



ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ

ΣΧΟΛΗ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ
ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ

ΤΟΜΕΑΣ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΥΠΟΛΟΓΙΣΤΩΝ
ΕΡΓΑΣΤΗΡΙΟ ΜΙΚΡΟΪΠΟΛΟΓΙΣΤΩΝ & ΨΗΦΙΑΚΩΝ
ΣΥΣΤΗΜΑΤΩΝ

Implementation of Computer Vision Algorithms on Embedded Architectures

Ανάπτυξη Αλγορίθμων Ρομποτικής Όρασης σε
Ενσωματωμένες Αρχιτεκτονικές

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

ΟΔΥΣΣΕΑΣ Γ. ΠΑΠΑΝΙΚΟΛΑΟΥ

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

Αθήνα, Νοέμβριος 2016



ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ

ΣΧΟΛΗ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ
ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ

ΤΟΜΕΑΣ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΥΠΟΛΟΓΙΣΤΩΝ
ΕΡΓΑΣΤΗΡΙΟ ΜΙΚΡΟΪΠΟΛΟΓΙΣΤΩΝ & ΨΗΦΙΑΚΩΝ
ΣΥΣΤΗΜΑΤΩΝ

Implementation of Computer Vision Algorithms on Embedded Architectures

Ανάπτυξη Αλγορίθμων Ρομποτικής Όρασης σε
Ενσωματωμένες Αρχιτεκτονικές

Διπλωματική Εργασία

Οδυσσέας Γ. Παπανικολάου

Εγκρίθηκε από την τριμελή εξεταστική επιτροπή την 4^η Νοεμβρίου 2016.

Δημήτριος Σούντρης
Αν. Καθηγητής Ε.Μ.Π.

Κιαμάλ Πεχμεστζή
Καθηγητής Ε.Μ.Π.

Γεώργιος Γκούμας
Επ. Καθηγητής Ε.Μ.Π.

.....

Οδυσσέας Γ. Παπανικολάου

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

Copyright © Οδυσσέας Γ. Παπανικολάου, 2016.

Με επιφύλαξη παντός δικαιώματος. All rights reserved.

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

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

Περίληψη

Η όραση υπολογιστών αποτελούσε κυρίως ένα πεδίο της ακαδημαϊκής έρευνας κατά τη διάρκεια των προηγούμενων ετών. Σήμερα, ωστόσο, είναι όλο και πιο δημοφιλής σε εφαρμογές πραγματικού κόσμου. Λόγω της εμφάνισης πολύ ισχυρών, χαμηλού κόστους, και ενεργειακά αποδοτικών μικρο - επεξεργαστών, έχει καταστεί δυνατό να ενσωματωθούν πρακτικοί αλγόριθμοι όρασης υπολογιστών σε ενσωματωμένα συστήματα και να δημιουργηθούν έξυπνες συσκευές ικανές να κατανοήσουν το περιβάλλον τους μέσω οπτικών μέσων.

Στην παρούσα διπλωματική εργασία, ασχολούμαστε με την ανάπτυξη του αλγορίθμου ανίχνευσης σημείων ενδιαφέροντος των Harris και Stephens, σε μια ενσωματωμένη συσκευή που έχει σχεδιαστεί για την επιτάχυνση εφαρμογών μηχανικής όρασης, τη Myriad 2 από την Movidius. Μετά την αξιολόγηση βασικών φίλτρων υπολογιστικής όρασης, δημιουργήσαμε ένα πλήθος κανόνων με βάση τους οποίους θα μπορούσαμε να επιτύχουμε μια αποδοτική υλοποίηση σε αυτή την πλατφόρμα. Στη συνέχεια, οι απαραίτητες μετατροπές έγιναν προκειμένου να μεταφερθεί ο αλγόριθμος από γενικού σκοπού επεξεργαστές σε μια ενσωματωμένη συσκευή. Ακολούθως, αρχίσαμε εισάγοντας αρκετές βελτιώσεις με στόχο την αξιοποίηση του υλικού της Myriad στο υψηλότερο επίπεδο και τη μέγιστη απόδοση. Αυτές οι βελτιστοποιήσεις, ήταν αρχικά στοχευμένες στη μείωση του εύρους μνήμης της εφαρμογής. Με τον περιορισμό του αριθμού των απαιτούμενων προσπελάσεων στη μνήμη, ήμασταν σε θέση να επιτρέψουμε στους VLIW επεξεργαστές της Myriad 2 να εκτελέσουν τον αλγόριθμο αποδοτικά, χωρίς ενδιάμεσες καθυστερήσεις από μεταφορές δεδομένων. Επιπλέον, μια αλγοριθμική μετατροπή εφαρμόστηκε προκειμένου να ευθυγραμμίσει τις πιο απαιτητικές συναρτήσεις της υλοποίησης με το σύστημα επεξεργασίας των SHAVEs. Η Myriad 2, σε αντίθεση με τους παραδοσιακούς μικρο-επεξεργαστές, έχει σχεδιαστεί για να λειτουργεί παράλληλα σε μεγάλο όγκο πληροφοριών. Ως εκ τούτου, κάνοντας μια σχετική βελτιστοποίηση στις βασικές συναρτήσεις, ήμασταν σε θέση να επιτύχουμε ακόμη υψηλότερα επίπεδα απόδοσης.

Τέλος, η εργασία αυτή απέδειξε ότι Myriad 2 μπορεί να επιταχύνει σημαντικά υψηλής πολυπλοκότητας αλγορίθμους μηχανικής όρασης, όπως αυτός των Harris και Stephens, και να ανταπεξέλθει αποτελεσματικά στο βαρύ υπολογιστικό φορτίο λειτουργώντας σε ένα εξαιρετικά χαμηλό επίπεδο κατανάλωσης ισχύος.

Λέξεις κλειδιά

Μηχανική όραση, Αλγόριθμος ανίχνευσης σημείων ενδιαφέροντος των Harris και Stephens, Myriad 2, Αναγνώριση Γωνιών, Ενσωματωμένα συστήματα.

Abstract

Computer vision has mainly been a field of academic research over the past several decades. Nowadays, however, it is becoming more and more popular in real world applications. Due to the emergence of very powerful, low-cost, and energy-efficient processors, it has become possible to incorporate practical computer vision capabilities into embedded systems and create devices capable of understanding their environment through visual means.

In this thesis we deal with the development of the Harris & Stephens corner detection algorithm into an embedded device designed to accelerate machine vision tasks, Myriad 2 by Movidius. After evaluating basic computer vision filters, we created a design space based on which we could achieve an efficient implementation in this platform. Subsequently, the necessary transformations were made in order to port the algorithm from a general purpose CPU into an embedded device. Then, we started introducing several optimizations with the goal of utilizing Myriad's hardware at the highest level and reaching a maximum performance. These optimizations were initially based in reducing the memory overhead of the application. By limiting the number of required memory accesses, we were able to allow the VLIW processors of Myriad 2 to run the algorithm efficiently, without too many stalls. In addition, an algorithmic transformation was applied in order to align the most demanding functions with the processing scheme of the SHAVEs. Myriad 2, instead of traditional processors, is designed for operating in parallel on tons of information that they are all coming through at once. Therefore, by making a relevant optimization in the basic functions, we were able to achieve even higher efficiency levels.

Finally, our implementation proved that Myriad 2 can accelerate significantly a high-complexity Computer Vision algorithm, like Harris corner detection, and deal with the intensive computational load efficiently and at an ultra low power envelope.

Key words

Computer Vision, Harris & Stephens algorithm, Myriad 2, Corner detection, VLIW, Embedded Vision, Embedded Architectures.

Acknowledgements

For the completion of this thesis, I would like to express my sincere gratitude to Prof. Dimitrios Soudris for inspiring me through his teaching and research. I would also like to thank him for trusting me with such a demanding, as well as interesting scientific task. Of course, this thesis could not have been completed successfully without the invaluable contribution of Dr. Lazaros Papadopoulos. I am grateful to him for the guidance and support he had so willingly offered throughout my research. In addition, I would like to express my gratitude to Dr. George Lentaris for supporting me with his expertise on the theoretical subject of this work.

Finally, I would also like to thank my friends and my family for the love and support they have showed me throughout the duration of this thesis and my studies in general.

Contents

Περίληψη	5
Abstract	7
Acknowledgements	9
Contents	11
List of Figures	13
Ανάπτυξη Αλγορίθμων Μηχανικής Όρασης σε ενσωματωμένες αρχιτεκτονικές	15
1. Computer Vision	35
1.1 Definition	35
1.2 Applications	38
1.3 Convolution Kernels	41
1.4 Feature Detection	43
1.4.1 Canny edge detection	44
1.4.2 Moravec detector	47
1.4.3 The Harris & Stephens / Plessey / Shi-Tomasi corner detection algorithm	49
2. Embedded Architecture Presentation	53
2.1 Myriad 2 Architecture	53
2.1.1 List of basic features	53
2.1.2 Myriad 2 Block Level Architecture Overview	54
2.1.3 Streaming Hybrid Access Vector Engine	57
2.1.4 Memory subsystem	60
2.1.5 Myriad 2 implementation	61
2.2 Myriad 2 Programming Paradigms	62
2.3 Streaming Image Processing Pipeline	63
2.3.1 Introduction to the SIPP framework	63
2.3.2 Filter Graphs	63
2.4 Myriad 2 applications	67
3. Evaluation of basic Computer Vision filters in Myriad 2	69
3.1 Convolution filters	70
3.1.1 Software implementation	70
3.1.2 Hardware accelerators	78
3.2 Edge detection	81

3.2.1	Canny edge detection software implementation	81
3.2.2	Hardware accelerator	83
3.3	Outcome of the evaluation	84
4.	Implementation of Harris Corner Detection	87
4.1	SIPP filters implementation	87
4.1.1	Software Filter	87
4.1.2	Hardware accelerator	88
4.2	Porting a generic C implementation in Myriad 2	90
4.2.1	Description of the used implementation	90
4.2.2	First stage - Initial Porting	94
4.2.3	Second stage - Working buffers in CMX	97
4.2.4	Third Stage - Arbitrary image size and buffers in CMX	99
4.2.5	Fourth Stage - Task Based Parallelism	104
4.2.6	Fifth stage - Algorithmic optimization	107
5.	Conclusion	111
6.	Future Work	115

List of Figures

0.1	Αυτόνομο όχημα	18
0.2	Gaussian συνάρτηση	20
0.3	Καθορισμός σημείων ενδιαφέροντος μέσω ιδιοτιμών	21
0.4	Αρχιτεκτονική της Myriad 2	24
0.5	Αρχιτεκτονική των SHAVE	26
0.6	Διάγραμμα ροής του αλγόριθμου των Harris Algorithm & Stephens	27
1.1	Human visual system perception of a flower	35
1.2	Some examples of computer vision algorithms and applications.	36
1.3	Autonomus vehicle	40
1.4	ExoMars Rover	40
1.5	Convolution kernel for Gaussian blur	42
1.6	Steam engine photograph	45
1.7	Canny edge detection output image	45
1.8	Feature Detection	49
1.9	Gaussian window function	50
1.10	Classification of Image points	51
2.1	Myriad2 Architecture Overview	54
2.2	SHAVE Internal Architecture	58
2.3	Myriad 2 die plot	61
2.4	Basic stages of an sRGB pipeline	64
2.5	Separate processing of Luma and Chroma	65
2.6	Separate processing of Luma and Chroma - no DDR	66
3.1	SIPP convolutions	70
3.2	512x384 image used as input in the examined algorithms	72
3.3	1584x1290 image used as input in the examined algorithms	72
3.4	Canny edge detection on 512x384 image	81
3.5	Canny edge detection on 1584x1290 image	81
4.1	Harris Algorithm implementation flow chart	91
4.2	Harris corner detector applied on a 512x384 image	95

Ανάπτυξη Αλγορίθμων Μηχανικής Όρασης
σε ενσωματωμένες αρχιτεκτονικές

Όραση Υπολογιστών

Ορισμός

Η μηχανική όραση, υπολογιστική όραση ή τεχνητή όραση, είναι ένα επιστημονικό πεδίο της τεχνητής νοημοσύνης το οποίο επιχειρεί να αναπαράγει αλγοριθμικά την αίσθηση της όρασης, συνήθως σε συστήματα βασισμένα σε ηλεκτρονικούς υπολογιστές, όπως τα ρομπότ. Η μηχανική όραση σχετίζεται με τη θεωρία και την τεχνολογία που εμπλέκονται στη σχεδίαση και κατασκευή συστημάτων που λαμβάνουν και αναλύουν δεδομένα από ψηφιακές εικόνες. Τα εν λόγω δεδομένα μπορούν να είναι φωτογραφίες, βίντεο, όψεις από πολλαπλές κάμερες και πολυδιάστατες εικόνες.

Εφαρμογές

Η όραση υπολογιστών χρησιμοποιείται σήμερα σε πλήθος εφαρμογών, οι οποίες περιλαμβάνουν :

- Έλεγχος διαδικασιών (π.χ. ένα βιομηχανικό ρομπότ ή ένα αυτόνομο όχημα)
- Ανίχνευση συμβάντων (π.χ. οπτική επιτήρηση)
- Οργάνωση πληροφοριών (π.χ. ευρετηριοποίηση βάσεων δεδομένων και ακολουθιών εικόνων)
- Εξομοίωση αντικειμένων και περιβαλλόντων (π.χ. βιομηχανική επιθεώρηση, ιατρική ανάλυση εικόνας ή τοπογραφική εξομοίωση)
- Αλληλεπίδραση χρηστών με υπολογιστικά συστήματα (π.χ. ως είσοδος σε μια συσκευή επικοινωνίας ανθρώπου / μηχανής).

Ένα από τα πιο σημαντικά πεδία εφαρμογής είναι η ιατρική όραση ή ιατρική επεξεργασία εικόνας. Αυτή η περιοχή χαρακτηρίζεται από την εξαγωγή πληροφοριών από τα δεδομένα της εικόνας για τον σκοπό της πραγματοποίησης ιατρικής διάγνωσης ενός ασθενούς. Σε γενικές γραμμές, τα δεδομένα εικόνας είναι σε μορφή εικόνων μικροσκοπίας, εικόνων ακτίνων X, αγγειογραφιών, υπερήχων και εικόνων τομογραφίας. Ένα παράδειγμα των πληροφοριών που μπορούν να εξαχθούν από τα εν λόγω δεδομένα εικόνας είναι η ανίχνευση όγκων, η αρτηριοσκλήρωση ή άλλες επιζήμιες αλλαγές. Μπορεί επίσης να είναι μετρήσεις των διαστάσεων του οργάνου, της ροή του αίματος, κλπ Αυτή η περιοχή υποστηρίζει επίσης την ιατρική έρευνα με την παροχή

Αλγόριθμος Ανίχνευσης Γωνιών των Harris & Stephens

Ο κύριος αλγόριθμος με τον οποίο αχολούμαστε σε αυτή τη διπλωματική εργασία είναι ο αλγόριθμος ανίχνευσης γωνιών των Harris & Stephens. Η ανίχνευση γωνιών είναι μια προσέγγιση που χρησιμοποιείται στο πλαίσιο των συστημάτων όρασης υπολογιστών για την εξαγωγή ορισμένων χαρακτηριστικών μιας εικόνας. Πεδία εφαρμογής της αποτελούν η ανίχνευση κίνησης, η καταγραφή εικόνων, η τρισδιάστατη μοντελοποίηση ενός χώρου και η αναγνώριση αντικειμένων. Η έννοια της ανίχνευσης γωνιών συμπίπτει με αυτή του εντοπισμού σημείων ενδιαφέροντος.

Ένας από τους πρώτους αλγορίθμους ανίχνευσης γωνιών είναι αυτός του Moravec, ο οποίος ορίζει μία γωνία ως ένα σημείο με χαμηλή αυτο-ομοιότητα. Ο αλγόριθμος εξετάζει κάθε pixel στην εικόνα για να δει εάν μια γωνιά είναι παρούσα, ελέγχοντας το επίπεδο ομοιότητας ενός μέρους της εικόνας με επίκεντρο το pixel και άλλων σημείων σε κοντινή απόσταση και με τα οποία υπάρχει αλληλοεπικάλυψη. Η ομοιότητα μετράται με τη λήψη του αθροίσματος των τετραγώνων των διαφορών (SSD) μεταξύ των αντίστοιχων pixels των δύο κομματιών. Όσο μικρότερος ο αριθμός αυτός, τόσο μεγαλύτερη η ομοιότητα.

Η τιμή που καθορίζει μια γωνία ορίζεται ως το μικρότερο SSD μεταξύ του κομματιού και των γειτόνων του (οριζόντια, κάθετα και επί των δύο διαγωνίων). Ο λόγος γι' αυτό είναι πως, εάν αυτός ο αριθμός είναι υψηλός, τότε η διακύμανση κατά μήκος όλων των μετατοπίσεων είναι είτε ίσος είτε μεγαλύτερος από αυτόν, δείχνοντας έτσι ότι όλες οι κοντινές περιοχές θα είναι διαφορετικές.

Οι Harris και Stephens βελτίωσαν τον αλγόριθμο του Moravec θεωρώντας τη διαφοροποίηση της “βαθμολογίας” μιας γωνίας σε σχέση με την κατεύθυνση άμεσα, αντί να χρησιμοποιούν μετατοπισμένα παράθυρα.

Έστω ότι η εικόνα συμβολίζεται ως I . Ας θεωρήσουμε ότι παίρνουμε ένα κομμάτι της εικόνας στην περιοχή (u, v) και το μετατοπίζουμε κατά (x, y) . Το σταθμισμένο άθροισμα των τετραγώνων των διαφορών (SSD) μεταξύ αυτών των δύο κομματιών, το οποίο συμβολίζεται S , δίνεται από την ακόλουθη εξίσωση:

$$E(u, v) = \sum w(x, y) \times [I(x + u, y + v) - I(x, y)] \quad (0.1)$$

όπου:

- E το άθροισμα των τετραγώνων των διαφορών
- $W(x,y)$ η συνάρτηση παραθύρου
- $I(x,y)$ η τιμή της έντασης του φωτός στο pixel
- $I(x+u,y+v)$ η τιμή της έντασης του φωτός σε γειτονικό σημείο

όπου το $I(u+x, v+y)$ μπορεί να προσεγγιστεί μέσω ενός αναπτύγματος Taylor. Ας θεωρήσουμε I_x και I_y τις μερικές παραγώγους του I , έτσι ώστε:

$$I(u + x, v + y) \approx I(u, v) + I_x(u, v)x + I_y(u, v)y \quad (0.2)$$

Επομένως, η συνάρτηση του αθροίσματος γίνεται:

$$S(x, y) \approx \sum \sum w(u, v) \times (I(u, v) + I_x(u, v)x + I_y(u, v)y)^2 \quad (0.3)$$

Η προηγούμενη σχέση μπορεί να γραφεί σε μητρική μορφή ως εξής:

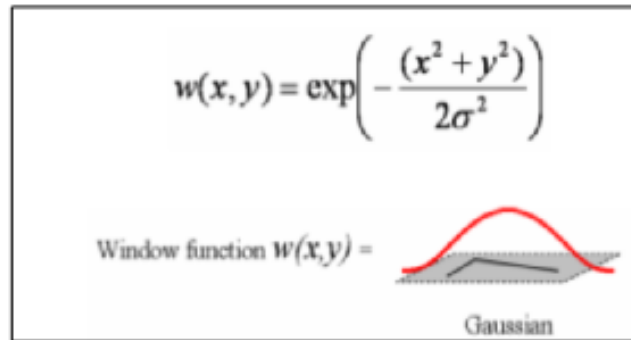
$$S(x, y) \approx (xy)A \begin{pmatrix} x \\ y \end{pmatrix} \quad (0.4)$$

όπου:

$$A = \sum_u \sum_v w(u, v) \begin{bmatrix} I_x^2 & I_x I_y \\ I_x I_y & I_y^2 \end{bmatrix} = \begin{bmatrix} \langle I_x^2 \rangle & \langle I_x I_y \rangle \\ \langle I_x I_y \rangle & \langle I_y^2 \rangle \end{bmatrix} \quad (0.5)$$

Η μήτρα αυτοσυσχέτισης A παρέχει ένα κάτω φράγμα για την αβεβαιότητα στη θέση ενός ταιριάζοντος παραθύρου. Ως εκ τούτου, είναι ένας χρήσιμος δείκτης με βάση τον οποίο μπορεί να ελεγχθεί η ταύτιση διαφορετικών κομματιών.

Μια άλλη βελτίωση σε σύγκριση με τον αλγόριθμο του Moravec είναι ότι η συνάρτηση παραθύρου είναι τώρα Gaussian, εξασφαλίζοντας ισοτροπική απόκριση. Η γενική μορφή μιας Gaussian συνάρτησης παραθύρου παρουσιάζεται στην ακόλουθη εικόνα:



Σχήμα 0.2: Gaussian συνάρτηση [5]

Στον αλγόριθμο Harris, μια γωνιά θεωρείται ότι έχει μεγάλη διακύμανση στο S σε όλες τις κατευθύνσεις του πίνακα $(x \ y)$. Σε μαθηματική μορφή, αυτό μπορεί να εκφραστεί μέσω των ιδιοτιμών του πίνακα A . Εάν ένα σημείο ενδιαφέροντος είναι υπό εξέταση, τότε ο πίνακας A θα πρέπει να έχει δύο ιδιοτιμές με μεγάλη τιμή. Λαμβάνοντας υπόψη τα μεγέθη των ιδιοτιμών, μπορούμε να καθορίσουμε τις ακόλουθες περιπτώσεις:

- $\lambda_1 \approx 0$ and $\lambda_2 \approx 0$, τότε αυτό το σημείο δεν αποτελεί σημείο ενδιαφέροντος
- $\lambda_1 \approx 0$ and λ_2 έχει μεγάλη θετική τιμή, τότε στο σημείο αυτό βρίσκεται μια ακμή
- Both λ_1, λ_2 έχουν μεγάλες θετικές τιμές, τότε στο σημείο αυτό έχουμε γωνία

Ωστόσο, επειδή ο υπολογισμός των ιδιοτιμών απαιτεί υψηλό υπολογιστικό φόρτο, οι Harris και Stephens προτείνουν την εναλλακτική συνάρτηση M_c η οποία παρατίθεται εδώ:

$$M_c = \lambda_1 \lambda_2 - \kappa(\lambda_1 + \lambda_2)^2 = \det(A) - \kappa \text{trace}^2(A) \quad (0.6)$$

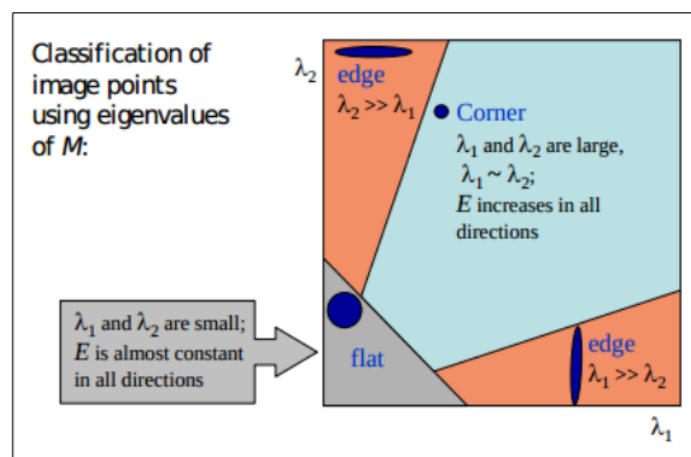
όπου $\det(M) = \lambda_1 \lambda_2$, $\text{trace}(M) = \lambda_1 + \lambda_2$ και κ είναι μία τιμή που επιλέγεται με βάση το επιθυμητό επίπεδο ευασθησίας στην έξοδο του αλγορίθμου. Αποδεκτές τιμές του αλγορίθμου είναι μεταξύ 0.04 και 0.15.

Σε αντίθεση με την ανάλυση ιδιοτιμών, η ποσότητα αυτή δεν απαιτεί τη χρήση των τετραγωνικών ριζών ενώ εξακολουθεί να αποφεύγει χαρακτηριστικά που είναι συνήθη σε ακμές όπου $\lambda_2 \gg \lambda_1$. Ο Triggs [1] προτείνει τη χρησιμοποίηση της ποσότητας

$$\lambda_1 - \kappa \lambda_2$$

η οποία μειώνει επίσης την απόκριση στις ακμές, όπου μερικές φορές σφάλματα παραποίησης διογκώνουν τη μικρότερη ιδιοτιμή. Δείχνει επίσης πώς η βασική 2x2 Hessian μπορεί να επεκταθεί σε παραμετρικές κινήσεις για την ανίχνευση σημείων που είναι επίσης με ακρίβεια εντοπίσιμα σε κλίμακα και περιστροφή.

Ακολούθως παρουσιάζεται μία γραφική αναπαράσταση της ταξινόμησης των σημείων εικόνας, σύμφωνα με τις ιδιοτιμές του M :



Σχήμα 0.3: Καθορισμός σημείων ενδιαφέροντος μέσω ιδιοτιμών. [2]

Τα βήματα ενός βασικού ανιχνευτή σημείων ενδιαφέροντος με βάση την αυτοσυσχέτιση συνοψίζονται στα ακόλουθα αλγοριθμικά στάδια:

1. Υπολογισμός των οριζόντιων και κάθετων παραγώγων της εικόνας I_x και I_y μέσω συνέλιξης της αρχικής εικόνας με παραγωγούς φίλτρων Gauss
2. Υπολογισμών των τριών εικόνων που αντιστοιχούν σε αυτές τις κλίσεις (Η μήτρα A είναι συμμετρική, οπότε μόνο τρεις εγγραφές απαιτούνται)
3. Συνέλιξη αυτών των εικόνων με ένα μεγαλύτερο πυρήνα φίλτρου Gauss.
4. Υπολογισμός ενός βαθμωτού μέτρου, χρησιμοποιώντας έναν από τους τύπους που συζητήθηκαν παραπάνω.
5. Εύρεση τοπικού μεγίστου πάνω από κάποιο όριο και αναγνώρισή του ως σημείο ενδιαφέροντος.

Παρουσίαση του ενσωματωμένου συστήματος

Το ενσωματωμένο σύστημα που χρησιμοποιήθηκε για την εφαρμογή του εξεταζόμενου αλγορίθμου είναι η Myriad 2 από τη Movidius, η πρώτη always-on Vision Processing Unit που έχει κατασκευαστεί. Η Myriad2 είναι ένα SoC πολλαπλών πυρήνων που υποστηρίζει εφαρμογές υπολογιστικής όρασης και οπτική αντίληψης για κινητές συσκευές, wearable και ενσωματωμένες πλατφόρμες. Αυτό το κεφάλαιο περιγράφει τη λειτουργικότητα και τη χρήση της Myriad 2. Επιπλέον, εισάγει λεπτομέρειες που αφορούν την αρχιτεκτονική αυτού του τσιπ.

