ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ
ΣΧΟΛΗ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ
ΤΟΜΕΑΣ ΤΕΧΝΟΛΟΓΙΑΣ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΥΠΟΛΟΓΙΣΤΩΝ

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

Design and implementation of image processing algorithms on embedded CPU-GPU-SoC devices

Ιωάννης Ορούτζογλου
Α.Μ. : 03112124

Επιβλέπων : Δημήτριος Ι. Σούντρης
Αναπληρωτής Καθηγητής ΕΜΠ

Αθήνα
Ιούνιος 2018

ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ
ΣΧΟΛΗ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟ
ΓΙΣΤΩΝ
ΤΟΜΕΑΣ ΤΕΧΝΟΛΟΓΙΑΣ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΥΠΟΛΟΓΙΣΤΩΝ

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

Design and implementation of image processing algorithms on embedded CPU-GPU-SoC devices

Ιωάννης Ορούτζογλου
Α.Μ. : 03112124

Επιβλέπων : Δημήτριος Ι. Σούντρης
Αναπληρωτής Καθηγητής ΕΜΠ

Τριμελής Επιτροπή Εξέτασης

(Υπογραφή)	(Υπογραφή)	(Υπογραφή)
....	....	....
Δημήτριος Σούντρης	Κιαμάλ Πεκμεστζή	Γεώργιος Γκούμας
Αναπληρωτής Καθηγητής	Καθηγητής	Επίκουρος Καθηγητής
ΕΜΠ	ΕΜΠ	ΕΜΠ

Ημερομηνία Εξέτασης:
19 Ιουνίου 2018

Ευχαριστίες

Καταρχάς, θέλω να εκφράσω την ευγνωμοσύνη μου στον επιβλέποντά καθηγητή μου κ. Δημήτριο Σούντρη που με εμπιστεύθηκε για αυτήν την διπλωματική εργασία. Επιπρόσθετα, θέλω να ευχαριστήσω τους υποψήφιους διδάκτορες Δημοσθένη Μασούρο και Βασίλειο Τσούτσουρα για την αμέριστη βοήθεια και τη συνεργασία τους καθ ΄όλη τη διάρκεια της διπλωματικής μου. Οι εύστοχες παρατηρήσεις τους και οι συνομιλίες μας, με ενέπνευσαν και με βοήθησαν να επεκτείνω τις γνώσεις μου γύρω από το θέμα της διπλωματικής αυτής εργασίας. Θέλω, επίσης, να ευχαριστήσω τον υποψήφιο διδάκτορα Ιωάννη Στρατάκο για τις πειραματικές μετρήσεις που μου παρείχε καθώς και όλα τα μέλη του Microlab για το ευχάριστο και φιλικό περιβάλλον εργασίας, παράγοντας πολύ σημαντικός για μένα.

Τέλος, θέλω να ευχαριστήσω τους γονείς μου Στράτο και Γαρυφαλλιά, την αδερφή μου Κατερίνα, το θείο μου Χρήστο και τους φίλους μου και κοντινούς μου ανθρώπους Ελπίδα, Γιώργο, Αργύρη, Νίκο, Βασίλη και Χρήστο για την όμορφη και ενθουσιώδη φοιτητική ζωή που μοιραστήκαμε. Είμαι ευγνώμων σε όλους τους, για τη δύναμη και το πάθος που μου δίνουν να ακολουθήσω τα όνειρά μου.

Acknowledgments

I would like to express my gratitude to my supervisor, Prof. Dimitrios Soudris for trusting me with this diploma thesis. Also i would like to thank the PhD candidates Dimosthenis Masouros and Vasileios Tsoutsouras for their help and cooperation throughout the course of my diploma thesis. Their insightful comments during our conversations helped me improve and expand my knowledge around the topic of this diploma thesis. I would also like to thank the PhD candidate Ioannis Stratakos providing me with experimental results and all the members of Microlab for the pleasant working environment.

Last but not least, I would like to thank my parents Stratos and Garyfallia, my sister Katerina, my uncle Christos and my friends Elpida, Giorgos, Argyris, Nikos, Vasilis and Christos for the great and beautiful student life we shared together. I am thankful to these people, giving me the strengh and the passion to pursue my goals.

Περίληψη

Σήμερα η συνεχής πρόοδος στην τεχνολογία έχει οδηγήσει στην ανάπτυξη πιο πολύπλοκων και υπολογιστικά απαιτητικών αλγορίθμων επεξεργασίας εικόνας. Πολλοί από αυτούς τους αλγόριθμους έχουν υιοθετηθεί σε ενσωματωμένα συστήματα τα οποία στοχεύουν σε μια ποικιλία εφαρμογών, όπως η αυτοκινητοβιομηχανία, 3D πλοήγηση, η επιτήρηση, κλπ. Ωστόσο, σε ενσωματωμένα συστήματα πραγματικού χρόνου, όπου η καθυστέρηρη μεταφοράς δεδομένων και η κατανάλωση ισχύος διαδραματίζουν σημαντικό ρόλο, εφαρμογές λογισμικού προσανατολισμένες να εκτελούνται σε επεξεργαστές γενικής χρήσης δεν μπορούν να προσφέρουν ικανοποιητικές λύσεις.

Σκοπός της παρούσας διπλωματικής εργασίας είναι ο σχεδιασμός ενός συστήματος επεξεργασίας εικόνας για ενσωματωμένες εφαρμογές, η υλοποίησή του σε Σύστημα-σε-Ψηφίδα και η αξιολόγησή του. Ως εφαρμογή επιλέχθηκε η ανίχνευση γωνιών με τη χρήση του αλγορίθμου Harris σε σύστημα πραγματικού χρόνου διαβάζοντας την είσοδό από την camera. Η επιτάχυνση του αλγορίθμου Harris - Corner Detector επιτυγχάνεται στην πλατφόρμα Tegra X1 CPU-GPU-SoC με χρήση της γλώσσας CUDA C. Ερευνήσαμε διαφορετικούς τρόπους επικοινωνίας μεταξύ CPU και GPU καθώς και προγραμματιστικές τεχνικές στη GPU με σκοπό την καλύτερη απόδοση όσον αφορά τον χρόνο εκτέλεσης της εφαρμογής αλλα και την κατανάλωση ενέργειας. Πιο συγκεκριμένα, ερευνήσαμε τεχνικές με χρήση της κοινής μνήμης της GPU, shared memory, της constant αλλά και της texture μνήμης. Τέλος, καθοριστική και πιο σημαντική για την επιτάχυνση της εφαρμογής ήταν η τεχνική της διαχωριστικότητας (separability), με την οποία οι δισδιάστατες συνελίξεις της εικόνας εκτελούνται ως δύο ξεχωριστές μονοδιάστατες συνελίξεις, πράγμα που βελτιώνει σημαντικά την απόδοση του συστήματος. Τα πειράματα της τελικής υλοποίησης δείχνουν κέρδος μέχρι $\times 73$ συγκριτικά με την αρχική υλοποίηση της εφαρμογής σε έναν επεξεργαστή ARM Cortex A57. Έτσι, μπορούμε να υλοποιήσουμε την εφαρμογή αυτή σε σύστημα πραγματικού χρόνου, διαβάζοντας την είσοδό μας απο την camera,χωρίς να παρατηρείται καθυστέρηση στα frames. Τέλος, στοχεύουμε στη σύγκριση μεταξύ των CPU-GPU και CPU-FPGA συνδυασμών,προκειμένου να αποφανθούμε για την πιο αποδοτική πλατφόρμα που μπορεί να υποστηρίξει την εφαρμογή αυτή.

Λέξεις Κλειδιά— GPU, CUDA C, Επεξεργασία Εικόνας, Σύστημα-σε-Ψηφίδα, Harris Corner Detector, Ενσωματωμένο Σύστημα, Tegra x1, Maxwell αρχιτεκτονική.

Abstract

Nowadays the ever-increasing advancements in technology has led to the deployment of more complex and computationally intensive image processing algorithms. Many of these algorithms have been adopted in present-day embedded systems targeting a variety of applications such as automotive, 3D navigation, surveillance, etc. However in real-time embedded systems, where latency and power play an important role, software-oriented implementations running on general purpose CPUs may not offer satisfactory solutions.

The purpose of this thesis is the design of an image processing system for embedded applications, its deployment on a System-on-Chip (SoC) platform and the evaluation of the developed system. As a case study was selected the Harris Corner Detector algorithm, in a real time system getting the input image from a camera. The thesis focus on the acceleration of the Harris-Corner Detector algorithm on the Tegra X1 GPU SoC. More specifically, we examine different ways of communication between CPU-GPU and various programming techniques on GPU with respect to the execution time and the power consumption. Experimental results show an acceleration of up to x74 compared to a pure software implementation on ARM Cortex A57 using CUDA C. Thus, we can easily implement a real-time corner detection getting the input from the camera, without observing any frames latency. In addition, the thesis focuses to the comparison between CPU-GPU and CPU-FPGA (ZC702) combination for the best acceleration of the application.

