear Interpolation operates on a 2x2 neighbourhood of pixels. Thus, to downscale the image, a stride is computed, as the inverse of the scaling factor. Successive pixels of the same output line are computed by using 2x2 neighbourhoods that have a distance equal to the calculated stride. Pixels of successive output lines are also computed in a similar manner. The four pixels that form the neighbourhood are assumed to be in positions (0,0), (0,1), (1,0) and (1,1). The desired pixel is assumed to be in position (0.5, 0.5). Therefore, the above equations become:

$$f(0.5, 0) = \frac{1}{2} \cdot f(0, 0) + \frac{1}{2} \cdot f(1, 0)$$

$$f(0.5, 1) = \frac{1}{2} \cdot f(0, 1) + \frac{1}{2} \cdot f(1, 1)$$

and thus:

$$f(0.5, 0.5) = \frac{1}{2} \cdot f(0.5, 0) + \frac{1}{2} \cdot f(0.5, 1) = \frac{1}{4} \cdot (f(0, 0) + f(0, 1) + f(1, 0) + f(1, 1))$$

We conclude that in this particular application, Bilinear Interpolation is equivalent to average pooling with a 2x2 pool size.

3.1.2 Bicubic Interpolation

Bucubic Interpolation can be seen as a generalization of cubic interpolation, where data points on a two-dimensional regular grid are interpolated. This method considers 16 pixels, in a 4x4 neighbourhood, compared to the 4 pixels that bilinear interpolation takes into account. As a result, the resized images appear smoother and with fewer artifacts, with the tradeoff of increased latency. The desired pixel can then be computed as:

$$p(x,y) = \sum_{i=0}^{3}\sum_{j=0}^{3} a_{\text{ij}} \cdot x^i \cdot y^j$$

The interpolation problem consists of determining the 16 coefficients a_{ij}. Directly calculating these coefficients is inefficient, since their values depend on the values of the examined pixels. This means that for each output pixel, these coefficients would need to be calculated time after time. Therefore, it was decided that we use the **Bicubic Convolution Algorithm**, proposed by Keys [19].

Bicubic Interpolation, as described above, is equivalent to applying a convolution with the following kernel in both dimensions:

$$W(x) = \begin{cases} (\alpha+2)\cdot|x|^3 - (\alpha+3)\cdot|x|^2 + 1 & |x| \leq 1 \\ \alpha\cdot|x|^3 - 5\alpha\cdot|x|^2 + 8\alpha\cdot|x| - 4\alpha & 1 < |x| < 2 \\ 0 & otherwise \end{cases}$$

where α is a parameter, typically set to -0.5 or -0.75, and x is the distance between the examined pixel and the desired pixel. Along the x-dimension, pixels are numbered with integer values ranging from -1 to 2. The same numbering applies for the y-dimension as well. Each line in the 4x4 neighbourhood will be convoluted with the above kernel. Subsequently, the resulting values will also be convoluted with that kernel, to obtain the desired pixel. As was the case with bilinear interpolation, the desired pixel is assumed to be in position (0.5, 0.5). Since both the position of the desired pixel and the positions of the 4x4 neighbourhood pixels are known, the coefficient of each pixel can be computed and be hardcoded inside our application. Furthermore, since $x = y = 0.5$, the coefficients for the 2 dimensions will be the same. Thus, only the following 4 values need to be calculated:

$$W(0.5-(-1)) = W(1.5) = -0.75\cdot|1.5|^3 + 5\cdot0.75\cdot|1.5|^2 - 8\cdot0.75\cdot|1.5| + 4\cdot0.75 = -0.09375$$

$$W(0.5) = 1.25\cdot|0.5|^3 - 2.25\cdot|0.5|^2 + 1 = 0.59375$$

$$W(1-0.5) = W(0.5) = 0.59375$$

$$W(2-0.5) = W(1.5) = -0.09375$$

where α was set to be equal to -0.75, in order to mimic the behaviour of the OpenCV library. Thus, the bicubic convolution in one dimension can be written in matrix form as:

$$f(P_{-1},P_0,P_1,P_2) = \begin{bmatrix} -0.09375 & 0.59375 & 0.59375 & -0.09375 \end{bmatrix} \cdot \begin{bmatrix} P_{-1} \\ P_0 \\ P_1 \\ P_2 \end{bmatrix}$$

And to compute the desired pixel:

$$DesiredPixel = f \begin{pmatrix} f(P_{(-1,-1)}, P_{(-1,0)}, P_{(-1,1)}, P_{(-1,2)}), \\ f(P_{(0,-1)}, P_{(0,0)}, P_{(0,1)}, P_{(0,2)}), \\ f(P_{(1,-1)}, P_{(1,0)}, P_{(1,1)}, P_{(1,2)}), \\ f(P_{(2,-1)}, P_{(2,0)}, P_{(2,1)}, P_{(2,2)}) \end{pmatrix}$$

Figure 3.3: Bicubic Convolution Algorithm. The red points form the 4x4 neighbourhood that is to be used to compute the value of the desired pixel. The green dots are the points that will be evaluated when convoluting in the x-dimension. The blue dot is the desired pixel and can be computed by applying convolution in the y-dimension, on the green-dot points.

3.1.3 Lanczos Resampling

Lanczos Resampling is the process of mapping samples of a digital signal to a translated and scaled copy of the Lanczos kernel, and then evaluating the desired points by summing these copies [43]. This method is frequently used for multivariate interpolation in image processing, to resize or rotate images, for example.

The Lanczos kernel consists of the normalized sinc function sinc(x), windowed by the Lanczos window, or sinc window, which is the central lobe of a horizontally stretched sinc function sinc(x/a) for -a ¡ x ¡ a. The parameter a is a positive integer, which determines the size of the kernel, ie the number of input samples to be interpolated to produce an eavluation.

$$L(x) = \begin{cases} sinc(x) \cdot sinc\left(\frac{x}{a}\right) & -a < x < a \\ 0 & otherwise \end{cases}$$

The OpenCV library [41] provides the option to resize images using the Lanczos resampling method with a set to be equal to 4. Therefore, in the scope of this thesis, the Lanczos kernel will be equal to:

$$L(x) = \begin{cases} 1 & x = 0 \\ \frac{sin(\pi x) \cdot sin\left(\frac{\pi x}{4}\right)}{4 \cdot \pi^2} & -a < x < a, x \neq 0 \\ 0 & otherwise \end{cases}$$

Interpolation of a one-dimensional digital signal f with a Lanczos filter of order a at an arbitrary real argument x is obtained through the discrete convolution of signal samples with the Lanczos kernel:

$$f(x) = \sum_{i=-a+1}^{a} f(\lfloor x \rfloor + i) \cdot L(i - x + \lfloor x \rfloor)$$

In two dimensions, the Lanczos kernel is simply the product of the one-dimensional Lanczos kernels of each dimension. This kernel is not separable.

$$L(x, y) = L(x) \cdot L(y)$$

Therefore, the interpolation of a two-dimensional image, let f, with a Lanczos filter of order a can be performed as follows:

$$f(x, y) = \sum_{i=-a+1}^{a} \sum_{j=-a+1}^{a} f(\lfloor x \rfloor + i, \lfloor y \rfloor + j) \cdot L(i - x + \lfloor x \rfloor) \cdot L(j - y + \lfloor y \rfloor)$$

The OpenCV library also aplies the filter weight w by division to preserve flux:

$$f(x, y) = \frac{1}{w} \cdot \sum_{i=-a+1}^{a} \sum_{j=-a+1}^{a} f(\lfloor x \rfloor + i, \lfloor y \rfloor + j) \cdot L(i - x + \lfloor x \rfloor) \cdot L(j - y + \lfloor y \rfloor)$$

$$w = \sum_{i=-a+1}^{a} \sum_{j=-a+1}^{a} L(i - x + \lfloor x \rfloor) \cdot L(j - y + \lfloor y \rfloor)$$

In the scope of this thesis, both x and y are equal to 0.5, since the desired pixel is set to be at the center of the considered 8x8 neighbourhood, thus their floor function is equal to 0. The sum operators can be interchanged and the interpolation can be performed equivalently as:

$$f(x, y) = \sum_{j=-3}^{4} \frac{1}{w} \cdot L(j - y) \cdot \left(\sum_{i=-3}^{4} f(i, j) \cdot \frac{1}{w} \cdot L(i - x) \right)$$

Unsurprisingly, the operation described by the above equation is similar to that of the Bicubic Convolution Algorithm. The coefficients of the weighted sums remain to be calculated. We only need to compute these coefficients once, since they are common for both dimensions in this particular case (a = 4, x = y = 0.5).

$$L(-3-0.5) = L(-3.5) = -0.01266087782123867$$

$$L(-2.5) = 0.05990948337726289$$

$$L(-1.5) = -0.16641523160350802$$

$$L(-0.5) = 0.6203830132406946$$

$$L(0.5) = L(-0.5)$$

$$L(1.5) = L(-1.5)$$

$$L(2.5) = L(-2.5)$$

$$L(3.5) = L(-3.5)$$

$$sum(L_i) = 1.0024327743864216$$

Therefore, the interpolation of the pixels in one direction can be expressed in matrix form as:

$$DesiredPixel = \begin{bmatrix} P_{-3} & P_{-2} & P_{-1} & P_0 & P_1 & P_2 & P_3 & P_4 \end{bmatrix} \cdot \begin{bmatrix} -0.012630151512143303 \\ 0.05976409082786907 \\ -0.1660113634107474 \\ 0.6188774240950217 \\ 0.6188774240950217 \\ -0.1660113634107474 \\ 0.05976409082786907 \\ -0.012630151512143303 \end{bmatrix}$$

3.2 UrsoNet: Pose Estimation for Satellites

UrsoNet [4] is a Convolutional Neural Network, proposed by Pedro Proenca and Yang Gao, that achieved third place on the synthetic test set and second place on the real test set of ESA's Pose Estimation Challenge in 2019. Contrary to the other contenders, UrsoNet demands a moderate amount of resources, and could potentially operate on embedded devices at the edge. Figure 3.4 illustrates a simplified overview of UrsoNet's network architecture.

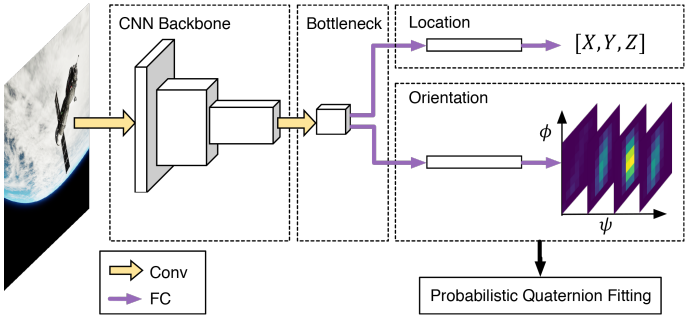


Figure 3.4: Simplified Overview of the UrsoNet Architecture [4]

3.2.1 Design and Architecture

The CNN backbone of the network is based on the ResNet architectures (ResNet34, ResNet50 or ResNet101) with pre-trained weights, since they contain a limited number of pooling layers. Pooling and Fully-Connected layers are generally not desirable, as they do not preserve spatial features. As a result, the ResNet architecture is modified, and the last fully-connected layer, as well as the global average pooling layer are substituted with one 3x3 convolution layer, with stride 2 to compress the CNN features (Bottleneck Layer).

The estimation of the location of the satellite is performed via a simple regression branch, consisting of two fully-connected layers. However, the network is not trained by minimizing the absolute Euclidean distance of the estimated and the ground truth location. Instead, the loss function employed is:

$$L_{\text{loc}} = \sum_{i}^{m} \frac{||t^{(\text{i})} - t_{\text{gt}}^{(\text{i})}||_2}{||t_{\text{gt}}^{(\text{i})}||_2}$$

where $t^{(\text{i})}$ and $t_{\text{gt}}^{(\text{i})}$ are the estimated and ground truth translation vector respectively. This function is first term of the total loss function:

$$L_{\text{total}} = \beta_1 L_{\text{loc}} + \beta_2 L_{\text{ori}}$$

This loss function minimizes the relative error of the estimated location and is chosen, because the fine-tuned loss weights β_1 and β_2 generalize better to other datasets.

A similar regression branch, consisting of two fully-connected layers, is responsible for the estimation of the orientation of the satellite. UrsoNet can be configured to either directly regress orientation or perform continuous orientation estimation via classification with soft assignment coding. In the first case, L2 or L1 losses are not sufficient to correctly train the network, as they do not represent the actual angular distance accurately, for any orientation system. Therefore, the orientation loss function chosen is one of the following:

$$L_{\text{ori}} = arccos(|{q^{(\text{i})}}^T \cdot q_{\text{gt}}^{(\text{i})}|)$$

$$L_{\text{ori}} = 1 - |{q^{(\text{i})}}^T \cdot q_{\text{gt}}^{(\text{i})}|$$

where $q^{(\text{i})}$ and $q_{\text{gt}}^{(\text{i})}$ are the estimated and ground truth quaternions respectively.

Alternatively, in the second case, in order to perform soft classification, each ground truth label is encoded as a Gaussian random variable in an orientation discrete output space, so that the network learns to output probability mass functions. The network output is set to be a 3D histogram, where each bin maps to a combination of discrete Euler angles. Assuming that $Q = \{b_1, ..., b_N\}$ are the quaternions corresponding to the histogram bins, then, during training, each bin is encoded with the soft assignment function:

$$f(b_i, q_{\text{gt}}) = \frac{K(b_i, q_{\text{gt}})}{\sum_j^N K(b_j, q_{\text{gt}})}$$

where the kernel function K(x; y) uses the normalized angular difference between two quaternions:

$$K(x, y) = e^{-\frac{\left(\frac{2cos^{-1}(|x^T y|)}{\pi}\right)^2}{2\sigma^2}}$$

and the variance σ^2 is given by the formula:

$$\sigma^2 = \frac{\left(\frac{\Delta}{M}\right)^2}{12}$$

where Δ/M represents the quantization step, Δ is the smoothing factor that controls the Gaussian width and M is the number of bins per dimension. To train the network, UrsoNet uses a Softmax Cross-Entropy loss function. At test time, given the bin activations $\{\alpha_1, ..., \alpha_N\}$ and the respective quaternions, in one hemisphere, a quaternion can be fitted by minimizing the weighted least squares:

$$\hat{q} = argmin_{\mathbf{q}} \sum_i^N w_i (1 - b_i^T q)^2$$

where α_i is assigned to w_i and the optimal solution is given by the right null space of the matrix $\sum_i^N w_i (b_i b_i^T)$.

3.2.2 Unreal Rendered Spacecraft on Orbit (URSO)

Unreal Rendered Spacecraft on Orbit (URSO) [4] is a simulator, that utilizes Unreal Engine 4 (UE4) features to render realistic images of non-cooperative spacecrafts, on orbit around the Earth. Lighting in this environment is made of a directional light and spotlight to simulate sunlight and Earth albedo respectively. The simulation of the sun, a body of emissive material with UE4 bloom scatter convolution is used. Earth is modelled as a high polygonal sphere textured with 21600x10800 Earth and cloud images from the Blue Marble Next Generation collection, and is further masked to obtain specular reflections from the ocean surface. Soyuz and Dragon spacecraft models, with their geometry imported from 3D model repositories, are also included.

Datasets are generated by randomly sampling 5000 viewpoints around the day side of the Earth from low Earth orbit altitude. The Earth rotation, camera orientation and target object pose are all randomized. Specifically, the target object is placed randomly within the camera's field of view at a distance ranging between 10 and 40 meters. A few examples of generated images are included below.

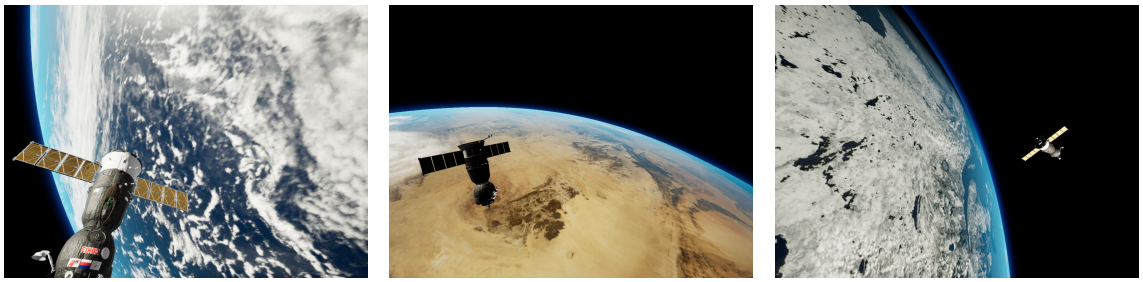


Figure 3.5: Example of Synthetic Images for the Soyuz Spacecraft [4]

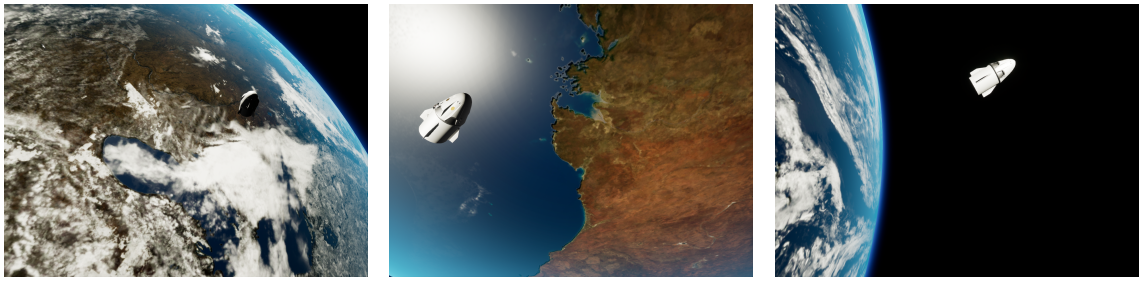


Figure 3.6: Example of Synthetic Images for the SpaceX Dragon Spacecraft [4]

Chapter 4

Design and Acceleration on Myriad X

Chapter 4 is dedicated to the implementation of the downsampling algorithms, discussed in Section 3.1, and the execution of UrsoNet, the Convolutional Neural Network discussed in Section 3.2, on Intel Movidius Myriad X. An in-depth analysis of the former is included in Section 4.1, since these algorithms constitute the Preprocessing Stage of our Pose Estimation system.The actual pose estimation, performed via the CNN, is accelerated on the Neural Compute Engine of the VPU. A thorough analysis of how this Hardware Accelerator can be utilized is provided in Section 4.2. Section 4.3 features the design of a custom Power Management and Measurement System, which proved to be valuable for the evaluation and power efficiency of the proposed application. Finally, Section 4.4 proposes a Pose Estimation & Tracking system for satellites, which is to be accelerated on the Intel Movidius Myriad X VPU, and essentially solves the "Lost in Space" problem.

4.1 Preprocessing Stage

The Preprocessing Stage consists of the image scaling functions, which aim to downsample the input. This downsampling is crucial to the application's overall performance. The original dataset contains RGB images with a 1280x960 resolution. This size would demand such an enormous amount of operations inside the Neural Network, that the overall system would not be able to run in a real-time scenario.

Unlike the Neural Network though, the algorithms employed during this stage operate on the full size of the input images, and thus need to be carefully designed, in order to avoid creating a bottleneck in this stage. Figure 4.1 illustrates the method that was followed, during this stage in development, to run the preprocessing stage.

The following sections provide an in-depth look at the design and acceleration of the Bilinear and Bicubic Interpolation, as well as the Lanczos Resampling Algorithms.