Λίστα βασικών χαρακτηριστικών

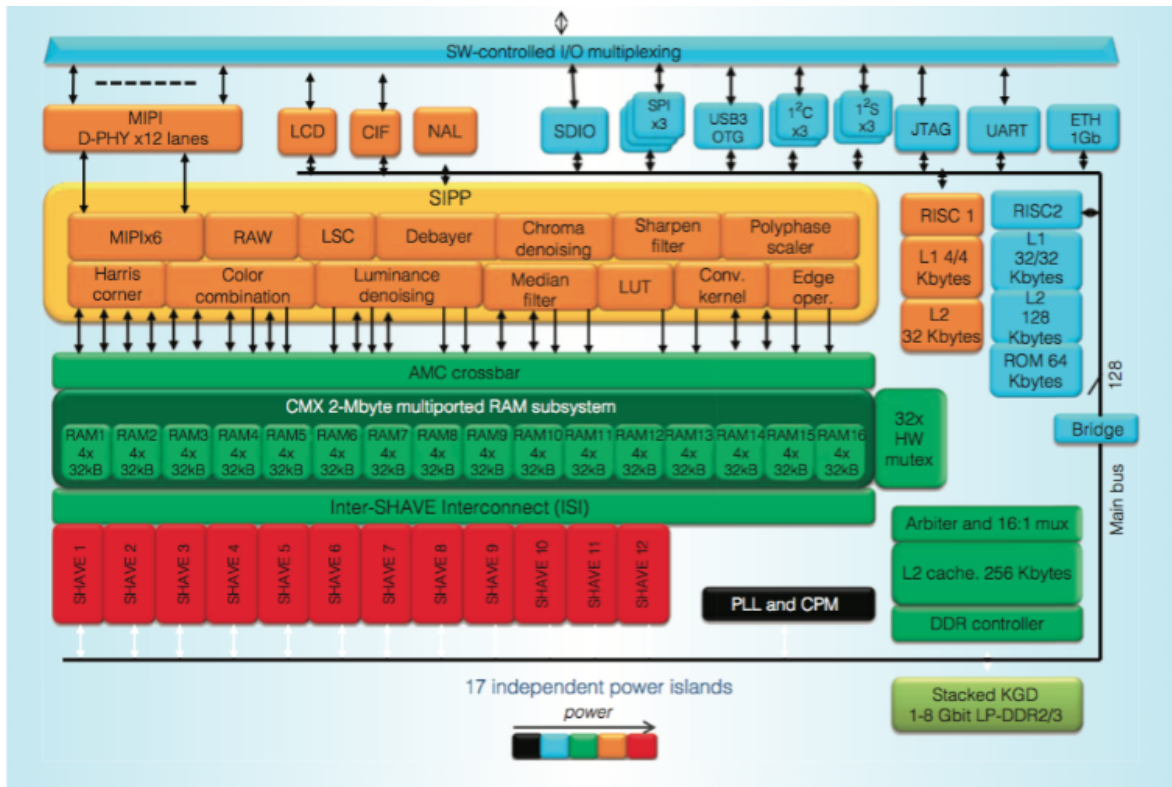
Τα χαρακτηριστικά της Myriad 2 είναι τα ακόλουθα:

- 12 x SHAVE (Streaming Hybrid Access Vector Engine) VLIW vector επεξεργαστές (128-bit) , 2 x RISC επεξεργαστές
- 2 MB on-chip μνήμης RAM (CMX)
- 128/512 MB μνήμης DDR
- Ο LEON RISC έχει 256 KB L2 κρυφή μνήμη
- Ο LEON RT has 32 KB L2 κρυφή μνήμη
- Τψηλό on-chip εύρος ζώνης
- SIPP Image Signal Processing επιταχυντές υλοποιημένους σε επίπεδο υλικού
- Εύρος περιφερειακών συσκευών εισόδου - εξόδου, όπως SPI, I2C, I2S, SDIO, Ethernet, USB
- Διεπαφές εικόνας, όπως MIPI, CIF, LCD

Η οικογένεια επεξεργαστών της Myriad 2 αποτελείται από τις εξής εκδόσεις:

- MA2x5x - MA2150 / MA2155 / MA2450 / MA2455

Για τους προγραμματιστικούς σκοπούς της εργασίας αυτής, χρησιμοποιήθηκε η έκδοση MA2150, η οποία έχει 128 MB DDR μνήμη ένα συχνότητα συστήματος στα 600 MHz. Το διάγραμμα της αρχιτεκτονικής της παρουσιάζεται στο παρακάτω σχήμα:



Σχήμα 0.4: Αρχιτεκτονική της Myriad 2 [3]

Ως VPU SoC, η Myriad 2 έχει ένα ελεγχόμενο από λογισμικό, πολλαπλών πυρήνων, υποσύστημα μνήμης και κρυφές μνήμες που μπορούν να ρυθμιστούν ώστε να επιτρέπουν το χειρισμό ενός μεγάλου φόρτου εργασίας. Επίσης, είναι δυνατή η επίτευξη υψηλού εύρους μεταφοράς δεδομένων και εντολών για την αποδοτική αξιοποίηση των 12 επεξεργαστών, των δύο επεξεργαστών RISC, και των φίλτρων που βασίζονται σε επιταχυντές υλικού. Ένα πολλαπλών καναλιών σύστημα άμεσης πρόσβασης στη μνήμη (DMA) μειώνει το φόρτο μεταφοράς δεδομένων μεταξύ των επεξεργαστών, των επιταχυντών υλικού και της μνήμης. Παράλληλα, μια μεγάλη γκάμα περιφερειακών συμπεριλαμβανομένων των φωτογραφικών μηχανών, LCD πάνελ, και συστημάτων μαζικής αποθήκευσης, επικοινωνούν αποδοτικά με τους επεξεργαστές και τους επιταχυντές υλικού.

Επειδή η ενεργειακή απόδοση είναι πρωταρχικής σημασίας, η συσκευή είναι σχεδιασμένη ώστε να έχει συνολικά 17 power islands, μεταξύ των οποίων μία για κάθε ένα από τους 12 SHAVE επεξεργαστές, επιτρέποντας έτσι βέλτιστο έλεγχο ισχύος σε επίπεδο λογισμικού. Η συσκευή υποστηρίζει 8-, 16-, 32-, και κάποιες λειτουργίες 64-bit σε ακεραίους, καθώς και fp16 (Open-EXR) και FP32 αριθμητική, ενώ μπορεί να φτάσει στα 1.000 Gflops (fp16). Η προκύπτουσα αρχιτεκτονική προσφέρει αυξη-

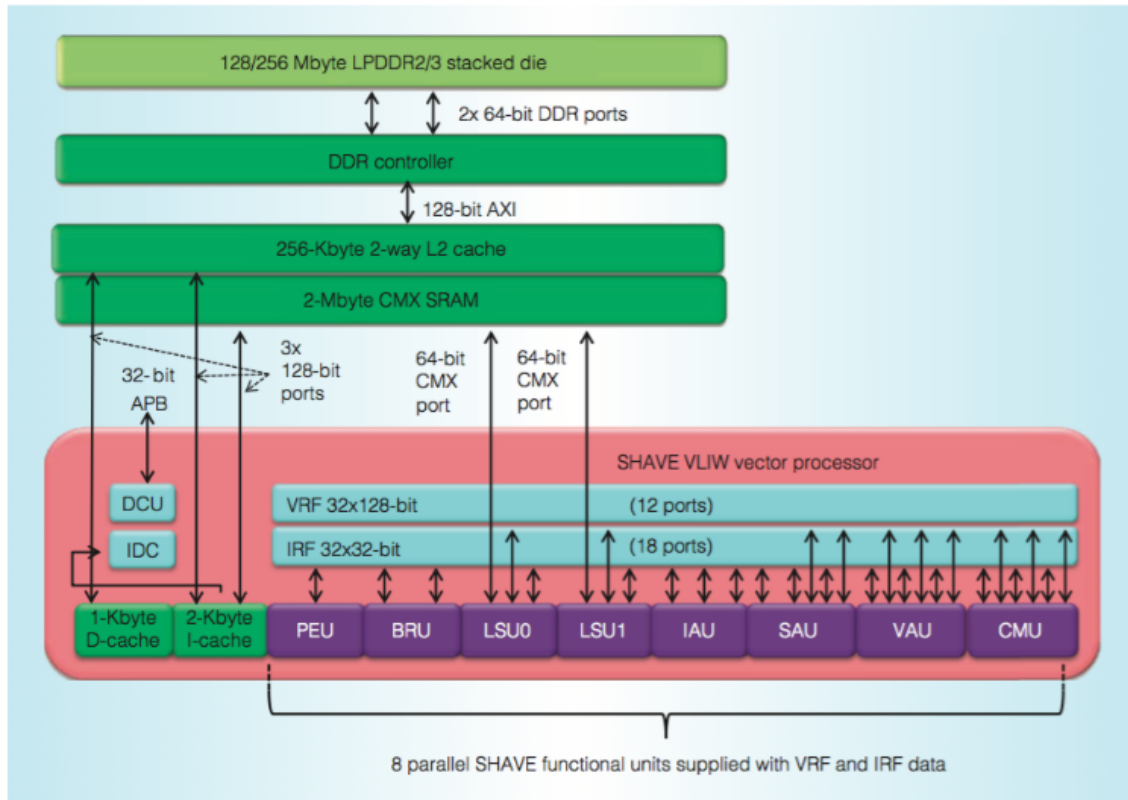
μένη απόδοση ανά watt σε ένα ευρύ φάσμα εφαρμογών από το πεδίο της όρασης υπολογιστών καθώς και της επαυξημένης πραγματικότητας.

Streaming Hybrid Access Vector Engines

Οι SHAVEs χρησιμοποιούνται για το μεγαλύτερο μέρος της επεξεργασίας κατά την εκτέλεση ενός αλγορίθμου. Κάθε επεξεργαστής περιέχει ευρύς καταχωρητές σε συνδυασμό με VLIW παρέχοντας έτσι μια μέθοδο για παραλληλισμό σε επίπεδο εντολών. VLIW πακέτα ελέγχουν πολλαπλές λειτουργικές μονάδες που έχουν ικανότητα SIMD, δηλαδή να πεξεργάζονται πολλά δεδομένα με έκδοση μιας εντολής. Έτσι, επιτυγχάνεται υψηλή παραλληλία και τη απόδοση σε λειτουργικές μονάδες και επεξεργαστές. Κάθε μία από αυτές τις μονάδες μπορεί να ξεκινήσει παράλληλα σε ένα ενιαίο πακέτο εντολών.

Ο SHAVE αποτελεί έναν επεξεργαστή υβριδικής αρχιτεκτονικής που συνδυάζει τα καλύτερα χαρακτηριστικά των GPUs, DSPs και RISC με 8-, 16-, και 32-bit αριθμητική ακεραίων και 16- και 32-bit αιθμιτική κινητής υποδιαστολής, καθώς και μοναδικά χαρακτηριστικά, όπως υλικό υποστήριξης αραιών δομών δεδομένων [3]. Η αρχιτεκτονική μεγιστοποιεί την απόδοση ανά watt, διατηρώντας παράλληλα την ευκολία του προγραμματισμού, ιδιαίτερα όσον αφορά την υποστήριξη για το σχεδιασμό και τη μεταφορά εφαρμογών λογισμικού για πολυπύρρηνα συστήματα.

Η αρχιτεκτονική των SHAVEs φαίνεται στο Σχήμα 0.5.



Σχήμα 0.5: Αρχιτεκτονική των επεξεργαστών SHAVE [3]

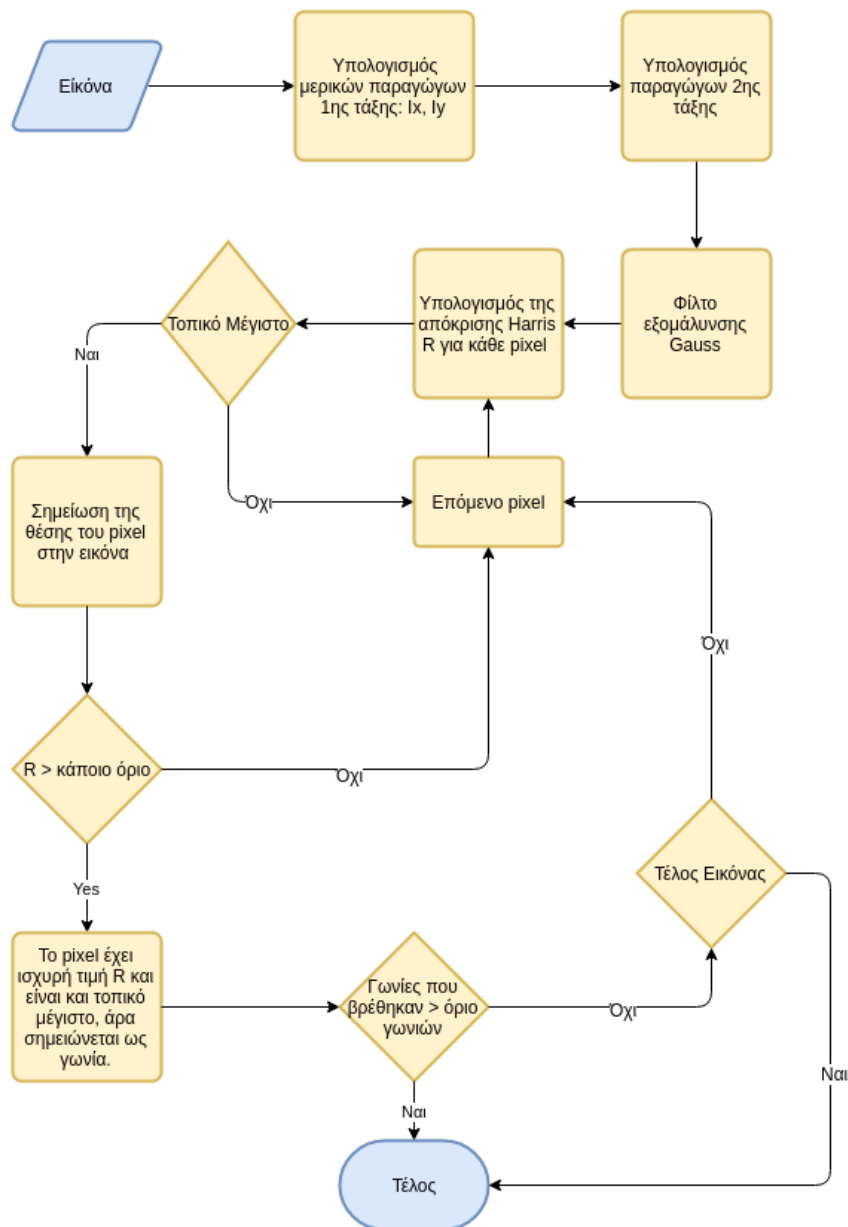
Τα SHAVEs υποστηρίζουν εντολές SIMD για πολλούς τύπους, που περιλαμβάνουν αλλά δεν περιορίζονται σε: 16 bits ακεραίους, 32 bits ακεραίους, 16 bits κινητής υποδιαστολής, 32 bits κινητής υποδιαστολής, 8 bits ακεραίους.

Οι εφαρμογές μπορούν να αναπτυχθούν σε Assembly, C και C ++ και να τρέξουν στους SHAVEs με τη χρήση των moniAsm και moniCompile, εργαλείων υλοποιημένων από τη Movidius.

Υπάρχουν δύο συστοιχίες αρχείων επεξεργαστών: IRF και VRF.

Στάδια ανάπτυξης του αλγόριθμου των Harris & Stephens στη Myriad 2

Για την ανάπτυξη του αλγορίθμου βασιστήκαμε σε μια C υλοποίηση η οποία είχε σχεδιαστεί για επεξεργαστές γενικού σκοπού. Το διάγραμμα της υλοποίησης αυτής φαίνεται παρακάτω :



Σχήμα 0.6: Διάγραμμα ροής του αλγόριθμου των Harris Algorithm & Stephens

Στο πρώτο βήμα, διαβάζονται τα δεδομένα της εικόνας και περνιούνται ως παράμετροι στη συνάρτηση που υπολογίζει τις μερικές παραγώγους πρώτης τάξης (I_x , I_y). Αυτό επιτυγχάνεται με την εφαρμογή ενός διαχωρίσιμου φίλτρο συνελίξεως που συνδυάζει τους ακόλουθους πυρήνες:

$$derivative : \begin{bmatrix} 1 & -3 & 0 & 3 & 1 \end{bmatrix}$$

και

$$smoothing : \begin{bmatrix} 1 & 6 & 12 & 6 & 1 \end{bmatrix}$$

Η συνάρτηση που επιτελεί αυτή τη λειτουργία αυτή ονομάζεται *imgradient5_smo()* και δίνεται παρακάτω σε μορφή ψευδοκώδικα. Σημειώστε ότι το σύμβολο "*" υποδηλώνει συνέλιξη, όχι πολλαπλασιασμό.

```
int imgradient5_smo(image[width, height], width, height, gradx, grady)
    for (i=[0,height-1])
        for (j=[0,width-1])
            wrkx[i, j]=image[i, j]*derivative_kernel;
            wrky[i, j]=image[i, j]*smoothing_kernel;
        next_line;

    for (i=[0,height-1])
        for (j=[0,width-1])
            gradx[i, j]=wrkx[i, j]*smoothing_kernel;
            grady[i, j]=wrky[i, j]*derivative_kernel;
        next_line;
```

Οι δύο πρώτοι δύο βρόχοι επανάληψης των *imgradient5_smo()* είναι οι οριζόντιες συνέλιξεις και ρόλος τους είναι ο υπολογισμός, για κάθε pixel, των μερικών παραγώγων αλλά και της εξομάλυνσης της εικόνας, αντίστοιχα, κρατώντας τα τοπικά αθροίσματα των πυρήνων 5 στοιχείων που διατρέχουν την εικόνα. Για το σκοπό αυτό, χρησιμοποιούνται οι τιμές των δύο προηγούμενων και επόμενων πίξελ στη συγκεκριμένη σειρά. Στη συνέχεια, κάθε τιμή κανονικοποιείται με βάση σχετικούς παράγοντες. Η ομαλοποίηση δεν εφαρμόζεται σε κάθε συντελεστή για να μειωθεί ο αριθμός των πράξεων κινητής υποδιαστολής.

Όμοια, στο δεύτερο ζευγάρι επαναλήψεων, γίνονται οι κάθετες συνέλιξεις. Τέλος, τα αποτελέσματα αποθηκεύονται στους πίνακες *gradx* και *grady*, οι οποίοι αντιστοιχούν στα I_x και I_y , αντιστοίχως.

ΕΠομένω βήμα αποτελεί ο υπολογισμός των τετραγώνων των μερικών παραγώγων, καθώς αυτές χρησιμοποιούνται στην βασική συνάρτηση του αλγόριθμου αναζήτησης γωνιών του Harris. Οι τιμές αυτές είναι οι I_x^2 , I_y^2 καθώς και $I_x I_y$

Έπειτα, τα παραπάνω περνιούνται ως όρισμα στη συνάρτηση *imgblurg()*, η οποία

καλείται τρεις φορές και εφαρμόζει ένα πυρήνα εξομάλυνσης Gauss.

Η συνάρτηση αυτή δίνεται παρακάτω. Εδώ, επίσης, το σύμβολο "*" υποδηλώνει συνέλιξη μεταξύ δύο πινάκων.

```
int imgblurg(gradx, grady, gradxy){
    /* separability: convolve horizontally ... */
    for (i=[0,height]){
        for (j=[0,width]){
            wrkx[i, j]=gradx * gaussian_kernel;
            wrky[i, j]=grady * gaussian_kernel;
            wrkxy[i, j]=gradxy * gaussian_kernel;

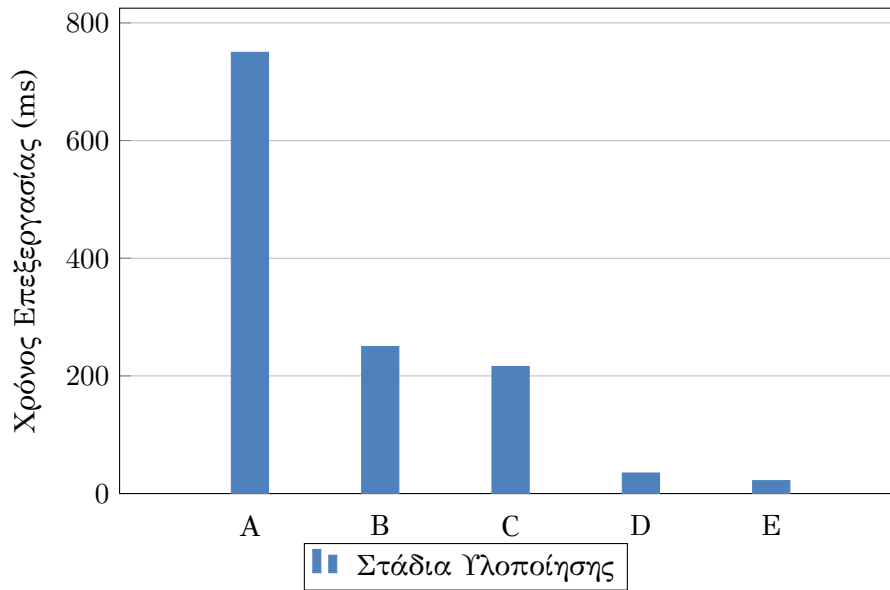
            /* ... then convolve vertically */
            for (i=[0,height]){
                for (j=[0,width]){
                    gradx2[i, j]=gradx * gaussian_kernel;
                    grady2[i, j]=grady * gaussian_kernel;
                    gradxy[i, j]=gradxy * gaussian_kernel;
                }
            }
        }
    }
}
```

Πλέον, είναι δυνατό να γίνει ο υπολογισμός της τιμής "γωνιότητας" για κάθε pixel, όπως αυτός περιγράφεται στο σχετικό αλγόριθμο. Αφού βρεθούν αυτές οι τιμές, ροκρτώνται μόνο όσες είναι πάνω από κάποιο ανώφλι και είναι επίσης μέγιστες σε ένα τοπικό πυρήνα. Οι τελευταίες, θεωρούνται σημεία ενδιαφέροντος, ή αλλιώς γωνίες.

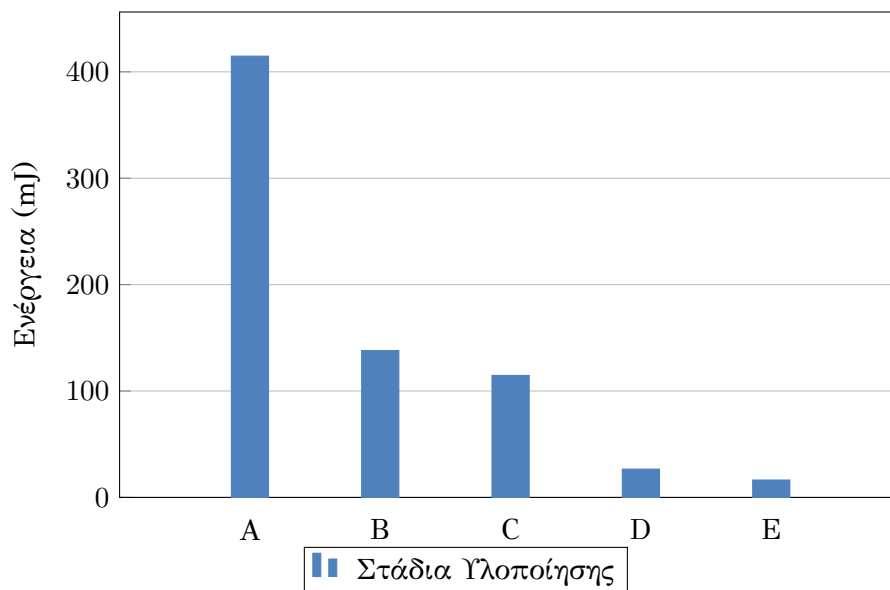
Επειδή η παραπάνω υλοποίηση είχε σχεδιαστεί αρχικά για γενικού σκοπού επεξεργαστές, η ανάπτυξη σε αυτή την εργασία απαίτησε διάφορα βήματα μετασχηματισμών και βελτιστοποιήσεων ώστε να επιτευχθεί ένα υψηλό επίπεδο απόδοσης στη Myriad 2.

Στα παρακάτω διαγράμματα δίνονται το κέρδος σε απόδοση και ενέργεια που επετεύχθη έπειτα από κάθε βήμα της υλοποίησης. Εφόσον η υλοποίηση αυτή στοχεύει ένα ενσωματωμένο σύστημα, ενέργεια που καταναλώνεται ανά εκτέλεση είναι πολύ σημαντική. Ακολούθως, δίνεται αναλυτική περιγραφή των βημάτων αυτών.

Κέρδος σε απόδοση



Κέρδος σε κατανάλωση ενέργειας



Κατάλληλη μνήμη για επεξεργασία δεδομένων στους SHAVEs - CMX vs DDR

Οι πίνακες αποθήκευσης δεδομένων μιας εφαρμογής που τρέχει στους SHAVEs πρέπει να τοποθετούνται στη μνήμη CMX. Ωστόσο, κατά το πρώτο στάδιο του porting ενός γενικού κώδικα C στη Myriad 2, το μικρό μέγεθος της CMX απαιτεί την τοποθέτηση αυτών των πινάκων στη DDR, δεδομένου ότι είχαν αρχικά κατασκευαστεί για μηχανές χωρίς περιορισμούς μνήμης. Στη συνέχεια, μετά την πραγματοποίηση ουσιαστικών τροποποιήσεων στην εφαρμογή, τα δεδομένα βρίσκονται στη σωστή μνήμη και η απόδοση που παίρνουμε αρχίζει να είναι πιο ευθυγραμμισμένη με τις δυνατότητες του Myriad 2, όπως μπορεί να φανεί με τη σύγκριση των φάσεων A και B των διαγραμμάτων. Ως εκ τούτου, η DDR μπορεί να χρησιμοποιηθεί για να κρατήσει τα

δεδομένα μόνο σε ένα πολύ αρχέγονο στάδιο της μεταφοράς κώδικα από επεξεργαστές γενικής χρήσης στη Myriad 2.

Αριθμός προσπελάσεων στη μνήμη μέσω DMA

Μετά την επίτευξη μιας πολύ υψηλότερη απόδοσης μέσω της τοποθέτησης των buffers στη CMX, η έμφαση μετατοπίζεται προς τη χρήση αυτής της μνήμης όσο το δυνατόν αποτελεσματικότερα. Η καθυστέρηση που εφαρμόζεται σε μια εφαρμογή από τις προσβάσεις στη μνήμη είναι ένα κοινό χαρακτηριστικό των ενσωματωμένων αρχιτεκτονικών. Για το σκοπό αυτό, γίνονται τροποποιήσεις προκειμένου να μειωθεί η επιβάρυνση της μνήμης της εφαρμογής και να εκτελούνται όσο το δυνατόν λιγότερο λειτουργίες DMA. Αυτή η βελτιστοποίηση έχει αποδώσει στην περίπτωση μας ένα κέρδος απόδοσης της τάξης του 15%.

Εφαρμογή παραλληλισμού μεταξύ πολλών SHAVE

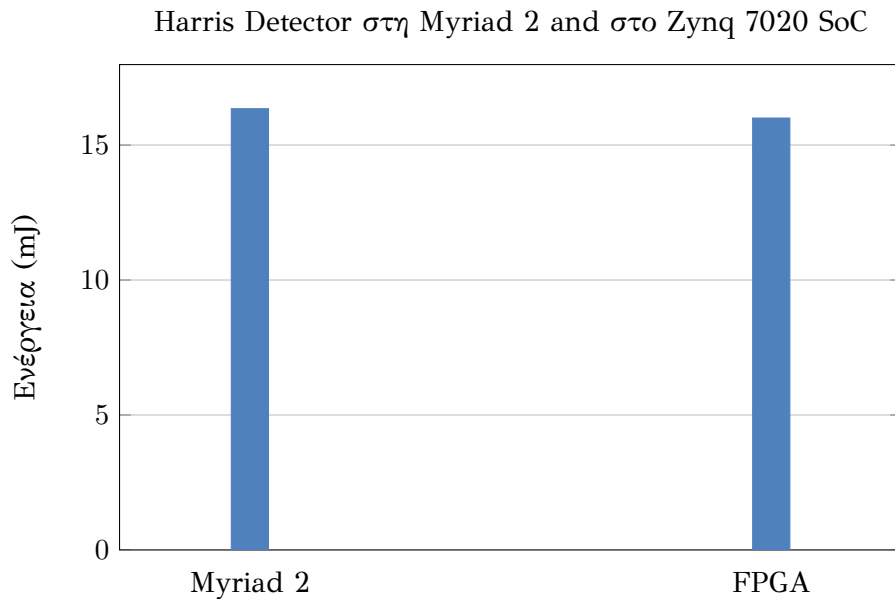
Τα προηγούμενα βήματα ήταν απαραίτητα προκειμένου να επιτευχθεί η πιο επαρκής απόδοση θα μπορούσαμε να πάρουμε με 1 SHAVE. Ωστόσο, η πραγματική δύναμη επεξεργασίας της Myriad 2 δεν έχει αξιοποιηθεί μέχρι να γίνει χρησιμοποιήσει όλων τα διαθέσιμων SHAVE. Ρυθμίζοντας 12 VLIW επεξεργαστές να τρέχουν τον αλγόριθμό μας παράλληλα οδηγούμαστε σε θεαματική αύξηση της αποτελεσματικότητας, όπως μπορεί να δει κανείς από τη διαφορά μεταξύ των σταδίων C και D των διαγραμμάτων.

Αλγοριθμική βελτιστοποίηση

Μέχρι αυτό το στάδιο έχουμε αξιοποιήσει όλα τα χαρακτηριστικά της Myriad 2 και έχουμε επιτύχει σημαντική αύξηση των επιδόσεων στην εφαρμογή μας. Ωστόσο, κατά την ανάπτυξη ενός κώδικα που αρχικά κατασκευάστηκε για CPU γενικής χρήσης, είναι σημαντικό να εξεταστούν βελτιστοποιήσεις όχι μόνο σε επίπεδο πλατφόρμας, αλλά και σε μια αλγοριθμικό επίπεδο. Στην περίπτωση μας, επικεντρωθήκαμε στην ενίσχυση της απόδοσης των λειτουργιών συνέλιξης, στις οποίες καταναλώνεται το μεγαλύτερο μέρος της επεξεργασίας. Αυτό επιτεύχθηκε με την αντικατάσταση των διαχωριζόμενων φίλτρων με φίλτρα τετράγωνου πυρήνα που, ακόμα κι αν χρειάζονται περισσότερες εργασίες, είναι περισσότερο ευθυγραμμισμένες με το σύστημα επεξεργασίας των SHAVEs. Η Myriad 2, σε αντίθεση με τους παραδοσιακές μικροεπεξεργαστές, έχει σχεδιαστεί για να λειτουργεί παράλληλα σε τόνους πληροφοριών. Ως εκ τούτου, ήταν σε θέση να επωφεληθεί από ένα μεγαλύτερο μέγεθος παραθύρου λειτουργίας (στάδιο E).

Συνδυάζοντας όλα τα παραπάνω στάδια, μία πολύ ικανοποιητική απόδοση στη Myriad 2 για μια σύνθετη εφαρμογή όρασης υπολογιστών, όπως η ανίχνευση γωνιών Harris. Το παρακάτω γράφημα απεικονίζει μια σύγκριση της ίδιας εφαρμογής σε Myriad 2 και μια συσκευή FPGA, το Zynq 7020 (XC7Z020) [4]. Η σύγκριση βασίζεται στην

κατανάλωση ενέργειας των συσκευών αυτών, ως δείκτη της συνολικής απόδοσης.