Keywords— GPU, CUDA C, Image Processing, System-on-Chip, Harris Corner Detector, Embedded System, Tegra x1,Maxwell architecture

Contents

List of Figures

Εκτεταμένη Περίληψη

Εισαγωγή

Ο τομέας της επεξεργασίας εικόνας και οι αλγόριθμοί του αποτελούν μεγάλο κομμάτι της συνεχούς προόδου που έχει επιτύχει η τεχνολογία. Πολλοί από τους αλγόριθμους αυτούς έχουν υιοθετηθεί από ενσωματωμένα συστήματα τα οποία στοχεύουν σε εφαρμογές όπως η αυτοκινητοβιομηχανία, η 3D πλοήγηση κλπ. Ωστόσο, τα ενσωματωμένα συστήματα πραγματικού χρόνου, λόγω της καθυστέρηρης μεταφοράς δεδομένων αλλά και της κατανάλωσης ισχύος δεν μπορούν να υποστηρίξουν ικανοποιητικά τέτοιου είδους εφαρμογές με επεξεργαστές γενικής χρήσης.

Προκειμένου να αυξηθεί η απόδοση/watt, διάφορες προσεγγίσεις έχουν προταθεί όπου εξειδικευμένο υλικό χρησιμοποιείται παράλληλα με την CPU για την επιτάχυνση κρίσιμων τμημάτων ή ακόμη και ολόκληρων αλγορίθμων. Οι προσεγγίσεις αυτές βασίζονται σε διάφορους συνδυασμούς σε επίπεδο συστήματος, όπως CPU-DSP, CPU-FPGA και CPU-GPU. Η παρούσα διπλωματική πραγματεύεται τον CPU-GPU συνδυασμό και σκοπεύει στην καλύτερη εκμετάλλευσή του ώστε να επιτύχει το καλύτερο δυνατό performance per watt σε μια εφαρμογή επεξεργασίας εικόνας, τον αλγόριθμο Harris, ανιχνευτή γωνιών, καθώς και να συγκριθεί με τον CPU-FPGA συνδυασμό,προκειμένου να αποφανθούμε για την πιο αποδοτική πλατφόρμα που μπορεί να υποστηρίξει την εφαρμογή αυτή.

Πλατφόρμα Υλοποίησης

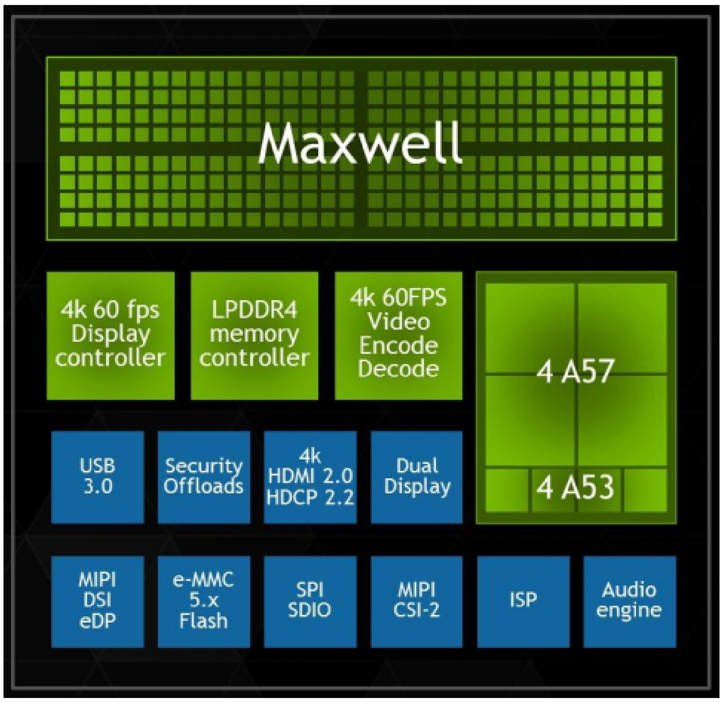
Η ενσωματωμένη πλατφόρμα πάνω στην οποία θα επιταχυνθέι ο αλγόριθμος Harris ειναι η NVIDIA Tegra x1. Πρόκειται για μία ενσωματωμένη πλατφορμα (Σύστημα σε Ψηφίδα) κατασκευασμένη απο την NVIDIA η οποία περιέχει έναν ARM επεξεργαστή και μία GPU αρχιτεκτονικής Μαξωελλ.

Πιο συγκεκριμένα η πλατφόρμα Τεγρα Ξ1 περιλαμβάνει:

- ARM Cortex A57/ A53 64/32-bit CPU αρχιτεκτονική η οποία προσφέρει απόδοση και χαμηλή κατανάλωση
- Maxwell GPU αρχιτεκτονική με 256 πυρήνες ώστε να επιτύχει κορυφαίες επιδόσεις και αποδοτικότητα ισχύος για εφαρμογές επεξεργασίας εικόνας και όχι μόνο

Στην Εικόνα 1 παρουσιάζεται η αρχιτεκτονική της πλατφόρμας NVIDIA Tegra x1.

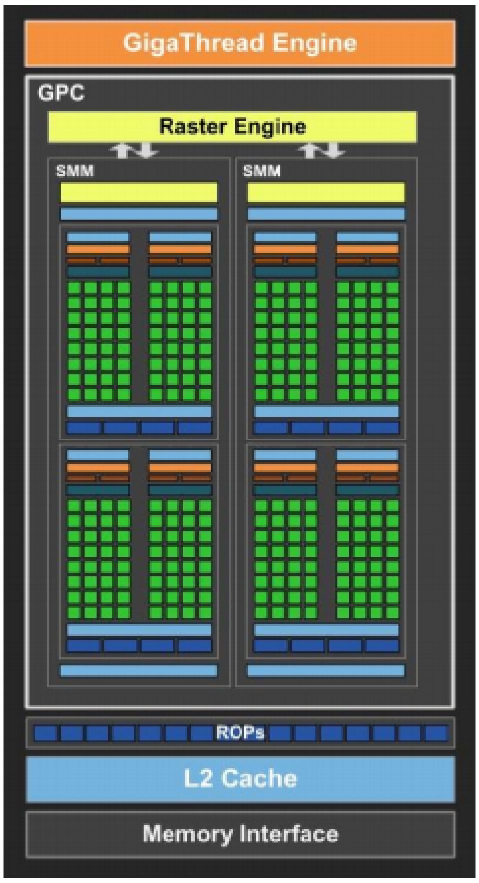
Όσον αφορά τη CPU αποτελείται από 4 ARM Cortex A57 υψηλής απόδοσης πυρήνες και 4 χαμηλής κατανάλωσης ARM Cortex A53 πυρήνες. O ARM Cortex A57 επεξεργαστής μοιράζεται μία 2MB L2 κρηφή μνήμη, καθώς κάθε πυρήνας διαθέτει μία 48KB L1 κρυφή μνήμη εντολών και μία 32KB L1 κρυφή μνήμη δεδομένων. Η GPU, ο επιταχυντής της ενσωματωμένης πλατφόρμας μας, αποτελείται απο Graphics Processing Clusters (GPC), Streaming Multiprocessors (SM) και ελεγκτές μνήμης. Η Maxwell GPU στο Tegra x1 αποτελέιται απο 2 SMs που το καθένα συνολικά



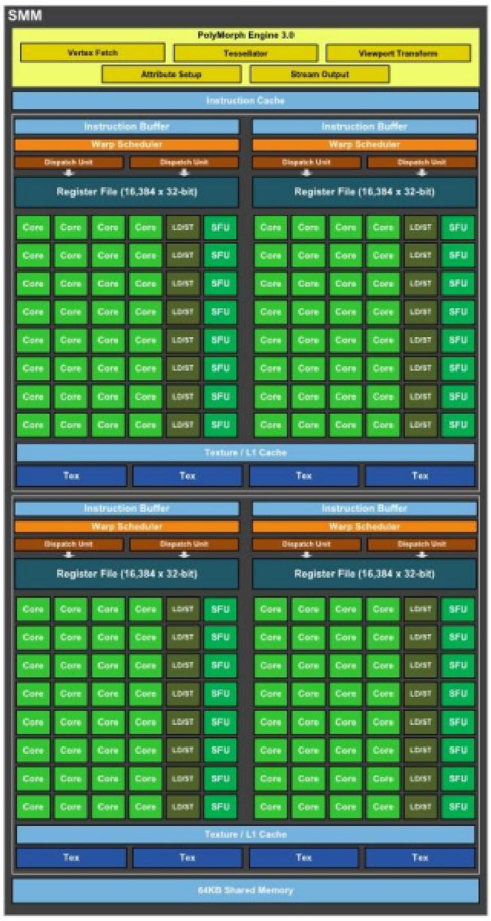
Εικόνα 1: Αρχιτεκτονική Nvidia Tegra x1.