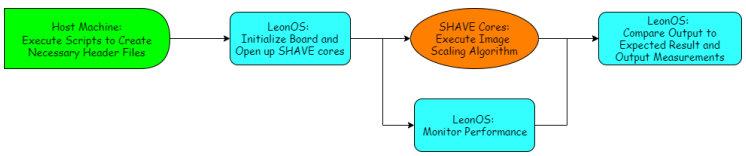


Figure 4.1: Process Followed to Execute the Downsampling on Myriad X

4.1.1 Implementation of Image Scaling Functions

The main task of downsampling the input image will be executed on the Microprocessor Array. This section focuses on the implementation of the correspondent code.

Several struct and Macro definitions need to be common between the SHAVE cores and LeonOS. Thus, they are placed in the "shared" folder and can be included both in the SHAVE and in the LOS application, by directly providing the header file that contains them. The code snippet below illustrates the main body of the FilterArgs.h header file, which consists of these definitions.

```
#define WIDTH   1280
#define HEIGHT  960

typedef struct {
    u8 red;
    u8 green;
    u8 blue;
} pixel_t;

typedef struct {
  pixel_t * inpImage;
  pixel_t * outImage;
  float ffactor;
  int workload;
  u32 shaveId;
} downsampleFilterArgs_t;
```

Listing 4.1: FilterArgs.h Main Body

Pixels are represented as a struct containing 3 bytes of RGB data. The arguments provided to the entry function of a SHAVE Core executing an image scaling task consist of:

- **A pointer to an array of input pixels and**
- **A pointer to an array of output pixels.** The input and output images are stored in these 1-dimensional arrays, residing in the DDR memory, in row-major order. The definition of row-major order is: Let N be the width of a 2-dimensional array and M be its height. This array can be stored in a 1-dimensional array of size NxM in row-major order, if the element stored in the i-th row and j-th column in the original array, is stored in [i * N + j] in the 1D array.
- **The scaling factor**, which is common for both dimensions. In this thesis, the scale factor was set to be 0.5, to maximize the ratio performance gain/accuracy loss. However, the code developed, was structured in such a way, that different scaling factors are supported, with the sole restriction that their inverse is an integer.
- **The workload**, that is the number of input lines to be processed by the particular core. This is particularly useful, since an arbitrary number of cores may be chosen to run the task in parallel. A more in-depth look at this feature is found in section 4.1.3.
- **The ID assigned to the SHAVE core**. Since the PipePrint utility is not thread safe, contention between processes may appear, when multiple SHAVE Cores attempt to print some data. Therefore, during debugging, a specific SHAVE may be chosen via the ID assigned to it, to provide debugging information. Alternatively, a more sophisticated implementation using hardware mutexes may be used, but the current approach meets the ends.

The code snippet below demonstrates the entrypoint of the SHAVE Cores executing, which is common, regardless of the algorithm being used.

```
1  __attribute__((dllexport)) void Entry( void * theArgsPointer )
2 {
3     /* Type-cast */
4     downsampleFilterArgs_t * theArgs = (downsampleFilterArgs_t *) theArgsPointer;
5     /* Retrieve Input Image  as an array of pixels */
6     pixel_t * inputImageDDR = theArgs->inpImage;
7     /* Get pointer to Output Image memory address */
8     pixel_t * outputImage = theArgs->outImage;
9     /* Downsampling factor */
10    float ffactor = theArgs->ffactor;
11    /* Inverse of Downsampling Factor */
12    int offset = 1 / ffactor;
13    /* # of lines processed by this SHAVE */
14    u32 workload = theArgs->workload;
15    u32 shaveId = theArgs->shaveId;
16
17    /* Each function call produces one output line.
18     * functionCall is replaced with the appropriate function
19     * implementing the desired resampling algorithm. The
20     * arguments of the function are common however. */
21    for (u32 i = 0; i < workload / offset; i ++) {
22        functionCall(&inputImageDDR[i * WIDTH * offset],
23                     &outputImage[i * (WIDTH / offset)],
24                     offset);
25    }
26
27    SHAVE_HALT;
28 }
```

Listing 4.2: SHAVE Entry Point

The reason why the ArgsPointer is a void pointer instead of a downsampleFilterArgs_t pointer will be thoroughly explained in Section 4.1.2.
In line 12, the offset is computed as the inverse of the scaling factor. This offset is equivalent to the stride of the operation. Therefore, this stride determines which input line will be used as an argument to the functionCall (line 22). Furthermore, in the case of bicubic and Lanczos interpolation, a second stride is also used, to omit the top and left padding.
The following sections demonstrate the application-specific functions called, depending on what algorithm is chosen.

Bilinear Interpolation

```
1  void bilinear_kernel ( pixel_t * lines, pixel_t * out, int offset) {
2      for( int i = 0; i < WIDTH / offset; i++) {
3          /* Use temporary u16 values here, to prevent overflow */
4          u16 red, green, blue;
5          red     =     lines[i * offset].red
6                      + lines[i * offset + 1].red
7                      + lines[i * offset + WIDTH].red
8                      + lines[i * offset + WIDTH + 1].red;
9
10         /* Code omitted for brevity */
11
12         out[i].red      = red   >> 2;
```

```
        out[i].green    = green >> 2;
        out[i].blue     = blue  >> 2;
    }
}
```

Listing 4.3: Bilinear Interpolation Kernel

The desired pixel is in position (0.5, 0.5). Thus, bilinear interpolation in this particular case, is equivalent to average pooling in a 2x2 region. The division by 4 operation is substituted with right shifting by 2 digits, since division is costly and redundant in this particular instance. It is important to note that, since we add 4 8-bit unsigned integers, the possibility of overflow is present. Thus, the results of the additions are stored in temporary 16-bit variables.

Bicubic Interpolation

```
typedef struct { float red; float green; float blue; } intermediate_t;

/* Bicubic Convolution Algorithm, alpha equals -0.75 */
static float coeff[4] = {-0.09375, 0.59375, 0.59375, -0.09375};
/* Bicubic Convolution Algorithm, alpha equals -0.5 */
// static float coeff[4] = {-0.0625, 0.5625, 0.5625, -0.0625}

void cubicKernelHConv1D ( pixel_t * pixels, intermediate_t * out) {
    out->red =      pixels[-1].red * coeff[0]
                +   pixels[ 0].red * coeff[1]
                +   pixels[ 1].red * coeff[2]
                +   pixels[ 2].red * coeff[3];

    /* Code omitted for brevity */
}

void cubicKernelVConv1D ( intermediate_t * pixels, pixel_t * out) {
    out->red =  roundf( pixels[-1].red * coeff[0]
                    +   pixels[ 0].red * coeff[1]
                    +   pixels[ 1].red * coeff[2]
                    +   pixels[ 2].red * coeff[3]);

    /* Code omitted for brevity */
}

void bicubicInterpolation ( pixel_t * lines, pixel_t * out) {
    intermediate_t arr[4];
    cubicKernelHConv1D(&lines[   -(WIDTH + 2)],    &arr[0]);
    cubicKernelHConv1D(&lines[              0],    &arr[1]);
    cubicKernelHConv1D(&lines[    (WIDTH + 2)],    &arr[2]);
    cubicKernelHConv1D(&lines[2 * (WIDTH + 2)],    &arr[3]);

    cubicKernelVConv1D(&arr[1], out);
}

void bicubic_kernel( pixel_t * inputImage, pixel_t * out, int offset){
    for (int i = 0; i < WIDTH / offset; i++){
        bicubicInterpolation(&inputImage[i * offset], &out[i]);
    }
}
```

Listing 4.4: Bicubic Interpolation Kernel

Each invocation of bicubicInterpolation produces 1 output pixel. As described in Section 3.1.2, bicubic interpolation is developed using the bicubic convolution algorithm, with an alpha value equal to -0.75 to mimic the behaviour of the OpenCV library. Thus, bicubicInterpolation consists of 4 horizontal 1-dimensional convolutions, followed by one vertical 1-dimensional convolution with the kernel. The output of horizontal convolutions is of intermediate_t type, since the results for each colour channel are floats. The output of the vertical convolution, on the other hand, is of pixel_t type, with the results for each colour channel being rounded.

Lanczos Resampling

```
typedef struct { float red; float green; float blue; } intermediate_t;

/* Normalized Lanczos Kernel */
static float coeff[8] = {-0.012630151512143303l, 0.059764090827869071,
                         -0.16601136341074741, 0.61887742409502171,
                          0.61887742409502171, -0.16601136341074741,
                          0.059764090827869071, -0.012630151512143303l};

void lanczosKernelHConv1D (pixel_t * in, intermediate_t * out){
    out->red   =  in[-3].red * coeff[0]
                + in[-2].red * coeff[1]
                + in[-1].red * coeff[2]
                + in[ 0].red * coeff[3]
                + in[+1].red * coeff[4]
                + in[+2].red * coeff[5]
                + in[+3].red * coeff[6]
                + in[+4].red * coeff[7];

   /* Code omitted for brevity */
}

void lanczosKernelVConv1D (intermediate_t * in, pixel_t * out){
    long float arr[3];
    arr[0]   =  in[-3].red * coeff[0]
                + in[-2].red * coeff[1]
                + in[-1].red * coeff[2]
                + in[ 0].red * coeff[3]
                + in[+1].red * coeff[4]
                + in[+2].red * coeff[5]
                + in[+3].red * coeff[6]
                + in[+4].red * coeff[7];

    /* Code omitted for brevity */

    /* !!!! Saturate-Cast !!!! */
    out->red = arr[0] < 0 ? 0 : arr[0] > 255 ? 255 : (int) arr[0];
    out->green = arr[1] < 0 ? 0 : arr[1] > 255 ? 255 : (int) arr[1];
    out->blue = arr[2] < 0 ? 0 : arr[2] > 255 ? 255 : (int) arr[2];
}

// Using a separable Lanczos Kernel
void lanczosKernelConv2D (pixel_t * in, pixel_t * out){
    intermediate_t arr[8];
    lanczosKernelHConv1D(&in[-3 * (WIDTH + 6)], &arr[0]);
    lanczosKernelHConv1D(&in[-2 * (WIDTH + 6)], &arr[1]);
    lanczosKernelHConv1D(&in[-    (WIDTH + 6)], &arr[2]);
```

```
47      lanczosKernelHConv1D(&in[                0], &arr[3]);
48      lanczosKernelHConv1D(&in[+    (WIDTH + 6)], &arr[4]);
49      lanczosKernelHConv1D(&in[+2 * (WIDTH + 6)], &arr[5]);
50      lanczosKernelHConv1D(&in[+3 * (WIDTH + 6)], &arr[6]);
51      lanczosKernelHConv1D(&in[+4 * (WIDTH + 6)], &arr[7]);
52
53      lanczosKernelVConv1D(&arr[3], out);
54  }
55
56  void lanczosDownsample(pixel_t * in, pixel_t * out, int offset){
57      for (int i = 0; i < WIDTH / offset; i++){
58          lanczosKernelConv2D(&in[i * offset], &out[i]);
59      }
60  }
```

Listing 4.5: Lanczos Resampling Kernel

The Lanczos 2-dimensional convolution operates in a similar manner to bicubic interpolation. Since the convolution is performed in an 8x8 region, top, bottom, left and right padding of 3 lines/columns needs to be taken into account when moving between lines in memory. The kernel, along with the intermediate results, is chosen to be of float type. During development, it was observed that using long floats provided no increase in accuracy, compared to the OpenCV library, but did dramatically increase overall latency of the operations. Thus, the coefficients and the intermediate results are of float type. Furthermore, it is vital that lines 35-38 be taken into consideration. The range of the output values in Lanczos resampling is not bound between 0 and 255. Saturate-cast must be performed on the output to ensure negative values are rounded to 0 and values greater than 255 are rounded to 255.

Makefile

The toolchain provided by Myriad X requires a subdir.mk makefile to be present in the "shave" directory. The compilation of the code is fairly straightforward, since this makefile is only required to include any source files present in the directory, that need to be compiled. Therefore, the sole content of this makefile is the following line:

```
1  srcs-shave-y += shave.c
```

Listing 4.6: SHAVE Makefile

4.1.2 Control Code and Scripts

This section focuses on the code run by the LeonOS processor, as well as the scripts executed on the host machine. We will initially examine the latter, since they are written in Python and are fairly easy to comprehend.

Header File Creation Script

createInputHeader.py is a CLI-tool that enables the creation of header files that contain the input image and the expected result. Running the CLI-tool with an -h flag set produces the following result:

```
$ ./createInputHeader.py -h
usage: createInputHeader.py [-h]
                            -interpolation INTERPOLATION
                            -image IMAGE
optional arguments:
-h, --help            show this help message and exit
-interpolation INTERPOLATION
                      Downsampling Method. Available options: linear,
                      cubic, lanczos.
-image IMAGE          Path to image.
```

Listing 4.7: Script Interface

The source code of this tool is provided below.

```
#!/usr/bin/env python3
import cv2

def createFiles(method, image):
  ### Read the image ###
  img = cv2.imread(image)

  ### Image is in BGR format ###
  red = []
  green = []
  blue = []
  for i in range(960):
    for j in range (1280):
      blue.append(img[i][j][0])
      green.append(img[i][j][1])
      red.append(img[i][j][2])

  ### Generate the header file ###
  f = open('inputImage.h', 'w')
  f.write("/************************************\n"+
          "    THIS IS AN AUTO GENERATED FILE    \n"+
          " ************************************/")
  f.write("\n#ifndef INPUT_FRAME_H_\n#define INPUT_FRAME_H_\n")
  f.write("#include \"app_config.h\"\n"+
          "#include \"FilterArgs.h\"\n\n")
  if method == cv2.INTER_LINEAR:
    f.write("DDR_DIRECT_DATA ALIGNED(64) pixel_t " +
            "inputFrame[WIDTH * HEIGHT] = {\n")
    for i in range(960*1280):
      f.write("{%d, %d, %d}" % (red[i], green[i], blue[i]))
      if i < 960*1280-1:
        f.write(",\n")
```

```
      else:
        f.write("\n};\n")
  elif method == cv2.INTER_CUBIC:
    '''
    Omitted for brevity
    '''
  elif method == cv2.INTER_LANCZOS4:
    '''
    Omitted for brevity
    '''
  f.write("#endif")
  f.close()
  ### Resize using selected interpolation method ###
  resized = cv2.resize(img, (640, 480), interpolation=method)
  ### Image is in BGR format ###
  '''
  Omitted for brevity
  '''

  ### Generate the header file ###
  f = open('expectedFrame.h', 'w')
  f.write("/***********************************\n"+
          "    THIS IS AN AUTO GENERATED FILE    \n"+
          " ***********************************/")
  f.write("\n#ifndef EXPECTED_FRAME_H_\n#define EXPECTED_FRAME_H_\n")
  f.write("#include \"app_config.h\"\n"+
          "#include \"FilterArgs.h\"\n\n")
  f.write("DDR_DIRECT_DATA ALIGNED(64) pixel_t "+
          "expectedFrame[WIDTH * HEIGHT / 4] = {\n")
  for i in range(480*640):
    f.write("{%d, %d, %d}" % (red[i], green[i], blue[i]))
    if i < 480*640-1:
      f.write(",\n")
    else:
      f.write("\n};\n")
  f.write("#endif")
  f.close()

if __name__ == '__main__':
  import argparse
  # Parse command line arguments
  parser = argparse.ArgumentParser()
  parser.add_argument('-interpolation', required=True,
                      help='Downsampling Method. '+
                        'Available options: linear, cubic, lanczos.')
  parser.add_argument('-image', required=True, help='Path to image.')

  args = parser.parse_args()

  if args.interpolation == 'linear':
    createFiles(cv2.INTER_LINEAR, args.image)
  '''
  Omitted for brevity
  '''
```

Listing 4.8: Script Source Code

The main function uses the argparse library to create the command line tools and invokes the createFiles function with the appropriate arguments. The createFiles function uses the OpenCV library to initially read the image (Line 6). The colour channels are then divided into 3 lists (Lines 8-16). An inputImage.h header file is created, which contains the definition of an inputFrame array residing inside the DDR memory. The array is filled with RGB pixels (Lines 19-34). The code for Bicubic and Lanczos Interpolation is omitted for brevity. It is similar to the one for Bilinear Interpolation, with the exception of some padding lines, initialized with {0, 0 , 0} pixels. The image is then downscaled by half in each dimension using the chosen interpolation method (Line 46). The output is written in an expectedFrame.h header file. These files are crucial, since they provide the data on which the algorithms are to operate and be evaluated on.

We will now dive into the header, C and script files needed by the LeonOS Processor.

RTEMS Configuration

RTEMS is provided in precompiled form by Movidius. The version of RTEMS shipped with the MDK comes with a board support package (BSP) for SPARC/LEON processors that is capable of configuring the system with simple RTEMS directives. These directives are included in an rtems_config.h file, that is inserted in the compilation toolchain. The source code of this file is presented below.

```
1  /* Omitted for brevity */
2  /* ask the system to generate a configuration table */
3  #define CONFIGURE_INIT
4
5  #ifndef RTEMS_POSIX_API
6  #define RTEMS_POSIX_API
7  #endif
8
9  #define CONFIGURE_MICROSECONDS_PER_TICK          1000    /* 1 millisecond */
10
11 #define CONFIGURE_TICKS_PER_TIMESLICE            10      /* 10 milliseconds */
12
13 #define CONFIGURE_APPLICATION_NEEDS_CONSOLE_DRIVER
14
15 #define CONFIGURE_APPLICATION_NEEDS_CLOCK_DRIVER
16
17 #define CONFIGURE_POSIX_INIT_THREAD_TABLE
18
19 #define CONFIGURE_MINIMUM_TASK_STACK_SIZE      4096
20
21 #define CONFIGURE_MAXIMUM_TASKS                 4
22
23 #define CONFIGURE_MAXIMUM_POSIX_THREADS         4
24
25
26 #define CONFIGURE_MAXIMUM_POSIX_KEYS            8
27
28 #define CONFIGURE_MAXIMUM_POSIX_SEMAPHORES      8
29
30 #define CONFIGURE_MAXIMUM_POSIX_MESSAGE_QUEUES  8
31
32 #define CONFIGURE_MAXIMUM_MESSAGE_QUEUES        2
33
34 #define CONFIGURE_MAXIMUM_POSIX_TIMERS          4
35
```

```
36 #define CONFIGURE_MAXIMUM_TIMERS                4
37
38 #define CONFIGURE_MAXIMUM_SEMAPHORES           16
39
40 #define CONFIGURE_MAXIMUM_DRIVERS               6
41
42 #define CONFIGURE_MAXIMUM_DEVICES               6
43
44 #define CONFIGURE_MAXIMUM_USER_EXTENSIONS    1
45 #define CONFIGURE_INITIAL_EXTENSIONS         { .fatal = Fatal_extension }
46
47 #define CONFIGURE_APPLICATION_EXTRA_DRIVERS  OS_DRV_INIT_TABLE_ENTRY
48
49 void *POSIX_Init (void *args);
50
51 #include <rtems/confdefs.h>
52
53 // System Clock configuration on start-up
54 BSP_SET_CLOCK(DEFAULT_OSC_CLOCK_KHZ, DEFAULT_APP_CLOCK_KHZ, 1, 1,
       DEFAULT_RTEMS_CSS_LOS_CLOCKS, 0, APP_UPA_CLOCKS, 0, 0);
55
56 // program L2 cache behavior
57 BSP_SET_L2C_CONFIG(1, DEFAULT_RTEMS_L2C_REPLACEMENT_POLICY,
       DEFAULT_RTEMS_L2C_LOCKED_WAYS,
58                     DEFAULT_RTEMS_L2C_MODE, 0, 0);
59
60 // program L1 cache behavior
61 BSP_SET_L1C_CONFIG(1,1);
62
63 // Configuring shave L2 cache
64 OS_DRV_INIT_SHAVE_L2C_MODE_DEFINE(OS_DRV_SHAVE_L2C_BYPASS);
```

Listing 4.9: RTEMS Configuration File

Since we do not develop a bare-metal application, it is integral that these configuration directives are well understood, as they may be modified during development. We provide a brief analysis of each RTEMS directive in lines 25-63:

- CONFIGURE_MICROSECONDS_PER_TICK sets the time interval between two succesive system clock ticks. The interval is measured in microseconds, therefore, in this case, a system clock tick occurs every millisecond.
- CONFIGURE_TICKS_PER_TIMESLICE sets the maximum time a thread can run on the processor before a context switch is invoked. The timeslice is set at 10 milliseconds.
- CONFIGURE_APPLICATION_NEEDS_CONSOLE_DRIVER specifies that the application will include a console device driver. This driver provides a single device named /dev/console. This device is used for Standard Input, Output and Error I/O Streams. This configuration directive is included, so that our application can run on the debug server provided by Movidius during development. In production, the bootable file does not need this directive.
- CONFIGURE_APPLICATION_NEEDS_CLOCK_DRIVER specifies that the application will include a clock tick device driver. Without a clock tick device driver, RTEMS has no way to know that time is passing and will be unable to support delays and wall time.
- CONFIGURE_POSIX_INIT_THREAD_TABLE is defined if the user wishes to use a POSIX

API Initialization Threads Table. The application may choose to use the initialization tasks or threads table from another API, which is the default option.