Δεδομένου ότι δεν υπάρχει σημαντική διαφορά στην αποτελεσματικότητα αυτών των ενσωματωμένων επεξεργαστών, η επιλογή μεταξύ αυτών πρέπει να βασίζεται σε άλλους παράγοντες. Για παράδειγμα, τα FPGAs μπορεί να χρησιμοποιηθεί όταν απαιτείται περισσότερη επεξεργαστική ισχύς, δεδομένου ότι ο χρόνος επεξεργασίας τους ήταν 6 ms με κατανάλωση 2 Watt, σε αντίθεση με τα 21 ms στη Myriad. Από την άλλη πλευρά, η Myriad 2 είναι εξαιρετική στο χειρισμό καθυκόντων όρασης υπολογιστών σε κατάσταση εξαιρετικά χαμηλής ισχύος, δεδομένου ότι καταναλώνει λιγότερο από 1 Watt. Ως εκ τούτου, είναι πιο κατάλληλη για κινητές συσκευές με περιορισμούς ισχύος, όπως μικρά ρομπότ, μη επανδρωμένα αεροσκάφη ή wearables.

Τέλος, η υλοποίησή μας απέδειξε ότι Myriad 2 μπορεί να επιταχύνει σημαντικά υψηλής πολυπλοκότητας αλγορίθμους μηχανικής όρασης, όπως η ανίχνευση γωνιών Harris, και να ανταπεξέλθει στο εντατικό υπολογιστικά φορτίο αποτελεσματικά, ακόμη και χωρίς τη χρήση των βελτιστοποιημένων βιβλιοθηκών CV που παρέχει η Movidius.

Implementation of Computer Vision
Algorithms on Embedded Architectures

Chapter 1

Computer Vision

1.1 Definition



Figure 1.1: The human visual system has no problem interpreting the subtle variations in translucency and shading in this photograph and correctly segmenting the object from its background. Figure reproduced from [5]

As humans, we perceive the three-dimensional structure of the world around us with apparent ease. Think of how vivid the three-dimensional percept is when you look at a vase of flowers sitting on the table next to you. You can tell the shape and translucency of each petal through the subtle patterns of light and shading that play across its surface and effortlessly segment each flower from the background of the scene (Figure 1.1). Looking at a framed group portrait, you can easily count (and name) all of the people in the picture and even guess at their emotions from their facial appearance. Perceptual psychologists have spent decades trying to understand how the visual system works but a complete solution to this puzzle remains elusive (Marr 1982;

Palmer 1999; Livingstone 2008).

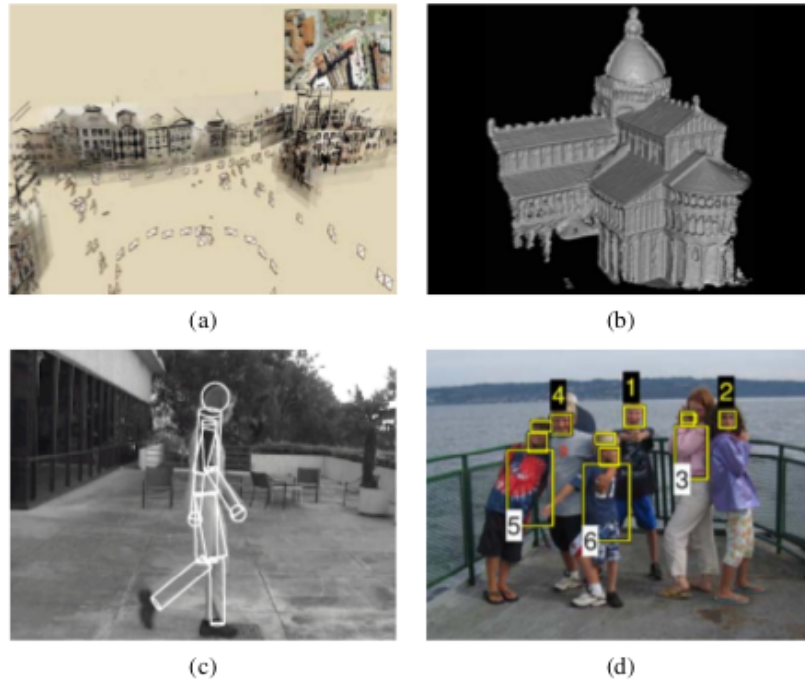


Figure 1.2: Some examples of computer vision algorithms and applications. (a) Structure from motion algorithms can reconstruct a sparse 3D point model of a large complex scene from hundreds of partially overlapping photographs [6]. (b) Stereo matching algorithms can build a detailed 3D model of a building from hundreds of differently exposed photographs taken from the Internet [7] (c) Person tracking algorithms can track a person walking in front of a cluttered background [8] (d) Face detection algorithms, coupled with color-based clothing and hair detection algorithms, can locate and recognize the individuals in this image [9]

Researchers in computer vision have been developing, in parallel, mathematical techniques for recovering the three-dimensional shape and appearance of objects in imagery. We now have reliable techniques for accurately computing a partial 3D model of an environment from thousands of partially overlapping photographs (Figure 1.2a). Given a large enough set of views of a particular object, we can create accurate dense 3D surface models using stereo matching (Figure 1.2b). We can track a person moving against a complex background (Figure 1.2c). We can even, with moderate success, attempt to find and name all of the people in a photograph using a combination of face, clothing, and hair detection and recognition (Figure 1.2d). However, despite all of these advances, the dream of having a computer interpret an image at the same level as a two-year old (for example, counting all of the animals in a picture) remains elusive.

Why is vision so difficult? In part, it is because vision is an inverse problem, in which we seek to recover some unknowns given insufficient information to fully specify the solution. We must therefore resort to physics-based and probabilistic models to disambiguate between potential solutions. However, modeling the visual world in all of its rich complexity is far more difficult than, say, modeling the vocal tract that produces spoken sounds.

The forward models that we use in computer vision are usually developed in physics (radiometry, optics, and sensor design) and in computer graphics. Both of these fields model how objects move and animate, how light reflects off their surfaces, is scattered by the atmosphere, refracted through camera lenses (or human eyes), and finally projected onto a flat (or curved) image plane. While computer graphics are not yet perfect (no fully computer-animated movie with human characters has yet succeeded at crossing the uncanny valley that separates real humans from android robots and computer-animated humans), in limited domains, such as rendering a still scene composed of everyday objects or animating extinct creatures such as dinosaurs, the illusion of reality is perfect.

In computer vision, we are trying to do the inverse, i.e., to describe the world that we see in one or more images and to reconstruct its properties, such as shape, illumination, and color distributions. It is amazing that humans and animals do this so effortlessly, while computer vision algorithms are so error prone. People who have not worked in the field often underestimate the difficulty of the problem. This misperception that vision should be easy dates back to the early days of artificial intelligence, when it was initially believed that the cognitive (logic proving and planning) parts of intelligence were intrinsically more difficult than the perceptual components (Boden 2006).

1.2 Applications

Computer vision is being used today in a wide variety of real-world applications, which include:

- **Optical character recognition (OCR):** reading handwritten postal codes on letters and automatic number plate recognition (ANPR)
- **Machine inspection:** rapid parts inspection for quality assurance using stereo vision with specialized illumination to measure tolerances on aircraft wings or auto body parts or looking for defects in steel castings using X-ray vision;
- **3D model building (photogrammetry):** fully automated construction of 3D models from aerial photographs used in systems such as Bing Maps;
- **Medical imaging:** registering pre-operative and intra-operative imagery or performing long-term studies of people's brain morphology as they age;
- **Automotive safety:** detecting unexpected obstacles such as pedestrians on the street, under conditions where active vision techniques such as radar or lidar do not work well.
- **Match move:** merging computer-generated imagery (CGI) with live action footage by tracking feature points in the source video to estimate the 3D camera motion and shape of the environment. Such techniques are widely used in Hollywood (e.g., in movies such as Jurassic Park); they also require the use of precise matting to insert new elements between foreground and background elements.
- **Surveillance:** monitoring for intruders, analyzing highway traffic (Figure 1.4f), and monitoring pools for drowning victims;
- **Fingerprint recognition and biometrics:** for automatic access authentication as well as forensic applications

One of the most prominent application fields is medical computer vision or medical image processing. This area is characterized by the extraction of information from image data for the purpose of making a medical diagnosis of a patient. Generally, image data is in the form of microscopy images, X-ray images, angiography images, ultrasonic images, and tomography images. An example of information which can be extracted from such image data is detection of tumours, arteriosclerosis or other malign changes. It can also be measurements of organ dimensions, blood flow, etc. This application area also supports medical research by providing new information, e.g., about the structure of the brain, or about the quality of medical treatments. Applications of computer vision in the medical area also includes enhancement of images that are interpreted by

humans, for example ultrasonic images or X-ray images, to reduce the influence of noise.

A second application area in computer vision is in industry, sometimes called machine vision, where information is extracted for the purpose of supporting a manufacturing process. One example is quality control where details or final products are being automatically inspected in order to find defects. Another example is measurement of position and orientation of details to be picked up by a robot arm. Machine vision is also heavily used in agricultural process to remove undesirable food stuff from bulk material, a process called optical sorting.

Military applications are probably one of the largest areas for computer vision. The obvious examples are detection of enemy soldiers or vehicles and missile guidance. More advanced systems for missile guidance send the missile to an area rather than a specific target, and target selection is made when the missile reaches the area based on locally acquired image data. Modern military concepts, such as "battlefield awareness", imply that various sensors, including image sensors, provide a rich set of information about a combat scene which can be used to support strategic decisions. In this case, automatic processing of the data is used to reduce complexity and to fuse information from multiple sensors to increase reliability.

One of the newer application areas is autonomous vehicles, which include submersibles, land-based vehicles (small robots with wheels, cars or trucks), aerial vehicles, and unmanned aerial vehicles (UAV). The level of autonomy ranges from fully autonomous (unmanned) vehicles to vehicles where computer vision based systems support a driver or a pilot in various situations. Fully autonomous vehicles typically use computer vision for navigation, i.e. for knowing where it is, or for producing a map of its environment (SLAM) and for detecting obstacles (Figure 1.3). It can also be used for detecting certain task specific events, e.g., a UAV looking for forest fires. Examples of supporting systems are obstacle warning systems in cars, and systems for autonomous landing of aircraft. Several car manufacturers have demonstrated systems for autonomous driving of cars, but this technology has still not reached a level where it can be put on the market. There are ample examples of military autonomous vehicles ranging from advanced missiles, to UAVs for recon missions or missile guidance. Space exploration is already being made with autonomous vehicles using computer vision, e.g., NASA's Mars Exploration Rover and ESA's ExoMars Rover (Figure 1.4).

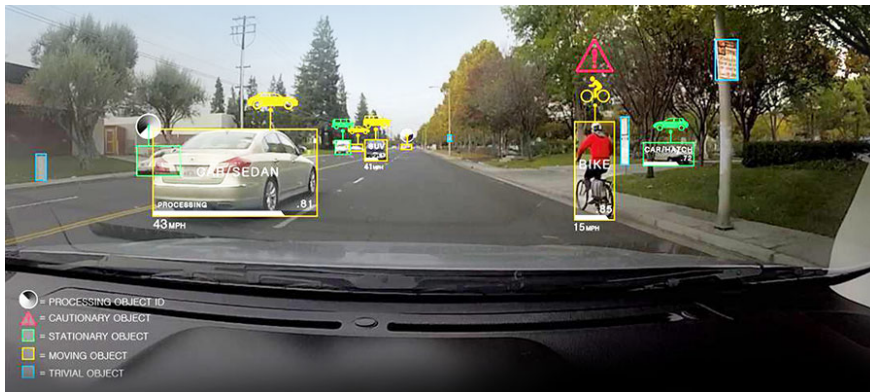


Figure 1.3: An autonomous vehicle that uses computer vision technology to see and understand nearby objects. *Source: Nvidia*



Figure 1.4: A prototype of the ExoMars Rover at the 2015 Cambridge Science Festival

1.3 Convolution Kernels

Convolution is an important operation in signal and image processing [10]. Convolution operates on two signals (in 1D) or two images (in 2D). A kernel, usually named convolution matrix or mask, is convolved with the image kernel (I.e. the matrix with the pixel values of the image) in order to produce the desired effect. Differently sized kernels containing different patterns of numbers produce different results under convolution. So, convolution takes two images as input and produces a third as output. Convolution is an incredibly important concept in many areas of math and engineering [5]. They are the building block for more complex computer vision algorithms, so they are presented first in order to ensure a solid understanding of the applications examined later in this thesis.

Definition

Beginning with 1D convolution, a 1D "image", is also known as a signal, and can be represented by a regular 1D vector. The convolution $f * g$ of f and g is defined as:

$$(f * g)(i) = \sum_{j=1}^m g(j)f(i - 1 + \frac{m}{2}) \quad (1.1)$$

where f is the input vector, g a convolution kernel and we assume that f has length n , and g has length m .

One way to think of this operation is that we're sliding the kernel over the input image. For each position of the kernel, we multiply the overlapping values of the kernel and image together, and add up the results. This sum of products will be the value of the output image at the point in the input image where the kernel is centered. Consider the following example.

Suppose the input 1D image is :

$$f = \begin{array}{|c|c|c|c|c|c|c|} \hline 10 & 50 & 60 & 10 & 20 & 40 & 30 \\ \hline \end{array}$$

and our kernel is:

$$g = \begin{array}{|c|c|c|} \hline 1/3 & 1/3 & 1/3 \\ \hline \end{array}$$

If we name the output image h , in order to compute $h(3)$, we slide the kernel so that it is centered around $f(3)$:

10	50	60	10	20	40	30
	1/3	1/3	1/3			

It can be assumed that the value of the input image and kernel is 0 everywhere outside the boundary, so the above could be rewritten as :

10	50	60	10	20	40	30
0	1/3	1/3	1/3	0	0	0

Then we multiply the corresponding (lined-up) values of f and g and add up the products.

Most of these products will be 0, except for at the three non-zero entries of the kernel. So the product is :

$$\frac{1}{3}50 + \frac{1}{3}60 + \frac{1}{3}10 = \frac{50}{3} + \frac{60}{3} + \frac{10}{3} = 40$$

Thus, $h(3) = 40$. From the above, it should be clear that what this kernel is doing is computing a windowed average of the image, i.e., replacing each entry with the average of that entry and its left and right neighbor. Using this intuition, the other values of h can be computed as well:

$$g = \begin{bmatrix} 20 & 40 & 40 & 30 & 20 & 30 & 23.333 \end{bmatrix}$$

For 2D convolution, just as before, the kernel is slid over each pixel of the image, multiplying the corresponding entries of the input image and kernel, and adding them up - the result is the new value of the image.

Figure 1.5 illustrates the result convolving an image with a kernel. In particular, a kernel that applies Gaussian blur in an image is used. The convolution matrix for this filter is:

$$\frac{1}{16} \begin{bmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{bmatrix}$$



Figure 1.5: (a) The original image (b) Image blurred with the gaussian matrix.

1.4 Feature Detection

One of the main targets of the field of Computer Vision is to extract features from images and supply the results as inputs to systems, in order to make important decisions (such as triggering an actuator to move a robot hand). This process is called feature detection and refers to all methods and operations that are necessary to calculate at every pixel of an image whether or not satisfies the criteria of each feature. The result is a subset of the image, containing either isolated points, continuous lines or connected regions. Although there is not a clear definition of the meaning of feature, usually we refer to feature as an interesting part of an image, which is repeated two or more times throughout the image.

There have been developed several feature detection algorithms, varying on the desired feature detected, the computational complexity and repeatability. We could divide them into the following groups:

- **Corners / Points of interest** : An interest point is a point in an image which has a well-defined position and can be robustly detected. This means that an interest point can be a corner but it can also be, for example, an isolated point of local intensity maximum or minimum, line endings, or a point on a curve where the curvature is locally maximal.
In practice, most so-called corner detection methods detect interest points in general, and in fact, the term "corner" and "interest point" are used more or less interchangeably through the literature [11]. As a consequence, if only corners are to be detected it is necessary to do a local analysis of detected interest points to determine which of these are real corners.
- **Edges** : With the term "edges" we refer to the locations in an image where there is a border (an edge) between two regions. There is no a predefined shape for an edge, since it can contain everything. To compute an edge, most algorithms rely on the fact that edges consist of sets that include pixels on an image that have high value gradient magnitude.
Hence, edges have one dimensional structure.
- **Blobs / Regions of Interest** : Blobs provide a complementary description of image structures in terms of regions, as opposed to corners that are more point-like. Nevertheless, blob descriptors may often contain a preferred point (a local maximum of an operator response or a center of gravity) which means that many blob detectors may also be regarded as interest point operators. Blob detectors can detect areas in an image which are too smooth to be detected by a corner detector.

- **Ridges** : In case we have stretched objects in an image, it is necessary to use ridges. A ridge could be described as an one-dimensional line that constitutes a symmetry axis and plus its width depends on the local ridge point. Nevertheless, calculating ridge points in gray-scale images is computationally heavier than detecting edges, corners or blobs.

In this thesis we are going to examine algorithms from the fields of corner and edge detection. A corner is calculated as an intersection of two edges or as a point where there are two strong and different edge directions in a local region around the point. Therefore, the basic steps of these feature detection techniques are similar and in order for a substantial understanding to be achieved, the following subsection constitutes a presentation of such algorithms.

1.4.1 Canny edge detection

The Canny edge detector is an edge detection operator that uses a multi-stage algorithm to detect a wide range of edges in images [12] .

Canny edge detection is a technique to extract useful structural information from different vision objects and dramatically reduce the amount of data to be processed. It has been widely applied in various computer vision systems. Canny has found that the requirements for the application of edge detection on diverse vision systems are relatively similar. Thus, an edge detection solution to address these requirements can be implemented in a wide range of situations. The general criteria for edge detection include :

- Detection of edge with low error rate, which means that the detection should accurately catch as many edges shown in the image as possible.
- The edge point detected from the operator should accurately localize on the center of the edge.
- A given edge in the image should only be marked once, and where possible, image noise should not create false edges.

To satisfy these requirements Canny used the calculus of variations - a technique which finds the function which optimizes a given functional. The optimal function in Canny's detector is described by the sum of four exponential terms, but it can be approximated by the first derivative of a Gaussian.

Among the edge detection methods developed so far, canny edge detection algorithm is one of the most strictly defined methods that provides good and reliable detection. Owing to its optimality to meet with the three criteria for edge detection and the simplicity of process for implementation, it became one of the most popular algorithms for edge detection.

Figures 1.6 and 1.7 show the result of canny edge detector applied in an image.

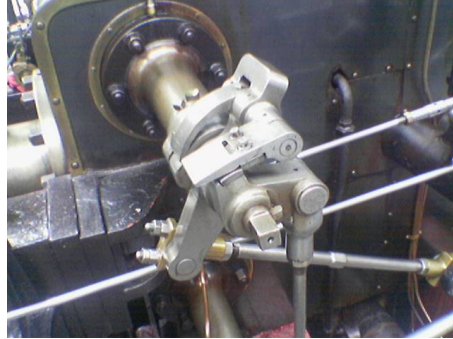


Figure 1.6: Photograph of a steam engine. *Source: Wikipedia*

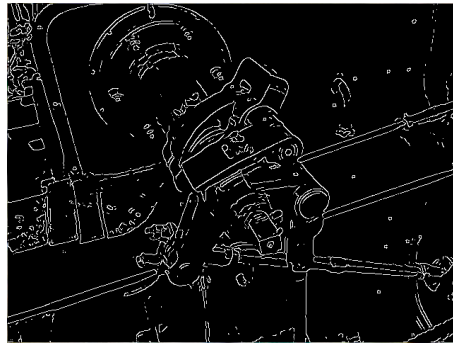


Figure 1.7: Canny edge detector applied on the above image. *Source: Wikipedia*

The process of Canny edge detection algorithm can be broken down to 5 different steps:

1. Apply Gaussian filter to smooth the image in order to remove the noise
2. Find the intensity gradients of the image
3. Apply non-maximum suppression to remove spurious response to edge detection
4. Apply double threshold to determine potential edges
5. Track edge by hysteresis: Finalize the detection of edges by suppressing all the other edges that are weak and not connected to strong edges.

These steps are described below in detail.

Gaussian filter

Since all edge detection results are easily affected by image noise, it is essential to filter out the noise to prevent false detection caused by noise. To smooth the image, a

Gaussian filter is applied to convolve with the image. This step will slightly smooth the image to reduce the effects of obvious noise on the edge detector. The equation for a Gaussian filter kernel of size $(2k + 1) \times (2k + 1)$ is given by:

$$H_{ij} = \frac{1}{2\pi\sigma^2} \exp\left(-\frac{(i - (k + 1))^2 + (j - (k + 1))^2}{2\sigma^2}\right) \quad (1.2)$$

, where $1 \leq i, j \leq (2k + 1)$.

Finding the intensity gradient of the image

An edge in an image may point in a variety of directions, so the Canny algorithm uses four filters to detect horizontal, vertical and diagonal edges in the blurred image. The edge detection operator returns a value for the first derivative in the horizontal direction (G_x) and the vertical direction (G_y). From this the edge gradient and direction can be determined:

$$G = \sqrt{G_x^2 + G_y^2}$$

$$\Theta = \text{atan2}(G_x, G_y)$$

, where where G can be computed using the hypot function and atan2 is the arctangent function with two arguments. The edge direction angle is rounded to one of four angles representing vertical, horizontal and the two diagonals.

Non-maximum suppression

Non-maximum suppression is an edge thinning technique.

Non-Maximum suppression is applied to "thin" the edge. After applying gradient calculation, the edge extracted from the gradient value is still quite blurred. With respect to criterion 3, there should only be one accurate response to the edge. Thus non-maximum suppression can help to suppress all the gradient values to 0 except the local maximal, which indicates location with the sharpest change of intensity value. The algorithm for each pixel in the gradient image is:

1. Compare the edge strength of the current pixel with the edge strength of the pixel in the positive and negative gradient directions.
2. If the edge strength of the current pixel is the largest compared to the other pixels in the mask with the same direction (i.e., the pixel that is pointing in the y direction, it will be compared to the pixel above and below it in the vertical axis), the value will be preserved. Otherwise, the value will be suppressed.

Double threshold

After application of non-maximum suppression, the edge pixels are quite accurate to present the real edge. However, there are still some edge pixels at this point caused by noise and color variation. In order to get rid of the spurious responses from these bothering factors, it is essential to filter out the edge pixel with the weak gradient value and preserve the edge with the high gradient value. Thus two threshold values are set to clarify the different types of edge pixels, one is called high threshold value and the other is called the low threshold value. If the edge pixel's gradient value is higher than the high threshold value, they are marked as strong edge pixels. If the edge pixel's gradient value is smaller than the high threshold value and larger than the low threshold value, they are marked as weak edge pixels. If the pixel value is smaller than the low threshold value, they will be suppressed. The two threshold values are empirically determined values, which will need to be defined when applying to different images.

Edge tracking by hysteresis

So far, the strong edge pixels should certainly be involved in the final edge image, as they are extracted from the true edges in the image. However, there will be some debate on the weak edge pixels, as these pixels can either be extracted from the true edge, or the noise/color variations. To achieve an accurate result, the weak edges caused by the latter reasons should be removed. Usually a weak edge pixel caused from true edges will be connected to a strong edge pixel while noise responses are unconnected. To track the edge connection, blob analysis is applied by looking at a weak edge pixel and its 8-connected neighborhood pixels. As long as there is one strong edge pixel is involved in the blob, that weak edge point can be identified as one that should be preserved.

1.4.2 Moravec detector

One of the first efforts in corner detection was Moravec detection algorithm which determines a corner as a point of low self-similarity [10]. The algorithm checks whether the neighborhood of a centered pixel resembles with the other local pixels, by computing the sum of squared differences between the two sections. If the sum has a low value, then the algorithm implies more similarity. So, if the pixel has intensity value that is similar to its neighbors, then the regions will not differ. However, if the pixel belongs to an edge, then it is obvious that the two regions that are in vertical direction to the edge will have strong differences in intensity values, whereas in a parallel direction there would be plenty of similarities. If the pixel belongs in a section that intensity values vary in all directions, then all of the neighborhood patches will look different. Hence, the corner strength is defined as the lowest sum of squared differences (SSD) between the region of the centralized pixel and its neighbors in all directions (horizontal, vertical and the

two diagonals). If the value of SSD is a local maximum, then that point is considered to be an point of interest. Nevertheless, the Moravec detector has a *strong drawback*: it is not isotropic, meaning that if there is an edge that is in a different direction of its neighbors, then the smallest SSD will be high and thus the edge will be considered a corner incorrectly.

The mathematical equation that represents this relationship is the following:

$$E(u, v) = \sum w(x, y) \times [I(x + u, y + v) - I(x, y)] \quad (1.3)$$

where:

- E is the computed sum of square differences
- $W(x,y)$ is the window function
- $I(x,y)$ is the intensity of the pixel
- $I(x+u,y+v)$ is the shifted intensity

We compute shifted intensity in four directions: $(u,v) = \{ (1,0), (1,1), (0,1), (-1,1) \}$.

The algorithm searches for the local maximal in $\min \{E(u,v)\}$. The mathematical formula of the Moravec detector confirms the problems mentioned above:

- because of the binary window function, the response of the algorithm is very vulnerable to noise.
- The step of the operator is 45 degrees and thus important information is eliminated.
- Only the minimum value of $E(u,v)$ is taken into account

1.4.3 The Harris & Stephens / Plessey / Shi-Tomasi corner detection algorithm

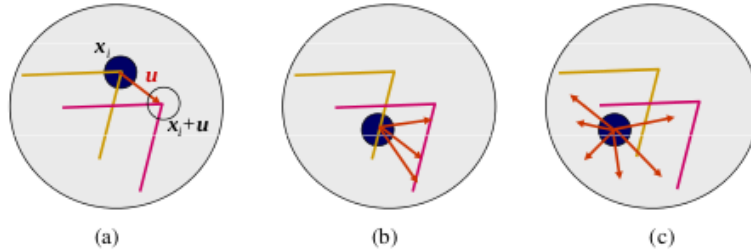


Figure 1.8: Aperture problems for different image patches: (a) stable (“corner-like”) flow; (b) classic aperture problem (barber-pole illusion); (c) textureless region. The two images I_1 (yellow) and I_2 (red) are overlaid. The red vector u indicates the displacement between the patch centers and the $w(x_i)$ weighting function patch window is shown as a dark circle. Figure reproduced from [5]

After having seen the problems that can occur in feature detection, it is normal ask what are good features to track [1, 13]. Patches of an image that have large contrast changes (gradients) are easier to localize, although straight line segments at a single orientation suffer from the aperture problem [14] , i.e., it is only possible to align the patches along the direction normal to the edge direction (Figure 4.4b). Patches with gradients in at least two (significantly) different orientations are the easiest to localize, as shown schematically in Figure 4.4a.

These intuitions can be formalized by looking at the simplest possible matching criterion for comparing two image patches, i.e., their (weighted) summed square difference, as shown in equation 1.3 .

Harris and Stephens[2] use the same function for this purpose but improved Moravec’s detector by taking into account the differential value of the corner, regarding the direction directly and not using shifted regions.

An important development compared with Moravec’s operator is that Harris algorithm considers all small shifts and not with 45 degree step. Thus, a Taylor expansion is used to compute $I(u + x, v + y)$ approximately. Letting I_x and I_y be the partial derivatives of the Intensity of an image, we have:

$$I(u + x, v + y) \approx I(u, v) + I_x(u, v)x + I_y(u, v)y \quad (1.4)$$

Thus, the sum expression becomes :

$$S(x, y) \approx \sum \sum w(u, v) \times (I(u, v) + I_x(u, v)x + I_y(u, v)y)^2 \quad (1.5)$$

The previous relationship can be written in a matrix form as follows :

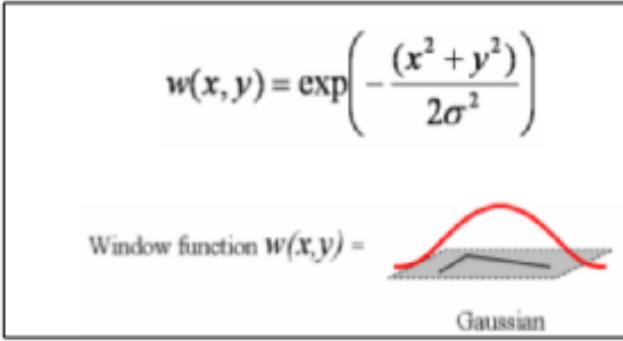
$$S(x, y) \approx (xy)A \begin{pmatrix} x \\ y \end{pmatrix} \quad (1.6)$$

where A is the structure tensor:

$$A = \sum_u \sum_v w(u, v) \begin{bmatrix} I_x^2 & I_x I_y \\ I_x I_y & I_y^2 \end{bmatrix} = \begin{bmatrix} \langle I_x^2 \rangle & \langle I_x I_y \rangle \\ \langle I_x I_y \rangle & \langle I_y^2 \rangle \end{bmatrix} \quad (1.7)$$

The auto-correlation matrix A provides a lower bound on the uncertainty in the location of a matching patch. It is therefore a useful indicator of which patches can be reliably matched.