περιέχει 128 πυρήνες. Συνεπώς συνολικά η GPU μας διαθέτει 256 πυρήνες, γεγονός που θα μας χρειαστεί στην αξιολόγηση της απόδοσης της εφαρμογής μας. Στην Εικόνα 2 παρουσιάζεται η αρχιτεκτονική της GPU στην NVIDIA Tegra x1.

Η καρδιά μίας GPU είναι το SM. Στην περίπτωσή μας η Tegra x1 διαθέτει 2 SMs, το καθένα από τα οποία, (SMM αποκαλείται στη Maxwell αρχιτεκτονική) περιέχει 4 δρομολογητές wrap όπου κάθε δρομοογητής είναι ικανός να τρέξει 2 εντολές του κάθε wrap ανα κύκλο. Κάθε SMM αποτελείται απο 128 πυρήνες, τους δικούς του πόρους καθώς και buffering εντολών. Στην Εικόνα 3 παρουσιάζεται η αρχιτεκτονική της GPU στην NVIDIA Tegra x1.



Εικόνα 2: Αρχιτεκτονική GPU στην Nvidia Tegra x1.



Εικόνα 3: Αρχιτεκτονική SMM στην Nvidia Tegra x1.

Προγραμματιστικό Μοντέλο CUDA

Το προγραμματιστικό μοντέλο CUDA επιτρέπει στους προγραμματιστές να πραγματοποιούν εφαρμογές σε ετερογενή συστήματα CPU-GPU. Από την σκοπιά του προγραμματιστή γίνεται η εξής διάκριση:

- Host: η CPU και η μνήμη της (host memory)
- Device: η GPU και η μνήμη της (device memory)

Λέξη κλειδί για το μοντέλο μας είναι ο kernel, ο κώδικας-συνάρτηση που εκτελείται στη GPU. Μία CUDA εφαρμογή αποτελείται από σειριακό κώδικα (host, γραμμένος σε ANSI C) και από παράλληλο κώδικα (device, γραμμένος σε CUDA C). Η καλύτερη απόδοση του συστήματος πέρα απο την καλύτερη δυνατή περιγραφή του σε παράλληλο κώδικα επηρεάζεται και από παράγοντες που σχετίζονται με το υλικό της GPU. Ο τρόπος οργάνωσης των threads και blocks και ο τρόπος μεταφοράς δεδομένων μεταξύ CPU-GPU είναι κάποιοι από τους πιο σημαντικούς παράγοντες.

Οργάνωση των threads / blocks αποτελεί η επιλογή του αριθμού των blocks μέσα σε ένα grid και των threads μέσα σε ένα block, καθώς και οι διαστάσεις τους. Ο γενικός κανόνας για τον προσδιορισμό τους είναι ότι πρέπει ο αριθμός των threads να είναι πολλαπλάσιο του 32 μέσα σε ένα block και να μην δημιουργούνται threads τα οποία να μένουν αδρανή.

Ο τρόπος μεταφοράς δεδομένων αποτελέι κρητικό σημείο όσον αφορά τις ετερογενείς αρχιτεκτονικές. Η περίπτωση μας βέβαια, μας ευνοεί αφού οι CPU και GPU μοιράζονται κοινή DRAM μνήμη. Παρ΄ ολα αυτά, η ανάγκη για την καλύτερη επιλογή του τρόπου επικοινωνίας CPU-GPU παραμένει.

Οι τρόποι μεταφοράς που επιτρέπει η GPU είναι οι εξής:

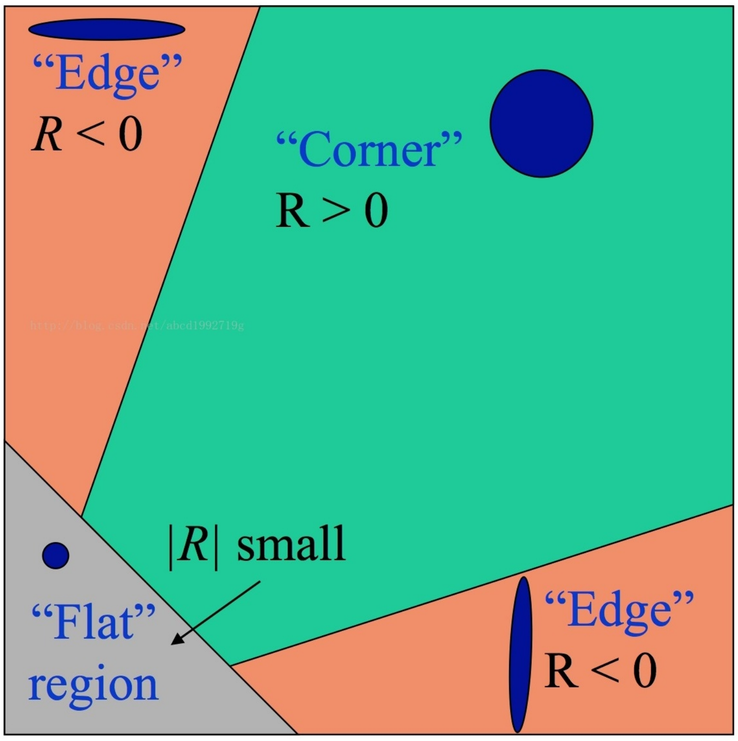
- Classic τρόπος μεταφοάς - απαιτείται δέσμευση χώρου μνήμης στο host και στο device, και μεταφορά των δεδομένων
- Pinned τρόπος μεταφοάς - απαιτείται δέσμευση χώρου μνήμης στο host και στο device, και μεταφορά των δεδομένων της οποίας η καθυστέρηση είναι πολύ μικρή
- Zero-Copy τρόπος μεταφοάς - απαιτείται δέσμευση χώρου μνήμης μόνο στο host Δεν απαιτείται μεταφορά των δεδομένων
- Unified Virtual Address (UVA) τρόπος μεταφοάς - απαιτείται δέσμευση χώρου μνήμης μόνο στο host Δεν απαιτείται μεταφορά των δεδομένων
- Pitch τρόπος μεταφοάς - απαιτείται δέσμευση χώρου μνήμης στο host και στο device, και μεταφορά των δεδομένων η οποία επιτρέπει από τη GPU πιο αποδοτική πρόσβαση στα δεδομένα

Τα αποτελέσματα των παραπάνω τρόπων θα φανούν παρακάτω στα πειραματικά αποτελέσματα. Τέλος για την καλύτερη απόδοση συνίστανται προγραμματιστικές τεχνικές, μία από τις οποίες είναι η χρήση της κοινής μνήμης (shared memory), μία χαμηλής καθυστέρησης μνήμη την οποία την μοιράζονται μεταξύ τους τα threads που ανήκουν σε ένα SMM. Αξίζει να σημειωθεί πως η shared μνήμη και η L1 κρυφή μνήμη απέχουν λιγότερο απο το SMM από ότι η L2 κρυφή μνήμη και η global μνήμη. Για τον λόγο αυτό αποτελεί πολύ δημοφιλή επιλογή η φόρτωση συχνά χρησιμοποιούμενων δεδομένων απο την global στην shared μνήμη και η εκτέλεση τους απο εκεί. Την τεχνική αυτή την εξετάζουμε παρακάτω.

Αλγόριθμος Harris και η υλοποίησή του

Ο αλγόριθμος Harris εισήχθηκε απο τους Chris Harris και Mike Stephens το 1988 με σκοπό τη βελτίωση του αλγορίθμου του Movarek για ανίχνευση γωνιών. Δέχεται ως είσοδο μία grayscale εικόνα και ανιχνεύει τις γωνίες της (σημεία στην εικόνα που παρουσιάζουν σημαντική τοπική διακύμανση στην ένταση σε όλες τις κατευθύνσεις), οι οποίες μπορούν να ανιχνευτούν επανειλημμένα με αρκετή ακρίβεια κάτω από διαφορετικές συνθήκες. Η δημοφιλία του Harris και οι ποικίλες υπολογιστικές τεχνικές του (π.χ. συνέλιξη, σταθερής και κινητής υποδιαστολής αριθμητική) κάνουν τον Harris μια ιδιαίτερα αντιπροσωπευτική εφαρμογή για τους σκοπούς της εργασίας.

Για κάθε εικονοστοιχείο, ο αλγόριθμος Harris υπολογίζει την δύναμη μιας γωνίας ("cornerness") σύμφωνα με τον τύπο $R = \mathrm{I}_x^2\mathrm{I}_y^2 - (\mathrm{I}_x\mathrm{I}_y)^2 - 0.04 \cdot (\mathrm{I}_x^2 + \mathrm{I}_y^2)^2$, όπου $\mathrm{I}_x^2, \mathrm{I}_y^2$ και $\mathrm{I}_x\mathrm{I}_y$ αντιπροσωπεύουν τα Gaussian-smoothed γινόμενα των παραγώγων της εικόνας, τα οποία υπολογίζονται μέσω ενός φίλτρου Sobel. Τιμές του cornerness που υπερβαίνουν ένα συγκεκριμένο όριο και τις αντίστοιχες τιμές των υπόλοιπων σε μια γειτονιά 3×3 του υπό εξέταση εικονοστοιχείου (αποτελούν τοπικό μέγιστο), ορίζουν γωνίες στην εικόνα όπως φαίνεται στην Εικόνα 4.