- CONFIGURE_MINIMUM_TASK_STACK_SIZE sets the minimum Task Stack Size. The Stack must be able to store the context of the idle tasks as well as any normal stack items used by the tasks (local variables, function call overhead, etc.) so the actual size required depends on what the idle tasks were doing - and will be at its very minimum if the idle tasks are doing nothing.
- CONFIGURE_MAXIMUM_TASKS is the maximum number of Classic API Tasks that can be concurrently active. Similarly, CONFIGURE_MAXIMUM_POSIX_THREADS is the maximux number of POSIX Threads that can be concurrently active.
- The configuration directives of line 42-54 are well-understood.
- CONFIGURE_MAXIMUM_DRIVERS defines the number of device drivers per node. If the application will dynamically install device drivers, then the configuration option value shall be larger than the number of statically configured device drivers.
- CONFIGURE_MAXIMUM_DEVICES is defined to the number of individual devices that may be registered in the device file system.
- CONFIGURE_MAXIMUM_USER_EXTENSIONS defines the maximum number of Classic API User Extensions that can be concurrently active. CONFIGURE_INITIAL_EXTENSIONS is used to initialize the table of initial user extensions. In this script, 1 user extension will be active, namely Fatal_extension, which is a logging tool for errors while initializing RTEMS.

Line 70 declares a POSIX_Init function. By default, this is the entry function of the system, directly after booting up. Therefore, the main function of LeonOS must be named as such.

We will now provide a brief analysis of the BSP directives located in lines 79-86:

- BSP_SET_CLOCK configures the various clocks of the platform. The arguments passed to it are:

  1. The clock frequency of the Reference Oscillator used.
  2. The target frequency of Phase-Locked Loop 0.
  3. Master Divider Nominator and
  4. Denominator used by PLL0.
  5. LEON OS Clocks.
  6. LEON RT Clocks.
  7. UPA Clocks.
  8. SIPP Clocks.
  9. Auxiliary Clocks.

  A Phase-Locked Loop is utilized to generate the operating frequency of 700MHz of the system, using the Reference Oscillator. The Reference Oscillator is clocked at 24MHz.

- BSP_SET_L2C_CONFIG configures the L2 cache of the LeonOS processor. The arguments passed to it are:

1. Enable or Disable Cache
2. Replacement Policy. Default is LRU.
3. Cache ways locked in cache. Default is unlocked cache.
4. Mode of operation. Default is Copy-Back.
5. Number of MTRR Registers
6. Pointer to an array of MTRR Configuration

Memory type range registers (MTRRs) are a set of processor supplementary capability 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. No MTRRs are used in our application.

- BSP_SET_L1C_CONFIG configures the L1 cache of the LeonOS processor. The arguments passed to it are:

1. Enable Data Cache
2. Enable Instruction Cache

Finally, Line 89 initializes the L2 Cache of the SHAVEs in bypass mode. Thus, SHAVE cores do not use their L2 Cache.

Application Configuration Files

We will now examine the app_config.h header file, which is based on the default app_config.h file provided by the MDK. The main body of this header file, as illustrated below, consists of various defines, many of which are used in rtems_config.h.

```
1  #define DEFAULT_APP_CLOCK_KHZ      (700000)
2  #define DEFAULT_OSC_CLOCK_KHZ      (24000)
3
4  #define SHAVES_USED                (1)
5
6  #define APP_UPA_CLOCKS  ( DEV_UPA_SH0      |  \
7                            DEV_UPA_SH1      |  \
8                            DEV_UPA_SH2      |  \
9                            DEV_UPA_SH3      |  \
10                           DEV_UPA_SH4      |  \
11                           DEV_UPA_SH5      |  \
12                           DEV_UPA_SH6      |  \
13                           DEV_UPA_SH7      |  \
14                           DEV_UPA_SH8      |  \
15                           DEV_UPA_SH9      |  \
16                           DEV_UPA_SH10     |  \
17                           DEV_UPA_SH11     |  \
18                           DEV_UPA_SH12     |  \
19                           DEV_UPA_SH13     |  \
20                           DEV_UPA_SH14     |  \
21                           DEV_UPA_SH15     |  \
22                           DEV_UPA_SHAVE_L2 |  \
23                           DEV_UPA_CDMA     |  \
24                           DEV_UPA_CTRL     |  \
25                           DEV_UPA_MTX )
26
```

```
27 #define ALIGNED(x)   __attribute__((aligned(x)))
28 #define DDR_TEXT     __attribute__((section(".ddr.text")))
29 #define DDR_DATA     __attribute__((section(".ddr.data")))
30 #define DDR_RODATA   __attribute__((section(".ddr.rodata")))
31 #define DDR_BSS      __attribute__((section(".ddr.bss")))
32 #define CMX_TEXT     __attribute__((section(".cmx.text")))
33 #define CMX_RODATA   __attribute__((section(".cmx.rodata")))
34 #define CMX_DATA     __attribute__((section(".cmx.data")))
35 #define CMX_BSS      __attribute__((section(".cmx.bss")))
36 #define DDR_DIRECT_DATA    __attribute__((section(".ddr_direct.data")))
37
38 int initClocksAndMemory(int * dataPartitionID);
```

Listing 4.10: App Configuration Header File

The main clock of the VPU is set to a frequency of 700MHz, while the reference oscillator frequency is set to 24MHz. Furthermore, in lines 6-25, the clocks of the Microprocessor Array are set to include all SHAVE cores and components. These Macros are all utilized in rtems_config.h, and thus, that file includes the app_config header.

In addition to these, a few more definitions are included in this header file. The Macro SHAVES_USED, which is initially set to one, provides a simple interface to run this stage on an arbitrary number of SHAVE cores. The Macros in lines 27-36 provide a user-friendly interface for declaration of variables in specific regions of the DDR or CMX memories, and with specific alignment. Finally, a function that initializes all the system clocks and memories is declared. The body of this function is the sole content of the app_config.c files, along with the necessary includes. It is listed in the following code snippet.

```
1  // Setup all the clock configurations needed by this application and also the ddr
2  int initClocksAndMemory(int * dataPartitionID){
3      s32 sc = 0;
4      uint32_t i;
5      u8 partID[2]; // We'll configure 2 partitions: one for code and one for data
6
7      // Shave L2 cache is initialized by RTEMS
8      // Configure shave L2 cache
9      rtems_status_code ret = RTEMS_SUCCESSFUL;
10
11     // Set and invalidate Shave L2 cache partitions 16K / shave is sufficient
12     ret += OsDrvShaveL2cAddPart(OS_DRV_SHAVE_L2C_PART_128_KB, 0, 1, &partID[0]);
13     ret += OsDrvShaveL2cAddPart(OS_DRV_SHAVE_L2C_PART_128_KB, 0, 1, &partID[1]);
14
15     for (i=0; i < SHAVES_USED; i++){
16         // Assign the second partition as instruction partition for all shaves
17         ret += OsDrvShaveL2cAssignPart(i,partID[1],
       OS_DRV_SHAVE_L2C_NON_WIN_INST_PART, 0);
18         // Assign the first partition as data partition for all shaves
19         ret += OsDrvShaveL2cAssignPart(i,partID[0],
       OS_DRV_SHAVE_L2C_NON_WIN_DATA_PART, 0);
20     }
21
22     if(RTEMS_SUCCESSFUL == ret){
23         // Return the data partition ID, that can be used to flush
24         // and invalidate the partition
25         *dataPartitionID = partID[0];
26
27         // Initialize the SHAVE driver
28         sc = OsDrvSvuInit();
```

```
    }
    else sc = (s32) ret;

    return sc;
}
```

Listing 4.11: App Configuration Source Code

Even though the L2 Cache is set to be bypassed by the SHAVE cores, a basic initialization is provided in this file, in case we need to turn it on. Two partitions of 128KB are created, with each one of them consisting of eight 16KB blocks. The first partition is assigned to the SHAVEs as instruction cache, while the second is assigned as data cache. The OsDrvSvuInit function is then called to initialize the SHAVE Driver.

LeonOS POSIX_Init Source Code

After the successful configuration of RTEMS and basic initialization of the board, permission to the POSIX_Init function of LeonOS to execute is granted. This function is responsible for booting up the SHAVE Cores and monitoring their performance during their operation. The source code of the main.c file that is included in the /leon directory is provided below.

```
#include "inputImage.h"
#include "expectedFrame.h"
#include "app_config.h"
// includes omitted for brevity

extern u32 bilinearShave0_Entry;
/* Code omitted for brevity */


u32*  bilinearShave_Entry[16]={
        &bilinearShave0_Entry,
/* Code omitted for brevity */
};

uint32_t pll0;
int sysclk_khz;
u64 cycles_start, cycles_end,cycles_elapsed;

static osDrvSvuHandler_t handler[SHAVES_USED];
DDR_DIRECT_DATA ALIGNED(64) downsampleFilterArgs_t threadArgs[SHAVES_USED];
DDR_DIRECT_DATA ALIGNED(64) pixel_t outputFrame[WIDTH * HEIGHT / 4];

void *POSIX_Init(void *args) {
    UNUSED(args);

    /* Variable declaration omitted for brevity */

    status = initClocksAndMemory(&l2cDataPartitionId);
    OsDrvCprGetClockFrequency(OS_DRV_CPR_CLK_PLL0,&pll0);
    sysclk_khz=pll0;
    if (status != OS_MYR_DRV_SUCCESS){
        printf("Error initializing clocks, memory or shaves L2 caches. Function
    returned error code %d\n", status);
        exit(1);
    };

```

```
    int LINES_PER_CORE = HEIGHT / SHAVES_USED;
    for(u32 i = 0; i < SHAVES_USED; i++) {
        threadArgs[i].inpImage = &inputFrame[LINES_PER_CORE * WIDTH * i];
        threadArgs[i].outImage = &outputFrame[LINES_PER_CORE * WIDTH / 4 * i];
        threadArgs[i].ffactor = 0.5;
        threadArgs[i].workload = LINES_PER_CORE;
        threadArgs[i].shaveId = i;
    }

    printf("Open shaves\n");
    OsDrvShaveL2cFlushInvPart(l2cDataPartitionId, OS_DRV_SHAVE_L2C_FLUSH_INV);

    for(i=0; i<SHAVES_USED; i++) {
        sc = OsDrvSvuOpenShave(&handler[i], i, OS_MYR_PROTECTION_SEM);
        if (sc == OS_MYR_DRV_SUCCESS) {
            sc = OsDrvSvuResetShave(&handler[i]);
            if (sc)
                exit(sc);
            sc = OsDrvSvuSetAbsoluteDefaultStack(&handler[i]);
            if (sc)
                exit(sc);
        }
        else {
            printf("[T2] cannot open shave %d\n", i);
        }

    }

    OsDrvFreeRunCntRead(OS_DRV_FREE_RUN_CNT_CSS, &cycles_start);
    for(i=0; i<SHAVES_USED; i++){
        sc = OsDrvSvuStartShaveCC(&handler[i],
                                  (u32) bilinearShave_Entry[i],
                                  "i",
                                  (void *) &threadArgs[i]);
        if (sc)
            exit(sc);
    }

    for(i=0; i<SHAVES_USED; i++) {
        sc = OsDrvSvuWaitShaves(1, &handler[i], OS_DRV_SVU_WAIT_FOREVER, &running);
        if (sc)
            exit(sc);
        sc = OsDrvSvuCloseShave(&handler[i]);
        if (sc)
            exit(sc);
    }
    OsDrvFreeRunCntRead(OS_DRV_FREE_RUN_CNT_CSS, &cycles_end);
    cycles_elapsed = cycles_end - cycles_start;
    OsDrvShaveL2cFlushInvPart(l2cDataPartitionId, OS_DRV_SHAVE_L2C_FLUSH_INV);

    u8 matching_data = 1;

    for (int i = 0; i < HEIGHT * WIDTH / 4; i += 1){
        if (expectedFrame[i].red != outputFrame[i].red){
            matching_data = 0;
            break;
        }
        /* Code omitted for brevity */
```

```
94      }
95
96      if (matching_data){
97          printf("Output matches golden data!\n");
98          printf("System running at %u.%02uMHz\n", sysclk_khz/1000, (sysclk_khz
        %1000+5)/10);
99          printf("SHAVE Cores used: %d\n", SHAVES_USED);
100         unsigned t_100us = (u32)( cycles_elapsed /( sysclk_khz / 10 ));
101         printf("Computation took: ");
102         printf( "%5u.%ums ", t_100us/10, t_100us%10 );
103         printf( "(%11llucycles)\n", cycles_elapsed );
104     }
105     else{
106         printf("Output does not match golden data!\n");
107     }
108
109     printf("Finish\n");
110     unitTestFinalReport ();
111     exit(0);
112 }
```

Listing 4.12: Main Function Source Code

Initially, we extern the entry functions of the SHAVE Cores. Memory is statically allocated in the DDR for the filter arguments of the entry function and for the output frame. The input and expected frame are included in the header files. POSIX_Init calls initClocksAndMemory, as defined in app_config.h. This function only initializes the SHAVE Cores and creates partitions of the L2 SHAVE Cache. The rest of the clocks and memories are initialized by RTEMS. In lines 41-48, all the necessary arguments for the execution of a SHAVE Core are created. Lines 53-67 open up the SHAVE Cores that will be utilized, resets them and sets their stack. Lines 70-77 start the execution of the used cores. The third argument passed to OsDrvSvuStartShaveCC is a string, whose length is equal to the number of arguments that will be provided. The value "i" stands for one argument provided, that is to be stored in the Integer Register File. Since the argument provided is not an integer but a pointer to a struct, we typecast it to a void pointer. Thus, a memory address is written in the IRF. Myriad X does not make use of virtual addresses, therefore each SHAVE core can typecast this pointer back to its original type, and access its contents. Lines 79-86 then wait for the SHAVEs to return and subsequently shut them down. This part of the code (lines 70-86) is "guarded" by reads of the CSS Free Run Counter, to measure the exact latency of the compute-intensive part of the application. The output frame is then compared to the expected output.

The source code above is used to execute the Bilinear Interpolation algorithm. Bicubic Interpolation and Lanczos Resampling can be invoked in a similar manner. However, a few more lines of code need to be included to create the appropriate padding of the image. The type of padding was chosen based on the padding performed by the OpenCV library, which is a form of replication padding. The following snippet contains the correspondent code for Bicubic Interpolation. In the case of Lanczos Resampling, padding is created similarly. Free Run Counters are also utilized, to measure the performance of padding being executed on the LeonOS.

```
1 OsDrvFreeRunCntRead(OS_DRV_FREE_RUN_CNT_CSS, &cycles_start);
2 for (int j = 1; j <= HEIGHT; j++){
3     // Left Padding
4     inputFrame[j * (WIDTH + 2)].red   = inputFrame[j * (WIDTH + 2) + 1].red;
5     // Right Padding
```

```
    inputFrame[(j+1) * (WIDTH + 2) - 1].red   = inputFrame[(j+1) * (WIDTH + 2) -
    2].red;
    /* Code omitted for brevity */
}

for(int i = 0; i < WIDTH + 2; i++){
    // Top Padding
    inputFrame[i].red   = inputFrame[i + WIDTH + 2].red;
    // Bottom Padding
    inputFrame[(WIDTH + 2) * (HEIGHT + 1) + i].red   = inputFrame[(WIDTH + 2) * (
    HEIGHT) + i].red;
    /* Code omitted for brevity */
}
OsDrvFreeRunCntRead(OS_DRV_FREE_RUN_CNT_CSS, &cycles_end);
```

Listing 4.13: Padding Code

Custom Linker Script

A custom script for the GNU Linker is provided by Movidius. This script contains a detailed memory map to be used by the application. Even though the default script proved sufficient for bilinear and bicubic interpolation, a few modifications were necessary for Lanczos resampling. That is due to the fact that the memory required by the code segment of the SHAVE cores was, in some cases, greater than the one assigned in the script. Therefore, the length of the SHVX_CODE regions was increased to 35KB, and the length of the corresponding SHVX_DATA regions was reduced to 93KB.