Another improvement comparing to Moravec detector is that the window function is now *Gaussian*, guaranteeing isotropic response. The general form of a Gaussian window function is presented below:



$w(x, y) = \exp\left(-\frac{(x^2 + y^2)}{2\sigma^2}\right)$

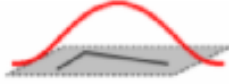
Window function $w(x, y) =$ 
 Gaussian

Figure 1.9: Gaussian window function

In Harris' algorithm, a corner is considered to have a large variation of S in all of the directions of the vector (x y) . In mathematical form, this can be expressed through the eigenvalues of the matrix A. If an interest point is examined, then matrix A should have two eigenvalues with grand value.

Considering the magnitudes of the eigenvalues, we can determine the following cases:

- $\lambda_1 \approx 0$ and $\lambda_2 \approx 0$, then this point is not of interest.
- $\lambda_1 \approx 0$ and λ_2 has a grand positive value, then at this point there is an edge
- Both λ_1 , λ_2 have grand positive values, then at this point we have found a corner.

Nevertheless, because computing the eigenvalues requires a big workload, Harris and Stephens suggested an alternative function M_c which is presented below:

$$M_c = \lambda_1 \lambda_2 - \kappa(\lambda_1 + \lambda_2)^2 = \det(A) - \kappa \text{trace}^2(A) \quad (1.8)$$

where $\det(M) = \lambda_1 \lambda_2$, $\text{trace}(M) = \lambda_1 + \lambda_2$ and κ is a values chosen depending on the required level of sensitivity in the response. Accepted values of κ are between 0.04 and 0.15

Unlike eigenvalue analysis, this quantity does not require the use of square roots and yet is still rotationally invariant and also downweights edge-like features where $\lambda_2 \gg \lambda_1$. Triggs [1] suggests using the quantity

$$\lambda_1 - \kappa \lambda_2$$

which also reduces the response at edges, where aliasing errors sometimes inflate the smaller eigenvalue. He also shows how the basic 2×2 Hessian can be extended to parametric motions to detect points that are also accurately localizable in scale and rotation.

A graphical representation of the classification of image points, according to the eigenvalues of M is shown below:

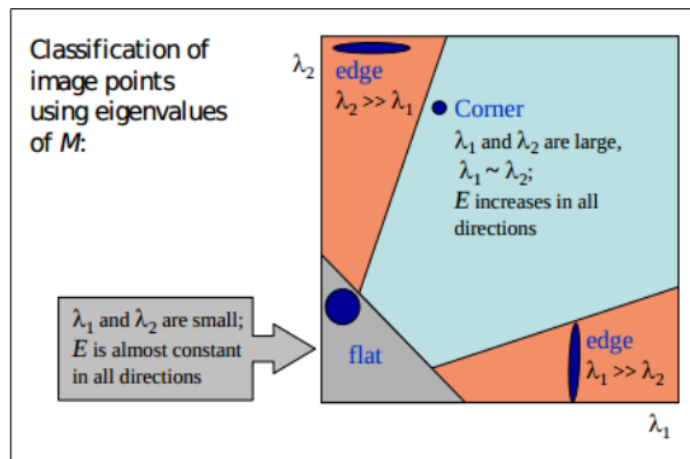


Figure 1.10: Classification of Image points. [2]

The steps in the basic auto-correlation-based keypoint detector are summarized in the following algorithmic stages :

1. Compute the horizontal and vertical derivatives of the image I_x and I_y by convolving the original image with derivatives of Gaussians
2. Compute the three images corresponding to the outer products of these gradients. (The matrix \mathbf{A} is symmetric, so only three entries are needed)
3. Convolve each of these images with a larger Gaussian.
4. Compute a scalar interest measure using one of the formulas discussed above
5. Find *local maxima* above a certain threshold and report them as detected feature point locations.

Chapter 2

Embedded Architecture Presentation

The embedded system used for the implementation of the examined algorithms is Myriad2 by Movidius, the world's first Vision Processing Unit. Myriad2 is a multicore, always-on System on Chip that supports Computational Imaging and Visual Awareness for mobile, wearable and embedded applications.

This chapter describes the functionality and use of the Myriad 2 multiprocessor SoC. In addition, it introduces details regarding the architecture of this chip as well as the programming paradigms it supports.

2.1 Myriad 2 Architecture

2.1.1 List of basic features

The Myriad 2 SoC device family offers twelve SHAVE vector processors with two 32-bit RISC (LEON) to provide performance efficiency and flexibility. A brief overview of the Myriad 2 common features are presented below. All of the components used for the implementation of the examined algorithms will be thoroughly explored later in this chapter.

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

- Imaging interfaces, such as MIPI, CIF, LCD

The Myriad 2 family consists of the following socket revisions:

- MA2x5x - MA2150 / MA2155 / MA2450 / MA2455

The one used for the programming purposes of this thesis is the MA2150 revision, which has 128 MB DDR memory and a system clock of 600 MHz. Its architecture diagram is presented in the following figure:

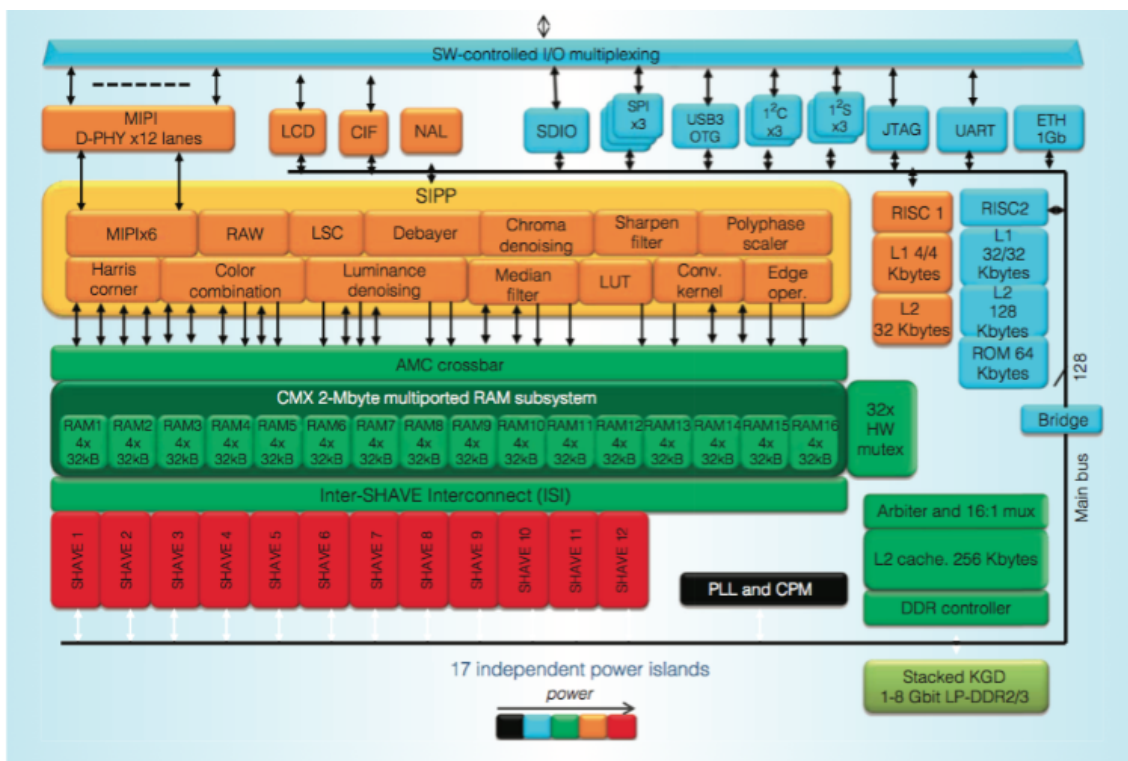


Figure 2.1: Myriad 2 vision processing unit (VPU) system on chip (SoC) block diagram shows the 12 SHAVE cores and associated Inter-SHAVE Interconnect, located below the multicore memory subsystem (connection matrix or CMX). Above the CMX are the hardware accelerators for computer vision and image processing controlled by a first reduced-instruction-set computing (RISC) processor, and above that again are the I/O peripherals, which are controlled by a second RISC processor. Finally, all of the processors in the system share access to the 64-bit DRAM interface. Figure reproduced from [3].

2.1.2 Myriad 2 Block Level Architecture Overview

Application processor systems on chip (SoCs) are typically based on one or more 32-bit reduced-instruction-set computing (RISC) processor surrounded by hardware acceler-

ators that share a common multilevel cache and DRAM interface. This shared infrastructure is attractive from a cost perspective but creates major bottlenecks in terms of memory access, where multimedia applications demand real-time performance but must contend with user applications and a platform OS such as Android. Thus, the platform often underdelivers in terms of computational video performance or power efficiency. A more attractive model is to build a software programmable architecture as a coprocessor to an application processor, rather than using fixed-function or configurable hardware. Such a coprocessor can then abstract away a lot of the hard real-time requirements in dealing with multiple camera sensors and accelerometers from the application processor and making the entire ensemble appear as a single super-intelligent MIPI camera below the Android camera's hardware abstraction layer. As a result, the architecture presented in [3] focuses on power-efficient operation, as well as area efficiency, allowing product derivatives to be implemented entirely in software where previously hardware and mask costs would have been incurred. The software programmable model is especially interesting in areas where standards do not exist, such as body-pose estimation.

As a VPU SoC, Myriad 2 has a software-controlled, multicore, multiported memory subsystem and caches that can be configured to allow handling of a large range of workloads and provide high, sustainable on-chip data and instruction bandwidth to support the 12 processors, two RISC processors, and high-performance video hardware accelerator filters. A multichannel (or multiagent) direct memory access engine offloads data movement between the processors, hardware accelerators, and memory, and a large range of peripherals including cameras, LCD panels, and mass storage, communicate with processors and hardware accelerators. Additional programmable hardware acceleration helps speed up hard-to-parallelize functions required by video codecs, such as H.264 CABAC, VP9, and SVC, as well as video processing kernels for always-on computer-vision applications. Up to 12 independent high-definition cameras can be connected to 12 programmable MIPI D-PHY lanes supporting CSI-2 organized in six pairs, each of which can be independently clocked, to provide an aggregate bandwidth of 18 Gbits per second. The device also contains a USB 3.0 interface and a gigabit Ethernet media access controller, as well as various interfaces such as Serial Peripheral Interface (SPI), Inter-Integrated Circuit (I^2C), and Universal Asynchronous Receiver Transmitter (UART), connected to a reduced number of I/O pins using a tree of software-controlled multiplexers. In this way, these interfaces support a broad range of use cases in a low-cost plastic ball-grid array package with integrated 2 to 4 Gbit low-power DDR2/3 synchronous DRAM stacked in package using a combination of flip-chip bumping for the VPU die and wire bonding for the stacked DRAM.

Because power efficiency is paramount, the device employs a total of 17 power is-

lands, including one for each of the 12 integrated SHAVE processors, allowing very fine-grained power control in software. The device supports 8-, 16-, 32-, and some 64-bit integer operations as well as fp16 (Open-EXR) and fp32 arithmetic, and it can aggregate 1,000 Gflops (fp16). The resulting architecture offers increased performance per watt across a broad spectrum of computer vision and computational imaging applications from augmented reality to simultaneous localization and mapping.

As Figure 2.1 shows, there are three major architectural units in the Myriad 2 processor: The Media Sub System (MSS), the CPU Sub system (CSS) and the Microprocessor Array (UPA).

The Media Sub System (MSS)

The MSS is designed for two main actions, allowing external connections with imaging devices as well as allowing use of the HW filters available in Myriad 2. As such it is comprised by the MIPI, LCD, CIF interfaces, the SIPP HW filters and well as the AMC block which enables connections between these and CMX (SRAM) memory.

The Leon RT RISC is offered as part of the MSS in order to be used for coordinating frame input and controlling the pipelines set in place. Leon RT (LRT) is a RISC processor with a fair amount of L2 cache memory (32 KB). Leon RT is only one arbiter away from any Interface or HW filter register settings so it can efficiently change any required parameters of the MSS blocks with the minimum amount of delay due to bus arbitration.

The CPU Sub System (CSS)

The CSS have been designed to be the main communication and control unit with the outside world via the external communication peripherals: I2C blocks, I2S blocks, SPI blocks, UART, GPIO, ETH and USB3.0. The control unit of this block is the Leon OS (LOS) RISC processor, but in this block the Leon owns much bigger L1 (32 KB) and L2 (256 KB) caches, which allows to put a modern RTOS on it. This block also offers an AHB DMA engine for more optimal data transfer via the external peripherals. Beside handling the external interfaces and communication Leon OS could also control SHAVE processors imaging algorithms.

The Microprocessor Array (UPA)

The UPA is the design unit in Myriad 2 holding the 12 VLIW SHAVE vector processors, the 2 MB CMX SRAM memory and a few other blocks from which we list: the

specialized DMA engine, the 256 KB L2 cache memory available to the SHAVE cores. This design unit's main purpose is to provide support for customized code required by many imaging or computer vision applications as well as any other general computation intensive algorithms.

Each SHAVE processor has preferential ports into a 128 KB slice of the CMX memory, which will be detailed in a following subsection. As such, $12 \times 128 \text{ KB} = 1536 \text{ KB}$ are preferentially used by SHAVE cores but the remaining 512 KB of CMX memory are generally usable by any other resources. The recommended usage for these 512 KB is for HW SIPP filters usage or Leon OS timing critical code which would otherwise not be able to be kept in DDR.

2.1.3 Streaming Hybrid Access Vector Engine

Steaming Hybrid Access Vector Engines, or SHAVEs, are used for the bulk of the processing. Each SHAVE contains wide and deep register files coupled with a Very Long Instruction Word (VLIW) that provides a method for instruction level parallelism to be achieved. VLIW packets control multiple functional units which have SIMD capability for high parallelism and throughput at a functional unit and processor level. Each of these units can be launched in parallel in a single instruction packet.

The SHAVE processor is a hybrid stream processor architecture combining the best features of GPUs, DSPs, and RISC with both 8-, 16-, and 32-bit integer and 16- and 32-bit floating-point arithmetic as well as unique features such as hardware support for sparse data structures [3]. The architecture maximizes performance per watt while maintaining ease of programmability, especially in terms of support for design and porting of multicore software applications.

As Figure 2.2 shows, VLIW packets control multiple functional units that have SIMD capability for high parallelism and throughput at a functional unit and processor level. The functional units are the predicated execution unit (PEU), the branch and repeat unit, two 64-bit load-store units (LSU0 and LSU1), the 128-bit vector arithmetic unit (VAU), the 32-bit scalar arithmetic unit, the 32-bit integer arithmetic unit, and the 128-bit compare move unit (CMU), each of which is enabled separately by a header in the variable-length instruction. The functional units are fed with operands from a $128\text{-bit} \times 32\text{-entry}$ vector register file with 12 ports and a $32\text{-bit} \times 32\text{-entry}$ general register file with 18 ports delivering an aggregate 1,900 Gbytes per second (GBps) of SHAVE register bandwidth across all 12 SHAVEs at 600 MHz. The additional blocks in the diagram are the instruction decode and the debug control unit.

A constant instruction fetch width of 128 bits for variable-length instructions - with an average instruction width of around 80 bits - packed contiguously in memory and the five-entry instruction prefetch buffer guarantee that at least one instruction is always

ready while taking account of branches.

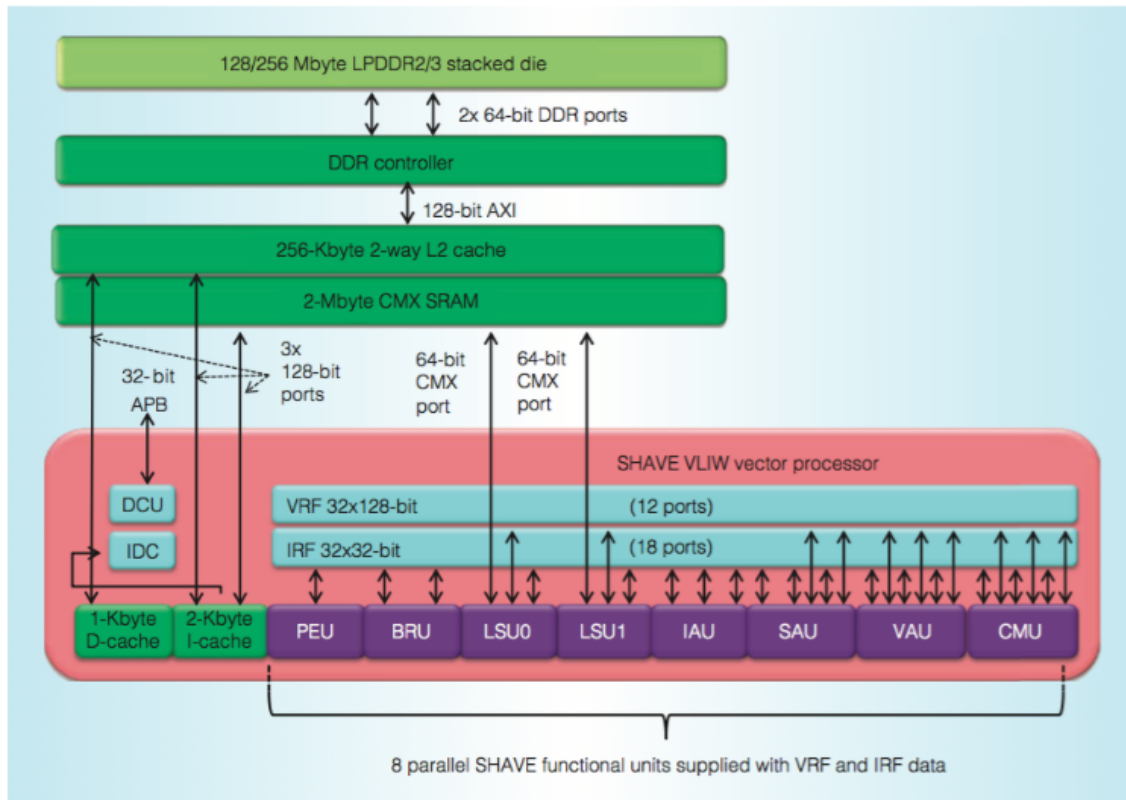


Figure 2.2: The Streaming Hybrid Architecture Vector Engine (SHAVE) processor microarchitecture is an eight-slot very long instruction word processor capable of performing multiple 128-bit vector operations in parallel with multiple load/store, scalar floating-point, integer, and control-flow operations in a single clock cycle. Figure reproduced from [3].

SHAVE supports SIMD instructions on multiple types, including but not limited to: 16 bits integer, 32 bits integer, 16 bits float, 32 bits float, 8 bits integers.

Applications can be developed in Assembly, C and C++ and run at the SHAVE through the use of moviAsm and moviCompile, both of which are Movidius internally developed tools.

There are two register file arrays: IRF and VRF which are described shortly below. The VAU, SAU, IAU, PEU, BRU and LSUs are also detailed.

IRF (Integer Register File)

The IRF consists of 32 registers, each 32 bits in length. These registers are implemented mainly to support integer operations, but are also used for load and store instructions. The execution units operating with these registers are the IAU (Integer Arithmetic Unit), and the SAU (Scalar Arithmetic Unit). There are SIMD operations operating with 16 bits and 8 bits integer data types performed by the SAU on the IRF.

VRF (Vector Register File)

The VRF consists of 32 registers, each 128 bits in length. The purpose of these registers is to provide SIMD operations to the SHAVE.

The arithmetic unit operating with VRF is the VAU (Vector Arithmetic Unit). This supports both integer and floating point operations directed towards multiple data types: 8, 16, 32-bit data types of integer or floating point.

IAU (Integer Arithmetic Unit)

This unit provides integer operation support on the IRF registers as well as support for different shifting and logic operations.

SAU (Scalar Arithmetic Unit)

This unit provides floating point operations support on the IRF.

Besides the most common floating point operations, this unit implements a few more complex functions on 16 bits floating point including: reciprocal, sine, square root, reciprocal of square root, cosine, arctangent, logarithm and exponential.

The unit also provides integer operations on the IRF registers. This feature may be used to launch more integer operations in parallel on the IRF if found useful.

VAU (Vector Arithmetic Unit)

This unit provides both floating point and integer operations on the VRF registers using 8, 16, and 32-bit data types of both integer or floating point.

CMU (Compare and Move Unit)

This unit provides functionality to copy (move) data from one register file to the other. Any combination is possible and multiple bit lengths are supported.

The unit also provides functionality for comparing different data types. Comparison is done setting a Condition Code register with multiple entries. This allows for comparisons to be made on VRF registers too, comparing multiple data at once.

LSU (Load Store Unit)

There are two load store units which provide functionality for loading and storing data to both register files. The LSU, used in conjunction with other units can also provide swizzling operations on various data types as described in the SHAVE ISA document.

BRU (Branching Unit)

The BRU provides functionality for branching. The SHAVE has a delay slot of 6 cycles which may be used to fill in other instructions.

PEU (Predicated Execution Unit)

The PEU is helpful for implementing conditional branching and also to make conditional stores on LSU or VAU units.

2.1.4 Memory subsystem

Myriad 2 offers an intelligent memory fabric that enables maximum processing with ultra-low power and minimum data movement [3].

The ability to combine image processing and computer vision processing into pipelines consisting of hardware and software elements was a key requirement because current application processors are limited in this respect. Thus, sharing data flexibly between SHAVE processors and hardware accelerators via the multiported memory subsystem was a key challenge in the design of the Myriad 2 VPU. In the 28-nm Myriad 2 VPU, 12 SHAVE processors, hardware accelerators, shared data, and SHAVE instructions reside in a shared 2-Mbyte memory block called Connection Matrix (CMX) memory, which can be configured to accommodate different instruction and data mixes depending on the workload.

The CMX block comprises 16 blocks of 128 Kbytes, which in turn comprise four 32-Kbyte RAM instances organized as 4,096 words of 64 bits each, which are independently arbitrated, allowing each RAM block in the memory subsystem to be accessed independently. The 12 SHAVEs acting together can move (theoretical maximum) 12×128 bits of code and 24×64 bits of data, for an aggregate CMX memory bandwidth of 3,072 bits per cycle (1,536 bits of data). This software-controlled multicore memory subsystem and caches can be configured to allow many workloads to be handled, providing high sustainable on-chip bandwidth with a peak bandwidth of 307 GBps (sixty-four 64-bit ports operating at 600 MHz) to support data and instruction supply to the 12 SHAVE processors and hardware accelerators (see Figure 2.1).

Furthermore, the CMX subsystem supports multiple traffic classes from latency-tolerant hardware accelerators to latency-intolerant SHAVE vector processors, allowing construction of arbitrary pipelines from a mix of software running on SHAVEs and hardware accelerators, which can operate at high, sustained rates on multiple streams simultaneously without performance loss and at ultra-low-power levels.

2.1.5 Myriad 2 implementation

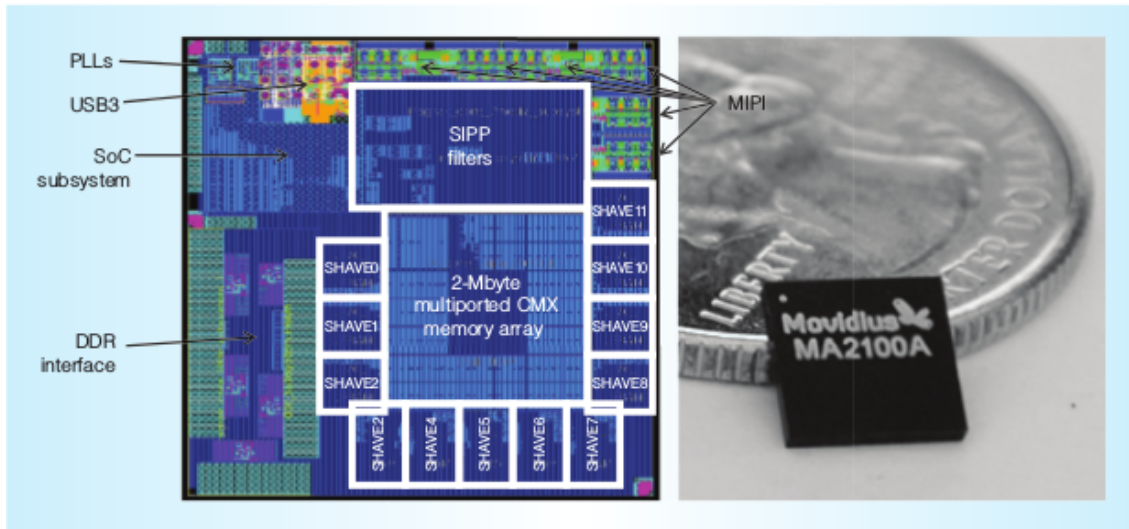


Figure 2.3: Myriad 2 VPU SoC die plot and 6 x 6 mm ball-grid array packaging. In the die plot, the SHAVE processors and hardware accelerators are clustered around the CMX memory system. On the outer rim of the die, the DRAM interface and MIPI, USB, and other peripherals as well as phase-locked loops dominate the periphery of the die, with the remainder of the die being occupied by the RISC processor and peripheral subsystems. Figure reproduced from [3].

Figure 2.3 shows a die plot of the 27mm^2 device, highlighting the major functional blocks [3]. Because power efficiency is paramount in mobile applications, Myriad 2 provides extensive clock and functional unit gating and support for dynamic clock and voltage scaling for dynamic power reduction. It also contains 17 power islands - one for each of the 12 SHAVE processors; one for each of the 12 SHAVE processors; one for the CMX memory subsystem; one for the media subsystem, including video hardware accelerators and RISC1; one for RISC2 and peripherals; one for the clock and power management; and one always-on domain.

This allows fine-grained power control in software with minimal latency to return to normal operating mode, including maintenance of the static RAM (SRAM) state that eliminates the need to reboot from external storage. The device has been designed to operate at 0.9 V for 600 MHz.

2.2 Myriad 2 Programming Paradigms

There are three different programming paradigms supported by Myriad 2 platform. Each one offers features suitable for specific applications and is selected based on the relevant requirements.

Standard Programming Paradigm

The standard programming paradigm for Myriad 2 involves using RTEMS running on LeonOS and the SIPP scheduler on Leon RT. In this way, it provides parallelization in an environment that is easily used and configured. The SIPP scheduler is designed to ensure parallel pipeline configurations for managing the HW filters and exterior interfaces with a low footprint. Hence, LeonRT optimized utilization is guaranteed.

The number of SHAVEs used for SIPP applications is configurable, so those that are not used for line based pipelines will remain free to be used by the RTEMS operating system running on Leon OS for various other purposes such as computer vision algorithms.

The One Leon Programming Paradigm

This paradigm is suitable for applications that might not require heavy line based processing. Such applications might choose to make the Leon RT processor completely inactive and instead use only the LeonOS with or without RTEMS. HW filters can still be used in this paradigm. In this programming model, Leon OS would control all of the applications running on the 12 SHAVE cores.

Bare Metal Programming Paradigm

It is often wanted by developers to write applications which will not be affected by any operating system overhead. For this reason, a bare metal programming paradigm is also supported by Myriad 2. This allows the usage of both LEON cores without any operating system. Only minimal schedulers are required in order to control the pipelines application.

Even though this paradigm requires more integration efforts, it offers the developers a model in which operating systems cost is absent.

2.3 Streaming Image Processing Pipeline

2.3.1 Introduction to the SIPP framework

The model used by many image processing libraries, such as OpenCV, consists of performing whole frame operations in series. This leads to high usage of DDR memory since frames need to be read from and written to the main memory between operations. Even though platforms with large CPU cache sizes can support this model, it is not suitable for embedded system where memory size as well as power usage are limiting factors. Therefore, a different approach is followed by Myriad2 which aims to maximize the usage of available resources.

The model used by SIPP framework consists of a graph of connected filters. Image data is read from the DDR to the CMX memory via DMA filters and after the processing the result is written back to the DDR. The processing is achieved by streaming data from one filter to the other in a scanline-by-scanline basis. The buffers used to hold the processed lines are located in the low-latency CMX memory, thus avoiding the need for DDR accesses except for those in the first and the last stage of the graph. Hence, benefits are gained by the SIPP framework regarding the performance as well as the power drain of the developed applications.

2.3.2 Filter Graphs

Processing under the SIPP framework is performed by filters. Applications construct pipelines consisting of filter nodes linked together in a DAG (Directed Acyclic Graph). Each filter is coupled with one or more output buffers. The output buffer stores the processed data output by the filter, and can store zero or more lines of data (zero lines in the case of a sink filter). When a filter is invoked, it produces at least one new line of data in its output buffer. For example, a 5x5 convolution filter requires at least 5 lines to function and outputs 1 line in each iteration. Lines are added to the output buffer in a circular fashion: the lines are written at increasing addresses, until the end of the buffer is reached, at which point the output position wraps back to the start of the buffer.

The filters used in the SIPP framework can be either software filters that run on SHAVEs or hardware filters. The latter are part of the Myriad2 SoC and can be used for computationally expensive ISP and computer vision tasks. A pipeline can be a mixture of software and hardware filters. Multiple instances of the same software filter can be used in a pipeline, whereas HW filters may be used only once.