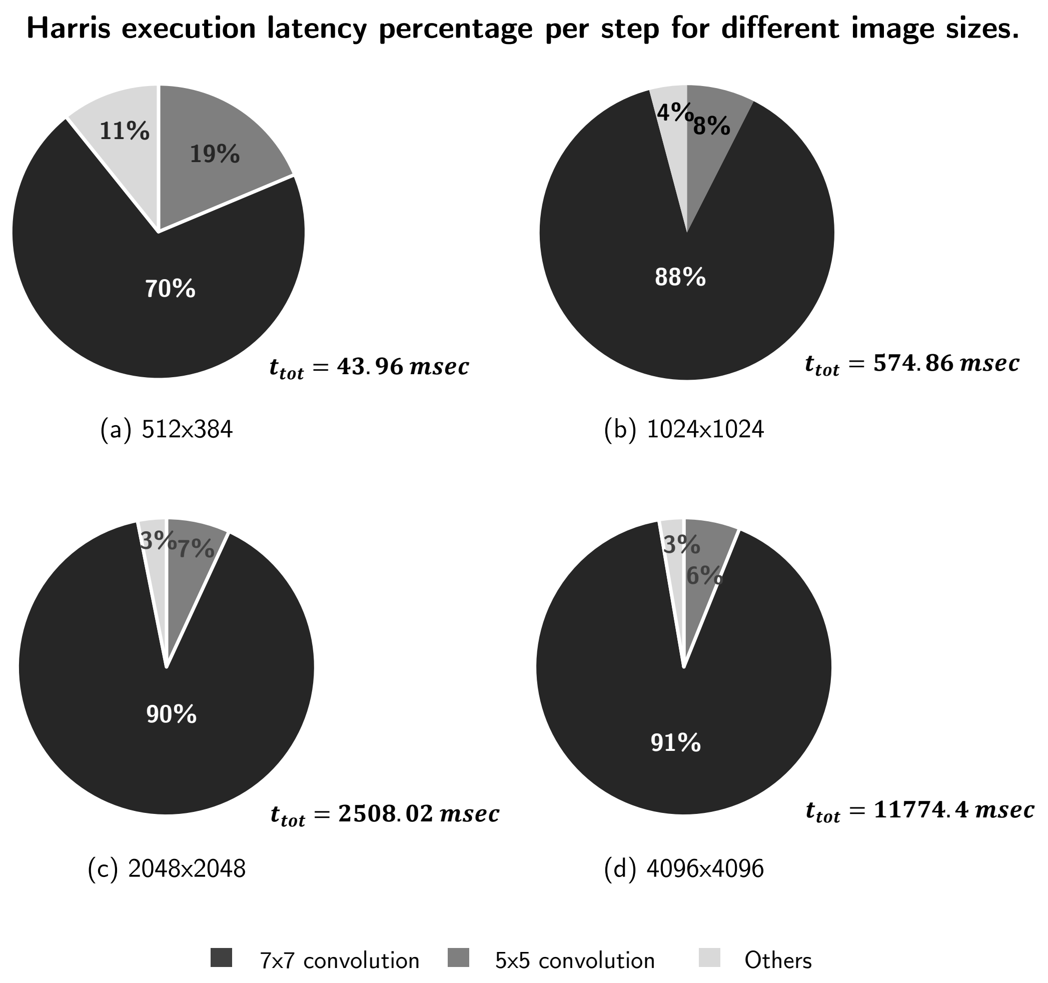


Εικόνα 4: Κριτήριο επιλογής γωνιών

Για την υλοποίηση του αλγορίθμου στην Tegra x1, απαιτούνται τα εξής βήματα:

- Κατασκευή του Ix, Iy με συνέλιξη της εικόνας με τον 5×5 Sobel operator
- Κατασκευή των Ix^2, Iy^2, Ixy δεδομένων των Ix, Iy
- Κατασκευή νέων Ix^2, Iy^2, Ixy με συνέλιξη των παλιών Ix^2, Iy^2, Ixy με 7×7 tap kernel
- Υπολογισμός του $R = \mathrm{I}_x^2\mathrm{I}_y^2 - (\mathrm{I}_x\mathrm{I}_y)^2 - 0.04 \cdot (\mathrm{I}_x^2 + \mathrm{I}_y^2)^2$ δεδομένων των νέων Ix^2, Iy^2, Ixy

Στην Εικόνα 5 φαίνονται τα ποσοστά της χρονικής διάρκειας ανά βήμα για διαφορετικές αναλύσεις εικόνας στην NVIDIA Tegra x1.



Εικόνα 5: Ποσοστά των επιμέρους συναρτήσεων του αλγορίθμου Ηαρρις για διαφορετικά μεγέθη εικόνας

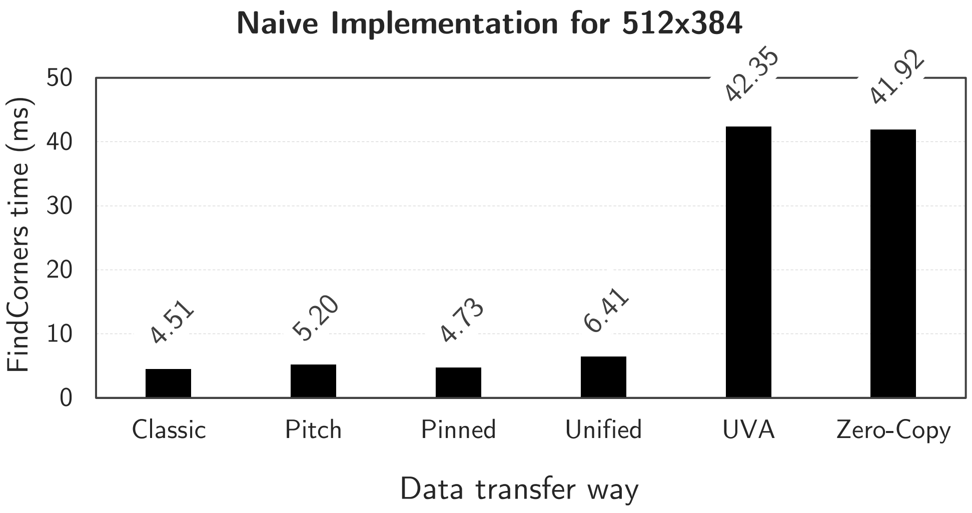
Είναι φανερό πως προτεραιότητα στην επιτάχυνση με GPU έχουν οι δύο δισδιάστατες συνελίξεις και ειδικά η 7×7 συνέλιξη. Παρ΄ όλα αυτά, για να αποφύγουμε την ανάγκη μεταφοράς δεδομένων μεταξύ CPU-GPU, λειτουργία ιδιαίτερα χρονοβόρα, εκτελούμε όλα τα βήματα υπολογισμών στη GPU. Ο τρόπος με τον οποίον το κάνουμε αυτό είναι ο εξής: κάθε thread αντιστοιχίζεται σε ένα pixel της εικόνας παράλληλα. Αυτομάτως αυτό σημαίνει πως ο συνολικός αριθμός των threads με τον οποίον οργανώνουμε τον kernel μας είναι ο συνολικός αριθμός των pixels της εικόνας μας που έχουμε να επεξεργαστούμε.

Πέρα απο τις τεχνικές που μας προσφέρει το προγραμματιστικό μοντελο της CUDA, εξαιρετικά αποτελέσματα επιφέρει η αλγοριθμική τεχνική της διαχωριστικότητας (separability). Πρόκειται στην ουσία για αλλαγή του τρόπου με τον οποίο υπολογίζεται μια δισδιάστατη συνέλιξη, αλλάζοντας τον τρόπο που φορτώνονται και αποθηκεύονται τα δεδομένα της εικόνας, προσφέροντας έτσι λιγότερες μεταβάσεις στην μνήμη και συνεπώς μικρότερο χρόνο εκτέλεσης.

Η συνέλιξη μίας $M \times N$ εικόνας με έναν $P \times Q$ kernel (όπως ονομάζουμε τον πίνακα με τον οποίον συνελίσουμε μία εικόνα) απαιτεί $M \times N \times P \times Q$ πολλαπλασιασμούς και προσθέσεις. Αν όμως ο kernel μας είναι διαχωρίσιμος (μπορει να προκύψει ως πολλαπλασιαμός μίας στήλης και μίας γραμμής) τότε ο υπολογισμός της δισδιάστατης συνέλιξής μας μπορεί να υπολογιστεί σε 2 βήματα. Σε μία μονοδιάστατη συνέλιξη της εικόνας με τη γραμμή-παράγοντα($M \times N \times P$ προσθέσεις και πολλαπλασιασμοί) και σε μία άλλη μονοδιάστατη συνέλιξη του αποτελέσματος με τη στήλη-παράγοντα($M \times N \times Q$ προσθέσεις και πολλαπλασιασμοί). Συνεπώς, ο συνολικός αριθμός των πράξεων είναι $(M \times N)(P + Q)$ σαφώς μικρότερος απο το $M \times N \times P \times Q$ όταν προκείται για P,Q μεγάλους αριθμούς. Με άλλα λόγια όπως θα δείξουμε και στα περιαματικά αποτελέσματα ο τρόπος της διαχωριστικότητας συνίσταται μόνο για kernels με μεγάλες διαστάσεις.