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

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

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

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

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

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

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

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

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

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

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

  SHV10_CODE (wx) : ORIGIN = 0x70000000 + 10 * 128K,       LENGTH = 32K
  SHV10_DATA (w)  : ORIGIN = 0x70000000 + 10 * 128K + 32K, LENGTH = 96K

  SHV11_CODE (wx) : ORIGIN = 0x70000000 + 11 * 128K,       LENGTH = 32K
  SHV11_DATA (w)  : ORIGIN = 0x70000000 + 11 * 128K + 32K, LENGTH = 96K

  SHV12_CODE (wx) : ORIGIN = 0x70000000 + 12 * 128K,       LENGTH = 32K
  SHV12_DATA (w)  : ORIGIN = 0x70000000 + 12 * 128K + 32K, LENGTH = 96K

  SHV13_CODE (wx) : ORIGIN = 0x70000000 + 13 * 128K,       LENGTH = 32K
  SHV13_DATA (w)  : ORIGIN = 0x70000000 + 13 * 128K + 32K, LENGTH = 96K

  SHV14_CODE (wx) : ORIGIN = 0x70000000 + 14 * 128K,       LENGTH = 32K
  SHV14_DATA (w)  : ORIGIN = 0x70000000 + 14 * 128K + 32K, LENGTH = 96K

  SHV15_CODE (wx) : ORIGIN = 0x70000000 + 15 * 128K,       LENGTH = 32K
  SHV15_DATA (w)  : ORIGIN = 0x70000000 + 15 * 128K + 32K, LENGTH = 96K

  CMX_DMA_DESCRIPTORS  (wx) : ORIGIN = 0x78000000 + 19 * 128K , LENGTH = 3K

  CMX_OTHER (wx) :            ORIGIN = 0x70000000 + 19 * 128K + 3K, LENGTH = 128K -
     6K

  LOS (wx) : ORIGIN = 0x80000000, LENGTH = 4 * 128K
  LRT (wx) : ORIGIN = 0x80000000 + 4 * 128K, LENGTH = 4 * 128K

  DDR_DATA (wx) : ORIGIN = 0x80000000 + 10 * 128K, LENGTH = 512M - 10 * 128K
}

SEARCH_DIR(output)

INCLUDE myriad2_leon_default_elf.ldscript
INCLUDE myriad2_shave_slices.ldscript
INCLUDE myriad2_SDL_general_purpose_sections.ldscript
```

Listing 4.14: Custom Linker Script

This script initially defines the CMX Slices, that are to be associated with each SHAVE Core. Notice that both the code and data segments occupy addresses that correspond to the CMX memory. On the other hand, LeonOS and LeonRT are assigned slices of the DDR memory to store their code segments. Subsequently, a small region is declared for the CMX DMA descriptors and the remainders of the CMX and DDR memories are also defined. Finally, a few more linker scripts are included.

4.1.3 Core Parallelization

A SHAVE Processor accelerates these preprocessing algorithms, compared to executing them on the LeonOS Core. However, the single core performance is nowhere near the one required by our application. It should be noted, that achieving real-time execution is not enough. The execution time of these downsampling algorithms must be negligible compared to that of the Neural Network. Thus, the need to parallelize the operations performed during this stage arises.

As we discussed previously, in section 4.1.2, the control code was developed in a way that enables in-parallel execution, without modifying the code. In fact, only the macro SHAVES_USED found in leon/app_config.h needs to be changed to the desirable value. Therefore, each SHAVE Core will run the correspondent kernel, producing only a fraction of the total output lines. This is possible, because the algorithms are embarrassingly parallel. Figure 4.2 illustrates which pixels of the input image are to be utilized per iteration for the bilinear interpolation algorithm. Each iteration, produces an output pixel, by reading the values of the correspondent pixels of the input image, and applying linear operations. Thus, no intermediate values are produced or needed to calculate the result. This means, that there are no true dependencies within the consecutive iterations of the algorithm. The same principle holds true for bicubic interpolation and Lanczos resampling as well. Both of these algorithms utilize a 2-dimensional convolution to produce an output pixel.

Consequently, parallelizing the execution of the downsampling on multiple cores is straightforward. Each core is assigned a number of output lines to be produced and is given access to the necessary input lines via a pointer.

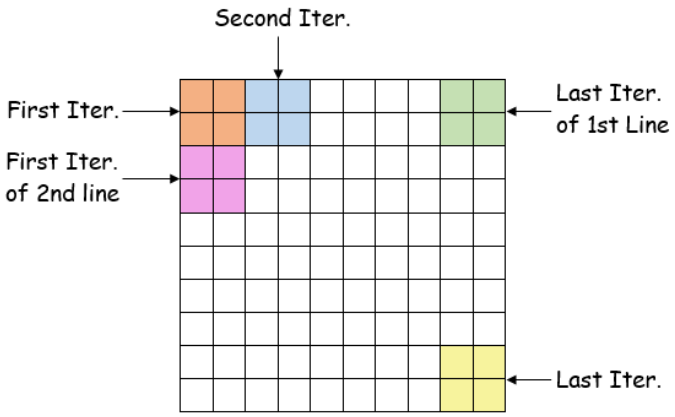


Figure 4.2: Execution of Bilinear Interpolation Downsampling

When increasing the number of active cores, the algorithms are quickly driven to saturation in terms of performance. Further increase of cores used, leads to performance degradation. The origins of this behaviour will be thoroughly explained in Section 5.1.2. Briefly, contention arises between the cores when reading data from the main DDR memory. This contention can be so severe, that in certain cases, operating on one core produces the same latency as operating on the maximum amount of cores, namely 16.

4.1.4 Scratchpad Memory Data Transfers

To eliminate the contention point described in the previous section, the Scratchpad Memory was utilized. As was already described in section 1.3.1, Scratchpads are SRAM memories that, unlike caches, neither flush nor request data to or from the main memory. Both the .text and .data sections of the code executed on the SHAVEs is stored in the CMX. Thus, any global or local variable declared in the shave/shave.c file resides in the CMX (Scratchpad) memory. Therefore, allocating buffers to store any input data is equivalent to defining them as global variables. To fill these buffers, the CMX DMA engine must be used.

Movidius provides a convenient API inside the MDK, to configure and utilize this component. The declarations of these functions are located in the ShDrvCmxDma.h header file. We provide several of them, along with their definitions, below:

```
/// Initializes the CMX DMA driver
///
/// This must be the first function called before using any other function form
/// the driver
///
/// @param [in] config Structure containing CMX DMA driver parameters
///
/// @return
///     MYR_DRV_SUCCESS the driver was successfully initialized
///     MYR_DRV_ERROR an error happened during driver initialization
///     MYR_DRV_ALREADY_INITIALIZED the driver was already initialized by
///      a previous call to this function
///
int32_t ShDrvCmxDmaInitialize(ShDrvCmxDmaSetupStruct *config);

// Creates a new 1D transaction
///
/// The function returns a handle to the new transaction
///
/// @param [out] handle a handle to a transaction list containing only the newly
///             created transaction
/// @param [in] transaction the new transaction to be created
/// @param [in] src source address for the transaction
/// @param [in] dst destination address for the transaction
/// @param [in] size transaction size in bytes
///
/// @return
///     MYR_DRV_SUCCESS the transaction was successfully created
///     MYR_DRV_ERROR the transaction could not be created
///
int32_t ShDrvCmxDmaCreateTransaction(ShDrvCmxDmaTransactionHnd *handle,
                                     ShDrvCmxDmaTransaction *transaction,
                                     uint8_t *src, uint8_t *dst, uint32_t size);

/// Creates and links a new 1D transaction to an existing transaction list
///
/// @param [out] handle for the list on which the new transaction will be added
/// @param [in] transaction the new transaction to be created and added to the
///            list
/// @param [in] src source address for the transaction
/// @param [in] dst destination address for the transaction
/// @param [in] size transaction size in bytes
///
/// @return
```

```
///     MYR_DRV_SUCCESS the transaction was successfully added
///     MYR_DRV_ERROR the new transaction could not be added
///
int32_t ShDrvCmxDmaAddTransaction(ShDrvCmxDmaTransactionHnd *handle,
                                  ShDrvCmxDmaTransaction *transaction,
                                  uint8_t *src, uint8_t *dst, uint32_t size);

/// Start the DMA transfer for a transaction list
///
/// @param [in] handle for the transaction list
///
/// @return
///     MYR_DRV_SUCCESS the transaction was successfully started
///     MYR_DRV_RESOURCE_BUSY the device was busy, the transaction was not started
///     MYR_DRV_ERROR the device wasn't initialized
///
///     This function doesn't wait for the transfer to finish. It returns
///     immediately after the transfer starts.
///
int32_t ShDrvCmxDmaStartTransfer(ShDrvCmxDmaTransactionHnd *handle);

/// Waits for one or more transactions started by ShDrvCmxDmaStartTransfer to
/// finish
///
/// @param [in] handle for a transaction or a list of transactions
///
/// @return
///     MYR_DRV_SUCCESS the transaction finished successfully
///     MYR_DRV_ERROR transfer finished with an error
///
int32_t ShDrvCmxDmaWaitTransaction(ShDrvCmxDmaTransactionHnd *handle);
```

Listing 4.15: CMX DMA Engine API

A few modifications need to be made to our shave.c code, to make use of the above API:

```
#include <ShDrvCmxDma.h>

// Omitted for brevity

static ShDrvCmxDmaTransaction CMX_DMA_DESCRIPTOR_DATA task, out_task;
static pixel_t ALIGNED(32)  inputImageCMX[WIDTH * 2];
static pixel_t ALIGNED(32)  outputImageCMX[(WIDTH / 2)];

// Omitted for brevity

__attribute__((dllexport)) void Entry( void * theArgsPointer ) {
    // Omitted for brevity

    int retStatus=MYR_DRV_SUCCESS;
    ShDrvCmxDmaTransactionHnd  handle, out_handle;

    retStatus = (int)ShDrvCmxDmaInitialize(NULL);
    assert((retStatus == MYR_DRV_SUCCESS) || (retStatus ==
    MYR_DRV_ALREADY_INITIALIZED));

    // Omitted for brevity

    for (u32 i = 0; i < workload / offset; i ++) {
        ShDrvCmxDmaCreateTransaction(&handle,
```

```
                                    &task,
                                    &inputImageDDR[i * WIDTH * offset],
                                    inputImageCMX,
                                    sizeof(pixel_t) * WIDTH * 2);
        ShDrvCmxDmaStartTransfer(&handle);
        ShDrvCmxDmaWaitTransaction(&handle);

        bilinear_kernel(inputImageCMX, outputImageCMX, offset);

        ShDrvCmxDmaCreateTransaction(&out_handle,
                                    &out_task, outputImageCMX,
                                    &outputImage[i * (WIDTH / offset)],
                                    sizeof(pixel_t) * (WIDTH / offset));
        ShDrvCmxDmaStartTransfer(&out_handle);
        ShDrvCmxDmaWaitTransaction(&out_handle);
    }

    SHAVE_HALT;
}
```

Listing 4.16: Extensions to SHAVE code for Bilinear Interpolation

The modifications made to the bicubic and Lanczos Interpolation algorithms are similar, but also account for the necessary padding lines.

In this code segment, 2 buffers are allocated, for input and output lines respectively. Per iteration, a DMA transaction is created, to fetch the necessary lines of the input image to produce one output line. The output line is initially stored in the CMX via the allocated buffer. Then, a new transaction is created that transfers the produced output line back to the DDR memory. Thus, all read and write operations are performed on the CMX. This accelerates the preprocessing performed significantly, not only due to the faster SRAM memory, but also due to the fact that each SHAVE has exclusive ports to their preferential CMX slice.

At this point, the preprocessing stage satisfies the timing constraints set, as it operates at at least one order of magnitude lower latency, than then Convolutional Neural Network. The following sections aim to further accelerate this stage, by fine-tuning several segments of code.

4.1.5 Offload Padding

Bicubic, as well as Lanczos Interpolation, require padding lines and columns in order to operate properly. The Python scripts provided in Section 4.1.2 provide zero-padding to the input frames. However, this is not the desirable form of padding and therefore, LeonOS is assigned the task to create the appropriate replication padding. This choice was not coincidental. In production, the camera used, will provide raw RGB data to our system. Thus, our system must be capable of padding these raw data.

The padding performed by LeonOS operates by issuing memory reads and writes to the DDR. This task is obviously embarrassingly parallel. However, as observed in Section 4.1.3, not much can be gained performance-wise solely by parallelizing the operation on the SHAVE Cores, due to contention for memory resources. Therefore, each SHAVE Core is set to be responsible for correcting the padding on the input lines, as they arrive to its preferential CMX slice through the DMA engine. All SHAVE Cores perform left and right padding, while the SHAVE ID is used to identify the two cores (first and last of the active group) that need to perform top and bottom padding.

The following 2 snippets contain the code that needs to be added to the entrypoint function of bicubic and Lanczos interpolation.

```
for(int i=0; i < workload / offset; i++){
    /* Omitted for brevity */

    for (int j = 0; j < 4; j++){
        inputImageCMX[j * (WIDTH+2)].red = inputImageCMX[j * (WIDTH+2) + 1].red;
        /* Omitted for brevity */

        inputImageCMX[(j+1) * (WIDTH+2) - 1].red =
                                        inputImageCMX[(j+1) * (WIDTH+2) - 2].red;
        /* Omitted for brevity */
    }
    if (shaveId == 0 && i == 0){
        for(int j = 0; j < WIDTH+2; j++){
            inputImageCMX[j].red = inputImageCMX[(WIDTH+2) + j].red;
            /* Omitted for brevity */
        }
    }
    else if (shaveId == SHAVES_USED-1 && i == workload / offset - 1){
        for(int j = 0; j < WIDTH+2; j++){
            inputImageCMX[3 * (WIDTH + 2) + j].red =
                                            inputImageCMX[2 * (WIDTH + 2) + j].red;
            /* Omitted for brevity */
        }
    }
    /* Omitted for brevity */
}
```

Listing 4.17: Padding Offloaded to SHAVE for Bicubic Resampling

```
for(int i=0; i < workload / offset; i++){
    /* Omitted for brevity */

    for (int j = 0; j < 8; j++){
        for(int k = 0; k < 3; k++){
            inputImageCMX[j * (WIDTH+6) + k].red =
                                inputImageCMX[j * (WIDTH+6) + 3].red;
```

```
            /* Omitted for brevity */

            inputImageCMX[(j+1) * (WIDTH+6) - 1 - k].red =
                            inputImageCMX[(j+1) * (WIDTH+6) - 4].red;
            /* Omitted for brevity */
        }
    }

    if( shaveId == 0){
        if (i==0){
            for(int j = 0; j < 3; j++){
                for(int k = 0; k < WIDTH + 6; k++){
                    inputImageCMX[j * (WIDTH+6) + k].red =
                                    inputImageCMX[k + 3 * (WIDTH+6)].red;
                    /* Omitted for brevity */
                }
            }
        }
        else if (i==1){
            for(int k = 0; k < WIDTH + 6; k++){
                inputImageCMX[k].red = inputImageCMX[k + (WIDTH+6)].red;
                /* Omitted for brevity */
            }
        }
    }
    else if( shaveId == SHAVES_USED-1 ){
        if ( i == workload / offset-2 ) {
            for(int k = 0; k < WIDTH + 6; k++){
                inputImageCMX[(WIDTH+6) * 7 + k].red =
                                inputImageCMX[(WIDTH+6) * 6 + k].red;
                /* Omitted for brevity */
            }
        }
        else if ( i == workload / offset - 1){
            for(int j = 0; j < 3; j++){
                for(int k = 0; k < WIDTH + 6; k++){
                    inputImageCMX[(j+5) * (WIDTH+6 ) + k].red =
                                    inputImageCMX[k + 4 * (WIDTH+6)].red;
                    /* Omitted for brevity */
                }
            }
        }
    }
    /* Omitted for brevity */
}
```

Listing 4.18: Padding Offloaded to SHAVE for Lanczos Resampling

It should be noted, that top and bottom padding occur only once, in the first iteration of the first SHAVE Core and in the last iteration of the last SHAVE Core respectively, for bicubic interpolation. On the other hand, during Lanczos resampling, padding occurs twice more. Once for the second iteration of the first SHAVE (in the form of top padding) and once for the second to last iteration of the last SHAVE (in the form of bottom padding).

This repetition of the top and bottom padding reveals an interesting property that is to be exploited in the following section.

4.1.6 Sliding Window Buffers

The algorithms have been developed in such a manner, that each iteration of the main loop produces exactly one output line. By setting the downsampling factor to 0.5, the stride used is lower than the number of lines needed per iteration of the loop for bicubic or Lanczos interpolation. This means, that each iteration of the loop, excluding the initial one, only needs to fetch two new input lines. This is evident in Figure 4.3, which depicts the contents of the input image CMX buffer for Lanczos resampling, during the first two iterations of the main loop.

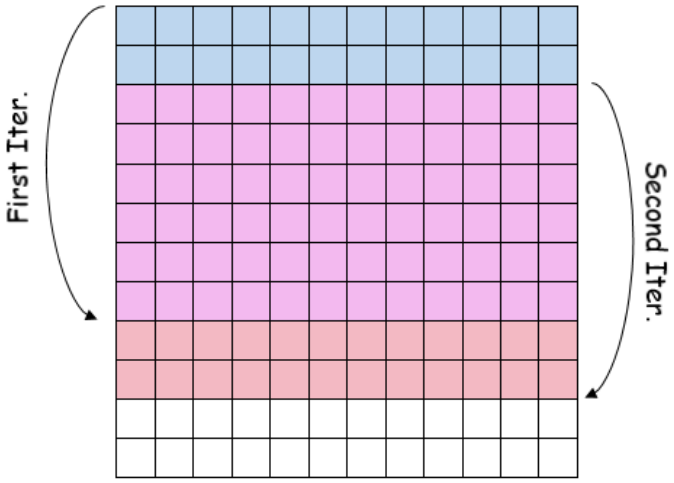


Figure 4.3: Contents of Input Image CMX Buffer for Lanczos Resampling

To take advantage of this property, we would need a data structure, similar to regular arrays, that additionally enables the user to shift its contents downwards and fill the last lines with data. Conceptually, this operation is illustrated in Figure 4.4. However, such a data structure is not provided by Movidius and needs to be created. The proposed architecture is illustrated in Figure 4.5. It consists of a double-linked list. Each node of the list contains a buffer that holds one line of input data. This architecture enables sliding the proposed buffer with a very low latency overhead, as only a simple rearrangement of the pointers connecting the nodes is required.

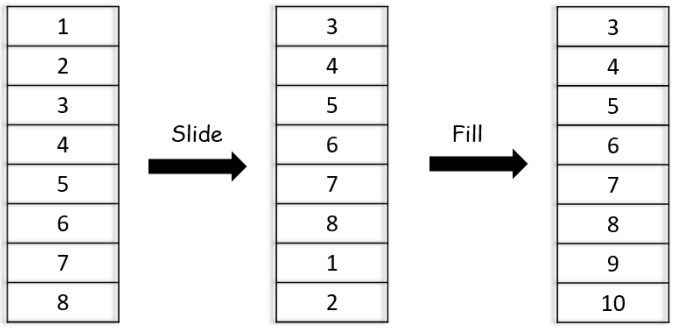


Figure 4.4: Basic Operation of Sliding Window Buffer

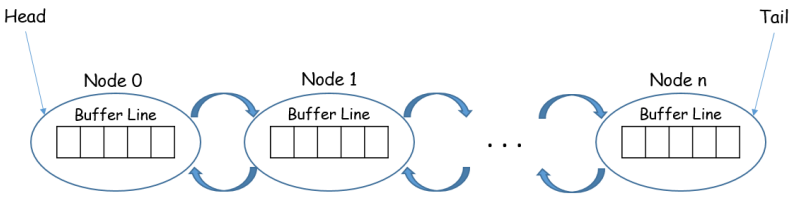


Figure 4.5: Sliding Window Buffer Backend Architecture

Based on the backend architecture proposed above, a user-friendly API, that provides such a data structure, was developed. The code is structured in such a way, so that it can be reused in other applications, where necessary. The Memory Manager is utilized by this data structure and needs to be included in the components section of the Makefile. In addition to this, the Myriad Mem Init component needs to be included, and the MemMgrInitialize() function, declared in the MyriadMemInit.h file, needs to be called inside LeonOS's initClocksAndMemory function. The proposed API is listed below.