The programmable hardware accelerators implemented in the Myriad 2 VPU include a poly-phase resizer, lens shading correction, harris corner detector, Histogram of Ori-

ented Gradients edge operator, convolution filter,sharpening filter, gamma correction, tone mapping, and luminance and chrominance denoising. Each accelerator has multiple memory ports to support the memory requirements and local decoupling buffers to minimize instantaneous bandwidth to and from the 2-Mbyte multicore memory subsystem. A local pipeline controller in each filter manages the read and writeback of results to the memory subsystem. The filters are connected to the multicore memory subsystem via a crossbar, and each filter can output one fully computed pixel per cycle for the input data, resulting in an aggregate throughput of 600 Mpixels per second at 600 MHz. [3]

Even though the bulk of the processing is performed by the SHAVE cores and the hardware accelerators, the SIPP framework runs on a RISC processor.

Below can be seen some simple pipeline examples :

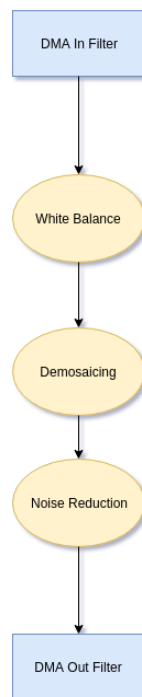


Figure 2.4: Basic stages of an sRGB pipeline

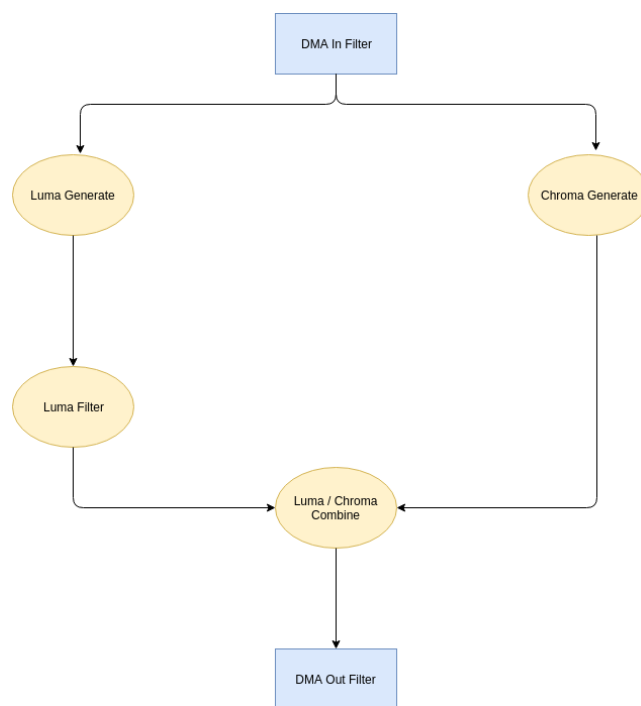


Figure 2.5: Separate Processing of Luminance and Chrominance of an image

In these examples DDR memory is used in the DMA transfers. However, it is possible to construct a pipeline in which data is processed in a streaming fashion, using only local memory (CMX). This mode of operation, which doesn't require DDR, is known as inline processing. In the following example, data coming from a camera is stored in a local memory buffer by the Mipi Rx filter (in the Mipi Rx filter's output buffer). The processed data is then transmitted directly from the Chroma/Luma combine filter's output buffer by the Mipi Tx filter. Thus, DDR memory can be completely omitted.

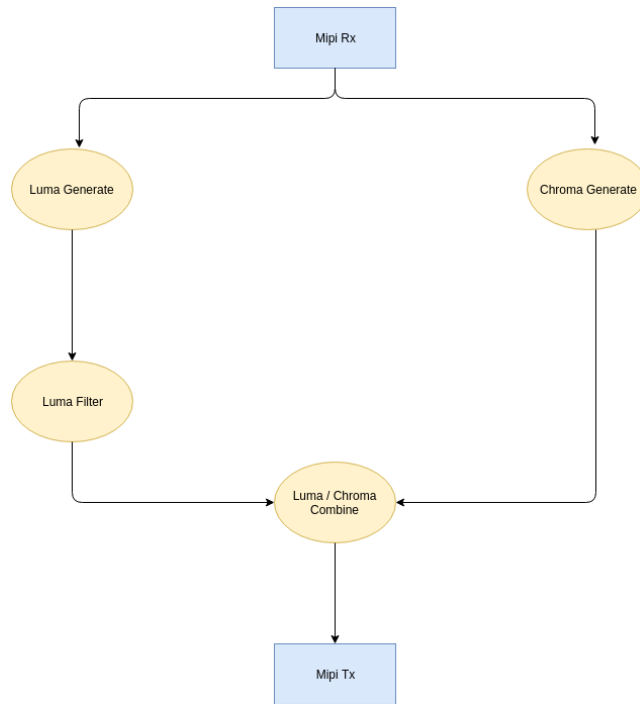


Figure 2.6: Separate Processing of Luminance and Chrominance without using DDR, streaming from MIPI sensor

2.4 Myriad 2 applications

The capabilities that were described previously make this embedded platform ideal for various computer vision applications, as stated in [3].

The SHAVE DSP supports streaming workloads from the ground up, making decisions about pixels or groups of pixels as they are processed by the 128-bit VAU. Because 128-bit comparison using the CMU and predication using the PEU can be performed in parallel with the VAU, higher performance can be achieved compared to a GPU in which decisions must be made about streaming data because GPUs suffer from performance loss due to branch divergence.

For example, the SHAVE processor excels when processing the FAST9 algorithm, where 25 pixels on a Bresenham circle around the center pixel must be evaluated for each pixel in, for instance, a 1080p frame. The FAST9 algorithm looks for nine contiguous pixels out of 25 on a Bresenham circle around the center that are above or below a center-pixel luminance value, meaning hundreds of operations must be computed for each pixel in a high-definition image. This requires hundreds of instructions on a scalar processor, and performance optimization requires the use of machine learning and training to improve detector performance [15].

In general, the application domain of Myriad 2 is Intelligent Machine Vision with some examples being (*Source: www.movidius.com*) :

Robotics

Drones and household robots are increasingly small and affordable enough to become serious consumer product categories. As new types of service, companion and collaborative robots emerge, these devices are demanding visual intelligence in order to navigate, understand and proactively assist us in our daily lives. Movidius provides the platform to create visually intelligent drones and robots without sacrificing size, battery life or performance.

Augmented & Virtual Reality

Virtual Reality (VR) and Augmented Reality (AR) devices are hitting the market and technological demands on the hardware are huge: gesture recognition, head tracking and object recognition are just a few of the necessary technologies to convincingly blend the real world with the digital. Myriad 2 allows VR and AR devices to crunch huge amounts of data at low power and ultra-low latency, two absolute musts in compact, immersive head-worn devices.

Wearables

Wearables are emerging as a category of devices that can augment our lives in meaningful ways. By passively filtering visual information and acting on cues relevant to their user, the dream of a truly capable digital assistant is in sight. Ultra-low power, high performance vision processors mean that even the smallest wearable devices can benefit from visual intelligence. The Myriad 2 platform allows devices to remain small and battery efficient, yet provide powerful new applications based on the rich variety of visual information available as users go about their daily lives.

Smart Security

Security and surveillance technology is getting a huge boost from visual intelligence. Imagine, a doorbell camera that not only alerts you to a visitor, but has already identified them as a courier. Visually intelligent cameras can detect fires from heat maps and alert authorities long before a fire builds up enough smoke to trigger a smoke detector; and motion detection cameras will be able to differentiate potential burglar from house pet. By bringing Myriad's visual intelligence to our security and surveillance, these new systems can detect and then intelligently act on data in real-time, providing safe and personalized security to homeowners and businesses alike.

Chapter 3

Evaluation of basic Computer Vision filters in Myriad 2

This chapter builds upon the knowledge presented in chapters 1 and 2 and introduces the techniques that were used in order to achieve efficient implementation of basic Computer Vision filters in Myriad 2.

In the first stage, an evaluation of basic CV filters was conducted. In particular, we examined the performance of convolution filters, with various kernel sizes. As detailed in section 1.2, these filters are very common in every computer vision application since most of the filters that are applied to images can be considered as a form of convolving a kernel with the image. Therefore, convolutions affect significantly the overall performance of the applications in which they are used.

Subsequently, we examined the performance of Canny edge detection in Myriad 2 as well as the features that affect it. As stated in section 1.3.1., this filter is useful for extracting valuable information of an image, as well as an intermediate step of a larger computer vision application, for example feature recognition.

The filters used in this chapter are part of the mdk that Movidius provides and do not constitute work of the writer of this thesis. The goal, however, of evaluating these filters in Myriad 2 is to grasp a deeper understanding of the embedded architecture and discover the means by which the main algorithm examined in this thesis, Harris Corner Detection, can be optimally developed.

3.1 Convolution filters

3.1.1 Software implementation

In this section, we examined two different sizes of convolution kernels, 5x5 and 11x11. The development is based on the Movidius SIPP framework (see section 2.3).

Figure 3.1 demonstrates the two pipelines that were built.

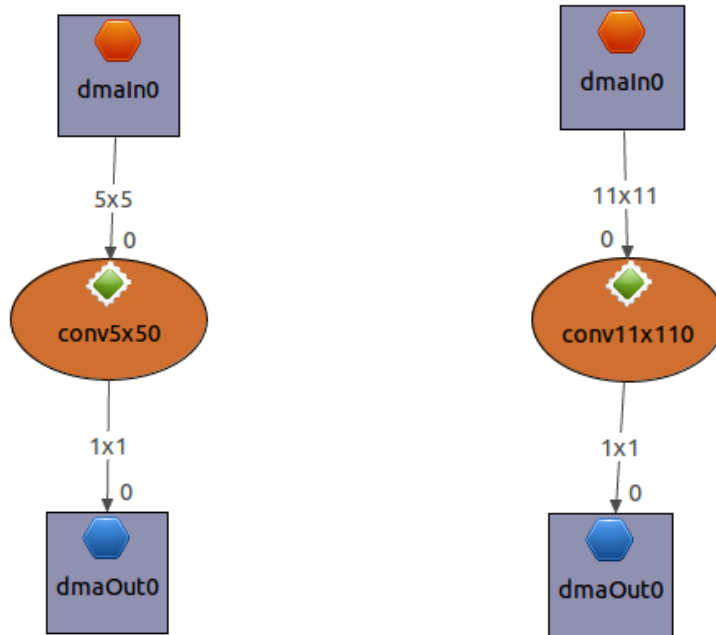


Figure 3.1: (a) 5x5 Convolution (b) 11x11 Convolution

Note that the "0" after the name of each filter indicates the index of such filters in a pipeline. In this example, it is zero since each filter appears only once.

As can be seen, the applications consist of two DMA filters and one convolution filter. The first DMA is used to bring the image from the DDR to the CMX. The number of lines that the DMA will transfer to the convolution filter depends on the size of the kernel. A 5x5 kernel requires at least 5 lines and a 11x11 at least 11.

Then, the filter is applied in a parallel manner, between various shaves. Each shave is responsible for a portion of the image. The SIPP framework divides the image in vertical stripes and allocates one to every shave. Then, through DMAs, the lines that correspond to each shave are brought to its CMX slice for processing. After the output is ready, the circular input buffers rotate and bring in the next necessary lines. In addition, the output lines are moved from CMX back to DDR via the DMA out filter.

The above process, which is presented in more detail in the section 2.3 regarding the SIPP framework, makes clear two very significant features that are used to gain increased performance of convolution filters in Myriad 2. Firstly, a pipeline is used in order to pass the image from DDR to CMX and apply the effect. This ensures high utilization of the platform's hardware. For example, if our graph consisted of more filters, one after the other, and no pipeline was used, the SHAVEs responsible for the later filters would have to wait for the earlier to finish. This practice would of course result in poor performance because of the generated stalls. Low performance would also occur even if all the shaves were used for each level of the graph, but processing between filters was done again in series. In this case, even though more shaves would work to apply the filters in the first level of the graph, they would have to complete the processing before moving to the next level.

However, SIPP ensures that images are processed in a scanline-by-scanline basis thus assuring another level of parallelism. In addition, task based parallelism is also used in our application because we deploy more than one shaves. All these VLIW processors run in parallel, each for a specific portion of the image, as said above.

Having described the way that SIPP uses both data as well as task based parallelism techniques, the following subsection presents information regarding the performance of convolution filters in Myriad 2.

Figures 3.2 and 3.3 demonstrate the images used as input in the following tests. Both of them are greyscale images, which means that the pixel values are in the range [0,255]. It is also worth noting that for the larger image, an extra configuration is necessary in order to increase the CMX pool size that is available to the the SIPP framework.

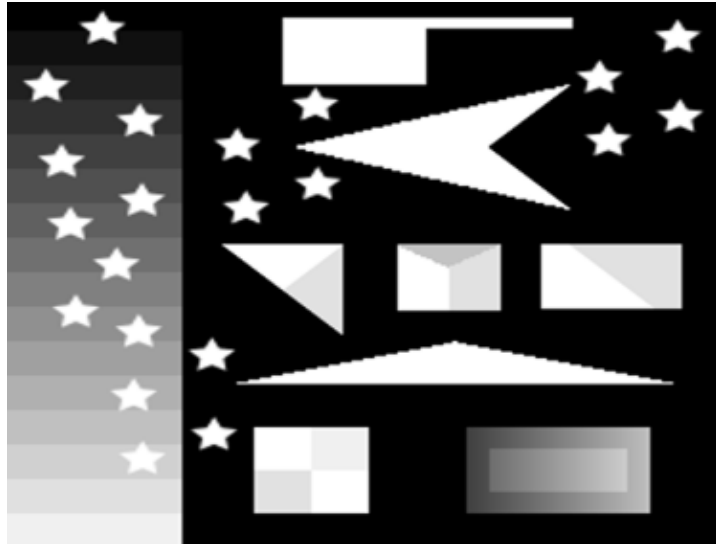


Figure 3.2: 512x384 test image.

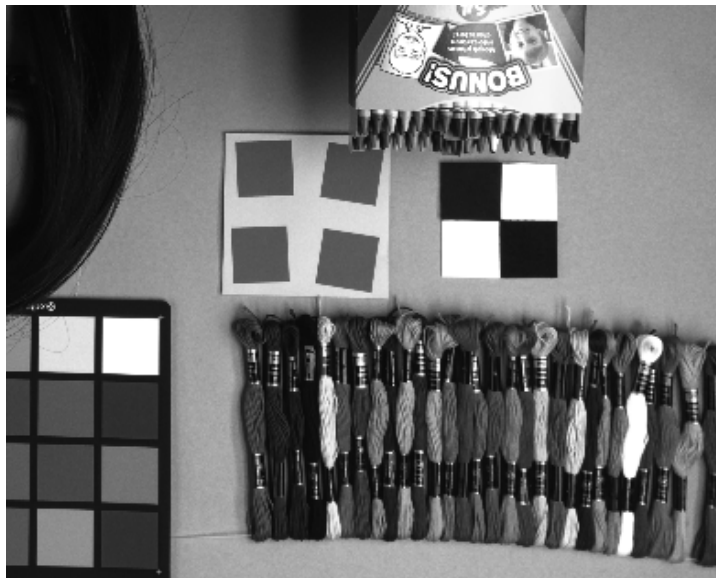
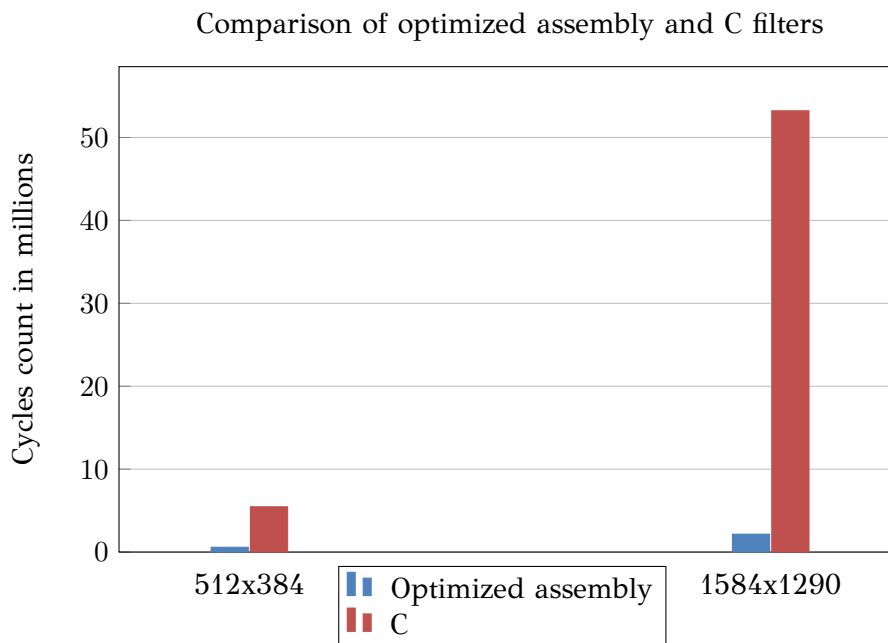


Figure 3.3: 1584x1290 test image.

C source code compared with optimized assembly filters

As a first step, we compare the performance of convolution filters developed in C and in optimized assembly. This evaluation of generic C code is important, as a first stage of working with an embedded platform and before deploying optimized filters, in order to identify and illustrate the significant differences between general purpose and embedded processors.

The following chart provides information regarding the comparison of C and assembly convolution filters. A convolution kernel of size 5 has been used and, in both cases, the number of shaves has been set to the maximum value of 12.



From the above data it is made clear that C implementations that are not optimized cause a tremendous performance drop. This issue is more clearly illustrated in the larger image, where the optimized assembly implementation is almost 25 times faster than the C source code.

The reason for this performance gap is that C implementations do not take into consideration the specific features of Myriad 2. In particular, they are generic C implementations that target any general purpose CPU and not a VLIW processor such as SHAVEs. Features that make SHAVE processors powerful, such as Single Instruction Multiple Data Operations (SIMD), are omitted in these C source implementations and in general there is an inadequate utilization of the platform's hardware.

On the other hand, the filters developed in assembly exploit the provided hardware as much as possible. Instruction level parallelism is used for almost every process. Instead

of using the IAU, optimized filters use the VAU to execute their commands in a SIMD fashion. For example, C source code processes one pixel at a time, whereas assembly exploits the fact that SHAVEs are 128-bit VLIW processors and evaluates **16 pixels in each instruction issue** (each pixel has an unsigned char value). Moreover, further optimizations have been applied in the assembly filters, such as removing unnecessary "nop's" after instructions and loop parallelization.

Overall, we conclude in the fact that optimized assembly implementations offer **an order of magnitude increase in performance** compared to those in C.

Therefore, for the following tests of this chapter only assembly written filters are used and the focus is shifted on identifying other features that affect the performance, such as the number of shaves that are deployed or the location of the buffers in memory.

Following is presented a comparison of optimized filters based on the :

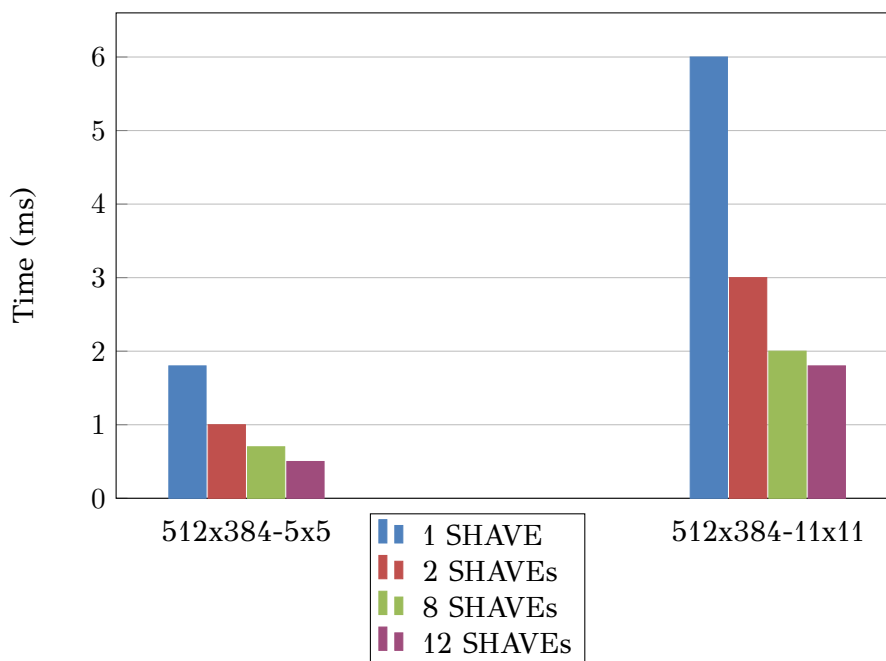
- Image size, 512x384 and 1584x1290
- Kernel size, 5x5 and 11x11
- Number of SHAVE processors used

Measurements have been made regarding both the performance (execution time in ms) as well as the Energy (Joule) for each implementation.

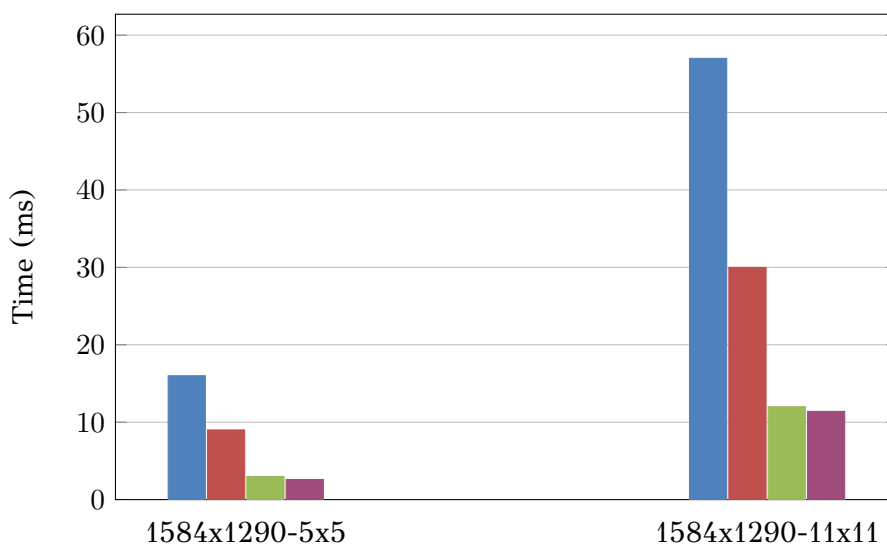
Myriad 2 provides specific functions and hardware to measure the power usage of an application. Energy is then calculated by multiplying the time with the power usage ($J = W*s$).

The results are provided in the following charts :

Convolution filter performance in Myriad 2 - 512x384 image



Convolution filter performance in Myriad 2 - 1584x1290 image

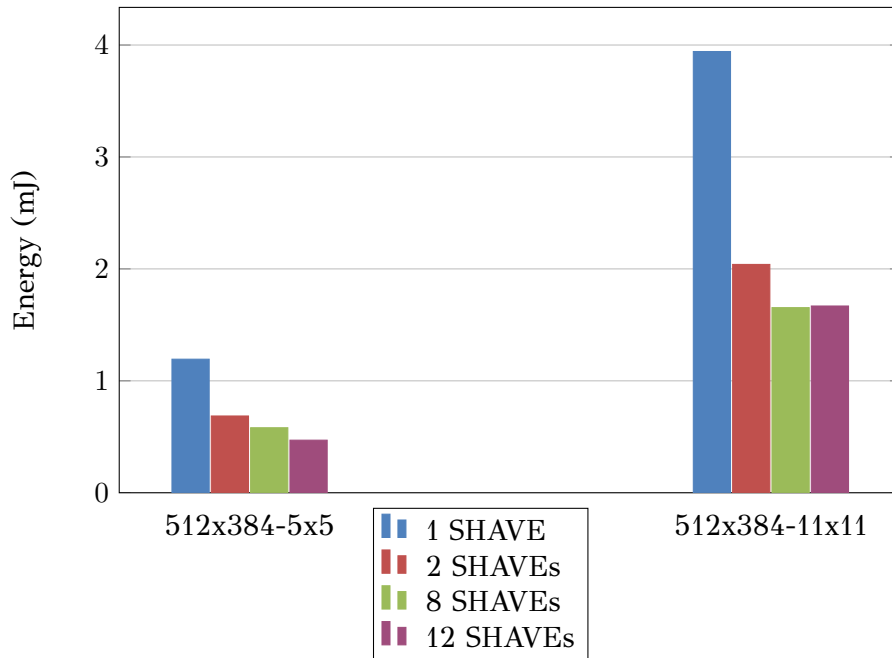


As expected, a larger convolution kernel (11x11) requires more processing time for the same image size. The performance gain achieved with the increase in the number of SHAVEs is more obvious when we move from just one SHAVE to two, where performance is doubled. In addition, significant increase in performance is noticed between two and eight SHAVEs. However, because the computation that is tested in this case is not very intensive (just a convolution kernel applied in an image), there is no significant gain when moving from eight to twelve SHAVE processors.

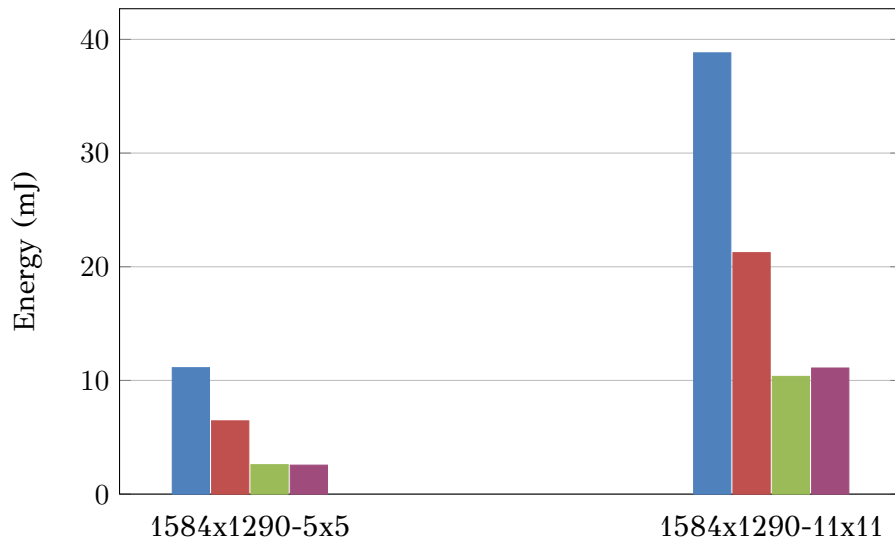
The above charts illustrate the performance gain that can be achieved when all SHAVE processors are deployed. However, as more processors are used, there is an increase in

the power usage of the platform. The Energy required for these operations can be seen below :

Convolution filter Energy Efficiency in Myriad 2 - 512x384 image

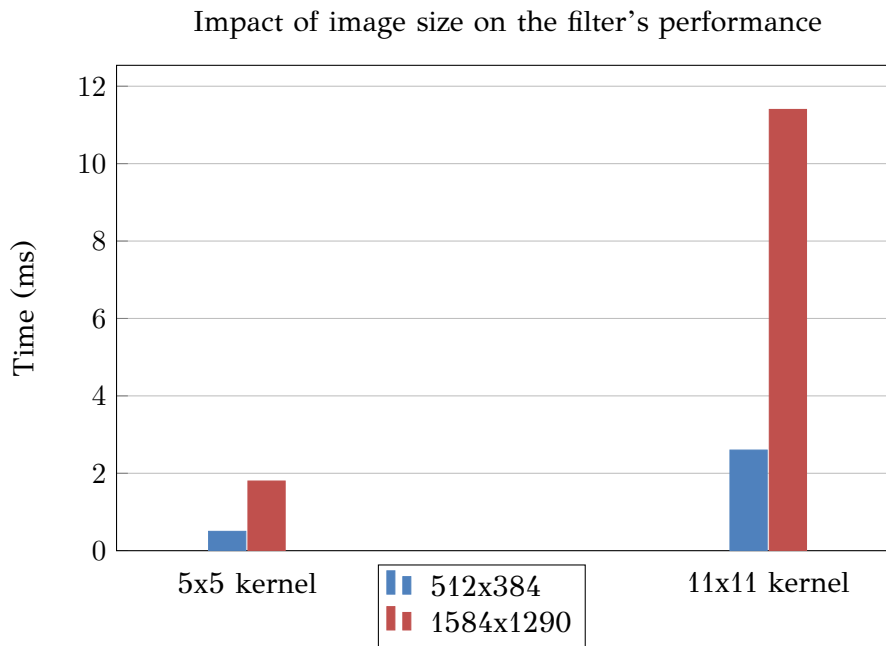


Convolution filter Energy Efficiency in Myriad 2 - 1584x1290 image



The power usage of the platform when only one SHAVE is used is at around 0.5 W, while this number reaches almost 1W when all SHAVES are deployed. Even though the power usage is doubled, the above graphs show that, in general, more SHAVES lead to better energy efficiency since the operation time is also significantly decreased. The only case when this is not valid is when we move from 8 to 12 SHAVES. In this case, the performance gain is not enough to negate the extra 100 mW of consumption. However, this is caused because the application tested is not complex enough to benefit from the extra SHAVES. Therefore, we conclude that using more SHAVES leads not only to higher efficiency but to lower overall energy consumption as well.