Πειραματικά αποτελέσματα

Όπως έχει ήδη αναφερθεί, ένας από τους πιο σημαντικούς παράγοντες για την απόδοση μιας εφαρμογής σε GPU είναι ο τρόπος μεταφοράς δεδομένων απο CPU σε GPU και αντίστροφα. Στην Εικόνα 6 παρουσιάζεται ο συνολικός χρόνος της εφαρμογής για διαφορετικούς τρόπους επιοινωνίας. Πρόκειται αλγοριθμικά για τον πιο απλό τρόπο εκτέλεσης της εφαρμογής, χωρίς τεχνικές που αναφέρθηκαν παραπάνω. Πρόκειται για τον λεγόμενο "naive" τρόπο όπου κάθε thread επεξεργάζεται ένα εικονοστοιχείο παράλληλα.



Εικόνα 6: Χρόνος ανίχνευσης γωνιών για διάφορους τρόπους επικοινωνίας για εικόνα 512x384 pixels

Έχοντας πλέον μια ιδέα για την συμπεριφορά του κάθε τρόπου μεταφοράς από την Εικόνα 6, μπορούμε εύκολα να συμπεράνουμε πως τουλάχιστον για την εφαρμογή μας, δεν συνίστανται οι τρόποι Unified Virtual Address (UVA) και Zero-Copy. Στην Εικόνα 7 παρουσιάζουμε τους χρόνους της ανίχνευσης γωνιών -της συνολικής εφαρμογής- για διαφορετικούς τρόπους μεταφοράς (εκτός των UVA , Zero-Copy που τους αποκλείσαμε) για διαφορετικές αναλύσεις εικόνας.

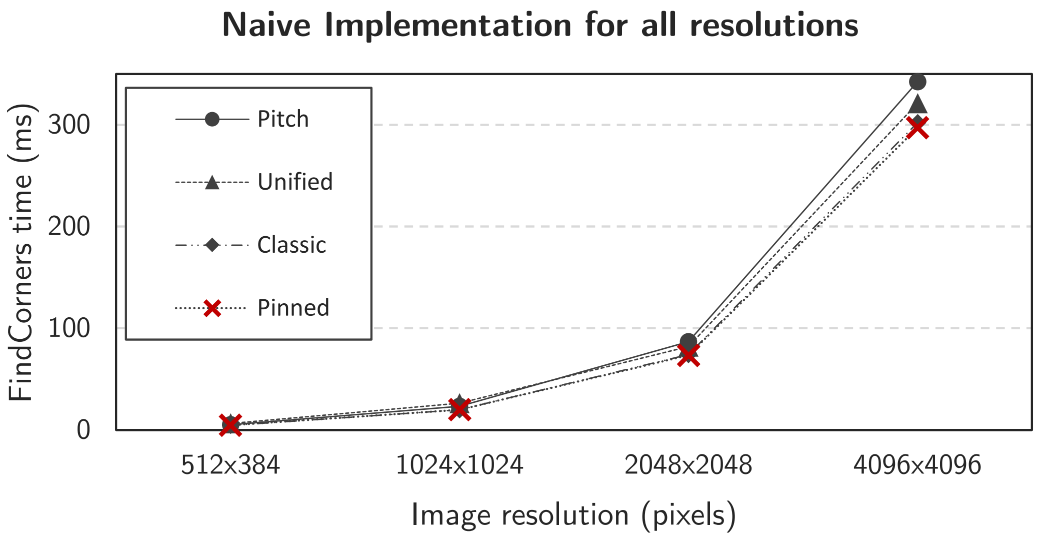
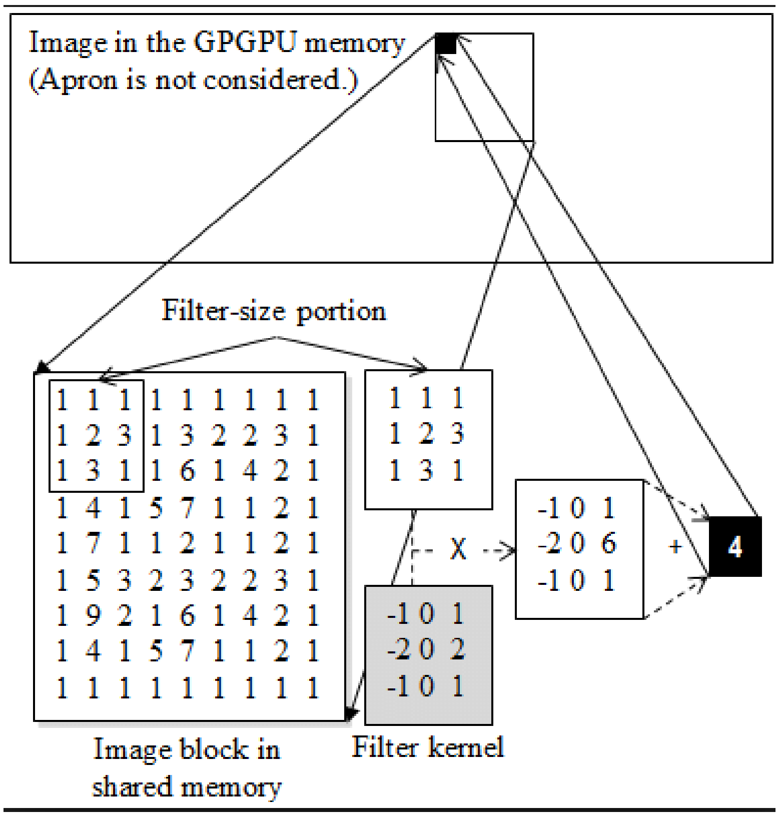


Figure 7: Χρόνος ανίχνευσης γωνιών με naive τρόπο για διάφορες αναλύσεις εικόνας

Παρατηρούμε πως για όλα τα μεγέθη τα οποία ερευνούμε, οι καλύτεροι τρόποι μεταφοράς είναι ο classic και ο pinned τρόπος με φαινομενικά καμία διαφορά στους χρόνους. Συνεπώς ο καλύτερος τρόπος θα κριθέι μεταξύ των δύο, με πειραάματα που θα δείξουν λεπτομερέστερα αποτελέσματα.

Όπως αναφέρθηκε παραπάνω, η κοινή μνήμη (shared memory) αποτελέι δημοφιλέστατη επιλογή των προγραμματιστών για τη βελτίωση της απόδοσης των εφαρμογών τους. Έτσι, και στην περίπτωσή μας αποτέλεσε προτεραιότητα να ερευνήσουμε αυτήν την τεχνική για την καλύτερη δυνατή απόδοση του συστήματος. Ο τρόπος με τον οποίο χρησιμοποιούμε αποδοτικά τη shared memory παρουσιάζεται στην Εικόνα 8:



Εικόνα 8: Τρόπος υλοποίησης με κρηφή μνήμη

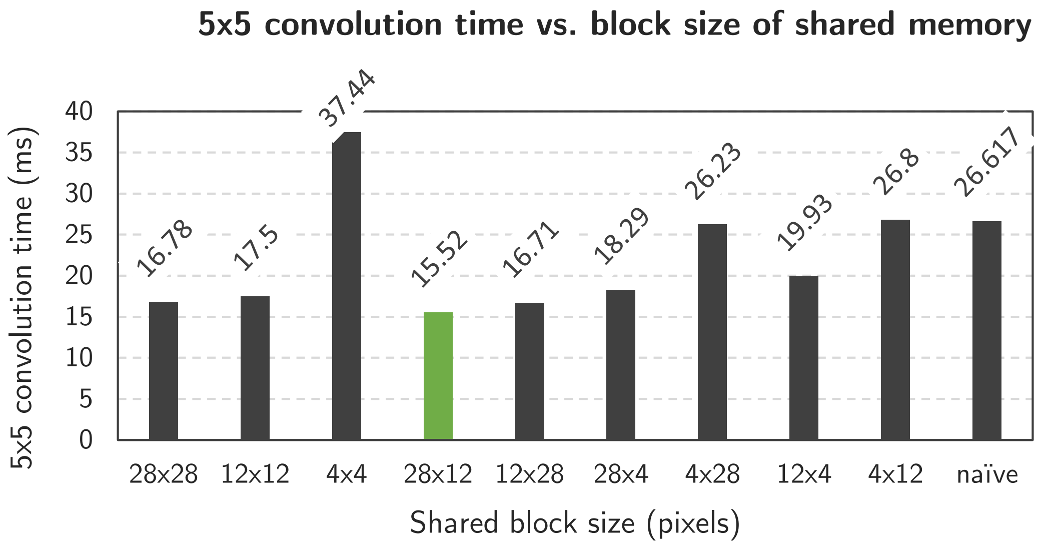
Κάθε block φορτώνει απο την global στη shared μνήμη εναν πίνακα με διαστάσεις της επιλογής μας. Απο εκεί, ο kernel τα φορτώνει για τις λειτουργίες του, πράγμα που κάνει τις προσβάσεις πιο γρήγορες. Το πρόβλημα το οποίο καλούμαστε να αντιμετωπίσουμε είναι τα ακριανά pixels που απαιτούνται για τον υπολογισμό της κάθε νέας τιμής του block της εικόνας. Αυτό σημαίνει πως αν η ακτίνα του kernel μας είναι a και το block μας θέλουμε να έχει διαστάσεις $b \times c$ τότε ο πίνακας στη shared μνήμη επιβαλλεται να έχει διαστάσεις $(b + a) \times (c + a)$