```
#ifndef _SLIDING_BUFFERS_H
#define _SLIDING_BUFFERS_H
#include <mv_types.h>
#include "memManagerApi.h"

struct BufferLine {
  u8 * line;                        // Line Buffer
  struct BufferLine * nextLine;     // Pointer to next node
  struct BufferLine * prevLine;     // Pointer to prev node
  int lineNum;                      // # of current node
};
typedef struct BufferLine* BufferHead;
typedef struct BufferLine* BufferTail;

struct SlidingWindowBufferStruct{
  BufferHead head;                  // Pointer to Head of the list
  BufferTail tail;                  // Pointer to Tail of the list
  u32 heightOfBuffer;               // Total number of nodes in the list
  u32 widthOfBuffer;                // Size of each node's buffer
  u32 sizeOfDatatype;               // Size of the individual elements stored
  MemMgrAreas CMXslice;             // The CMX Slice used to store the structure
};
typedef struct SlidingWindowBufferStruct* SlidingWindowBuffer;

// Creates a Sliding Window Buffer of given height and width, that contains
// objects of given datatype size.
// The function returns a pointer to the new buffer.
//
// @param [in] shaveID: The ID of the SHAVE that requests this buffer.
// @param [in] height: Height of the buffer.
// @param [in] width: Width of the buffer.
// @param [in] datatypeSize: Size of the objects that the buffer stores in bytes.
//
// @return A pointer to a SlidingWindowBufferStruct
//
SlidingWindowBuffer CreateSlidingWindowBuffer (u8 shaveID, u32 height,
                                               u32 width, u32 datatypeSize);
// Deallocates all the allocated memory of the structure
// and then deletes the struct itself.
```

```
//
// @param [in] buffer: The Sliding Window Buffer to be deleted.
void CleanupSlidingWindowBuffer(SlidingWindowBuffer buffer);

// This function is called to get one line of the structure.
//
// @param [in] buffer: The Sliding Window Buffer that contains the requested line.
// @param [in] linenum: The number of the line inside the buffer.
// @param [out] dst: The pointer to the requested line.
void SWBGetLine(SlidingWindowBuffer buffer, int lineNum, u8 * dst);

// This function is called to get all lines of the structure.
// The lines are returned in the form of a 2-dimensional array.
//
// @param [in] buffer: The Sliding Window Buffer that contains the requested lines.
// @param [out] dst: The pointer to the requested array.
void SWBGetAllLines(SlidingWindowBuffer buffer, u8 ** dst);

// This function is called to slide the buffer by an offset.
//
// @param [in] buffer: The Sliding Window Buffer to be shifted.
// @param [in] offset: Offset by which the buffer will be shifted.
void SWBSlideWindow(SlidingWindowBuffer buffer, int offset);

// This function is called to fill the entire buffer with data.
//
// @param [in] buffer: The Sliding Window Buffer to be filled.
// @param [in] src: Pointer to the array that contains the data to be copied.
void SWBFillWindow(SlidingWindowBuffer buffer, u8 * src);

// This function is called to slide the buffer by an offset,
// and fill an offset number of lines with new data.
// @param [in] buffer: The Sliding Window Buffer to operate on.
// @param [in] offset: Offset by which the buffer will be shifted.
// @param [in] src: Pointer to the array that contains the data to be copied.
void SWBSlideAndFillWindow(SlidingWindowBuffer buffer, int offset, u8 * src);
#endif // _SLIDING_BUFFERS_H
```

Listing 4.19: SWBuffer API (SlidingBuffer/SlidingBuffer.h)

The source code of the proposed API is included below.

```
#include "SlidingBuffer.h"
#include <stdio.h>
#include <ShDrvCmxDma.h>
#include <svuCommonShave.h>
#include "memManagerApi.h"

SlidingWindowBuffer CreateSlidingWindowBuffer (u8 shaveID, u32 height, u32 width,
    u32 datatypeSize){
    // Create the head of the list
  struct BufferLine * head = (struct BufferLine *) MemMgrAlloc (sizeof(struct
    BufferLine), shaveID, 32);
  head->lineNum = 0;
  head->line = (u8 *) MemMgrAlloc(datatypeSize * width, shaveID, 32);
  struct BufferLine * iter = head;
  for (u32 i=1; i<height; i++){
      // Create each subsequent node of the list
```

```
15      struct BufferLine * newLine = (struct BufferLine *) MemMgrAlloc (sizeof(struct
        BufferLine), shaveID, 32);
16      newLine->line = (u8 *) MemMgrAlloc(datatypeSize * width, shaveID, 32);
17      newLine->lineNum = i;
18      newLine->prevLine = iter;
19      iter->nextLine = newLine;
20      iter = newLine;
21    }
22
23      // Allocate memory of the SlidingWindowBufferStruct, then set its config.
24    SlidingWindowBuffer buffer = (SlidingWindowBuffer) MemMgrAlloc(sizeof(struct
        SlidingWindowBufferStruct), shaveID, 32);
25    buffer->head = head;
26    buffer->tail = iter;
27    buffer->heightOfBuffer = height;
28    buffer->widthOfBuffer = width;
29    buffer->sizeOfDatatype = datatypeSize;
30    buffer->CMXslice = shaveID;
31    return buffer;
32  }
33
34  void CleanupSlidingWindowBuffer(SlidingWindowBuffer buffer){
35    struct BufferLine* currDealloc = buffer->head;
36    for(u32 i=0; i<buffer->heightOfBuffer; i++){
37      struct BufferLine* nextToDealloc = currDealloc->nextLine;
38      // First deallocate the buffers, then the node itself.
39      MemMgrFree(currDealloc->line);
40      MemMgrFree(currDealloc);
41      currDealloc = nextToDealloc;
42    }
43    // Finally, deallocate the SlidingWindowBufferStruct
44    MemMgrFree(buffer);
45  }
46
47  void SWBGetLine(SlidingWindowBuffer buffer, int lineNum, u8 * dst){
48    struct BufferLine * iter = buffer->head;
49    for(int i=0; i<lineNum; i++)
50      iter = iter->nextLine;
51    dst = iter->line;
52  }
53
54  void SWBGetAllLines(SlidingWindowBuffer buffer, u8 ** dst){
55    struct BufferLine * iter = buffer->head;
56    for(u32 i=0; i<buffer->heightOfBuffer; i++){
57      dst[i] = iter->line;
58      iter = iter->nextLine;
59    }
60  }
61
62  void SWBSlideWindow(SlidingWindowBuffer buffer, int offset){
63    // Shift offset number of lines from head,
64    // keep the old head to use it to locate the new tail.
65    struct BufferLine* oldHead = buffer->head;
66    struct BufferLine* newHead = buffer->head;
67
68    for(int i=0; i<offset; i++){
69      newHead = newHead->nextLine;
70    }
```

```
  buffer->head = newHead;
  // Place the tail at the old head, and iteratively search
  // for the new tail.
  buffer->tail->nextLine = oldHead;
  for(int i=0; i<offset-1; i++){
    oldHead = oldHead->nextLine;
  }
  buffer->tail = oldHead;

  // Correct the line numbers
  struct BufferLine *iter = newHead;
  for(u32 i=0; i < buffer->heightOfBuffer - offset; i++){
    iter->lineNum = i;
    iter = iter->nextLine;
  }
}

void SWBFillWindow(SlidingWindowBuffer buffer, u8 * src){
    // Create a list of CMX DMA transactions to fill the buffers
  struct BufferLine * iter = buffer->head;
  ShDrvCmxDmaTransactionHnd handle;
  ShDrvCmxDmaTransaction taskList[buffer->heightOfBuffer];
  ShDrvCmxDmaCreateTransaction(&handle, &taskList[0], src, iter->line, buffer->
    sizeOfDatatype * buffer->widthOfBuffer);
  for(u32 i=1; i<buffer->heightOfBuffer; i++){
    iter = iter->nextLine;
    // Some cheeky pointer arithmetics here.
    ShDrvCmxDmaAddTransaction(&handle, &taskList[i],
                      &src[i * buffer->sizeOfDatatype * buffer->widthOfBuffer],
                      iter->line, buffer->sizeOfDatatype * buffer->widthOfBuffer);
  }
  ShDrvCmxDmaStartTransfer(&handle);
  ShDrvCmxDmaWaitTransaction(&handle);
}

void SWBSlideAndFillWindow(SlidingWindowBuffer buffer, int offset, u8 * src){
    // Fill the old head and subsequent offset-1 nodes with new data
    struct BufferLine * iter = buffer->head;
    ShDrvCmxDmaTransactionHnd handle;
    ShDrvCmxDmaTransaction taskList[offset];
    ShDrvCmxDmaCreateTransaction(&handle, &taskList[0], src, iter->line,
                                 buffer->sizeOfDatatype * buffer->widthOfBuffer);
    for(int i=1; i<offset; i++){
        iter = iter->nextLine;
        ShDrvCmxDmaAddTransaction(&handle, &taskList[i],
                        &src[i * buffer->sizeOfDatatype * buffer->widthOfBuffer],
                       iter->line, buffer->sizeOfDatatype * buffer->widthOfBuffer);
    }
    ShDrvCmxDmaStartTransfer(&handle);
    ShDrvCmxDmaWaitTransaction(&handle);

    // Slide the Window by the given offset
    SWBSlideWindow(buffer, offset);
}
```

Listing 4.20: SWBuffer Source Code (SlidingBuffer/SlidingBuffer.c)

The comments in the source code are adequate for understanding how the functions operate. However, a few more details need to be provided for the buffers residing inside the nodes. These

buffers will store objects of arbitrary data type. Thus, they are declared to hold unsigned 8-bit data. Since the user provides the size of the datatype being stored, we can identify the stride between consecutive elements of the array. It should also be noted, that no direct assignment is ever performed to any position of the buffer lines. They are exclusively filled with data through DMA transactions, preserving endianness.

Integration into SHAVE code

A few modifications need to be made, so that the Sliding Window Buffers can be used by our application. Initially, the kernel functions need to be adjusted to operate on 2-dimensional arrays. The code below demonstrates how this can be done for bicubic interpolation. The modifications performed on the Lanczos resampling kernel function are similar.

```
void cubicKernelHConv1D ( pixel_t * pixels, intermediate_t * out) {
    out->red =      pixels[0].red * coeff[0]
                +   pixels[1].red * coeff[1]
                +   pixels[2].red * coeff[2]
                +   pixels[3].red * coeff[3];
    /* Omitted for brevity */
}

void cubicKernelVConv1D ( intermediate_t * pixels, pixel_t * out) {
    out->red =  roundf( pixels[0].red * coeff[0]
                    +   pixels[1].red * coeff[1]
                    +   pixels[2].red * coeff[2]
                    +   pixels[3].red * coeff[3]);
    /* Omitted for brevity */
}


void bicubic_kernel( pixel_t ** inputImage, pixel_t * out, int offset){
    for (int i = 0; i < WIDTH / offset; i++){
        intermediate_t arr[4];
        cubicKernelHConv1D(&inputImage[0][i * offset],    &arr[0]);
        cubicKernelHConv1D(&inputImage[1][i * offset],    &arr[1]);
        cubicKernelHConv1D(&inputImage[2][i * offset],    &arr[2]);
        cubicKernelHConv1D(&inputImage[3][i * offset],    &arr[3]);

        cubicKernelVConv1D(arr, &out[i]);
    }
}
```

Listing 4.21: Modified Kernel Functions

The entrypoint function also needs to be adjusted:

```
__attribute__((dllexport)) void Entry( void * theArgsPointer ){
    /* Omitted for brevity */
    retStatus = (int)ShDrvCmxDmaInitialize(NULL);
    assert((retStatus == MYR_DRV_SUCCESS) || (retStatus ==
    MYR_DRV_ALREADY_INITIALIZED));

    SlidingWindowBuffer buffer = CreateSlidingWindowBuffer(shaveId,
                                                           4,
                                                           WIDTH+2,
                                                           sizeof(pixel_t));
    SWBFillWindow(buffer, inputImageDDR);
```

```
11      pixel_t ** matrix;
12      matrix = (pixel_t **) MemMgrAlloc (sizeof(pixel_t *) * 4, shaveID, 32);
13
14      for(int i=0; i < workload / offset; i++){
15          SWBGetAllLines(buffer, matrix);
16          bicubic_kernel(matrix, outputImageCMX, offset);
17          SWBSlideAndFillWindow(buffer, 2,
18                                &inputImageDDR[4 * (WIDTH+2) + i * 2 * (WIDTH+2)]);
19
20          ShDrvCmxDmaCreateTransaction(&out_handle, &out_task, outputImageCMX,
21                                       &outputImage[i * (WIDTH / offset)],
22                                       sizeof(pixel_t) * (WIDTH / offset));
23          ShDrvCmxDmaStartTransfer(&out_handle);
24          ShDrvCmxDmaWaitTransaction(&out_handle);
25      }
26       CleanupSlidingWindowBuffer(buffer);
27       SHAVE_HALT;
28  }
```

Listing 4.22: Modified Entrypoint Function

Finally, the contents of the subdir.mk Makefile need to be modified, so that SlidingBuffer.c is included in the compilation. Thus, the Makefile becomes:

```
1  srcs-shave-y += shave.c SlidingBuffer.c
```

Listing 4.23: Modified Makefile

4.1.7 Single Instruction, Multiple Data

The Intel/Movidius Myriad X provides SIMD utilities, that can be used to further accelerate our application. In this version of the code, it was determined that not utilizing Sliding Window Buffers as well, was in our best interests, since they result in more complex code and do not provide any significant performance gains. The latter reason will be thoroughly discussed in Section 5.1.2.

To enable SIMD processing, vector types need to be used, to pack the data that is to be used in the computation. These vectors have a length of 128 bytes. Thus, they can fit 16 unsigned chars or 4 integers, or 4 floats, or 4 FP16-types (half precision).Therefore, for the 3 different methods:

- **In Bilinear Interpolation**, each colour channel of the considered 2x2 neighbourhood is fitted into a uint4 data type variable.
- **In Bicubic Interpolation**, the kernel is fitted into a float4 variable. Each of the colour channels of the 4 pixels that will be convoluted with the kernel are also stored in a float4 variable.
- **In Lanczos Resampling**, both the coefficients and the colour channels are reduced to half precision, in order to be stored in two half8 variables.

The kernel functions are then modified to include SIMD instructions:

```
#include <moviVectorUtils.h>
void bilinear_kernel ( pixel_t * lines, pixel_t * out, int offset) {
    for( int i = 0; i < WIDTH / offset; i++) {
        uint4 red_val = {lines[i * offset].red,
                         lines[i * offset + 1].red,
                         lines[i * offset + WIDTH].red,
                         lines[i * offset + WIDTH + 1].red};
        /* Omitted for brevity */

        out[i].red      = __builtin_shave_sau_sumx_u32_r(red_val) >> 2;
        /* Omitted for brevity */
    }
}
```

Listing 4.24: SIMD Bilinear Interpolation

```
static float4 coeff = {/* Omitted for brevity */};
void cubicKernelHConv1D ( pixel_t * pixels, intermediate_t * out) {
    float4 red_val = {pixels[-1].red,
                      pixels[ 0].red,
                      pixels[ 1].red,
                      pixels[ 2].red};
    /* Omitted for brevity */

    out->red   = mvuDot(coeff, red_val);
    /* Omitted for brevity */
}

void cubicKernelVConv1D ( intermediate_t * pixels, pixel_t * out) {
    /* Omitted for brevity */
    out->red   = roundf( mvuDot(coeff, red_val) );
    /* Omitted for brevity */
}
```

Listing 4.25: SIMD Bicubic Interpolation

The modifications made to the Lanczos Resampling method are identical to the ones made to Bicubic Interpolation, with the sole difference that half8 data types are used.

It should be noted that both SIMD instructions used, namely __builtin_shave_sau_sumx_u32_r and mvuDot operate on the Scalar Arithmetic Unit (SAU). The Vector Arithmetic Unit (VAU) also supports SIMD operations but is not utilized. Therefore, the kernels can be further accelerated by utilizing both SAU and VAU in parallel. In order to achieve this, these functions would need to be coded in SHAVE Assembly.

4.2 CNN Inference Stage

The Inference Stage is responsible for estimating the pose of the satellite, located somewhere inside the input image. The input provided to this part of the application is the downsampled frame created by the Preprocessing Stage. Thus, the network was trained on RGB images with 640x512 resolution. The output of the network consists of two tensors. Each tensor contains a float value, that is either the encoded location of the satellite in the source image, or the encoded orientation.

Traditionally, in embedded development, one would need to design an Inference Engine for Neural Networks, to successfully implement this stage of our application. This is a rather complicated task, since the Engine must support a plethora of layers and operations, which, additionally, need to be fine-tuned to ensure optimum performance. Thankfully, this is not the case with Myriad X, since the software development kit includes such an engine, in the form of the Movidius Neural Compute Interface (mvNCI).

The mvNCI is a user-friendly API, enabling developers to utilize the Neural Compute Engine, Myriad X's dedicated neural network accelerator. It is build on top of the mvTensor, Myriad X's library that acts as the engine used to make a prediction about an input, using a pre-trained Neural Network model. Individual network operations are performed on either SHAVE or NCE units. A particular unit is chosen depending on the specific operation.

The following figure illustrates the process of running an inference on Myriad X. Initially, the Intermediate Representation of the frozen graph of our trained model is produced via the OpenVINO Model Optimizer. The Myriad Compiler, a tool included in OpenVINO, then uses this IR to generate a blob file that contains the network structure and its weights, alongside with directives for the mvNCI. The mvNCI API offers an abstraction layer for the developers and is used to load the input image and the network on which the image is to be evaluated on. Finally, the output of the inferencing is given in the form of the tensors containing blob data, which consist of the encoded location and orientation of the satelite in the image.

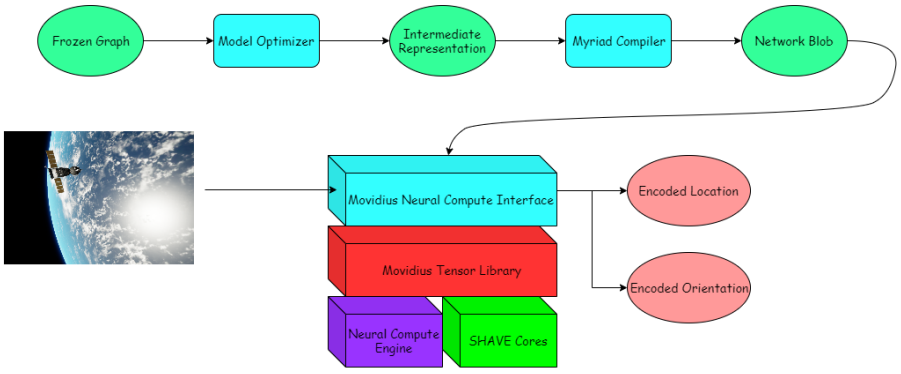


Figure 4.6: Inference Stage Workflow

4.2.1 The mvNCI API

In this section, we will examine the API provided by the built-in MvNCI component. This API is located inside the mvnci.h header file. Therefore, this file needs to be included in any source code that intends to utilize the Inference Engine. The contents of the file itself cannot be displayed here, since it is protected under an NDA agreement. However, an abstract representation of the typical workflow when working with the API is provided in Figure 4.7.

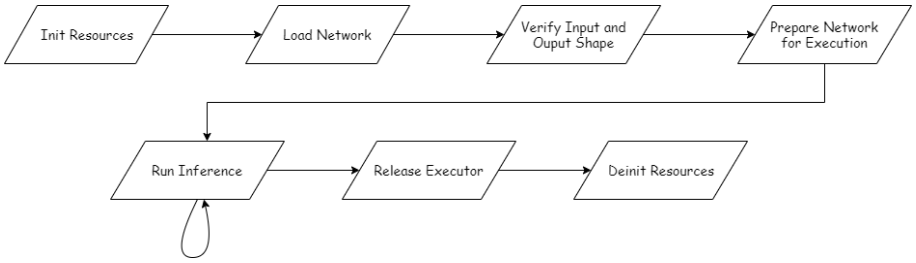


Figure 4.7: Typical Workflow for MvNCI Component

A brief explanation of each stage is provided below:

- **Init Resources:** Initially, the developer is required to acquire the necessary hardware resources of the accelerator.
- **Load Network:** Load the neural network structure from binary blob into the memory.
- **Verify Input and Output Shape:** Get the dimensions of each input and output tensor of the network. Then, verify that these dimensions are the ones expected.
- **Prepare Network for Execution:** Acquire resources for the particular executor and prepare the network to be run with certain processing resources.
- **Run Inference:** Run the tensor data through the neural network and get the output. This function is typically invoked very often and most of the processing time on the accelerator is spent inside it.
- **Release Executor:** Release resources for the particular executor.
- **Deinit Resources:** Release all acquired hardware resources of the accelerator.

The MvNCI Component has access to 2 different kinds of managers: a Resource Manager and a Power Manager.

The Resource Manager is responsible for keeping track of which hardware resources are occupied and which are available. The MvNCI, after all, does offer the option for asynchronous execution of requests. This means that multiple inference requests may be run in parallel. Therefore, a Resource Manager is essential for the correct operation of the Inference Engine, in this scenario. On the other hand, a Resource Manager is needed to determine whether there are enough resources to run even a single instance of a neural network. The Resource Manager is embedded inside the functions provided by the API, and thus, the developer is not needed to perform any additional initialization, or management of this feature.

The Power Management is also internally handled by the API. Unlike the Resource Manager though, the developer may choose the mode of operation of this manager. The available modes of operation are:

- Full Power, where all the initialized resources are powered on.
- Power on/off per Inference, where the resources are powered on when starting to execute a new inference request, and are then powered off after the completion of the inference.
- Power on/off per Layer, where the resources responsible for a particular layer are powered on before its execution and powered off immediately after.
- Power on/off per Layer, only for the Layers executing on SHAVE Cores
- Power on/off per Layer, only for the Layers executing on CNN Units of the Neural Compute Engine.
- A transition state, where the power manager is disabled.

The configuration of the Power Manager needs to be carefully selected, in order for the application developed to meet the constraints set. Powering on the different power isles of the Myriad X chip does come with a significant overhead in terms of latency. Keeping the board at full power and pre-warming the Inference Engine with an initial inference request leads to the optimum execution speed, at the cost of increased power consumption. On the other hand, powering off all resources associated with a layer after its execution, and powering them back on only when needed, provides the lowest power consumption, at the cost of increased latency per inference. Powering off all the resources after the successful completion of an inference request, may prove beneficial, in terms of power consumption, depending on how sparse the requests are, but still comes with an overhead, which does not permit us to reach the optimum performance.

4.2.2 The OpenVINO Inference Engine API

The OpenVINO Toolkit offers an attractive alternative to the MvNCI API, if the developers only aim to accelerate the inference stage of the application on Myriad X. OpenVINO is designed to be compatible with the Intel Neural Compute Stick 2, which features a Myriad X VPU. By using OpenVINO, programmers are able to develop their entire application in Python, C, or C++, and offload the inferencing on the VPU, while the rest of the application runs on the host system, whether it is embedded (for instance, a Raspberry Pi) or not.

In the scope of this thesis, the built-in benchmarkming application, included in the OpenVINO Toolkit, proved to be extremely useful. To demonstrate the C++ API for offloading to Myriad X, several critical parts of the main.c file of this application are included below:

```
// Copyright (C) 2018-2021 Intel Corporation
// SPDX-License-Identifier: Apache-2.0

/* Included libraries omitted for brevity */

using namespace InferenceEngine;

/* Profiling-specific declarations omitted for brevity */

/* Command-Line Parsing specific declarations omitted for brevity */

/**
* @brief The entry point of the benchmark application
*/
int main(int argc, char *argv[]) {
        ExecutableNetwork exeNetwork;

        std::string device_name = FLAGS_d;

        // Parse devices
        auto devices = parseDevices(device_name);

        // Parse nstreams per device
        std::map<std::string, std::string> device_nstreams =
                              parseNStreamsValuePerDevice(devices, FLAGS_nstreams);

        // Create an Inference Engine Core
        Core ie;

        if (!isNetworkCompiled) {
           // 4. Reading the Intermediate Representation network
            /* Omitted for brevity */
            // Read the network from the IR file
            CNNNetwork cnnNetwork = ie.ReadNetwork(FLAGS_m);
            /* Omitted for brevity */

            // Load the network to the device
            exeNetwork = ie.LoadNetwork(cnnNetwork, device_name);

            /* Omitted for brevity */
        } else {
            // Import Network from blob file
            exeNetwork = ie.ImportNetwork(FLAGS_m, device_name, {});

            exeNetwork.GetInputsInfo());
```

```
            if (batchSize == 0) {
                batchSize = 1;
            }
        }

        // Create a Queue of the requests that are to be executed
        InferRequestsQueue inferRequestsQueue(exeNetwork, nireq);
        // Fill the input blobs
        fillBlobs(inputFiles, batchSize, app_inputs_info, inferRequestsQueue.
    requests);

        auto inferRequest = inferRequestsQueue.getIdleRequest();
        // Warm-up of the Inference Engine
        if (FLAGS_api == "sync") {
            inferRequest->infer();
        } else {
            inferRequest->startAsync();
        }
        inferRequestsQueue.waitAll();

        /* Omitted for brevity */

        /** Start inference & calculate performance **/
        /** to align number if iterations to guarantee that last infer requests are
     executed in the same conditions **/

        while ((niter != 0LL && iteration < niter) ||
               (duration_nanoseconds != 0LL && (uint64_t)execTime <
    duration_nanoseconds) ||
               (FLAGS_api == "async" && iteration % nireq != 0)) {

            // Get an idle inference request
            inferRequest = inferRequestsQueue.getIdleRequest();

            if (FLAGS_api == "sync") {
                // Run synchronously
                inferRequest->infer();
            } else {
                // As the inference request is currently idle, the wait() adds no
    additional overhead (and should return immediately).
                // The primary reason for calling the method is exception checking/
    re-throwing.
                inferRequest->wait();
                // Run asynchronously
                inferRequest->startAsync();
            }
            iteration++;

            /* Profiling specific code omitted for brevity */
        }

        // wait the latest inference executions
        inferRequestsQueue.waitAll();

        /* Profiling specific code omitted for brevity */
    return 0;
}
```

Listing 4.26: Main Source Code of Benchmarking App

The process of utilizing the OpenVINO is straightforward, as is evident in the above code segment. An InferenceEngine::Core object needs to be created in order to read, load or import a neural network. If the network to be used is already compiled with the Myriad Compiler, then the method ImportNetwork is to be called. Otherwise, if the network is in IR format, an InferenceEngine::CNNNetwork object needs to be defined. This object is used to store the output of the ReadNetwork method of the Core object. Then, an InferenceEngine::ExecutableNetwork object is used to store the output of the LoadNetwork method of the Core object. The InferRequestQueue object is a queue that contains InferenceEngine::InferRequest objects, that are bound to the given ExecutableNetwork. The fillBlobs function fills the InferenceEngine::Blob objects that are bound to a specific InferRequest with data provided by the user. Finally, the InferRequests start their execution either synchronously or asynchronously.

4.3 Power Management and Measurement

Embedded Systems meant to be operating on the edge are not designed with the sole purpose of optimizing performance. The ability to operate in a small power envelope is integral to such systems, as their main power supply consists, most commonly, of batteries.

As we have already discussed in Section 4.2.1, the mvNCI Component internally utilizes its own Power Manager, providing several different modes of operation to the developers. On the other hand, the custom preprocessing stage, that was designed, possesses no such utility. Regardless of the number of SHAVE Cores used, all power islands associated with the individual SHAVE units are powered on, leading to a completely unnecessary increase in power consumption. In addition to this, the LeonOS is configured to only monitor the performance of the system in terms of latency. Therefore, it was determined, that a system for power management and measurement should be designed, and that the execution flow of the preprocessing stage, as depicted in Figure 4.1, should be slightly modified. The modified execution flow is illustrated in Figure 4.9.

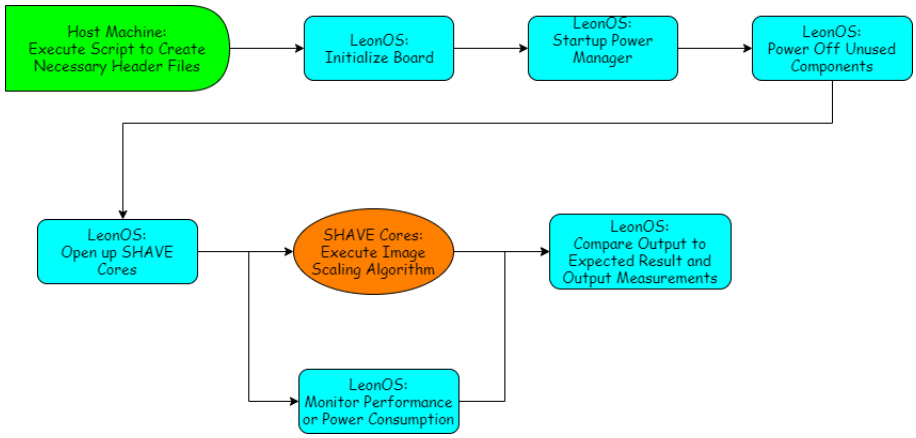


Figure 4.8: Modified Execution for Preprocessing Stage

4.3.1 The PwrManager Component

The Myriad X Development Kit does contain a Power Manager Component. However, this component proved to be unsuitable for our needs. This Power Manager is dedicated to controlling the power state of the entire board. Specifically, Myriad X offers a low-power mode of operation.

This low-power mode of operation is similar to the "sleep" state of a process. When entering this mode of operation, the execution state of all active cores is saved. Additionally, all the clocks, resets, PLL and GPIO configurations are stored, as they will be modified during the transition to the low-power state. The systems ticks of RTEMS are also disabled, in order to stop the interrupts they produce. Subsequently, all power islands associated with the CSS, MSS and UPA Subsystems are powered off. The DDR memory is configured to operate on a self-refresh mode and the DRAM controller enters a sleep state. The system clock is reduced to 32KHz. The system can exit the low-power mode of operation when a hardware interrupt is invoked, either by a switch triggered or by a timer.

It is evident, that depending on the setting where the system is to operate at, entering a low-power state for extended periods of time can prove to be extremely beneficial. However, this component provides none of the utilities that our application demands. The desired power manager should power off only the unused power islands, and not disrupt execution altogether. Furthermore, it is desirable for the manager to have power measurement capabilities, in order for us to deduce the power consumption of the designed system. Therefore, the provided Power Manager is insufficient and we are required to utilize the low level components provided, to design our own, custom Power Management and Measurement system.

4.3.2 The MV0257 Board

The MV0257 Board is a separate daughtercard, provided on the development kit to monitor the current usage of each of the Intel Movidius Myriad X supplies. A number of the Myriad X voltage supplies are also monitored. This enables the user to accurately monitor and measure the power usage and performance of Myriad X when used within their own application.

Figure 4.9: Power Measurement Board MV0257 [6]

The measurement board consists of 7 analog-to-digital converters (ADCs). Each one of them is dedicated to monitoring 4 different rails, thus 28 rails are to be monitored totally. To acquire a measurement, a specific ADC must be configured to start sampling one of its 4 rails. Depending on the accuracy that is desired, as set in the configuration of the ADC, the execution time required for one measurement may vary. The time required for the ADCs to generate a new sample is specified in the following list:

- 12 bits resolution: sample time 5ms.
- 14 bits resolution: sample time 17ms.
- 16 bits resolution: sample time 67ms.
- 18 bits resolution: sample time 267ms.

4.3.3 Proposed API for Power Management & Measurement

In this section, the proposed system for Power Management and Measurement will be analyzed. The main focus of this system was to enable accurate power measurement during the execution of the Preprocessing Stage. During its development, it was determined, that a form of static power management could also be provided. Specifically, the user would declare which resources would be used during the execution of the application, and the system would decide which power islands should be powered off to reduce power consumption. In the scope of this thesis, the static power management applied proved to be sufficient for our needs.

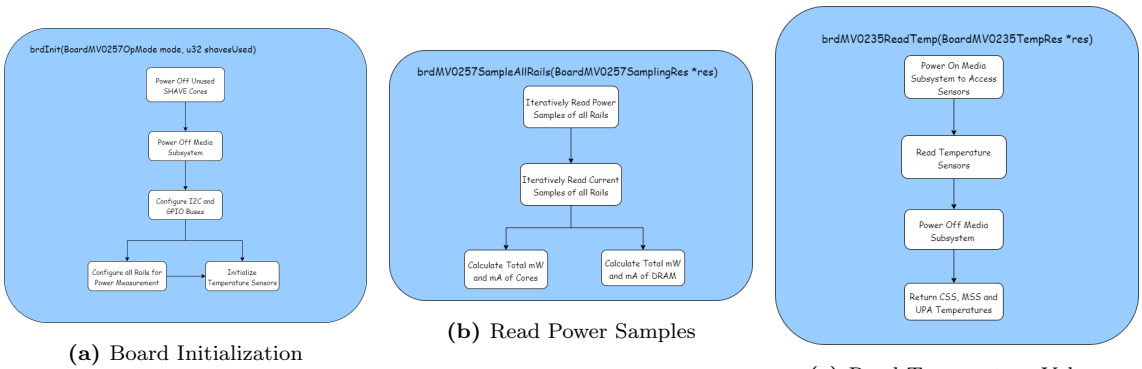
The API of the proposed Power Management and Measurement system is located in the PowerMeasurement.h header file. The contents of this file are listed below.

```
#ifndef POWER_MEASUREMENT_H
#define POWER_MEASUREMENT_H

/* Includes omitted for brevity */

// Temp sensors names
#define TSENS_CSS_NAME "/dev/tempsensor0"
#define TSENS_MSS_NAME "/dev/tempsensor1"
#define TSENS_UPA0_NAME "/dev/tempsensor2"
#define TSENS_UPA1_NAME "/dev/tempsensor3"

#define DDR_DATA __attribute__((section(".ddr.data")))

#define TEMP_SENSORS (4)

#define RAIL_MAX (28)

#define ADC_MAX (7)

#define START_RAIL (0)

typedef enum {
  READ_TEMP,  // Read Temperature Mode
  READ_PWR,   // Read Power Mode
  READ_ALL    // Read Both
}BoardMV0257OpMode;

typedef struct {
  float ddrPwr;   //mW consumed by DDR
  float ddrAmps;  //mA measured for DDR
  float corePwr;  //mW consumed by cores
  float coreAmps; //mA measured for cores
}BoardMV0257SamplingRes;

typedef struct {
  float cssTemp;  // Temperature for CPU Subsystem
  float mssTemp;  // Temperature for Media Subsystem
  float upaTemp0; // Temperature for Microprocessor Array
  float upaTemp1; // Temperature for Microprocessor Array
}BoardMV0235TempRes;

// This function performs all necessary initializations.
// It powers off any unused power islands,
// and sets all required configurations.
//
```

```
// @param[in] mode: The mode of operation of the system.
// @param[in] shavesUsed: The number of SHAVE processors to be utilized.
//
// On success, returns RTEMS_SUCCESSFUL
rtems_status_code brdInit(BoardMV0257OpMode mode, u32 shavesUsed);

// This function performs all necessary deinitializations.
//
// On success, returns RTEMS_SUCCESSFUL
rtems_status_code brdUnInit(void);

// This function samples all rails and computes
// the power consumption and current flow values
// for the cores and the DDR memory.
//
// @param[out] res: Pointer to the output struct, that contains all measurements.
//
// On success, returns RTEMS_SUCCESSFUL
rtems_status_code brdMV0257SampleAllRails(BoardMV0257SamplingRes * res);

// This function reads all temperature sensors and
// returns their values.
//
// @param[out] res: Pointer to the output struct, that contains all measurements.
//
// On success, returns RTEMS_SUCCESSFUL
rtems_status_code brdMV0235ReadTemp(BoardMV0235TempRes * res );

#endif // POWER_MEASUREMENT_H
```

Listing 4.27: PowerMeasurement Header File

The functions provided, internally perform multiple operations, that would otherwise need to be manually integrated to the application by the user. These functions require knowledge of the underlying hardware, and thus may require significant time to be fully-understood by a developer.



(a) Board Initialization

(b) Read Power Samples

(c) Read Temperature Values

Figure 4.10: Power Management and Measurement API Internal Operations

The brdInit function initially disables all unused SHAVE Processors. The SHAVE Cores that are still powered on after this operation are the ones with an ID less than the shavesUsed parameter provided. Then, the Media Subsystem is powered off. This includes the LeonRT processor, the ISP and CV hardware filters and the Neural Compute Engine. The Board Power Measurement Driver is initialized immediately after, and all the necessary GPIO and I2C buses are configured with the

default configuration. An additional custom configuration is provided for the I2C0 bus, that is set to transfer the power measurement data from the MV0257 board. This causes the creation of 7 file descriptors, one for each ADC. Finally, depending on the mode of operation, the rails and/or the temperature sensors are configured. The rails are configured to continuous operation with 12-bit resolution and gain equal to 1. The temperature sensors are configured to continuous operation as well.

The brdMV0257SampleAllRails function iteratively reads a power sample of each rail. To achieve this, the ADCs are first instructed to start sampling the desired rail. After 5ms, the power sample produced is ready to be read. Rails associated with different ADCs can be set to start sampling in-parallel. Thus, at least 4 * 5 = 20ms are demanded to successfully calculate the total power consumption of the Intel Movidius Myriad X. The operation is identical for reading current samples. Therefore, the total execution time of this function is approximately 40ms.

The brdMV0235ReadTemp initially powers back on the CPU and ISP power islands of the Media Subsystem. This particular operation creates an overhead of 100ms. Then, the files associated with the 4 temperature sensors are read to acquire the desired values. The power islands are powered back off and the results are returned.

The brdUnInit function simply uninitializes the Board Power Measurement Driver.

4.3.4 Modifications to the RTEMS Configuration

When using this power management and measurement system, several devices are utilized and file descriptors are created. Naturally, this means that additional resources need to be specified to the RTEMS configuration, for our system to be compatible with the underlying Operating System. Therefore, a few of the RTEMS directives declared in section 4.1.2 need to be modified. These directives are listed below.