Another feature worth measuring is how the performance scales from the small to the large image. Hence, we use a chart that compares the time needed for a 5x5 convolution with each of the used images. For both cases we use a maximum number of 12 SHAVES. The results can be seen below :



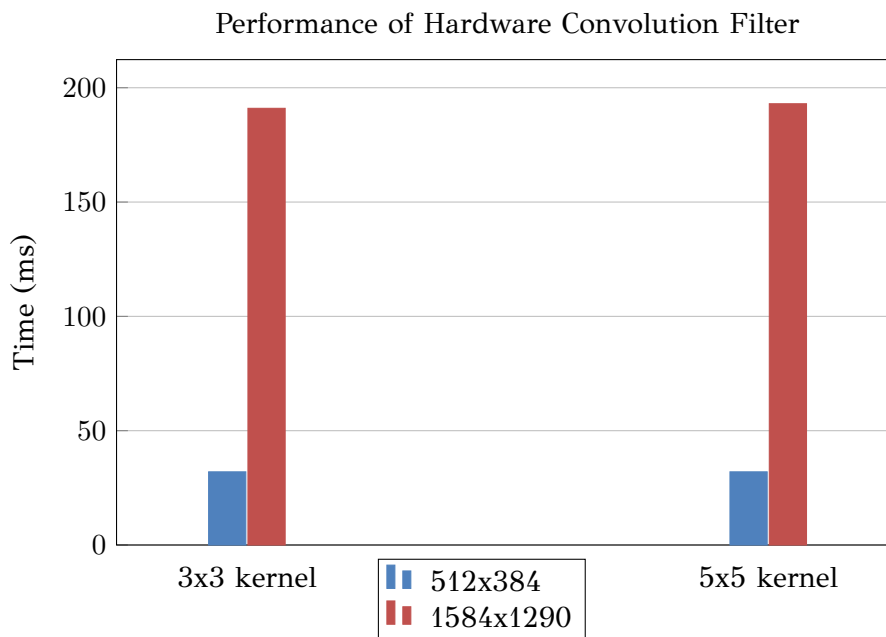
This chart shows that the size of the input affects immediately the processing time, which was normal to happen. However, even though the larger image is 10.39 times the size of the small, the time needed for each operation did not increase by this factor. This is again due to the simplicity of the examined kernel.

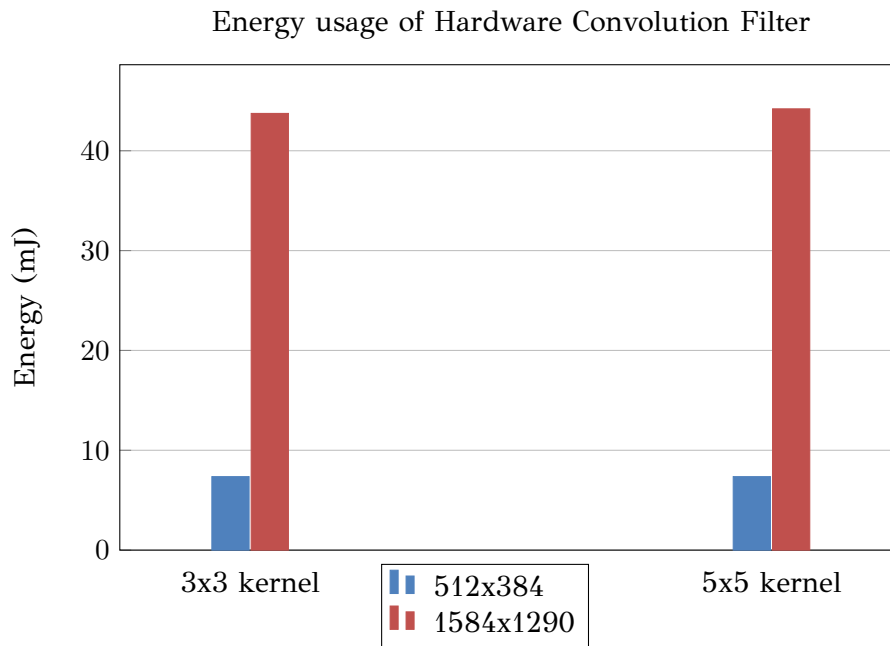
The explicit impact of the image size to the performance is not caused only by the extra processing allocated to SHAVES. It can also be explained by looking at the pipeline images presented in the start of this chapter, in Figure 3.1. Initially, the images are placed in DDR memory and are transferred to CMX via DMA operations. Even though the DMAs are processed by specialized hardware, they pose a limiting factor in terms

of performance. Image slices need to be in CMX for the SHAVEs to process them efficiently, so a larger image leads to a higher number of DMA operations. The real processing stalls between DMAs, thus causing a decrease in the overall performance. Hence, we conclude that another very significant feature that affects the performance of image processing and computer vision applications in Myriad 2, is **the number of DMA operations** needed to process the whole image.

3.1.2 Hardware accelerators

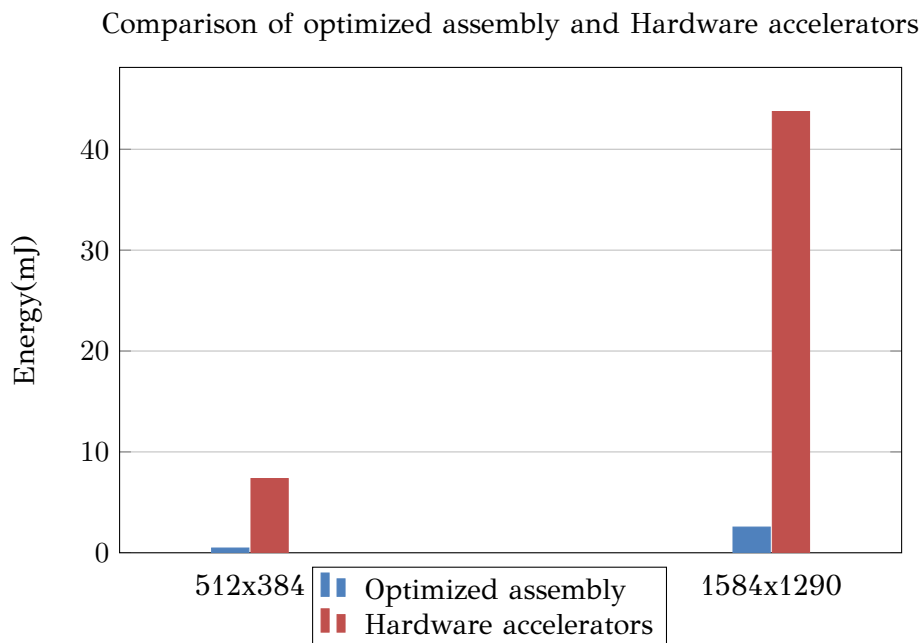
Movidius, recognizing the importance of convolution in image processing applications, has added a specialized hardware engine for this filter. This filter is configurable with two kernel sizes available, 3x3 and 5x5. The performance as well as the Energy consumption of these filters can be seen in the following charts :





Again, the image size affects the performance of the hardware filter. However, a bigger kernel size leads to more processing time only in the large image, where more computations take place, because the difference in the number of computations of a 3x3 and a 5x5 kernel is not so significant.

A valuable measurement is that of the comparison of the Energy, which entails both time as well as power usage, between the optimized assembly filters and the hardware kernels.



The software filters consume just below 1W in order to achieve maximum performance, while hardware filters consume only 250mW. However, the increased performance of

software filters leads to a better overall efficiency.

From the above data we can deduce that for small applications such as a convolution of a kernel with an image, it is preferable to use optimized software filters. Nevertheless, real computer vision applications can benefit from the usage of hardware filters. For example, by allocating the work of specific filters to the hardware engines, the SHAVEs are free to be used for other filters. In this way, the overall performance is increased by applying another level of parallelism, between SW and HW implementations. In addition, ultra low power applications can be built in Myriad 2 by using only hardware filters.

3.2 Edge detection

The convolutions tested in the previous section provided an important insight of the features of Myriad 2 that affect an application's performance. In this section, we test edge detection, as a more complex application, in order to make similar evaluations under a more computationally demanding algorithm.

3.2.1 Canny edge detection software implementation

Movidius provides an optimized software filter for Canny Edge detection. This filter performs the steps of the algorithm as described in Chapter 1, section 1.3.1. The kernel's size is 9x9.

The output that is generated by applying this filter to the test images is presented in Figures 3.4 and 3.5.

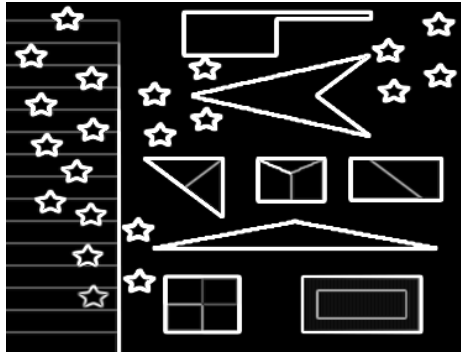


Figure 3.4: Canny edge detection on 512x384 test image.

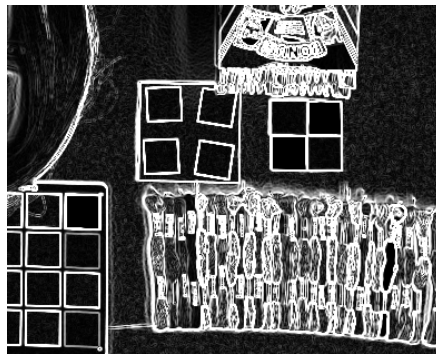
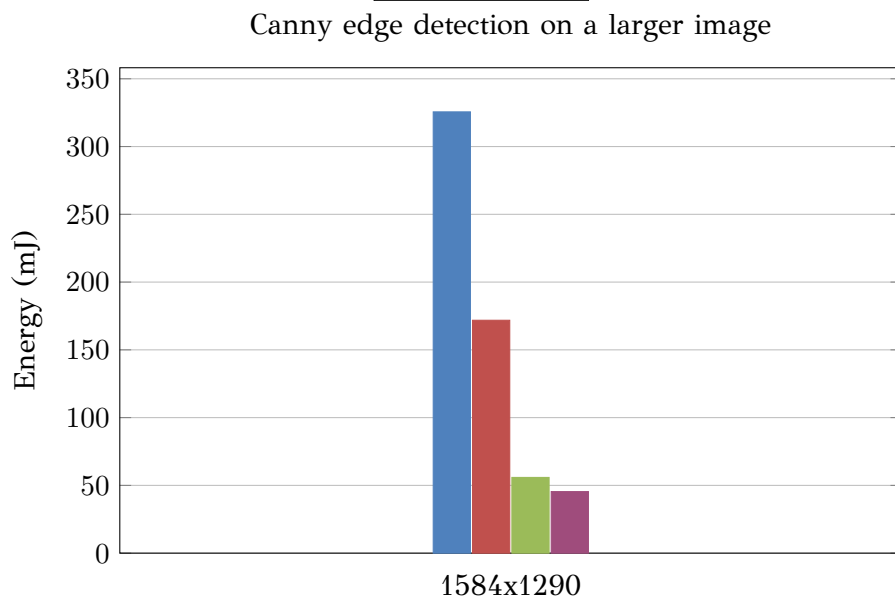
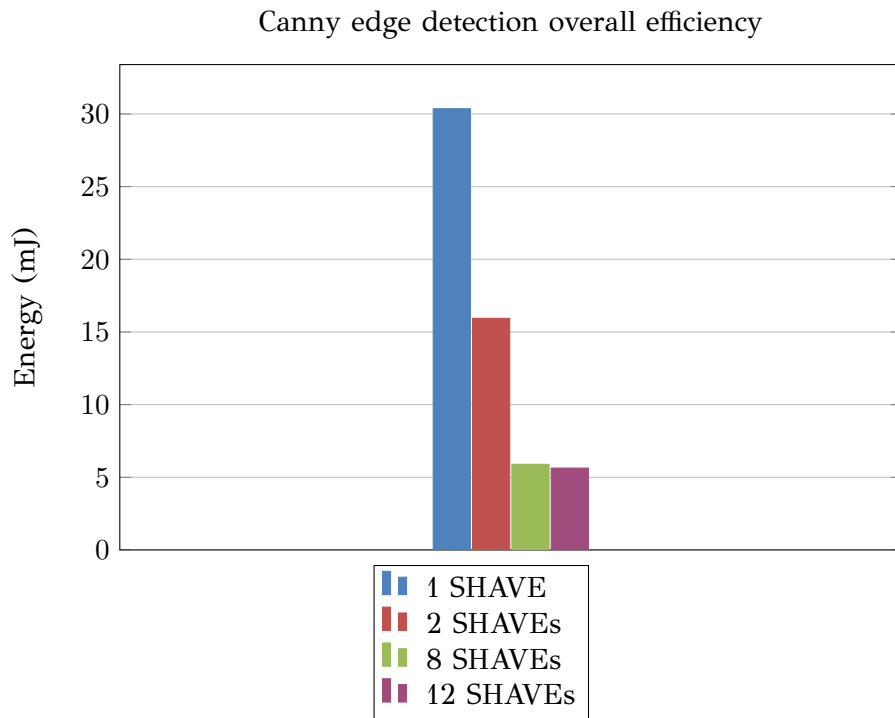


Figure 3.5: Canny edge detection on 1584x1290 test image.

Similarly to the previous section, our focus is in how this algorithm is affected by the number of SHAVEs and the image size (number of DMA operations). For the presentation of the results we will use the Energy consumption, since it contains both the performance as well as the power usage measurements ($J = W*s$), and it can be used as an index of overall efficiency.



These data, similar to those that we extracted from simple convolution operations in the previous section, show that the number of SHAVEs affects in a very significant way the overall efficiency.

However, there are some differences in our observations for this algorithm, compared to convolutions. In particular, when moving from eight to twelve SHAVEs, there is now a substantial performance gain because the algorithm is computationally intensive and can benefit from the extra SHAVEs. That was not the case when we tested simple convolution filters.

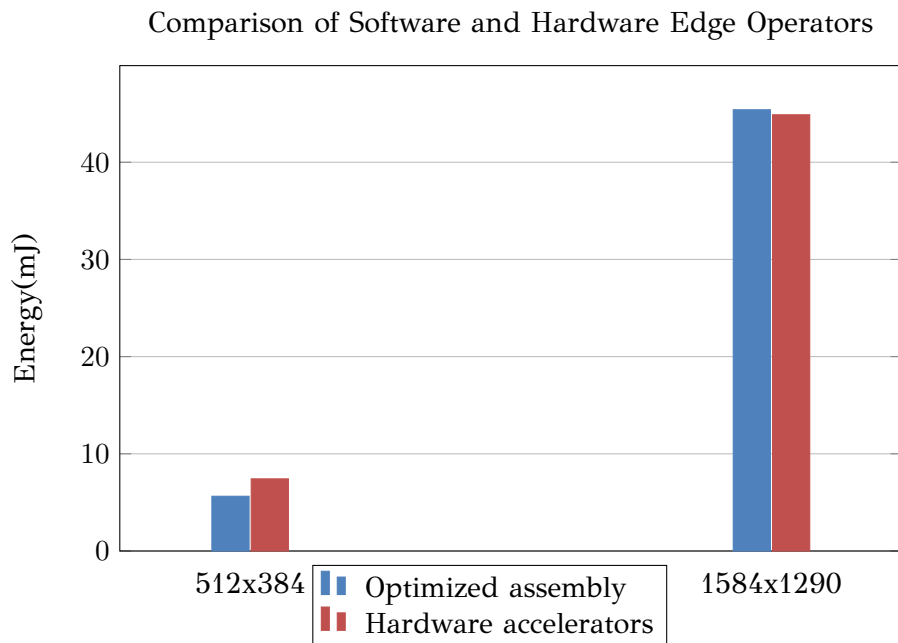
In addition, the image size has a greater effect to the overall performance. As stated earlier, the ratio between the 1584x1290 and the 512x384 image is 10,39. Even though the processing time of a convolution kernel did not increase by this factor, it can be seen now that the larger image consumes ten times more energy than the small.

Since this algorithm is a better representative of complex applications than a basic convolution filter, we can conclude in the fact that the image size and the performance of Myriad 2 have a linear relationship.

3.2.2 Hardware accelerator

Myriad 2 offers an Edge Operator hardware filter which implements an extension to the functionality of the standard 3x3 Sobel filter.

We have evaluated the performance of this filter and compared it with the optimized software filters. The results are provided below, again in terms of overall Energy consumption:



These results are much different than the corresponding comparison we did for convolution filters.

However, it is not correct to compare these two filters directly, because they constitute different implementations. The canny edge detector is a more complex algorithm than the hardware edge detector that Movidius has implemented as an extension to the Sobel filter. Actually, the sobel operator is just one out of the four steps that canny edge detection contains.

Having this in mind, and since both of them can be used for edge detection with similar results, it is safe to make a general comparison. In particular, we can see that for more complex applications such as edge detection, this hardware filter is almost as efficient as the optimized software filters that run on SHAVEs and performs canny edge detection. Moreover, for large images, there is better energy efficiency in the case of the hardware accelerator.

In general, the trade - off between hardware and software filters remains the same in this case also, as with simple convolution filters.

Hardware accelerators can provide the means for ultra low power applications and can also be used to achieve parallelism by combining software and hardware implementations.

3.3 Outcome of the evaluation

As stated in the beginning of this chapter, the goal of the evaluation of these filters was to achieve a better understanding of Myriad 2 and to identify the features that affect the performance of the developed applications.

These features concern mainly software filters, since hardware accelerators cannot be altered by the developer. Moreover, the knowledge extracted from this evaluation does not target only the development of applications with the optimized filters that Movidius provides. In contrast, it was made in order to generate a design space based on which the main algorithm developed in this thesis, Harris Corner Detection, can be optimally deployed in Myriad 2 from a general purpose CPU implementation.

The factors that affect the performance of software filters in Myriad 2 are :

- The location of SHAVE code and data (DDR is much slower than CMX but CMX is just 2MB)
- The Number of SHAVE processors used
- The number of DMA operations required in any iteration

- The CMX bandwidth (since it is the memory where SHAVEs keep their working buffers)

Chapter 4

Implementation of Harris Corner Detection

In this chapter we introduce the main focus of this thesis, the Harris Corner Detector algorithm [2]. Two different approaches were developed regarding this algorithm. The first one consists of using optimized filters that Movidius provides, both software and hardware. These filters perform the basic functionality of the Harris & Stephens algorithm, which is to calculate a response value for every pixel of the image.

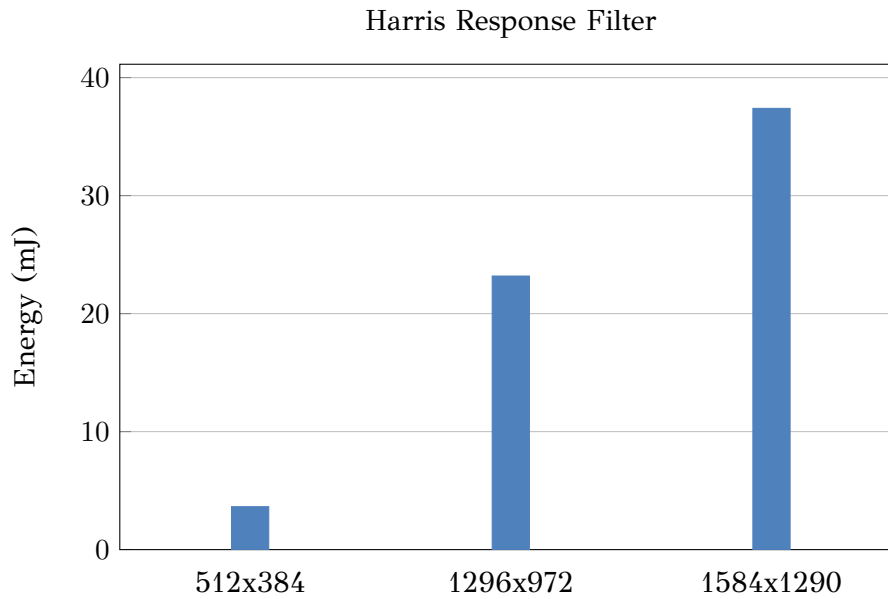
Then, a more complex C-code implementation of the Harris & Stephens algorithm was employed in Myriad 2 from scratch and the full capabilities of the architecture were used in order to achieve the best possible performance per Watt.

The results that we obtained as well as the development steps that were taken are detailed in the following sections.

4.1 SIPP filters implementation

4.1.1 Software Filter

Movidius provides an optimized software filter that performs Harris corner detection, as part of the CV library. This filter operates on a 8x8 window, containing the borders, and calculates the Harris Response R for each pixel of the image. It can be used as a part of a more complex pipeline that performs the complete Harris & Stephens algorithm. This pipeline would require, except for the Harris response filter, kernels that perform gaussian blur and non-maximum suppression. The performance of the Harris response filter is illustrated below in terms of overall energy efficiency for three different image sizes. The number of used SHAVEs has been set to the maximum value of 12.

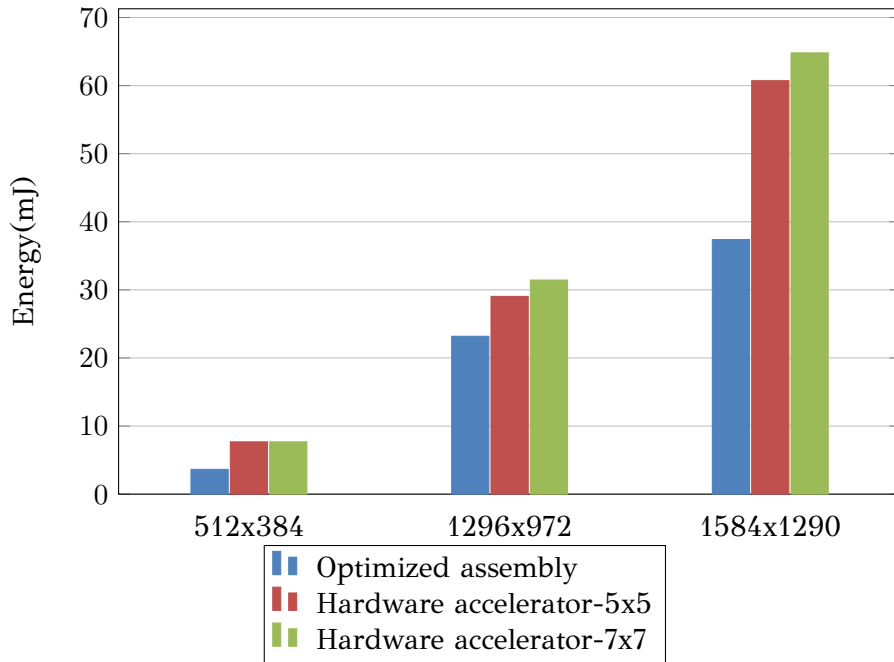


It is obvious that the filter that Movidius provides achieves very high energy efficiency. Applying an 8x8 kernel on an image and performing several computations is a quite intensive operation. However, Myriad 2 was able to output the Harris response for a 512x384 image in only 4ms, consuming less than 1 W. It can also be seen that the image size and the performance of this filter have a linear relationship, as was the case for Canny edge detection.

4.1.2 Hardware accelerator

Myriad 2 offers the same filter as a hardware accelerator. This filter is configurable and except for the k value, the developer can choose between a kernel size of 5x5 or 7x7. We have evaluated the performance of this filter and compared it with the optimized software filter. The results are provided below, again in terms of overall energy efficiency:

Comparison of Software and Hardware Harris Response Filters



From the above data we deduce that the trade - off between hardware and software filters remains the same in this case also, as with the edge detection filters. The optimized filters running on SHAVEs are more efficient than the HW filters. However, hardware accelerators can provide the means for ultra low power applications and can also be used to achieve parallelism by combining software and hardware implementations. For example, in a pipeline that would perform the complete Harris Corner Detection algorithm, we could get significant benefit by using the hardware accelerator. In particular, instead of applying all the processing to the SHAVEs, the harris response could be offloaded to the hardware filter and the other filters of the pipeline would run by the SHAVEs, leading to better overall efficiency.

4.2 Porting a generic C implementation in Myriad 2

4.2.1 Description of the used implementation

A significant feature of embedded platforms that concerns developers is the effort as well as the time that is required to deploy source code that was originally built for general purpose CPUs into an embedded architecture.

In order to examine this issue in our platform, we ported in Myriad 2 a C-code implementation of Harris & Stephens [2] algorithm, provided by Dr. Manolis Lourakis (<http://users.ics.forth.gr/lourakis>) [16]. Figure 4.1 illustrates the various functions that constitute this specific implementation. It is obvious that the functionality of this code is much more complex than the Harris response filter provided by Moividius, which can be thought of as a part of what Figure 4.1 shows.

In the first step, the image data is read and passed in the function that computes the first order partial derivatives (I_x, I_y). This is accomplished by applying a separable convolution filter that combines the following kernels :

$$derivative : \begin{bmatrix} 1 & -3 & 0 & 3 & 1 \end{bmatrix}$$

and

$$smoothing : \begin{bmatrix} 1 & 6 & 12 & 6 & 1 \end{bmatrix}$$

The function that performs this operation is called *imgradient5_smo()* and can be seen below in pseudocode. Note that the symbol "*" denotes convolution, not multiplication.

```
int imgradient5_smo(image[width, height], width, height, gradx, grady)
for (i=[0,height-1])
for (j=[0,width-1])
wrkx[i,j]=image[i,j]*derivative_kernel;
wrky[i,j]=image[i,j]*smoothing_kernel;
next_line;

for (i=[0,height-1])
for (j=[0,width-1])
gradx[i,j]=wrkx[i,j]*smoothing_kernel;
grady[i,j]=wrky[i,j]*derivative_kernel;
next_line;
```

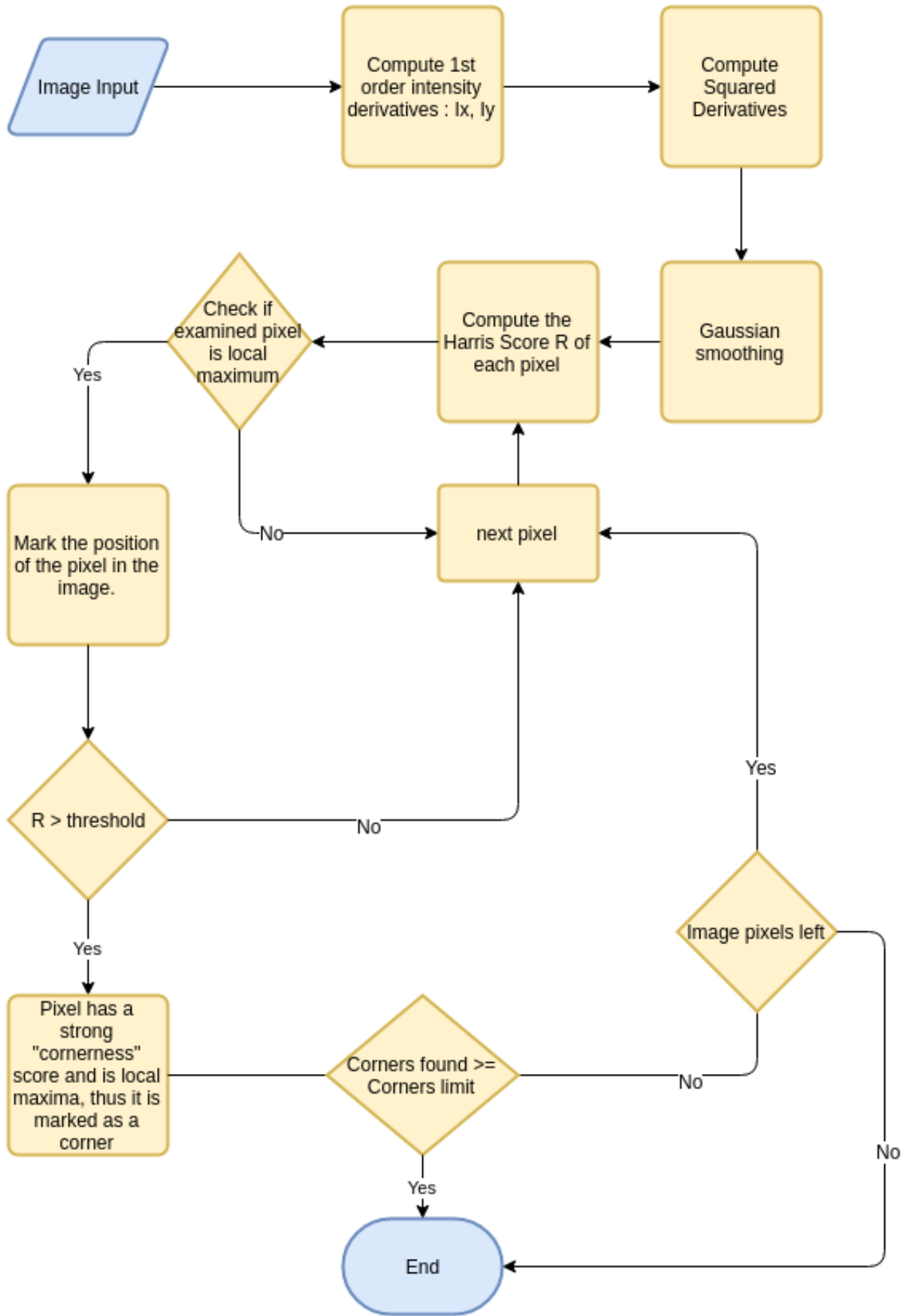



Figure 4.1: Harris Algorithm flow chart

The first two for loops of *imgradient5_smo()* are the horizontal convolutions and consist of computing for each pixel the derivative and the smoothed value respectively by accumulating the local sum of the 5 value kernel. For this purpose, the values of the previous two and the next two pixels in the particular row are used. Then, each value is normalized by the equivalent factors. The normalization is not applied in each coefficient in order to reduce the number of floating point operations.