Έτσι, βάσει αυτού, αλλά και του γενικού κανόνα ότι το σύνολο των threads σε ένα block πρέπει να είναι το πολύ 1024 και πολλαπλάσιο του 32, συμπεραίνουμε πως οι πιθανές διαστάσεις των πινάκων στις οποίες θα αποθηκευτούν τα δεδομένα στη shared μνήμη είναι πολύ περιορισμένες και καταγράφονται στον Πίνακα 1.

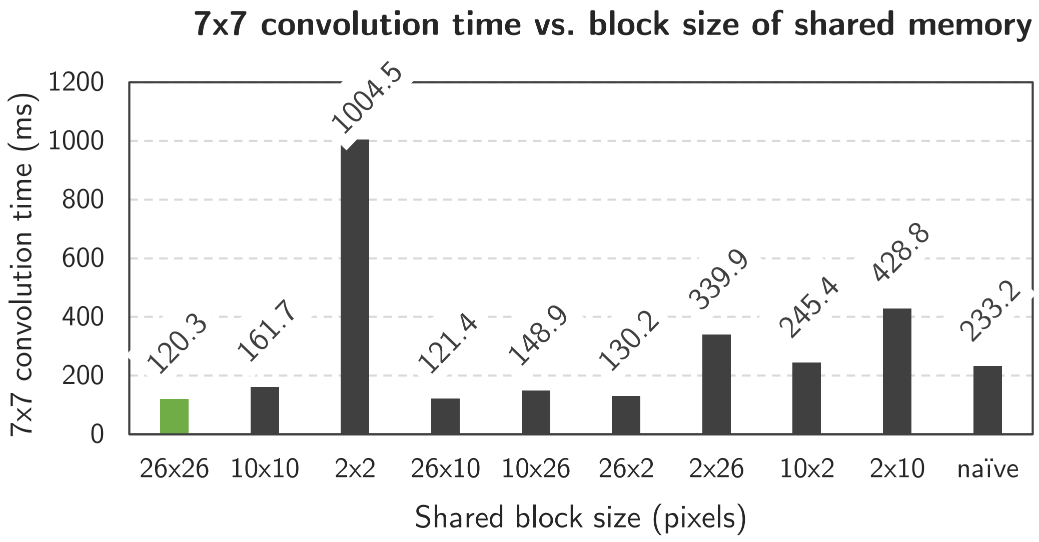
Table 1: Possible dimensions of blocks-arrays in shared memory

5x5 tap kernel	**7x7 tap kernel**
4x4	2x2
12x12	10x10
28x28	26x26
28x12	26x10
12x28	10x26
28x4	26x2
4x28	2x26
12x4	10x2
4x12	2x10

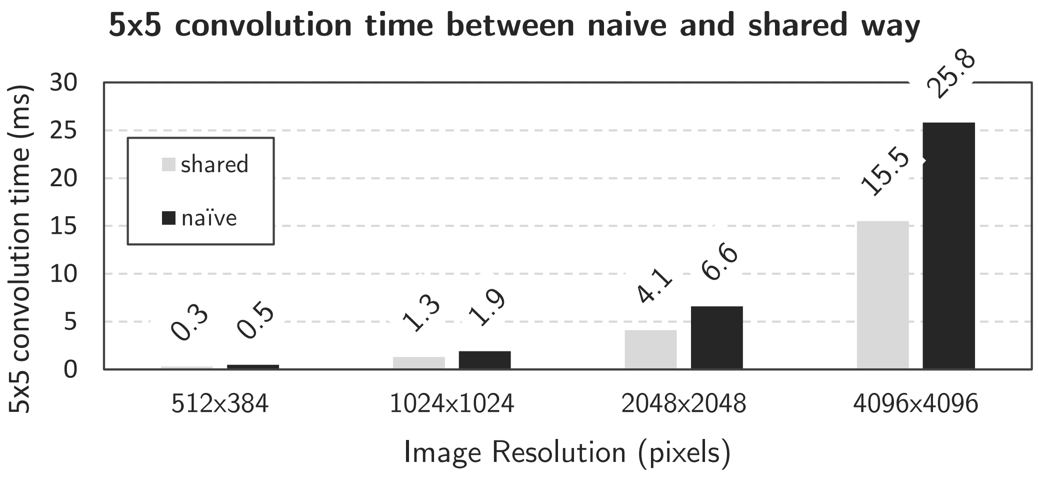
Αναζητώντας τις διαστάσεις που θα μας προσφέρουν βέλτιστες λύσεις, στις Εικόνες 9 και 10 παρουσιάζονται οι χρόνοι εκτέλεσης των δύο δισδιάστατων συνελίξεων (5×5 και 7×7) για 4096×4096 pixels ώστε να φαίνονται ευκολότερα οι διαφορές των χρόνων υπολογισμών τους.



Εικόνα 9: Υπολογισμός της 5×5 συνέλιξης μεταξύ naive και shared τρόπου



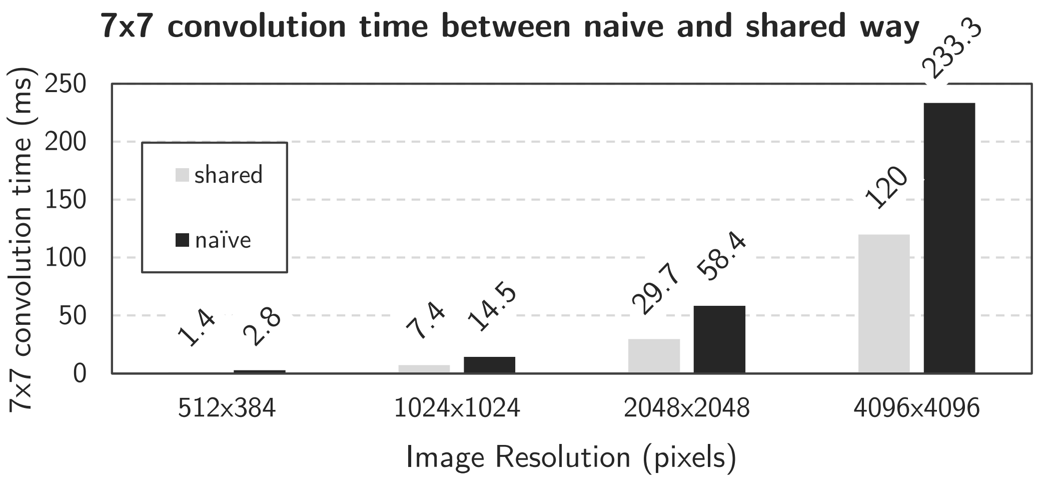
Εικόνα 10: Υπολογισμός της 7×7 συνέλιξης μεταξύ naive και shared τρόπου



H]

Εικόνα 11: Υπολογισμός 5 × 5 συνέλιξης μεταξύ naive τρόπου και shared

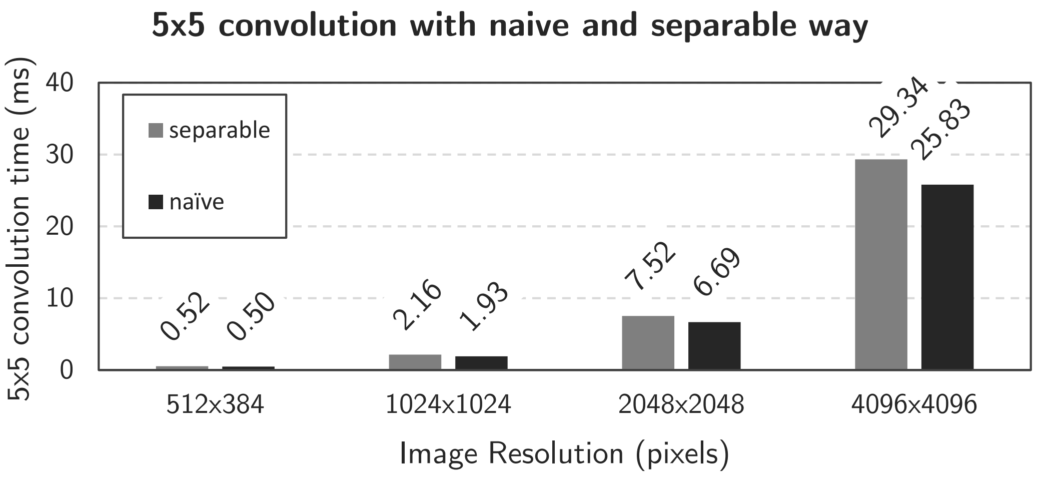
Από την παραπάνω εικόνα επιλέγουμε τις διαστάσεις 28 × 12 και 26 × 26 για τις 5 × 5 και 7 × 7 συνελίξεις αντίστοιχα. Έτσι, στην Εικόνα 11 και 12 παρουσιάζουμε τους χρόνους υπολογισμού των δύο συνελίξεων ξεχωριστά για τις ιδανικές διαστάσεις για όλες τις εικόνες που μας ενδιαφέρουν συγκριτικά με τη naive υλοποίηση τους.