```
1 #define CONFIGURE_MAXIMUM_DEVICES              18
2 #define CONFIGURE_MAXIMUM_FILE_DESCRIPTORS     20
```

Listing 4.28: Modifications to RTEMS Configuration

At least 11 new devices are to be used by the RTEMS (7 ADCs and 4 Temperature Sensors). Therefore, the maximum amount of devices is reconfigured from 6 to 18.
Each ADC and each temperature sensor requires a unique file descriptor. This means that up to 11 file descriptors will be needed for the power management and measurement system alone. Therefore, the maximum number of file descriptors is set to be 20 to satisfy this need.

4.4 Application Scenario: Pose Estimation & Tracking on VPU

The preprocessing algorithms, as well as the CNN inferencing for pose estimation, can be combined with a Pose Tracking System, to create a low-power, reliable solution to the "Lost in Space" problem in real-time. The Pose Tracking algorithm is adopted from previous work, namely [insert ref here]. The main limitation of this algorithm is that it attempts to evolve an initial pose, which is considered to be known in advance. UrsoNet can provide this initial pose of the satellite, extracted from the initial frame. Therefore, the Preprocessing and CNN Inference Stage are to be run only once, to acquire the initial 6-DoF pose. Then, the Pose Tracking algorithm is executed. Figure 4.11 illustrates the underlying architecture of the proposed system.

Since the two systems (pose estimation & pose tracking) were developed separately, certain modifications need to be made in order for them to be compatible to one another. UrsoNet operates on RGB images, while the Pose Tracking System accepts grayscale images. Thus, the camera must provide RGB images, that will be preprocessed to 8-bit grayscale pixels. This preprocessing is not compute-intensive and can be integrated in the Intensity Edge Detection Stage of the Pose Tracking algorithm. Furthermore, the Pose Tracking algorithm is set to operate on 1024x1024 resolution images. This means that UrsoNet needs to be retrained on 512x512 resolution images. This option is supported by setting a resizing factor of 0.4 and enabling the squaring of the training images. However, it is uncertain whether the accuracy achieved by training the network in such a manner will be sufficient. Therefore, a better option would be to create a new dataset that meets our needs, possibly by using URSO to generate images. Finally, the location and orientation encoding option of UrsoNet must be disabled, in order for the results of the network to be in 6-DoF format.

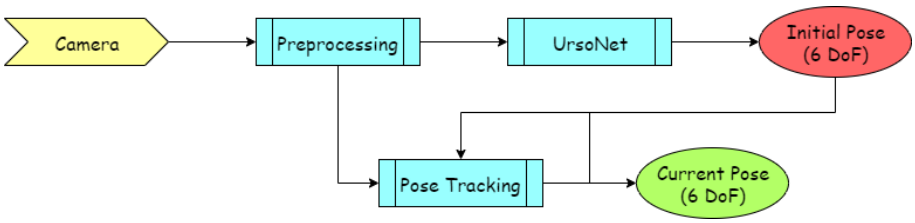


Figure 4.11: Proposed System for Pose Tracking of Satellite

Chapter 5

Experimental Evaluation

This section is dedicated to the evaluation of the Preprocessing and Inference Stage of the proposed system.

5.1 Evaluation of Preprocessing Stage

5.1.1 Experimental Setup

The implementation of the different downsampling algorithms was evaluated using the MV0235 board, which contains an ma2485 Myriad X chip. The platform was connected to the host computer with an ARM-USB-TINY-H JTAG Dongle. This interface was used to invoke the Movidius Debugger and offload our application on the board. Therefore, the results of each execution were presented on the standard output of the host computer's terminal.

To acquire measurements, the Free Running Counters and the custom Power Management and Measurement system were utilized. The former were read before and immediately after the critical sections of the code executed, that is SHAVE Core initialization and execution, as well as the padding of the images, if that was performed separately on LeonOS. The latter was used after the startup of the SHAVE Cores, to measure the power consumption of the board during their operation. In order to avoid measuring the power consumption of idle cores, when the downsampling algorithm finishes its execution before all the necessary rails are sampled, we re-executed the algorithms on the cores, to keep them operating throughout the measurement.

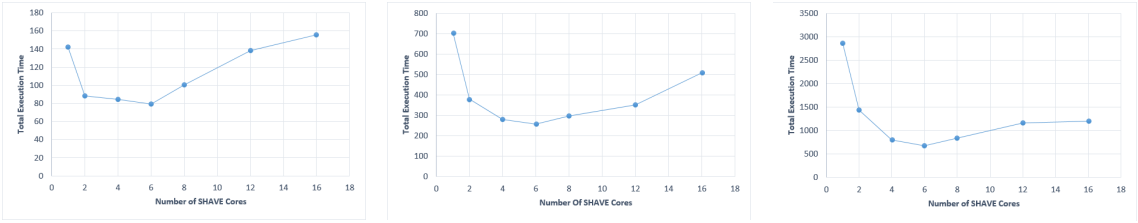
The Preprocessing Stage was evaluated on a 1280x960 resolution image from the "Soyuz Hard" dataset. The image chosen, namely 134_rgb, was resized to a 640x480 resolution.

5.1.2 Latency Results

Each implementation of the downsampling methods was initially ported to a single LEON/SPARC Core (LeonOS) and subsequently ran on a single SHAVE Core. As discussed in Section 4.1.3, porting the algorithms to the SHAVE Cores does significantly speed up our application, as is evident in Table 5.1 and Figure 5.1, but does not fulfil our need for real-time processing.

Table 5.1: Initial Porting on Intel/Movidius Myriad X

Downsampling Method	Execution Time (LEON/SPARC)	Execution Time (SHAVE)	Speedup
Bilinear Interpolation	468.2 ms	141.9 ms	3.3x
Bicubic Interpolation	2112 ms	702.4 ms	3x
Lanczos Resampling	7925.3 ms	2859 ms	2.8x

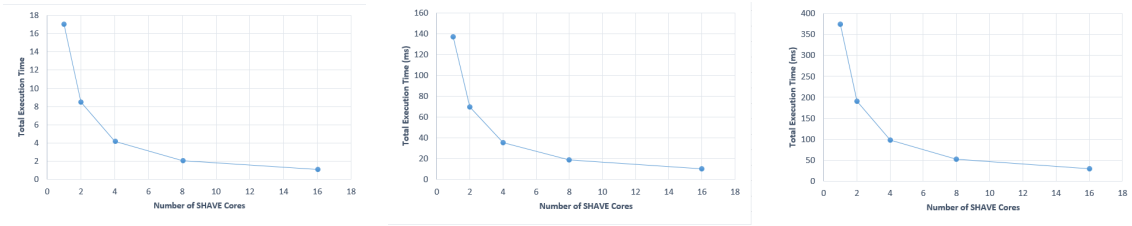


(a) Bilinear Interpolation **(b)** Bicubic Interpolation **(c)** Lanczos Resampling

Figure 5.1: Execution Time Scaling on SHAVE Cores

When executing the algorithms on multiple cores, an interesting property is revealed. All the methods show a performance gain when executing on up to 6 cores. When eight or more cores are utilized, a degradation in performance appears. For instance, executing the bilinear interpolation resampling method on 16 cores, proves to be more time-costly than executing on a single core. This is due to the fact that the main LPDDR Memory is common. This memory possesses a specific number of read and write ports. When multiple cores attempt to access the memory, only a subset of them is granted permission to do so. The rest of them stall their execution for several clock cycles and then retry to acquire access to the ports. As long as they fail to secure access priviliges, they stall for a set amount of clock cycles. Therefore, when the number of cores utilized is greater than the number of the available cores, this race condition occurs, which is only worsened by increasing the number of cores executing code.

On the other hand, all SHAVE Cores have access to their preferential CMX slice through their private, unique ports. Therefore, having the SHAVE Cores read and write to their preferential CMX slice eliminates all stalls in execution, caused by inability to access memory resources. Since the algorithms are embarrassingly parallel, and no race conditions exist between the processors, one would expect to observe linear scaling in the speedup of the parallelized methods. This is not the case for bicubic interpolation and Lanczos resampling, as illustrated in Figure 5.3. These methods differ from bilinear interpolation, in that they require the input frame to be padded. Since the padding was performed in the LEON/SPARC Core, an additional overhead of 2.1 and 6.2 ms respectively is added to the total execution time. By assigning the padding operation to the individual SHAVE Cores, we managed to eliminate this overhead, as no increase in SHAVE execution time was observed.



(a) Bilinear Interpolation **(b)** Bicubic Interpolation **(c)** Lanczos Resampling

Figure 5.2: Execution Time Scaling on SHAVE Cores with CMX enabled

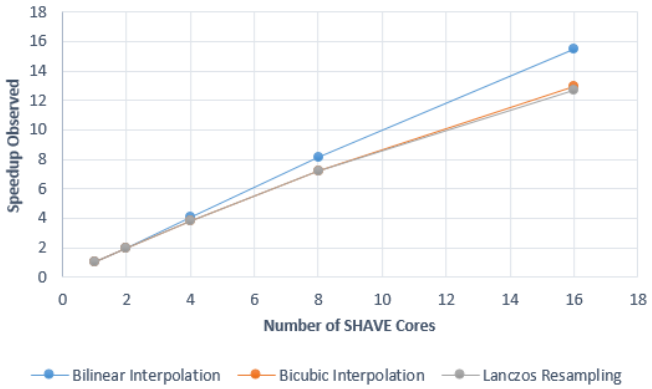
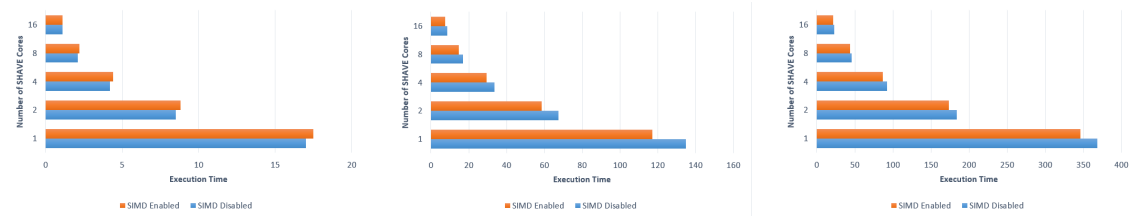


Figure 5.3: Speedup Observed during Parallelization

Further acceleration of the Preprocessing Stage was achieved by using the SIMD Utilities of Myriad X. Figure 5.4 compares the latency of the two different versions of code. Unsurprisingly, for Bicubic Interpolation and Lanczos Resampling, exploiting the SIMD capabilities of the SHAVE Cores, results in at least 5% speedup of the execution time required to perform the operation. In fact, even up to 15% acceleration was observed. On the other hand, Bilinear Interpolation performs worse when SIMD is enabled. This is due to the fact, that in each iteration of the algorithm, 32 total bytes of data participate in the computations. SIMD Instructions, however, operate on vector registers of 128-bytes length, which means that 96 unnecessary bytes will also be a part of the computation. Thus, enabling SIMD execution in this case is redundant, since the additional overhead outscales any performance gains.



(a) Bilinear Interpolation **(b)** Bicubic Interpolation **(c)** Lanczos Resampling

Figure 5.4: Execution Time Scaling on SHAVE Cores depending on whether SIMD is enabled/disabled. CMX Memory transfers are enabled and the padding operation is offloaded to the SHAVEs.

By offloading the padding operation, utilizing the CMX Scratchpad Memory and (conditionally) using the SIMD utilities of the SHAVE Cores, we achieved the best results in terms of latency.

These results are illustrated in Table 5.2 along with the correspondent speedup, compared to the initial porting of the algorithms on the LEON/SPARC processor. The execution times achieved satisfy our needs, since they not only support real-time operation (909, 135 and 46 frames per second respectively), but they are also at least an order of magnitude lower than the execution time of the chosen CNN.

Table 5.2: Final Results on Intel/Movidius Myriad X

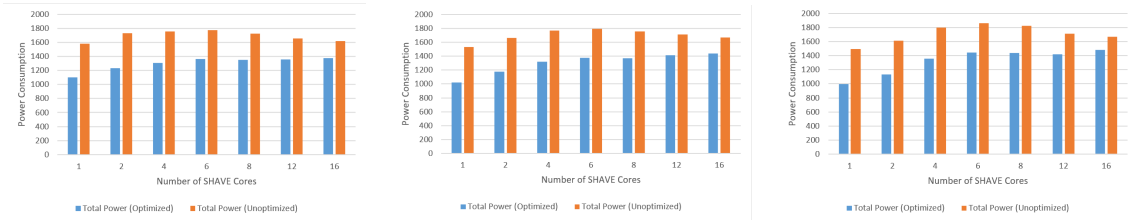
Downsampling Method	**Execution Time (LEON/SPARC)**	**Execution Time (SHAVEs)**	**Speedup**
Bilinear Interpolation	468.2 ms	1.1 ms	425.6x
Bicubic Interpolation	2112 ms	7.4 ms	285.4x
Lanczos Resampling	7925.3 ms	21.8 ms	363.5x

Contrary to the optimizations discussed above, the utilization of Sliding Window Buffers was not beneficial. By integrating this data structures to our implementations of the algorithms, the resulting execution times were increased to **9.8 ms** for 16-core execution of the bicubic interpolation algorithm, and **28.3 ms** for 16-core execution of the Lanczos resampling algorithm. The root of this degradation in performance is the use of the dynamic Memory Manager itself. This data structure was created by allocating memory during runtime, which invokes the dynamic memory management system of Myriad X. However, the data structures utilized are essentially static, since their size is known during compilation time. Therefore, by using the Memory Manager, an additional overhead is created, that is, in fact, completely unnecessary. For instance, cleaning up the the data structure after execution take an astounding 1.0 ms for 16-core execution of the bicubic interpolation algorithm, since the Memory Manager tries to defragmentize the memory. Thus, instead of using the Memory Manager, a simple 2-dimensional array should be statically allocated, and a function that correctly shifts the pointers stored in its first dimension should replace the overly complicated, proposed structure.

This modification of the underlying architecture of the Sliding Windows Buffers was never implemented though. The reason for this was that we observed that the CMX DMA Engine was highly optimized and performed data transfers between the memories with such low latency, that hardly 0.1 ms in execution time was to be gained by integrating this data structure.

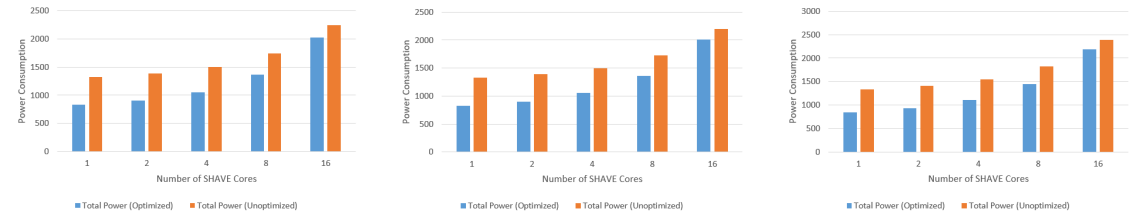
5.1.3 Power Consumption Results

The power consumption of the preprocessing stage was measured using the system proposed in Section 4.3. Two sets of measurements were taken. The first set was acquired by measuring the power consumption during normal initialization and execution. The other set was acquired in a similar manner, with the difference that the custom Power Manager was utilized. Figures 5.5 and 5.6 show the scaling of the power consumption of the Preprocessing Stage, for the initial porting of the algorithms, which has the SHAVEs reading directly from the DDR, and for the final implementation, where each SHAVE utilizes its preferential CMX slice and pads the input lines read. It is observed, that by using the custom Power Manager, up to **30%** of the power consumed can be saved.



(a) Bilinear Interpolation **(b)** Bicubic Interpolation **(c)** Lanczos Resampling

Figure 5.5: Power Consumption of Initial Porting of the Algorithms on the SHAVE Cores



(a) Bilinear Interpolation **(b)** Bicubic Interpolation **(c)** Lanczos Resampling

Figure 5.6: Power Consumption of Final Implementation of the Algorithms on the SHAVE Cores

Table 5.3 contains a more detailed analysis of the power consumption, during the execution of the initial porting of the algorithms and with the Power Manager operating. Since unused cores are powered off by the manager, when increasing the amount of cores utilized, the core power rises. On the other hand, the power consumed by the DDR memory is increased only when executing in up to 6 cores, and is decreased afterwards. As we discussed in the previous section, these algorithms only scale up to 6 cores, since the read/write ports of the DDR memory are limited. When more than 6 cores are used, SHAVEs that are not granted access to memory resources stall their execution. This leads to time periods, where several of the ports of the DDR Memory remain idle. Therefore, we observe this decrease in power consumption.

In a similar manner, Table 5.4 contains an analysis of the power consumption of the final implementation of the algorithms. Compared to the initial porting, a significantly lower DDR power consumption is observed during execution, as all read and write operations are taken over by the CMX Scratchpad Memory, and only the refreshing and memory transfer operations are executed. An equally significant increase in the power consumed by the computational units, however, is also observed. These measurements include the power consumed by the -now active- CMX memory, and may even reach 2 Watts.

Table 5.3: Power Consumption Analysis for Initial Porting

Downsampling Method	Number of Cores	Core Power (mW)	DDR Power (mW)
Bilinear Interpolation	1	822.34	276.2
	2	863.34	370.55
	4	908.88	396.75
	6	936.14	427.35
	8	942.36	407.7
	12	1026.48	330.5
	16	1061.01	314.4
Bicubic Interpolation	1	805.81	218.3
	2	865.81	307.9
	4	936.08	383.3
	6	973.81	401.15
	8	994.14	378.15
	12	1047.54	367.4
	16	1101.74	336.1
Lanczos Resampling	1	804.81	190
	2	864.21	268.7
	4	965.76	388.9
	6	1029.09	417.1
	8	1031.68	405.2
	12	1054.14	368.3
	16	1109.14	369.55

Table 5.4: Power Consumption Analysis for Final Implementation

Downsampling Method	Number of Cores	Core Power (mW)	DDR Power (mW)
Bilinear Interpolation	1	776.41	51,1
	2	850.87	54,4
	4	990.14	61,55
	8	1294.74	75,3
	16	1923.01	103,25
Bicubic Interpolation	1	778.47	51.1
	2	848.41	54.95
	4	992.16	61.55
	8	1281.81	75.75
	16	1907.41	102.25
Lanczos Resampling	1	789.14	51.65
	2	874.62	55.5
	4	1038.14	63.2
	8	1372.47	79.05
	16	2072.69	111.05

5.2 Evaluation of CNN Inference Stage

5.2.1 Experimental Setup

During development, it was determined that a bug in the mvNCI Component prevented us from utilizing the Neural Compute Engine, and the built-in Inference Engine of Myriad X on the MV0235 Board. To overcome this obstacle, the Intel Neural Compute Stick 2 (NSC2) was used. This stick hosts an ma2480 Myriad X chip and is programmed through the OpenVINO API for Neural Network Inferencing. Thus, the NCS2 was connected to a USB port of the host machine. The benchmark app provided by the OpenVINO Toolkit was utilized to gather measurements. Unfortunately, this tool does not provide power consumption analytics.

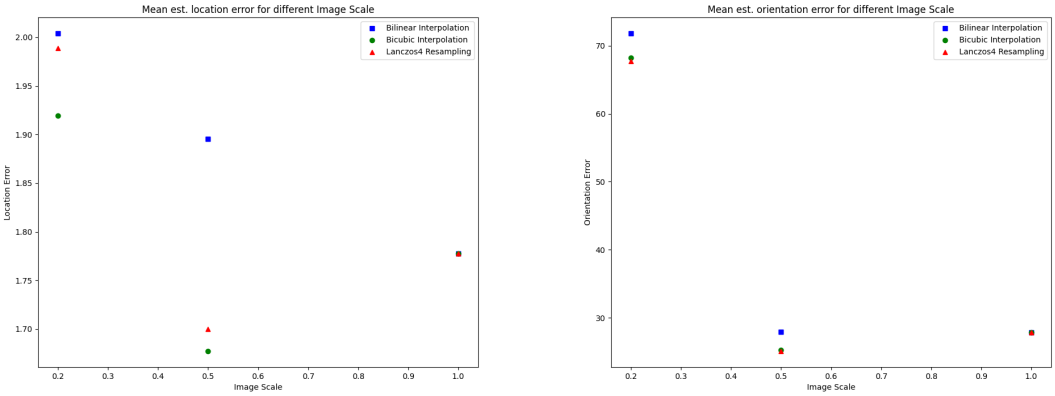
The Inference Stage was evaluated, in terms of latency, on an image from the "Soyuz Hard" dataset, namely 0_rgb, which was resized to a 640x512 resolution. On the other hand, the Neural Network itself was evaluated on a subset of the "Soyuz Hard" dataset, consisting of 10% of its total images, to determine the accuracy achieved.

During training, the training set (80% of total images) and the validation set (10% of total images) were resized to a resolution of 640x512, using one of the downsampling algorithms examined, and zero constant padding. For each one of the downsampling methods, a separate instance of a trained UrsoNet model was created. The initial weights of the ResNet backbone of all these instances were ImageNet pre-trained weights. The training of the models itself, as well as their evaluation in terms of accuracy, was performed on an NVIDIA Tesla V100 Volta GPU. Each model was trained for 100 epochs.

5.2.2 Network Accuracy Results

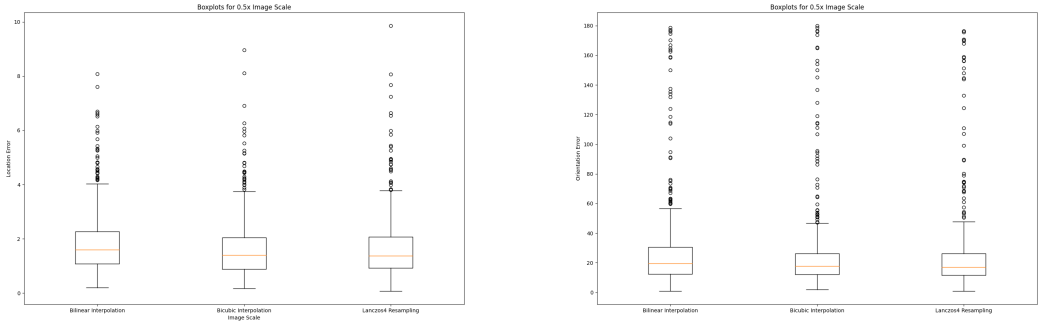
The accuracy of the trained models was evaluated with respect to two hyperparameters, namely resampling method and image scaling factor. The latter was chosen to be 1x (initial network), 0.5x (4x reduced input size) and 0.2x (25x reduced input size). The resampling methods chosen were Bilinear Interpolation, Bicubic Interpolation and Lanczos4 Resampling. The mean location and orientation errors of the estimated 6-DoF pose compared to the ground truth pose is illustrated in Figure 5.7. Surprisingly, the minimum mean orientation error and second lowest mean location error is observed for an image scale different than 0.5, and by utilizing the Lanczos4 algorithm. This particular model scores the highest for the Soyuz dataset. The model trained under Bicubic Interpolation resizing for 0.5x image scaling is an attractive alternative since it achieves a comparable score to the Lanczos4/0.5x model.