In the second pair of for loops, the kernels are interchanged since vertical convolutions are now applied. The multiplication is done by the previous two and next two pixels vertically, following the same procedure as before. Finally, the arrays *gradx* and *grady* hold the derivative values computed for each pixel and correspond to I_x and I_y respectively.

The next step of the algorithm takes the partial derivatives computed above and calculates the squared derivatives I_x^2 , I_y^2 as well as the $I_x I_y$, for the whole image.

Then, the above are passed as input in the *imgblurg()* function which applies the Gaussian blur. This operation is performed three times, one for each of the I_x^2 , I_y^2 and $I_x I_y$. This function is shown below. Again, the symbol "*" denotes convolution between two matrices.

```
int imgblurg(gradx , grady , gradxy){
/* separability: convolve horizontally ... */
for (i=[0,height]){
for (j=[0,width]){
wrkx[i , j]=gradx * gaussian_kernel;
wrky[i , j]=grady * gaussian_kernel;
wrkxy[i , j]=gradxy * gaussian_kernel;
}
}
/* ... then convolve vertically */
for (i=[0,height]){
for (j=[0,width]){
gradx2[i , j]=gradx * gaussian_kernel;
grady2[i , j]=grady * gaussian_kernel;
gradxy[i , j]=gradxy * gaussian_kernel;
}
}
}
```

The horizontal convolution is held by a window that is 7 points wide and takes into consideration three pixels before and three after the examined. Similarly, the vertical convolution considers three pixels above and three below the center pixel.

After the squared derivatives have been calculated, they are used to calculate the harris score for each pixel. Offsets are used for the height and width in order to avoid producing values for pixels in the images edges. It is reminded here that the harris score R is calculated by the following equation :

$$R = det - harris_kappa * trace^2$$

, where $det = I_x^2 * I_y^2 - (I_x I_y)^2$, $trace = I_x^2 + I_y^2$ and $harris_kappa \in [0.04, 0.15]$.

```
for (i=[vertical_offset , height - height_offset]){
for (j=[horizontal_offset , width - width_offset]){
det[i , j] = gradx2*grady2 - gradxy*gradxy;
trace[i , j] = gradx2 + grady2;
cornerness[i , j]= det - harris_kappa*trace*trace;
}
}
```

The cornerness buffer keeps the harris score for each pixel of the image. Then, if this score of each pixel is local maximum in a 3x3 window, its position is marked in a binary buffer that represents locally maximum pixels with the the logical value of 1 and all the others with zero.

In addition, a threshold is set depending on whether the image has more corners than the user is looking for.

Finally, if the cornerness of each pixel is high enough and if it is also locally maximum, it is marked as a corner and its coordinates are saved.

4.2.2 First stage - Initial Porting

After having described the implementation that we will be using, we can move to the main part of this thesis, which entails the efficient porting of this algorithm in Myriad 2. The images that we used for our tests are the same as in Chapter 3.

This implementation does not use the SIPP framework. This means that the developer has to program explicitly operations that this framework did automatically, such as the allocation of image slices in SHAVEs, DMA operations etc.

We have used the One Leon Programming Paradigm. The execution starts from the LeonOS processor, which runs the RTEMS operating system. This processor then starts the SHAVEs and they perform the bulk of the processing by applying the various steps of the implementation of the Harris algorithm. More details about the development will be given in each of the steps that we have followed.

In the first stage of the implementation, the main challenge concerned the memory footprint of the application. Since it was originally developed for general purpose CPUs where memory is not a limiting factor, there are used big buffers to store the intermediate data. In particular, these arrays are :

- `ibuf[]`, which contains integer values for I_x, I_y, I_x^2, I_y^2 and $I_x I_y$
- `fbuf[]`, which contains all the R-values with float arithmetic

The bigger demand for memory is caused by the `ibuf[]` array that is configured to hold the first order as well as the squared derivatives for each pixel of the image. Thus, the size of this buffer is $5 * IMAGE_WIDTH * IMAGE_HEIGHT$. In addition, this number must be multiplied by four if we want to calculate its size in bytes, since the buffer is used to hold integer values (4 bytes each).

By simple calculations, for an image with dimensions 512x384 such as the one that was used in this stage, only this buffer would require a memory size of :

$$5 * 512 * 384 * 4 = 3932160bytes.$$

This number is almost double than the 2MB CMX memory that is available in Myriad 2. It is, therefore, immediately apparent that as a first step towards reaching a working stage of the implementation, the buffers should be placed in the DDR memory, which size is more than enough, at 128 MB. In addition, since there were other issues to address before applying optimizations, we have just used one SHAVE for this stage. The CMX memory is used to hold only the image data. Their size for this image is $512 * 384 = 196608$ bytes and it can easily fit in CMX. The first stage of the code that runs in the SHAVE is to bring with 1 DMA operation the image data from the DDR to

the CMX. The image data is placed, in all Myriad 2 applications, in the higher parts of the CMX, since the lower are kept for the SHAVE's code and data.

Subsequently, after making the necessary modifications, we were able to run the application and get correct results. Figure 4.2 shows the corners that have been found on a 512x384 image, marked with white rectangles.

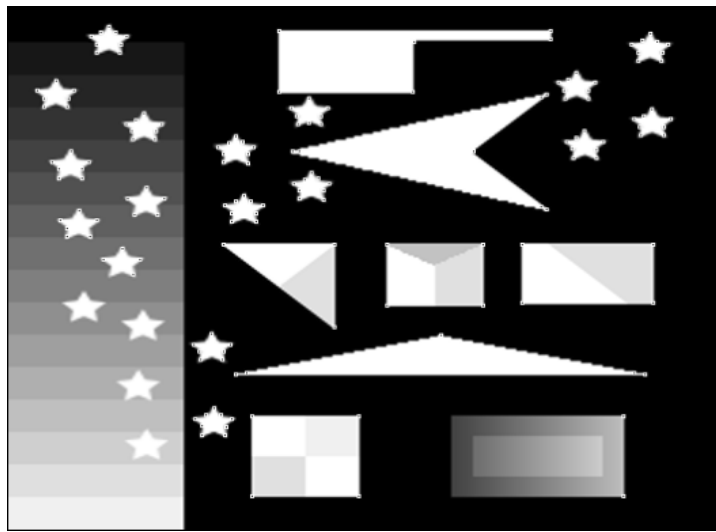
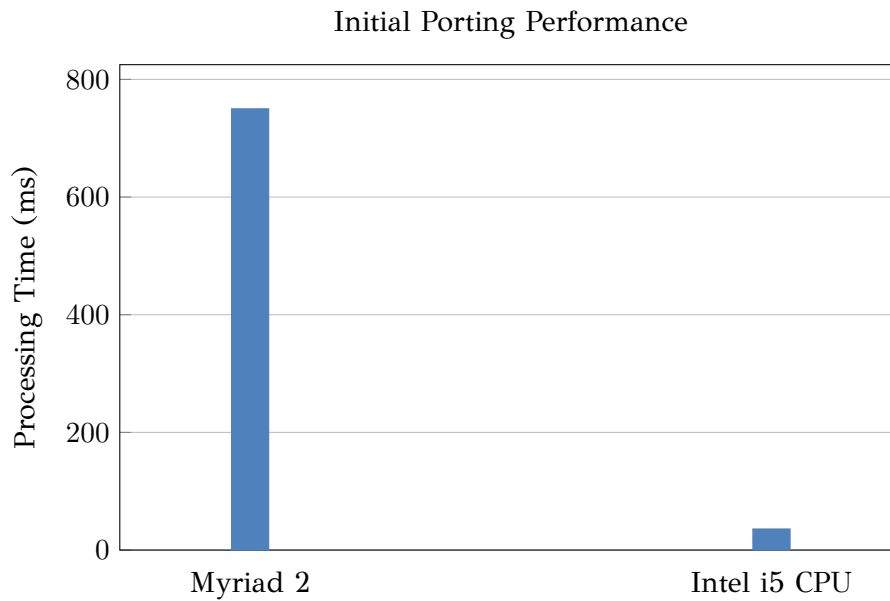


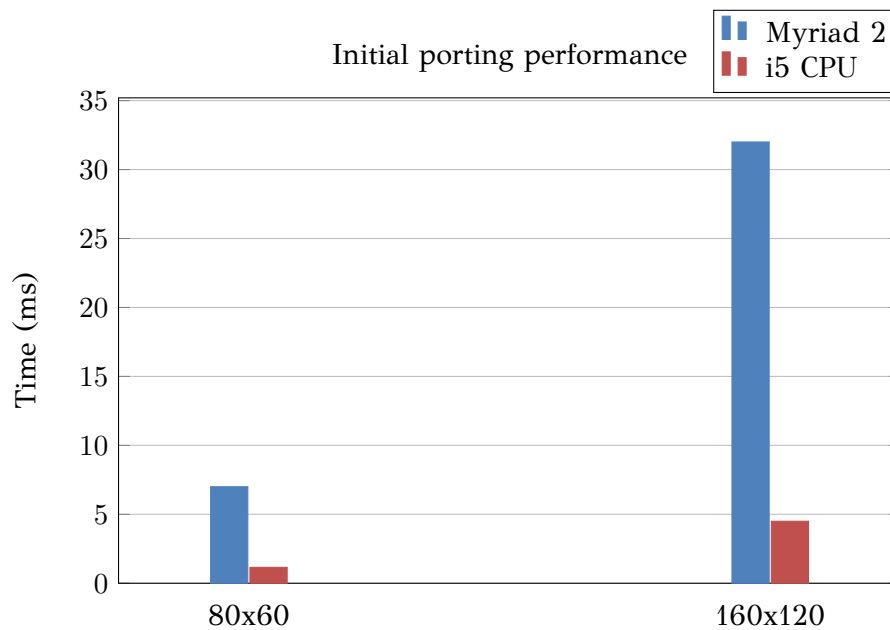
Figure 4.2: Harris corner detector applied on a 512x384 image

The performance results can be seen in the following charts. We compare the performance of this initial porting with an Intel Core i5 2435M CPU that runs at 2.4 GHz and consumes around 15W for operations such as this. This comparison is used in order to see where the implementation stands in terms of efficiency and how the design space in which we have concluded in Chapter 3 can help us reach a more optimal implementation.

The processing time that is presented below corresponds to the real processing conducted by the SHAVE. The clock started when the SHAVE was initialized by the LEON processor and the measurements were gathered upon the finish of the SHAVE's computations.

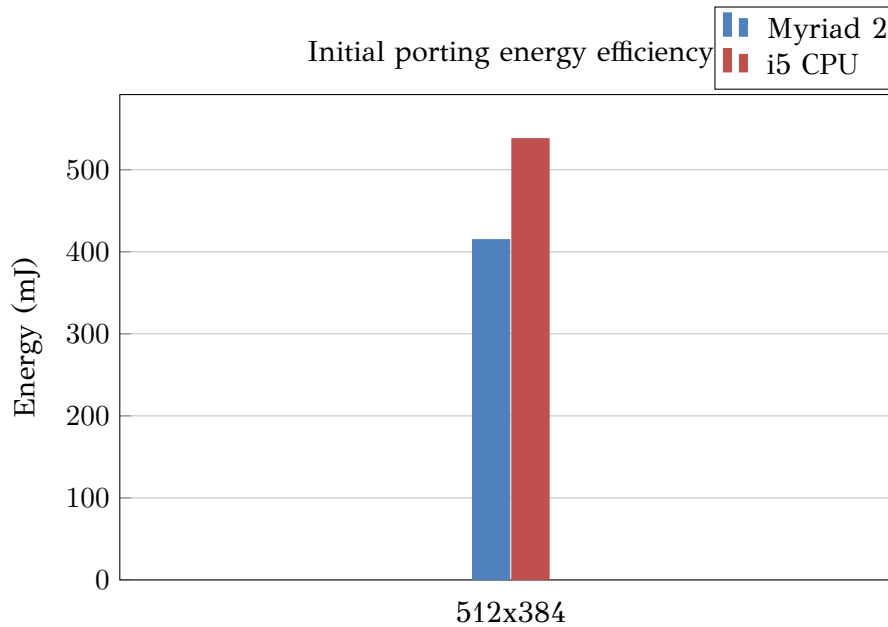


Except for the 512x384 image, we have also made tests for a 160x120 and a 80x60 version of this image. The results can be seen below:



As expected, even though for the small image the performance gap is not so great, the allocation of the working buffers in the DDR as well as the fact that we use only one SHAVE have lead to very poor performance.

The platform consumes around 530mW for each of these tests, because only one SHAVE processor is used. The following charts compare the overall energy efficiency of Myriad 2 and the Intel i5 CPU, for the case of the 512x384 image.



Even though the performance of Myriad 2 at this stage is much worse than the i5 CPU, its power consumption, as a platform that targets low power embedded applications, is much lower and it leads to a better overall Energy efficiency.

4.2.3 Second stage - Working buffers in CMX

In this step, the working buffers of the applications are placed in the CMX memory. As said earlier, the size of the CMX is not sufficient for the large arrays of the 512x384 image. Therefore, we have used the smaller images 160x120 and 80x60. Even though we want to be able to use images of arbitrary size, in this stage the focus is in measuring the impact that the location of the buffers has in the performance of our applications.

Normally, each SHAVE has 128KB of memory in the CMX, 96KB for data and 32KB for its code. However, this size is not enough to fit even the buffers of the 80x60 image. Only the `ibuf[]` for this image would have a size of $5 \times 80 \times 60 \times 4 = 96000$ bytes. Therefore, we have used a configuration LD script that changes the memory that is available for each SHAVE. Since, in this stage, there is only one SHAVE used, we can allocate to it the CMX memory that corresponds to all SHAVEs, with size 12x128KB. Now, there is enough CMX memory for the buffers of the SHAVE as well as any other variables that the implementation contains.

Below can be seen part of the ldscript that makes the described configuration. Even though there is code that defines the memory of all the SHAVEs, only the 1st SHAVE is allocated space in the CMX since it is the only one that is deployed. Memory addresses

that start from 0x70000000 point to the CMX, while the DDR addresses start from 0x80000000.

MEMORY

{

SHV0_CODE (wx) : ORIGIN = 0x70000000 + 0 * 128K, LENGTH = 32K
SHV0_DATA (w) : ORIGIN = 0x70000000 + 0 * 128K + 32K, LENGTH = 1504K

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

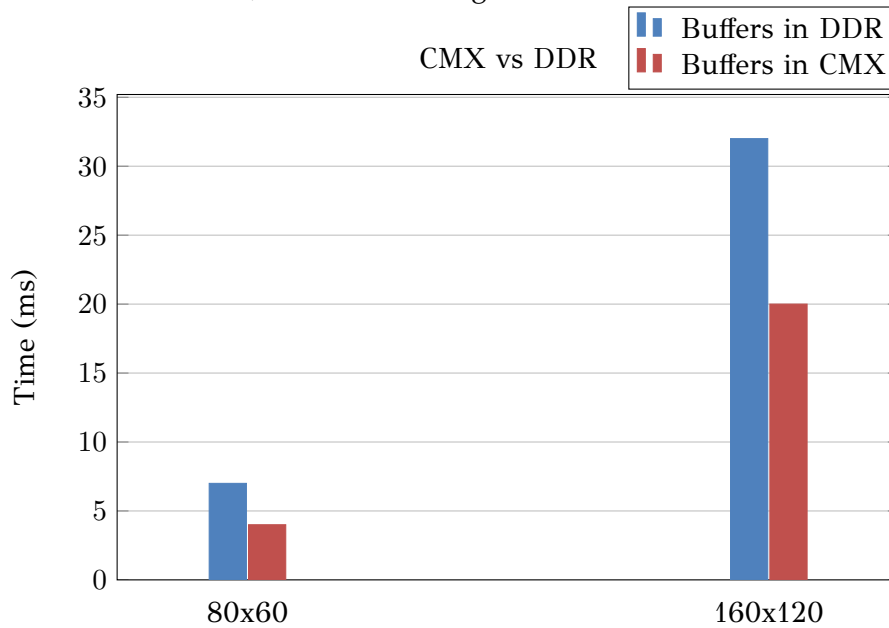
SHV10_CODE (wx) : ORIGIN = 0x70000000 + 10 * 128K, LENGTH = 32K
SHV10_DATA (w) : ORIGIN = 0x70000000 + 10 * 128K + 32K, LENGTH = 96K

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

CMX_DMA_DESCRIPTOR (wx) : ORIGIN = 0x78000000 + 12*128K , LENGTH = 3K
 CMX_OTHER (wx) : ORIGIN = 0x70000000 + 12*128K + 3K, LENGTH = 128K - 3K
 DDR_DATA (wx) : ORIGIN = 0x80000000, LENGTH = 128M

LOS (wx) : ORIGIN = 0x80000000 + 100M , LENGTH = 3*128K-10K
 LRT (wx) : ORIGIN = 0x80000000 + 100M + 3*128K-10K , LENGTH = 10K
 }

The following charts demonstrate a comparison between having the buffers in the DDR and the CMX, for the two images :



It is obvious that after placing the buffers in the correct memory, we have achieved a significant performance gain. The decrease in processing time is more apparent for the larger image, where more computations are needed. The memory accesses for reads and writes are now much faster since the CMX is very close to the SHAVEs and it has a higher bandwidth than the DDR memory. More details about the benefits of the CMX have been given in Chapter 2.

4.2.4 Third Stage - Arbitrary image size and buffers in CMX

After making a first test of the performance gain that CMX offers, we have modified the code in order to be able to process any image size. Previously, only 1 DMA operation was needed before the SHAVE could initialize its processing. This was possible because there was enough memory in the CMX to fit the buffers for the whole image.

However, this is not the case for larger images. As stated in the beginning of this section, the CMX is too small to fit even the `ibuf[]` for a 512x384 image. The solution to this problem is given by performing more DMA operations. Image data is transferred from DDR to CMX in *slices*. After the SHAVE has finished the processing of one slice, the next can be brought with another DMA operation. In this stage, the image will be split in horizontal slices only. This is possible because we are using only one SHAVE and there is enough space in the CMX for dimensions 512xSLICE_HEIGHT, where SLICE_HEIGHT will be tested with various values, starting from 32.

This concept requires the addition of *padding* to the slices in order to ensure that correct results are given as output. As described in Chapter 1, a nxn convolution kernel calculates the value of the center pixel by examining n/2 pixels in each direction, left , right, up and down. So, when our algorithm runs on the horizontal edges of the slice, it will need to consider some pixels from the previous and some from the next image slice. The amount of padding that is necessary can be calculated by examining the size of the kernels that the algorithm applies on the image. In our case, the kernels are the following :

- 5x5 convolution in `imgradient5_smo()`
- 7x7 convolution in `imgblurg()`
- 3x3 kernel for non-maximum suppression

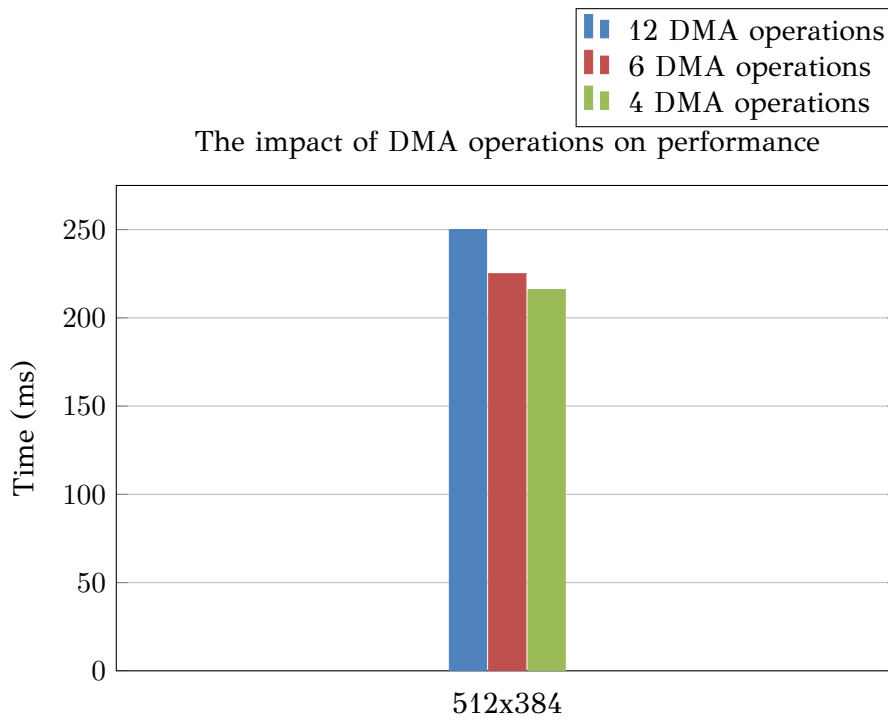
Since these kernels are applied one after the other, the final padding is calculated as the sum of the padding that is needed by each kernel. Therefore,

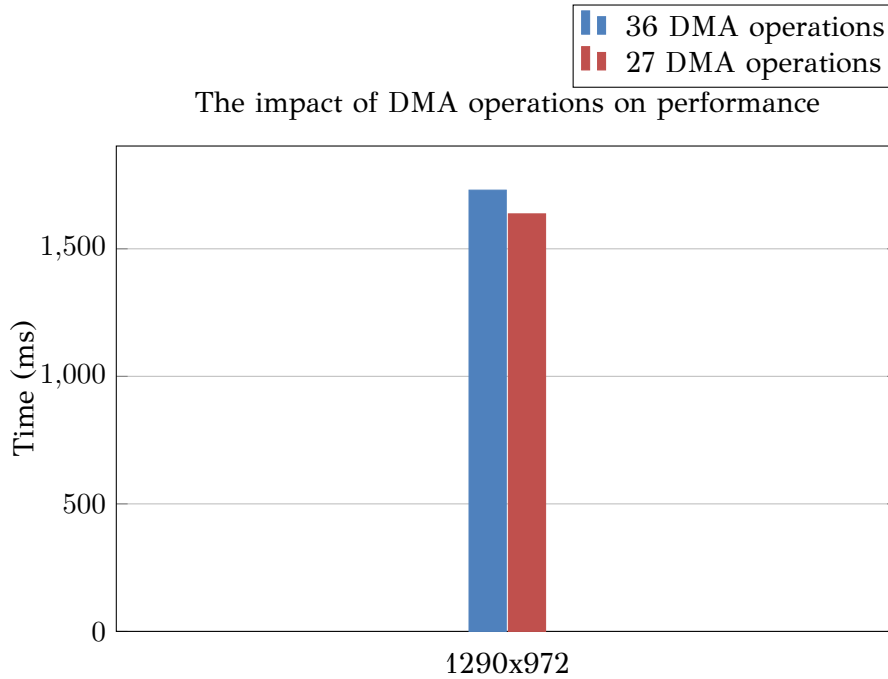
$$Padding = 5/2 + 7/2 + 3/2 = 2 + 3 + 1 = 6.$$

The first and the last slices will have only 6 lines of padding added, while those in the interim must have double padding, since they need 6 lines from the previous as well as from the next image slices.

In addition, since the SHAVE will perform the same operation for different slices of the image, it is essential to save the results of one slice before moving to the next. This is done by using two arrays in LeonOS that will receive the results after each SHAVE iteration. In particular, before moving to a next slice, the SHAVE transfers with DMA from the CMX back to the DDR the cornerness values that it has calculated for each of the pixels of this slice, as well as the values of the binary array that holds the pixels that are locally maximum. Finally, after the SHAVE has finished computing these values for the whole image, the Leon processor takes them and calculates the coordinates of the corners in the image.

After the implementation is configured to perform the algorithm on the slices of the image, the focus is shifted towards measuring the impact that the *number of DMA operations* has on the performance. This feature was identified in the previous Chapter as one of the factors that affect significantly the efficiency of software applications in Myriad 2. For our tests we have used the usual 512x384 image as well as a bigger image with dimensions 1296x972. In order to decrease the number of DMA operations, we choose image slices with bigger heights. The results can be seen in the following charts :





The above charts show that a lower number of DMA operations can increase significantly the performance of the implementation. In the case of the 512x384 image, we achieved a 13% gain in performance when reducing the DMA operations from 12 to 4.

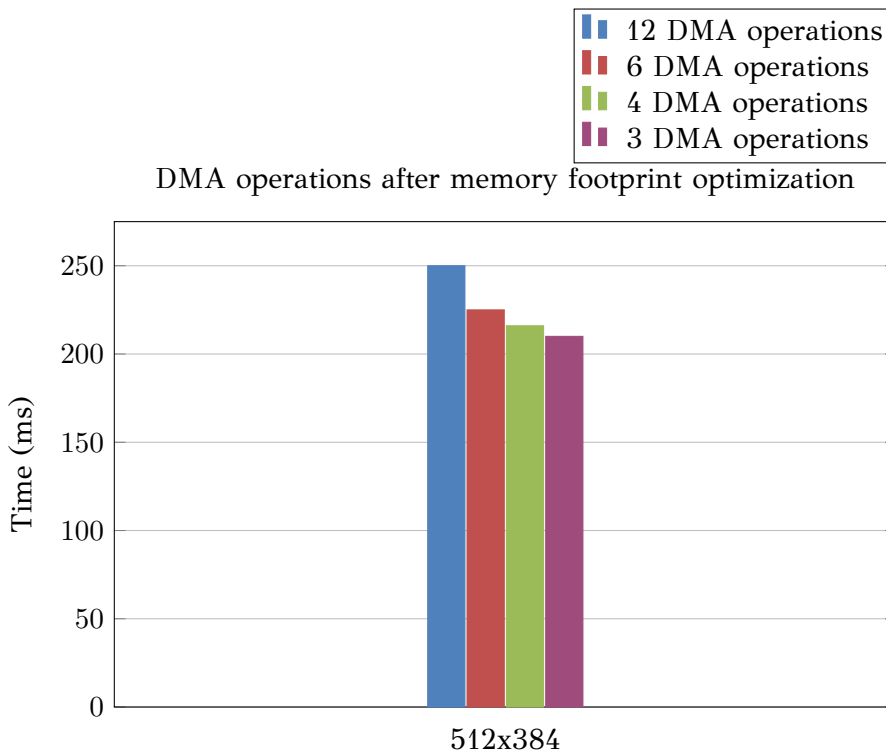
Regarding the larger image, even though the processing was complete almost 100ms earlier after reducing the number of DMA operations, we have achieved a smaller performance gain of around 5%. The much larger width that this image has does not allow us to increase the height even further in order to reduce the number of DMAs in a number that can give as a substantial drop in the processing time.

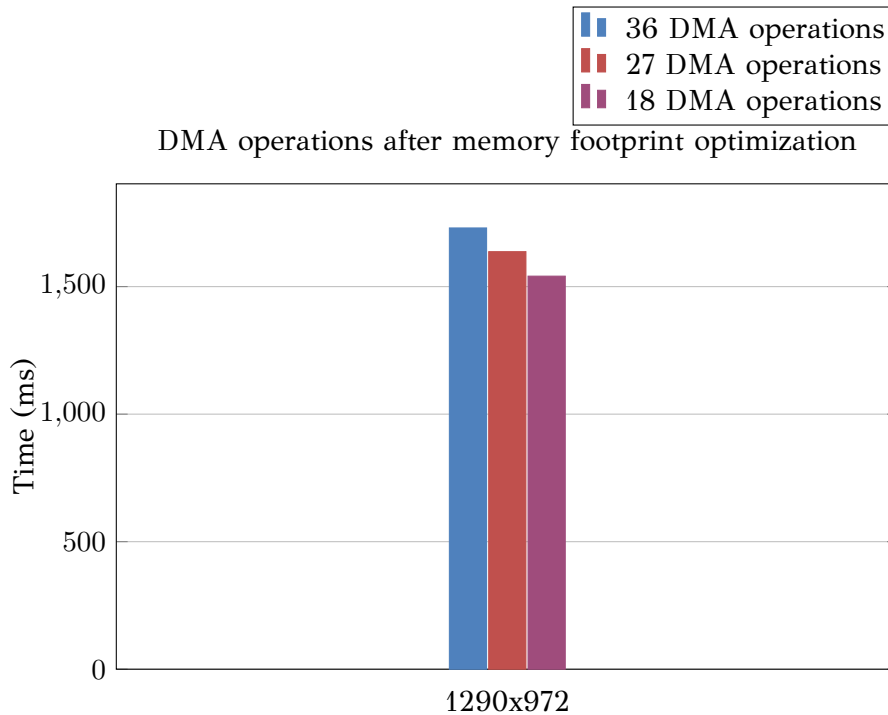
Therefore, we deduce that in order to reduce the number of necessary DMA operations, we must decrease the memory footprint of the application. As stated in the beginning of this section, the main challenge is to reduce the size of the `ibuf[]` array, which holds the `gradx`, `grady`, `gradx2`, `grady2` and `gradxy` values. The equation that performs the calculation of the Harris score R needs only the I_x^2 , I_y^2 and $I_x I_y$. After examining the implementation of Harris Corner Detection, we deduce that it is not mandatory to hold in this array the values of the first order as well as of the squared partial derivatives. It was developed this way because it targeted general purpose CPUs with large amounts of available memory.