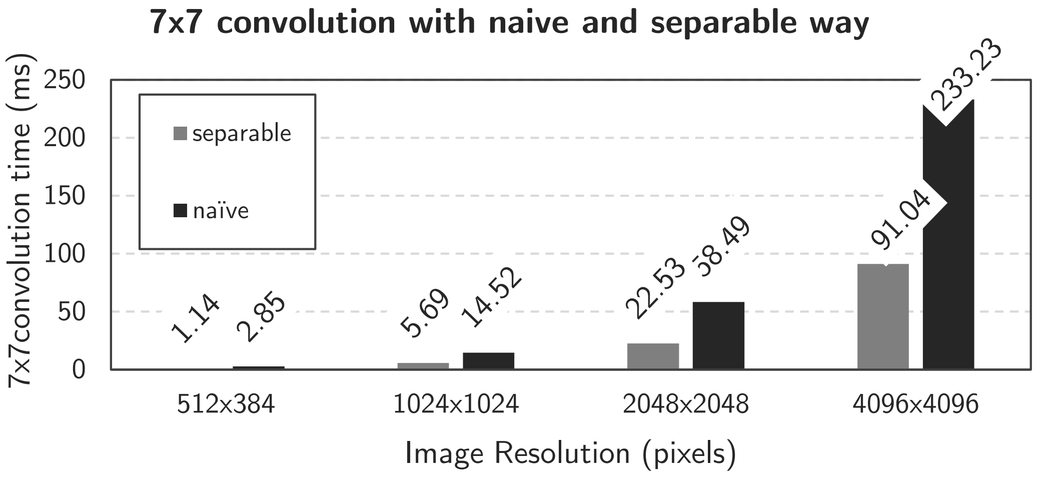
Εικόνα 12: Υπολογισμός 7 × 7 συνέλιξης μεταξύ naive τρόπου και shared

Απο τις Εικόνες 11 και 12 για την μέχρι στιγμής αναζήτησή μας για καλύτερη απόδοση τα συμπεράσματα προτείνουν την χρήση της shared μνήμης και για τις δύο συνελίξεις.

Όπως όμως αναφέρθηκε, η τεχνική της διαχωριστικότητας φαίνεται πολλά υποσχόμενη όταν πρόκειται για δισδιάστατες συνελίξεις με kernel μεγάλης ακτίνας. Στις Εικόνες 13 και 14 παρουσιάζονται οι χρόνοι υπολογισμού των δύο συνελίξεων ξεχωριστά με τον naive και τον διαχωριστικό τρόπο.

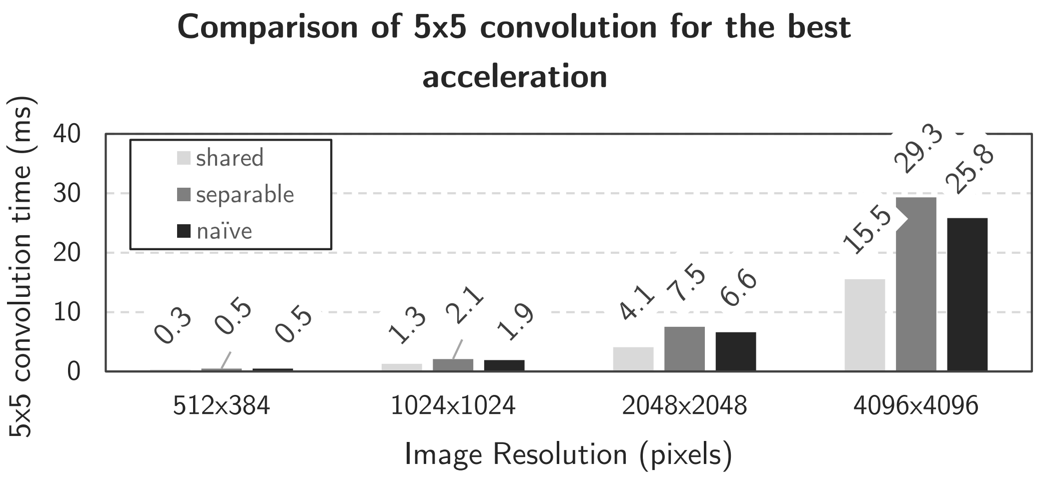


Εικόνα 13: Χρόνος εκτέλεσης του υπολογισμού της συνέλιξης 5×5 για naive και για separable τρόπο

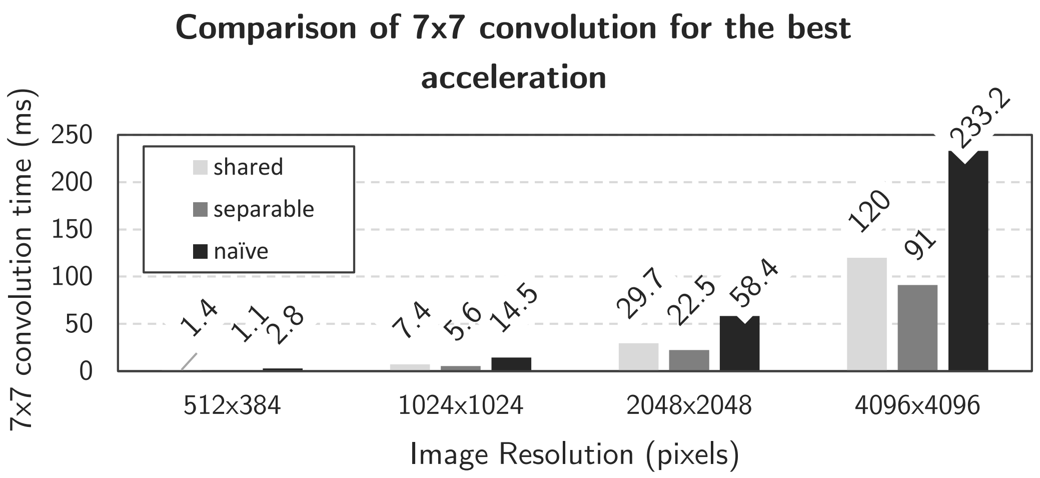


Εικόνα 14: Χρόνος εκτέλεσης του υπολογισμού της συνέλιξης 7×7 για naive και για separable τρόπο

Από τις παραπάνω εικόνες, συμπεραίνουμε όπως περιμέναμε, πως η διαχωριστικότητα όχι μόνο δεν ωφελεί την συνέλιξη 5×5 αλλά την επιβαρύνει αυξάνοντας τον χρόνο του υπολογισμού της. Αντίθετα, παρατηρούμε πως για τη συνέλιξη 7×7, ο χρόνος μειώνεται εντυπωσιακά πέφτοντας παραπάνω από το μισό. Για την επιλογή της καλύτερης επιτάχυνσης που αναζητούμε, στις Εικόνες 15 και 16 παρουσιάζουμε τη σύγκριση μεταξύ των προγραμματιστικών τεχνικών που δοκιμάσαμε για τα επιμέρους βήματα των συνελίξεων.