The box plots for these models, illustrated in Figure 5.8, provide some precious insight into the distribution of the errors observed. The two models have almost identical median, upper and lower quartile values, and interquartile range. On the other hand, even though the Bicubic/0.5x model has a slightly worse score, it also appears to have slightly less outlier values compared to Lanczos4/0.5x. Given that the Bilinear Interpolation algorithm exceeds both Bicubic and Lanczos methods significantly in terms of latency overhead added to the final system, the Bilinear/0.5x model is also an excellent candidate for our final system. All things considered, any of these three models provides a CNN, whose accuracy is equivalent to that of the original UrsoNet model, but unlike that model, it supports real-time operation on the Intel/Movidius Myriad X chip.



(a) **(b)**

Figure 5.7: Mean Est. Location and Orientation Error



(a) **(b)**

Figure 5.8: Box Plots of Location and Orientation Error for 0.5x Scaling

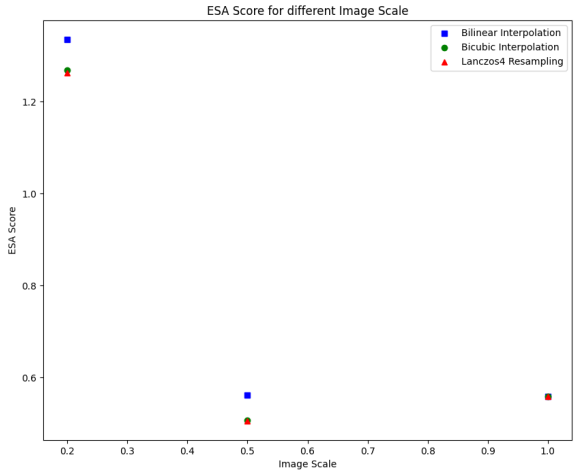


Figure 5.9: ESA Scores achieved for different Models

5.2.3 Latency Results

The average delay between feeding the input tensor, which contains the image, to the network and it producing a result is presented in Table 5.3. The original network, operating on 1280x960 resolution, RGB images, demands an astounding **2297.62 ms** for a single inference, or just 0.44 FPS! Thus, it is evident, that the Preprocessing Stage is absolutely fundamental, to enable us to work with smaller, faster networks.

In Section 5.2.1, we discussed how working with images resized to half the original dimensions, produces models with the same or even slightly better accuracy than the original CNN for this particular dataset. In terms of latency, working with such resized images, results in speeding up the inference process by 5 times. This speedup is crucial, as it delivers a network that can operate on 2.23 frames per second. For this particular application, this performance can be considered real-time, since the pose of the satellite changes in a slow pace.

We experimented with an increased scaling factor, namely resizing the input image to one fifth of the original dimensions. This led to a mere **67.40 ms** for a single inference, which amounts to 14.84 frames per second. Thus, by reducing the size of the input image by 25 times, we achieve 33.7 times better latency during inferencing. However, it should be noted that this comes at a price of reduced accuracy as well.

Table 5.5: Latency Results for Synchronous Inference on Intel NCS2

Input Tensor Shape	**Latency (ms)**	**Frames Per Second**
1x192x256x3	67.40	14.84
1x512x640x3	449.26	2.23
1x960x1280x3	2297.62	0.44

5.3 System Evaluation

5.3.1 Preprocessing and Inferencing

The proposed system for satellite pose estimation is designed to accept 1280x960 resolution, RGB images, resize them to 640x512 and then infer an estimated pose on the resized image, using UrsoNet. Each inference demands **449.26 ms**. Thus, depending on the resampling method chosen for the preprocessing, the proposed system operates with a **450.36 - 472.26 ms** latency for a single output, or **2.12 - 2.22 FPS**.

The power consumption of the proposed system cannot be calculated, since there are no data for the power consumption of the CNN Inference Stage. However, it should be noted that 7 SHAVE Cores and the CNN Hardware Accelerator (Neural Compute Engine) are utilized, in order to run the network. Furthermore, this accelerator uses the CMX Scratchpad Memory. Therefore, the expected power consumption should be slightly higher than the one observed for the final implementation, when executed on 8 SHAVE Cores, with the Power Manager not being utilized.

5.3.2 Application Scenario: Pose Estimation & Tracking

As discussed in Section 4.4, the Preprocessing and CNN Inference Stages need to be modified to be compatible with the Pose Tracking algorithm. The input to the Preprocessing Stage is going to be a 1024x1024 RGB image. Table 5.6 illustrates how our final implementations of the three downsampling methods scale for such an input size. Obviously, the execution time required is reduced in any case, since fewer output pixels are produced.

This is also the case for the Inference Stage. The latency for a single inference of a 512x512 input frame was measured to be **372.70 ms**. The Inference Stage is approximately **17%** faster, than the one proposed in the previous section. This is fairly significant, since the total latency of the system that estimates the initial pose of the satellite is, depending on the downsampling method, **373.70 - 391.3 ms**, or **2.56 - 2.66 FPS**, which is comparable to the latency of the Pose Tracking System.

Table 5.6: Latency of Image Scaling Algorithms

Downsampling Method	Number of Cores	Execution Time (ms)
Bilinear Interpolation	1	15.0
	2	7.5
	4	3.8
	8	1.9
	16	1.0
Bicubic Interpolation	1	100.3
	2	50.1
	4	25.1
	8	12.5
	16	6.3
Lanczos Resampling	1	295.5
	2	147.8
	4	73.9
	8	37.0
	16	18.6

The following table illustrates the execution time required for each stage of the CV algorithm, which is employed to perform pose tracking. Intensity edge detection is not added in the total time, because it is masked by pose refinement. The same applies for the image reception via the Camera Interface (CIF). This porting of the CV Algorithm on Myriad2 achieves **263 - 388 ms** latency between successive frames. It should be noted that MyriadX contains four additional SHAVE cores, as well as a larger Scratchpad Memory. Therefore, further speedup of this execution time can be reached, when porting the CV Algorithm to MyriadX.

Table 5.7: Latency of CV Functions of Pose Tracking System

CV Functions	**Execution Time (ms)**
I. Edge Detection	36-37
D. Edge Detection	39-40
Depth Rendering	119-212
Edge Matching	5-6
Pose Refinement	100-130
CV Algorithm	**263-388**

Chapter 6

Epilogue

6.1 Conclusion

In this thesis, we proposed a system that can solve the "Lost in Space" problem and efficiently accelerated it on the Intel/Movidius Myriad X VPU. The solution was based on a convolutional, residual neural network, "UrsoNet", which was trained to perform pose estimation of the Soyuz spacecraft. The initial pose estimated by the network would then be evolved and refined by a traditional CV algorithm, which had already been ported to the Myriad 2 VPU.

It was quickly observed, however, that, as is the case with most of these types of networks, that "UrsoNet" could not be executed in real-time due to the size of the input images. To this end, we downscaled the input images to half of their original resolution, experimenting with three different algorithms of increasing computational intensity: bilinear interpolation, bicubic interpolation and Lanczos resampling. This allowed for real-time operation as the execution time of a single inference was accelerated from 2297.62 to 449.26 ms, excluding the latency of the downsampling. The execution time of the downsampling algorithms is at least an order of magnitude lower than that of the network, with the most intensive algorithm needing a mere 21.8 ms for execution. To achieve this, we utilized all sixteen available SHAVE cores, offloaded the padding operation to them, utilized their preferential slices on the Scratchpad Memory, and exploited the SIMD utilities of these cores. The power consumption of the system was monitored through a custom power measurement system, which also performed static power management, reducing the total power consumption by up to 30%.

To successfully load and run the CNN on the VPU we utilized the Inference Engine provided by the OpenVINO toolkit. We experimented both with the offered API provided by OpenVINO and with the mvNCI component included in the Myriad X development kit, to evaluate the network and integrate it into our application. Finally, we provided clear directions, on how the two discrete systems, the pose estimating CNN and the CV tracking algorithm, need to be modified in order to be compatible with one another.

6.2 Future Work

Even though this thesis has, naturally, come to an end, the author holds the belief that there is still room left for optimizing and fine-tuning the proposed system:

- In the Preprocessing Stage, the algorithms could employ double buffering techniques, to mask the latency of the DMA transactions behind the execution of downsampling operations. Furthermore, it was observed that the API for SIMD instructions was built onto the Scalar Arithmetic Unit of the VPU. This means that only additions were truly performed in SIMD. Thus, assembly code can be written to fully utilize the capabilities of the SHAVE cores. A suggestion by the author would be to experiment with loop unrolling and utilizing both the SAU and VAU in-parallel.

- UrsoNet was configured to have a ResNet-50 as its backbone. It was determined that, for a small deterioration in accuracy, we could reach an astounding 67.4 ms latency for a single inference. The accuracy to performance ratio should be studied for lighter backbones, such a shallower ResNet of 34 layers or a MobileNet. Moreover, the network should be trained for more than 100 epochs, to reach its full potential. It should be noted, that in the original paper, UrsoNet was trained for 500 epochs.

- Since URSO is not open-source, we could experiment with creating our own simulator on the Unreal Engine 4, to generate more synthetic datasets of uncooperative spacecrafts on orbit around the Earth. This is crucial, as our network needs to be trained with square images, in order to be compatible with the tracking algorithm.

- The tracking algorithm needs to be ported to Myriad X. Myriad X is largely backwards compatible, but several differences in the development kits of the 2 platforms exist. Myriad X can further accelerate this application, since it offers an additional 4 SHAVE Cores, increased CMX memory space, and a higher clock frequency.

Bibliography

[1] Z. Meng, Y. Hu, and C. Ancey, "Using a data driven approach to predict waves generated by gravity driven mass flows," vol. 12, no. 2, p. 600, Feb. 2020. [Online]. Available: https://doi.org/10.3390/w12020600

[2] Y. LeCun and C. Cortes, "MNIST handwritten digit database," 2010. [Online]. Available: http://yann.lecun.com/exdb/mnist/

[3] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," *CoRR*, vol. abs/1512.03385, 2015. [Online]. Available: http://arxiv.org/abs/1512.03385

[4] P. F. Proenca and Y. Gao, "Deep learning for spacecraft pose estimation from photorealistic rendering," *arXiv preprint arXiv:1907.04298*, 2019.

[5] E. Petrongonas, V. Leon, G. Lentaris, and D. Soudris, "ParalOS: A Scheduling & Memory Management Framework for Heterogeneous VPUs," in *2021 24th Euromicro Conference on Digital System Design (DSD)*, 2021, pp. 221–228.

[6] Intel/Movidius Ltd, "Myriad X Development Kit: A Programmer's Guide."

[7] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng, "TensorFlow: Large-scale machine learning on heterogeneous systems," 2015, software available from tensorflow.org. [Online]. Available: http://tensorflow.org/

[8] Intel Ltd. (2021) OpenVINO Toolkit Overview. [Online]. Available: https://docs.openvino.ai/2021.2/index.html

[9] ——. (2021) OpenVINO Toolkit Documentation. [Online]. Available: https://docs.openvino.ai/latest/documentation.html

[10] Wikipedia, "Aliasing — Wikipedia, the free encyclopedia," http://en.wikipedia.org/w/index.php?title=Aliasing&oldid=1052296249, 2021.

[11] M. B. Khambete and M. A. Joshi, "Blur and ringing artifact measurement in image compression using wavelet transform," *World Academy of Science, Engineering and Technology, International Journal of Computer, Electrical, Automation, Control and Information Engineering*, vol. 1, pp. 341–344, 2007.

[12] S. Haykin, *Neural Networks and Learning Systems.* Pearson, 2009.

[13] S. Kotsiantis, "Supervised machine learning: A review of classification techniques," vol. 31, pp. 249–268, Jan. 2007.

[14] M. Alloghani, D. Al-Jumeily, J. Mustafina, A. Hussain, and A. J. Aljaaf, "A systematic review on supervised and unsupervised machine learning algorithms for data science," in *Unsupervised and Semi-Supervised Learning*. Cham: Springer International Publishing, 2020, pp. 3–21.

[15] A. Anwar, "A beginner's guide to regression analysis in machine learning." [Online]. Available: https://towardsdatascience.com/a-beginners-guide-to-regression-analysis-in-machine-learning-8a828b491bbf

[16] Stanford, "CS231n: Convolutional Neural Networks for Visual Recognition."

[17] A. Xygkis, "Implementation of Convolutional Neural Networks on Embedded Architectures," Sep. 2017. [Online]. Available: http://artemis.cslab.ece.ntua.gr:8080/jspui/bitstream/123456789/13550/1/DT2017-0208.pdf

[18] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich, "Going deeper with convolutions," 2014.

[19] R. Keys, "Cubic convolution interpolation for digital image processing," vol. 29, no. 6, pp. 1153–1160, Dec. 1981. [Online]. Available: https://doi.org/10.1109/tassp.1981.1163711

[20] V. Leon, G. Lentaris, E. Petrongonas, D. Soudris, G. Furano, A. Tavoularis, and D. Moloney, "Improving Performance-Power-Programmability in Space Avionics with Edge Devices: VBN on Myriad2 SoC," *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 20, no. 3, pp. 1–23, Mar. 2021.

[21] D. Moloney, B. Barry, R. Richmond, F. Connor, C. Brick, and D. Donohoe, "Myriad 2: Eye of the Computational Vision Storm," in *IEEE Hot Chips Symposium (HCS)*, Aug. 2014.

[22] G. Furano, G. Meoni, A. Dunne, D. Moloney, V. Ferlet-Cavrois, A. Tavoularis, J. Byrne, L. Buckley, M. Psarakis, K.-O. Voss, and L. Fanucci, "Towards the use of artificial intelligence on the edge in space systems: Challenges and opportunities," vol. 35, no. 12, pp. 44–56, Dec. 2020. [Online]. Available: https://doi.org/10.1109/maes.2020.3008468

[23] R. Banakar, S. Steinke, B.-S. Lee, M. Balakrishnan, and P. Marwedel, "Scratchpad memory." ACM Press, 2002. [Online]. Available: https://doi.org/10.1145/774789.774805

[24] C. Alippi, S. Disabato, and M. Roveri, "Moving convolutional neural networks to embedded systems: The AlexNet and VGG-16 case." IEEE, Apr. 2018. [Online]. Available: https://doi.org/10.1109/ipsn.2018.00049

[25] G. Lentaris, G. Chatzitsompanis, V. Leon, K. Pekmestzi, and D. Soudris, "Combining arithmetic approximation techniques for improved CNN circuit design." IEEE, Nov. 2020. [Online]. Available: https://doi.org/10.1109/icecs49266.2020.9294869

[26] S. I. Venieris and C.-S. Bouganis, "fpgaconvnet: A framework for mapping convolutional neural networks on FPGAs," in *2016 IEEE 24th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, 2016, pp. 40–47.

[27] V. Leon, S. Mouselinos, K. Koliogeorgi, S. Xydis, D. Soudris, and K. Pekmestzi, "A TensorFlow Extension Framework for Optimized Generation of Hardware CNN Inference Engines," *Technologies*, vol. 8, no. 1, 2020. [Online]. Available: https://www.mdpi.com/2227-7080/8/1/6

[28] V. Leon, T. Paparouni, E. Petrongonas, D. Soudris, and K. Pekmestzi, "Improving power of DSP and CNN hardware accelerators using approximate floating-point multipliers," *ACM Transactions on Embedded Computing Systems*, vol. 20, no. 5, pp. 1–21, Jul. 2021.

[29] V. Leon, K. Pekmestzi, and D. Soudris, "Exploiting the potential of approximate arithmetic in DSP & AI hardware accelerators," in *2021 31st International Conference on Field-Programmable Logic and Applications (FPL)*, 2021, pp. 263–264.

[30] T.-Y. Lu, H.-H. Chin, H.-I. Wu, and R.-S. Tsay, "A Very Compact Embedded CNN Processor Design based on Logarithmic Computing," 2020.

[31] A. Jahanshahi, "TinyCNN: A tiny modular CNN accelerator for embedded FPGA," 2019.

[32] L. Puglia, M. Ionică, G. Raiconi, and D. Moloney, "Passive Dense Stereo Vision on the Myriad2 VPU," in *IEEE Hot Chips Symposium (HCS)*, 2016, pp. 1–5.

[33] V. Leon, C. Bezaitis, G. Lentaris, D. Soudris, D. Reisis, E.-A. Papatheofanous, A. Kyriakos, A. Dunne, A. Samuelsson, and D. Steenari, "FPGA & VPU Co-Processing in Space Applications: Development and Testing with DSP/AI Benchmarks," in *2021 28th IEEE International Conference on Electronics, Circuits and Systems (ICECS)*, 2021, pp. 1–5.

[34] C. Marantos, N. Karavalakis, V. Leon, V. Tsoutsouras, K. Pekmestzi, and D. Soudris, "Efficient Support Vector Machines Implementation on Intel/Movidius Myriad 2," in *2018 7th International Conference on Modern Circuits and Systems Technologies (MOCAST)*, 2018, pp. 1–4.

[35] A. Xygkis, L. Papadopoulos, D. Moloney, D. Soudris, and S. Yous, "Efficient Winograd-based Convolution Kernel Implementation on Edge Devices," in *ACM/ESDA/IEEE Design Automation Conference (DAC)*, 2018, pp. 1–6.

[36] F. Tsimpourlas, L. Papadopoulos, A. Bartsokas, and D. Soudris, "A Design Space Exploration Framework for Convolutional Neural Networks Implemented on Edge Devices," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 37, no. 11, pp. 2212–2221, Nov. 2018.

[37] X. Xu, J. Amaro, S. Caulfield, A. Forembski, G. Falcao, and D. Moloney, "Convolutional Neural Network on Neural Compute Stick for Voxelized Point-Clouds Classification," in *International Congress on Image and Signal Processing, BioMedical Engineering and Informatics (CISP-BMEI)*, Oct. 2017, pp. 1–7.

[38] J. Hochstetler, R. Padidela, Q. Chen, Q. Yang, and S. Fu, "Embedded Deep Learning for Vehicular Edge Computing," in *IEEE/ACM Symposium on Edge Computing (SEC)*, Oct. 2018, pp. 341–343.

[39] F. Chollet *et al.* (2015) Keras. [Online]. Available: https://github.com/fchollet/keras

[40] D. Merkel, "Docker: lightweight linux containers for consistent development and deployment," *Linux journal*, vol. 2014, no. 239, p. 2, 2014.

[41] G. Bradski, "The OpenCV Library," *Dr. Dobb's Journal of Software Tools*, 2000.

[42] Intel Ltd. (2021) OpenVINO Toolkit: Open Model Zoo. [Online]. Available: https://github.com/openvinotoolkit/open_model_zoo

[43] K. Turkowski, "Filters for common resampling tasks," *Graphics gems*, pp. 147–165, 1990.