Hence, we can reduce the size of the array by removing `gradx` and `grady`. The 1st order partial derivatives can be placed in the same memory address where the squared derivatives will be calculated. However, before modifying the size of the array, we had to make some adjustments to the code. The function `imgradient5_smo()` places the final results in these buffers, while it also uses `gradx2` and `grady2` as working buffers, for

the intermediate results. Our work around was to keep the final results in gradx2 and grady2 and use gradxy and cornerness as the working buffers. Note that cornerness is an with float values, so it had to be cast to integer for this modification. in addition, the function imgblurg() , performs the gaussian blur on the squared derivatives by using gradx as working memory. Here we have again set cornerness for this use.

After performing this memory optimization, the size of the ibuf[] has been reduced from 5 x WIDTH x HEIGHT to 3 x WIDTH x HEIGHT. In the case of the 512x384 image, the data of the SHAVE have been reduced from 541408 bytes to 361184, for slices with height set to 32. This means that we have achieved a **33% decline in the memory footprint**. Now that more memory is available for the SHAVE in the CMX, we can increase the slice's height and have even less DMA operations. Our results are presented in the below charts :





The above charts demonstrate that we have achieved substantial performance gain, especially for the larger image, where the initial 5% is increased to 11.

In general, we can conclude that for each reduce of the number of DMAs to half, we have a 10% performance gain.

Finally, this decreased memory demand allows us to test a 2Mpixel image, with dimensions 1584x1290, despite the fact that we do not fragment the images vertically until this stage. The current implementation took 2691 ms to apply Harris corner detection in this image.

4.2.5 Fourth Stage - Task Based Parallelism

The number of DMA operations can be decreased only until a certain point, which is determined by the amount of available CMX memory. In addition, the numbers regarding the performance of the previous stage illustrate that there is still much room for development. A more significant optimization is to apply task based parallelism to our implementation, by using the maximum number of available SHAVEs.

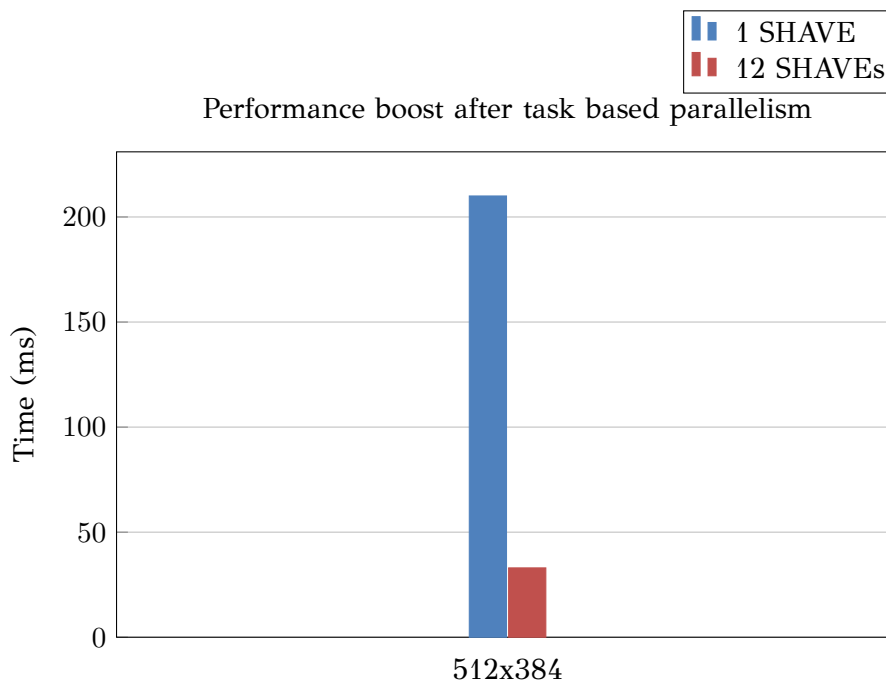
Now, each SHAVE is responsible of applying the Harris algorithm in a specific slice of the image. The height of the slice that is allocated in each SHAVE is determined by the height of the image and the number of shaves used. For example, in the case of the 512x384 image, we will have slices with height $384/12 = 32$.

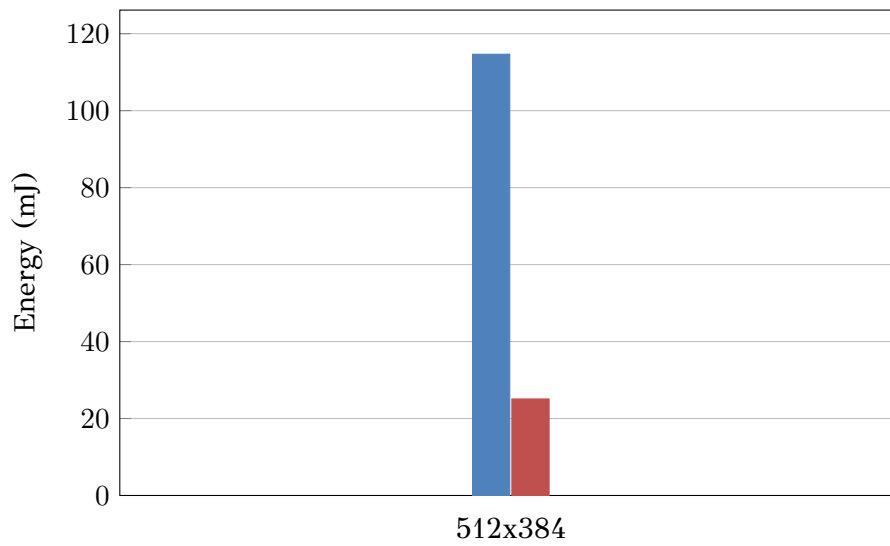
In addition, since we will be using all the SHAVEs, each one will have only 128KB of data in the CMX. This has forced us to perform fragmentation of the image not

only horizontally, but vertically too. This means that, based again on the convolution operations, we must also add vertical padding in the image slices.

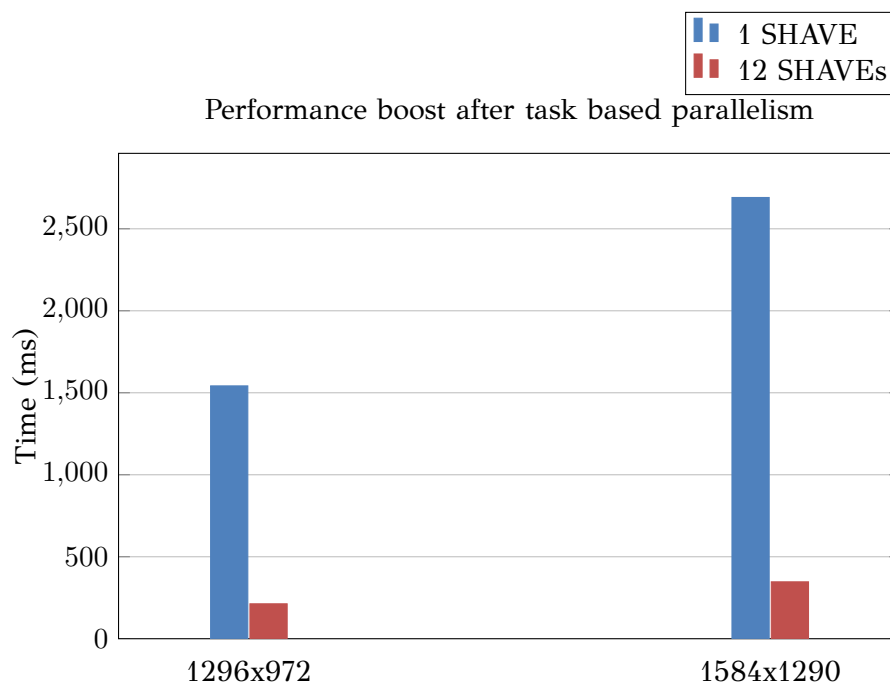
The target here is also on achieving an as high as possible utilization of the CMX memory available for each save. In this direction, we noticed that the size of our code is just above 25 KB. Therefore, we have allocated 26KB to the code of each SHAVE and have increased the data segment from 96 to 102KB. This extra space has allowed us to reach a *95% CMX utilization*, meaning that we will perform the minimum number of necessary DMA operations.

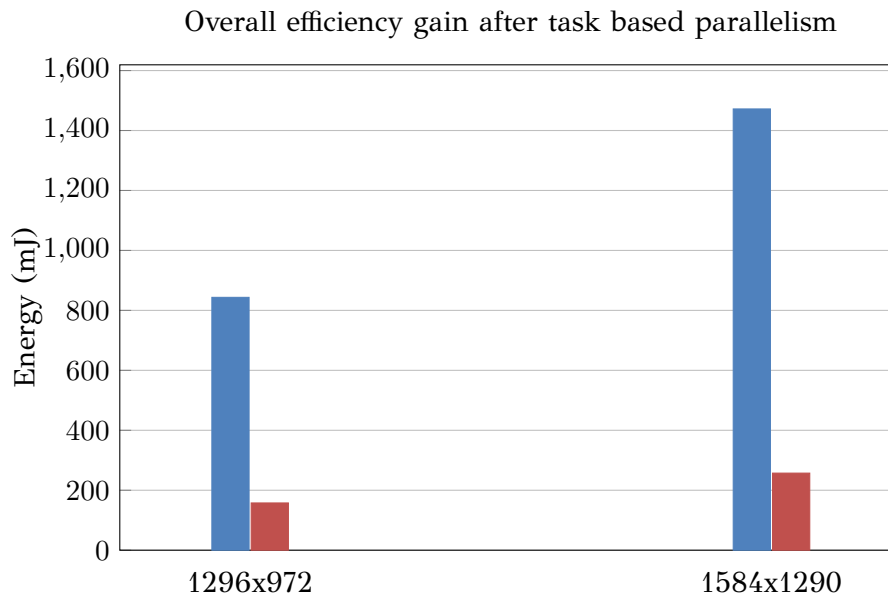
Since we will be using more SHAVEs, there is an increase of the power usage of the Myriad 2. Particularly, we move from around 550 mW in the case of 1 SHAVE, to 800mW when using all the available SHAVEs. Hence, it is essential to provide results not only for the time, but for the Energy consumption as well. After making all the necessary modifications, we can present the following results :





As expected, despite the increased power drain, the 12 shaves have offered a spectacular increase in the overall efficiency of the implementation. The results concerning the time as well as the energy (as an overall efficiency index) are also given for the two larger images :





It is obvious that in this stage we have achieved the most significant increase in the efficiency of the implementation and we have also reached a sufficient performance. This is due to the fact that we have made all the necessary steps towards transforming the initial code from a generic C to a Myriad 2 - specific implementation. Now, the code and the data is placed in the correct memory and we have also achieved the minimum number of DMA operations. In addition, the algorithm runs in parallel between the SHAVEs leading to a spectacular increase in processing power.

4.2.6 Fifth stage - Algorithmic optimization

In all the previous stages we have applied, step by step, transformations that were based on the design space that was built in Chapter 3. After reaching a sufficient level of performance, we try to examine the implementation from a higher level and see what transformations could be done in the implementation in order to achieve a more optimal performance.

The functions that consume the more processing time are `imggradient5_smo()` and `imgblurg()`. The latter, especially, is invoked three times, one for each of the I_x^2 , I_y^2 and $I_x I_y$, and is responsible for the largest part of the total computations.

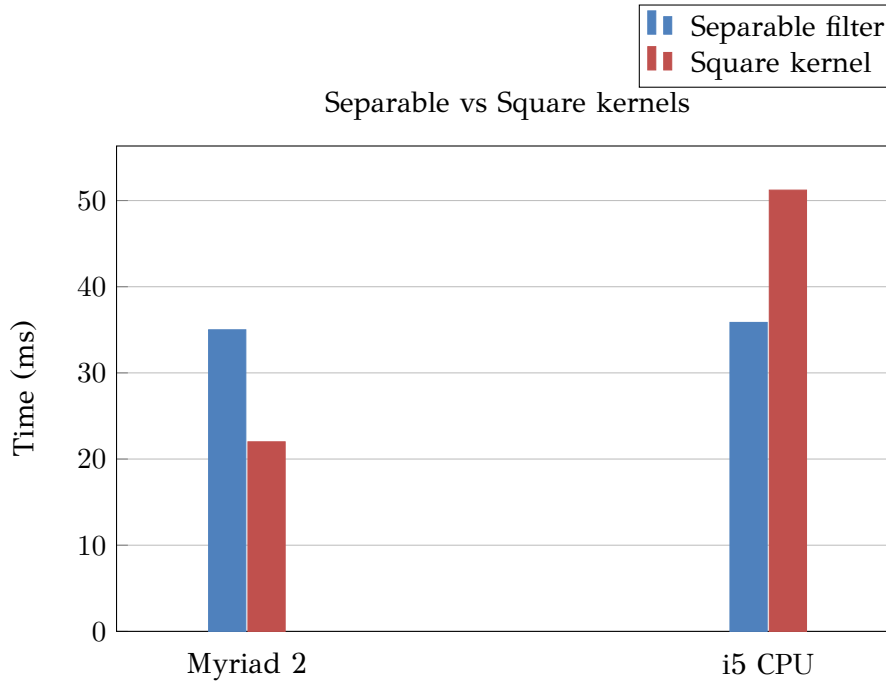
As stated in section 4.2.1, these functions apply separable convolutions on the image. The developer of the initial code has chosen to use separable filters because they offer a reduced number of computations. In particular, if we want to apply a convolution of an image with dimensions $N \times N$ and a convolution kernel $M \times M$, the number of multiplies needed for this operation would be $N * N * W * W = N^2 W^2$. However, in our case, the filters are separable, meaning that we can get the square kernel by

convolving a single column vector by some row vector. By taking advantage of the associative property of the convolution, the operation can be described as :

$$G = H * F = (C * R) * F = C * (R * F)$$

, where G is the operation of convolving a square kernel H with an image F . Since, as we said, the square kernel H can be replaced by the separable $C * R$, this operation can be performed by applying the separable filters on the image, horizontally and vertically. Now, the number of necessary multiplies is $2 * W * N^2$, which can be much smaller than $N^2 W^2$, given that W is big enough.

From the above we can see that, in general, it is preferable to use separable convolutions when it is possible because they lead to better performance. However, the knowledge we have achieved regarding Myriad 2 and the architecture of the SHAVEs, points to the fact that they can benefit by applying square kernels in chunks of the image. A traditional microprocessor, operates on information that it receives in series. Myriad 2, instead, is designed for operating in parallel on tons of information that they are all coming through at once. Contrary to general purpose CPUs, such as the Intel i5 we have used earlier, the SHAVEs are more powerful when they are given a larger window to operate in. The former are more aligned with the concept of computational complexity as we know it. Since they perform computations one - by - one, yielding in our case the values for each pixel of the image, the reduced number of operations that separable filters provide leads to a better performance. However, in the case of the SHAVEs, we got better performance by replacing the separable filters with square kernels that perform more operations but by considering larger windows at each iteration. The following charts present a comparison between these two approaches, both for Myriad 2 and for the Intel i5 CPU.



The above data illustrate that the performance of Myriad 2 has increased significantly with the square kernels. Even though the separable kernel has a reduced number of multiplies, the SHAVEs have performed better with the square because they apply larger windows of convolution on the image slices. However, the square kernels have resulted in a substantial performance drop for the Intel i5 CPU, since the number of required operations is increased. It is also worth noting that in the case of the i5 CPU, the image is not fragmented. The algorithm is applied gradually on the whole image, which means that when the function `imgblurg()` is invoked, it has a very large image matrix to apply the square kernel on. Therefore, the larger number of multiplies has a significant impact on its performance.

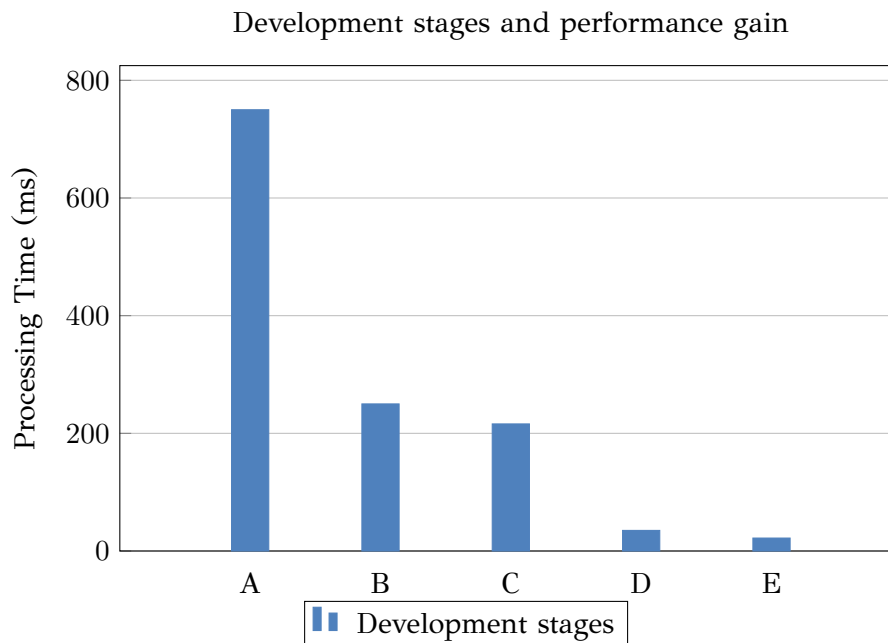
Chapter 5

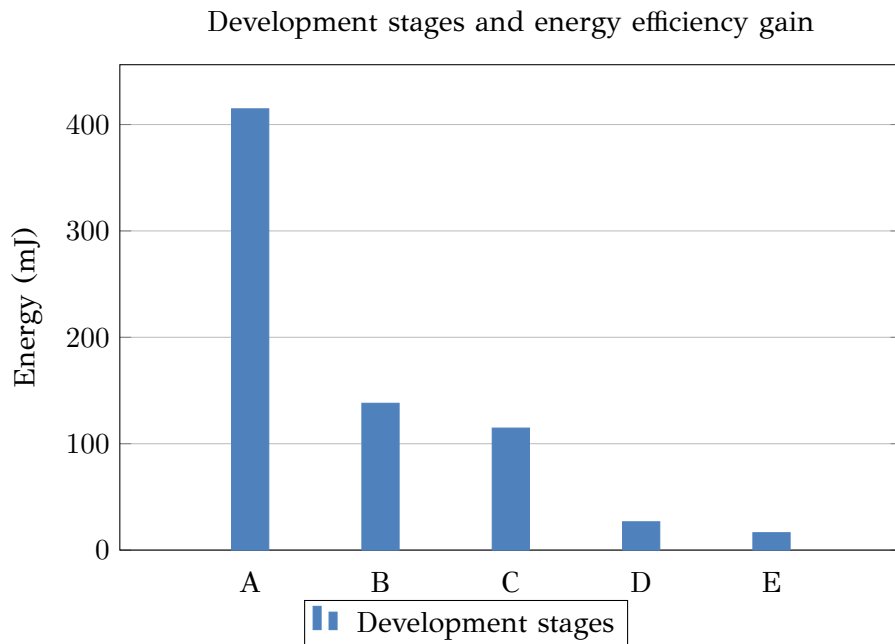
Conclusion

In this thesis, we examined a methodology for developing sophisticated Computer Vision applications into embedded architectures. As a case study, we worked on efficiently implementing the Harris & Stephens algorithm in a platform specialized for vision tasks, Movidius' Myriad 2. Deploying a complex algorithm such as Harris corner detection into an embedded architecture is an interesting concept for many scientists and software engineers who have access to relevant software libraries for general purpose machines and want to know the effort needed to deploy them into an embedded platform.

As a first stage, we evaluated basic Computer Vision filters in order to build a design space on which we based the steps of our Harris implementation. The main challenges of this implementation were posed by the constraints in the memory size, a common feature of embedded systems. Apart from modifications that reduced the memory overhead of the application, several optimizations have also been applied in order to reach high efficiency levels.

The performance gain that we achieved in each step is presented in the following chart, in terms of time as well as energy efficiency, for an image with dimensions 512x384.





From the above data we can make the following observations regarding the performance gain we have achieved in each stage :

DDR vs CMX

As explained in Chapter 2, the working buffers of an application that runs on the SHAVE processors should be placed in the CMX memory. However, in the first stage of porting a generic C code into Myriad 2, the small size of CMX requires placing those buffers in DDR, since they were originally built for machines with no memory limitations. Then, after making the essential modifications in the implementation, the buffers are located in the correct memory and the performance that we get starts to be more aligned with the capabilities of Myriad 2, as can be seen by comparing stages A and B of the charts. Therefore, DDR can be used to hold data only at a very primal stage of porting code from general purpose CPUs into Myriad 2.

Number of DMA operations

After reaching a much higher performance by placing the buffers in CMX, the focus is shifted towards utilizing this memory as efficiently as possible. The overhead that is applied in an application by memory accesses is a common feature of embedded platforms. For this purpose, modifications are used in order to reduce the memory overhead of the application and perform less DMA operations. This optimization has yielded in our case a performance gain of around 15%.

Task based parallelism

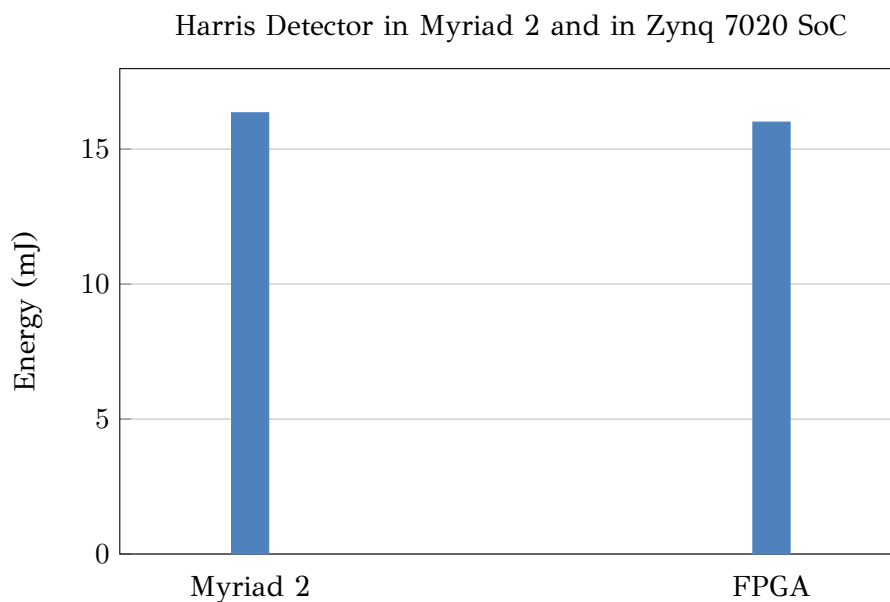
The previous steps were necessary in order to reach the most sufficient performance

we could get with 1 SHAVE. However, the real processing power of Myriad 2 is not exploited until we use all the SHAVE processors available. Having 12 VLIW run our algorithm in parallel has resulted in spectacular efficiency gain, as can be seen from the difference between stages C and D of the charts.

Algorithmic Transformations

Until this stage we have exploited all the features of Myriad 2 and have achieved significant performance gain in our application. However, when deploying a source code that was originally built for a general purpose CPU, it is essential to examine optimizations not only to a platform level but to an algorithmic level as well. In our case, we focus on enhancing the performance of the convolution operations, which consumed the largest part of the processing. This was achieved by replacing the separable filters with square kernel filters that, even if they require more operations, are more aligned with the processing scheme of the SHAVEs. Myriad 2, instead of traditional processors, is designed for operating in parallel on tons of information that they are all coming through at once. Therefore, it was able to benefit from a larger operating window size (stage E).

Combining all the above stages, we have managed to reach a very solid performance in Myriad 2 for a complex computer vision application such as Harris corner detection. The following chart illustrates a comparison of the same implementation in Myriad 2 and an FPGA device, the Zynq 7020 (XC7Z020) [4]. The comparison is based on the Energy consumption of these devices, as an index of overall efficiency.



As can be seen from the above chart, these devices have almost the same energy efficiency when running the Harris & Stephens algorithm. In particular, the total processing time in FPGA is 8ms (1.5 for communication and 6.5 for processing) and its power consumption is around 2 W. In Myriad 2, we have reached a processing time of 22ms at 740 mW, resulting in a very similar energy efficiency between these devices. Since there is no significant difference in the efficiency of these embedded platforms, the choice between them should be based on other factors. For example, FPGAs can be used when more processing power is wanted, since by porting more devices it can reach even higher performance levels, at the expense of power consumption and cost. On the other hand, Myriad 2 is exceptional at handling computer vision tasks at an ultra low power state, since it consumes less than 1 W. Therefore, it is more suitable for mobile devices with power limitations, such as small robots, drones or wearables.

Finally, our implementation proved that Myriad 2 can accelerate significantly a high-complexity Computer Vision algorithm, like Harris corner detection, and deal with the intensive computational load efficiently, even without using the optimized CV library that Movidius provides.

Chapter 6

Future Work

Nowadays, we are emerging into a new era of personal computing, with a demand for intelligent devices who are able to understand their environment. In order to be able to run the complex algorithms needed for such tasks, the traditional ways of performance improvement on these devices have to be reconsidered. Moore's law is slowing down which is causing the power and performance benefits in transitioning to the next process technology node to decrease. Therefore, designers of embedded platforms have to come up with more artful ways of creating powerful and at the same time, power efficient devices.

In this thesis we worked with a device which was designed in this way, based on the understanding that there is a deep interdependency between algorithms and chip architecture. Myriad 2 is a device specialized for efficiently performing machine vision tasks. Therefore, it was used in this thesis to develop a complex computer vision algorithm, the Harris & Stephens Corner detector. However, the basic techniques described in this thesis can also be used to improve other similar applications. For example, a very promising scientific area with algorithms that could be accelerated by Myriad 2 is Deep Learning. Even though such algorithms were developed to run mainly on supercomputers, they can now be ported into powerful embedded devices and be used in real world applications.

In addition, an interesting step forward would be to make similar implementations on other embedded devices that were designed to offer high processing power into mobile applications. Such a device is Nvidia's Tegra processors, with applications varying from mobile gaming platforms to autonomus vehicles and intelligent robots.

It is believed that the next decade will mark the start of a new era of special-purpose processors focused on decreasing the energy per operation in a new way. Using these processors to bring complex algorithms, such as the one examined in this thesis, in mobile devices used in everyday life is a very interesting concept with promising results.

Bibliography

- [1] B. Triggs, “Detecting keypoints with stable position, orientation, and scale under illumination changes,” *Eighth European Conference on Computer Vision (ECCV 2004)*, pp. 100 – 113, 2004.
- [2] C. Harris and M. Stephens, “A combined corner and edge detector,” *Plessey Research Roke Manor, United Kingdom*, 1988.
- [3] B. Barry, D. Moloney *et al.*, “Always - on vision processing unit for mobile applications,” *IEEE Computer Society*, 2015.
- [4] I. Stratakos, G. Lentaris, K. Maragos, D. Reisis, and D. Soudris, “A co-design approach for rapid prototyping of image processing on soc fpgas,” *Proceedings of the 20th Panhellenic Conference on Informatics, ACM*, pp. 1 – 6, 2016.
- [5] R. Szeliski, *Computer Vision : Algorithms and Applications*. Springer, 2010.
- [6] N. Snavely, S. M. Seitz, and R. Szeliski, “Photo tourism: Exploring photo collections in 3d,” *ACM Transactions on Graphics*, vol. 25, pp. 835 – 846, 2006.
- [7] M. Goesele *et al.*, “Multi-view stereo for community photo collections,” *Eleventh International Conference on Computer Vision (ICCV 2007), Rio de Janeiro, Brazil*, 2007.
- [8] H. Sidenbladh, M. J. Black, and D. J. Fleet, “Stochastic tracking of 3d human figures using 2d image motion,” *Sixth European Conference on Computer Vision (ECCV 2000), Dublin, Ireland*, pp. 702 – 718, 2000.
- [9] J. Sivic, C. L. Zitnick, and R. Szeliski, “Finding people in repeated shots of the same scene,” *British Machine Vision Conference (BMVC 2006), Edinburgh*, pp. 909 – 918, 2006.
- [10] L. G. Shapiro and G. C. Stockman, *Computer Vision*. Pearson, 2001.
- [11] A. Willis and Y. Sui, “An algebraic model for fast corner detection,” *IEEE 12th International Conference on Computer Vision*, pp. 2296 – 2302, 2009.
- [12] J. Canny, “A computational approach to edge detection,” *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 8, pp. 679 – 698, 1986.
- [13] J. Shi and C. Tomasi, “Good features to track,” *IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR’94)*, vol. 106, pp. 593–600, 1994.
- [14] B. K. P. Horn and B. G. Schunck, “Determining optical flow,” *Artificial Intelligence*, vol. 17, pp. 185 – 203, 1981.

- [15] E. Rosten and T. Drummond, "Machine learning for high-speed corner detection," *Proc. European Conf. Computer Vision*, pp. 430 – 440, 2006.
- [16] G. Lentaris, I. Stamoulias, D. Soudris, and M. Lourakis, "Hw/sw co-design and fpga acceleration of visual odometry algorithms for rover navigation on mars," *IEEE Transactions on Circuits and Systems for Video Technology*, vol. 26(8), 2015.