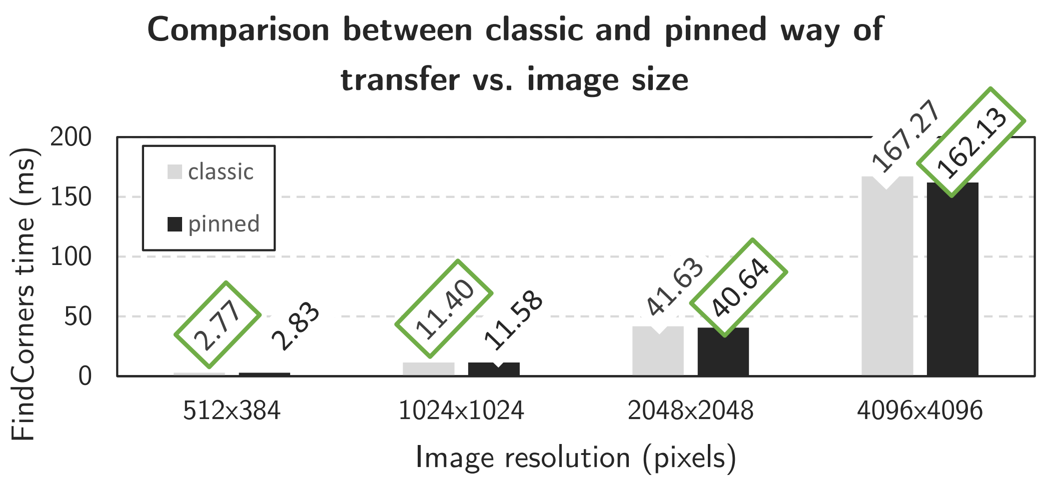
Εικόνα 15: Χρόνος εκτέλεσης του υπολογισμού της συνέλιξης 5×5 για naive , separable και shared τρόπο



Εικόνα 16: Χρόνος εκτέλεσης του υπολογισμού της συνέλιξης 7×7 για naive , separable και shared τρόπο

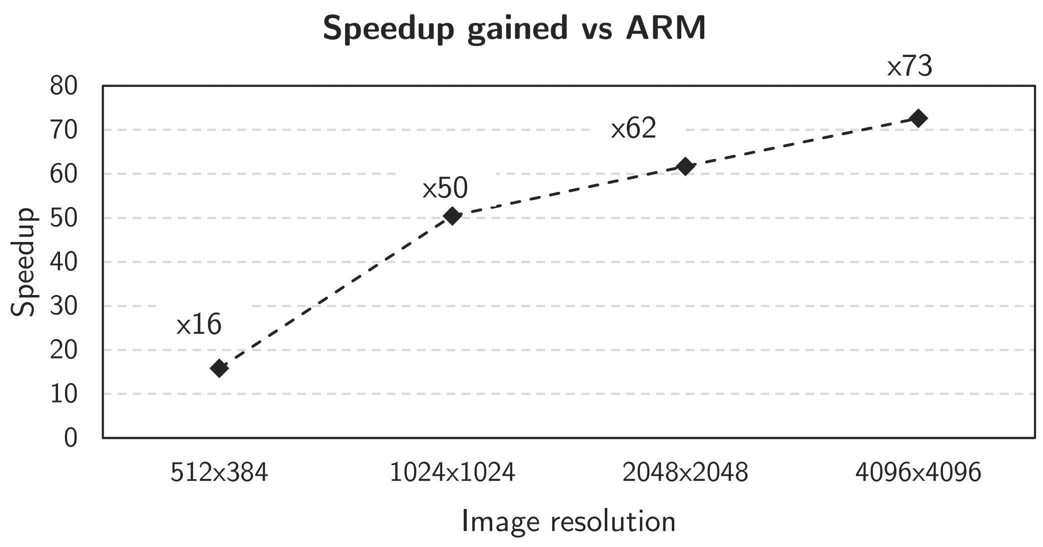
Όπως ήταν αναμενόμενο η διαχωριστικότητα δεν ευνοεί την 5×5 συνέλιξη. Ευνοεί όμως την 7×7 συνέλιξη ρίχνοντας τον χρόνο υπολογισμού της στο μισό. Μέσα απο την σύγκριση των Εικόνων 15 και 16 προκύπτει η δομή της βέλτιστης λύσης που προσφέρουμε στην παρούσα διπλωματική. Χρήση shared μνήμης για την 5×5 συνέλιξη, χρήση διαχωριστικών συνελίξεων για την 7×7 συνέλιξη και naive μέθοδος για τα υπόλοιπα βήματα της εφαρμογής που εκτελούνται στη GPU.

Έχοντας γνώση αυτού, στην Εικόνα 17 συγκρίνουμε τους χρόνους εκτέλεσης της συνολικής εφαρμογής Finding Corners ανίχνευσης γωνιών για τους classic και pinned τρόπους μεταφοράς.



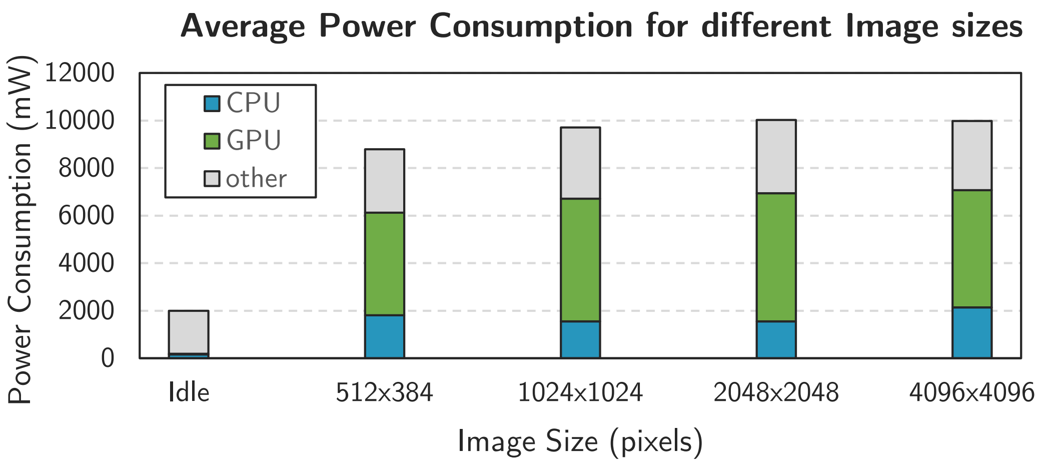
Εικόνα 17: Σύγκριση των δύο επικρατούντων τρόπων μεταφοράς

Από την Εικόνα 17,συμπεραίνουμε πως για εικόνες μικρότερες των 1024 × 1024 εικονοστοιχείων συνίσταται ο classic τρόπος και για εικόνες μεγαλύτερες συνίσταται ο pinned τρόπος μεταφοράς. Έχοντας καταλήξει στην βέλτιση επιτάχυνση που μπορούμε να προσφέρουμε με την πλακέτα Tegra x1, στην Εικόνα 18 παρουσιάζουμε το κέρδος το οποίο κερδίσαμε συγκριτικά με την εκτέλεση της εφαρμογής σε έναν ARM Cortex A57 CPU.



Εικόνα 18: Σύγκριση μεταξύ Tegra x1 και ARM Cortex A57 CPU

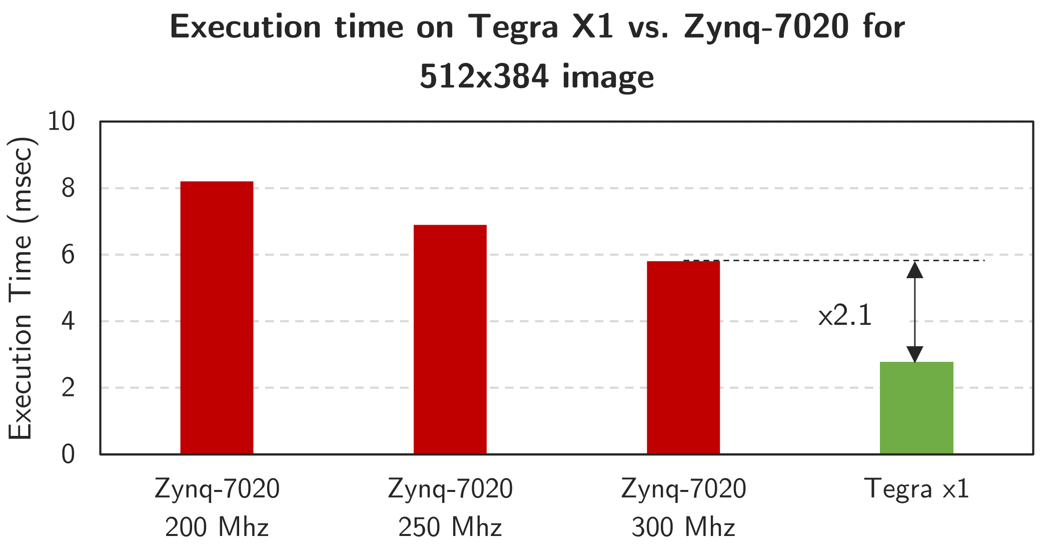
Όπως έχει ήδη αναφερθεί, η κατανάλωση ισχύος αποτελεί ένα πολύ κριτικό σημείο στην επιτάχυνση εφαρμογών με ενσωματωμένα συστήματα. Έχοντας καταλήξει στη βέλτιστη -κατά τον χρόνο εκτέλεσης- μέθοδο, στην Εικόνα 19 παρουσιάζουμε την συνολική κατανάλωση ισχύος του Tegra x1 για καμία εφαρμογή, εφαρμογή του αλγορίθμου για 512×384, 1024×1024, 2048×2048 και 4096×4096 pixels εικόνας.



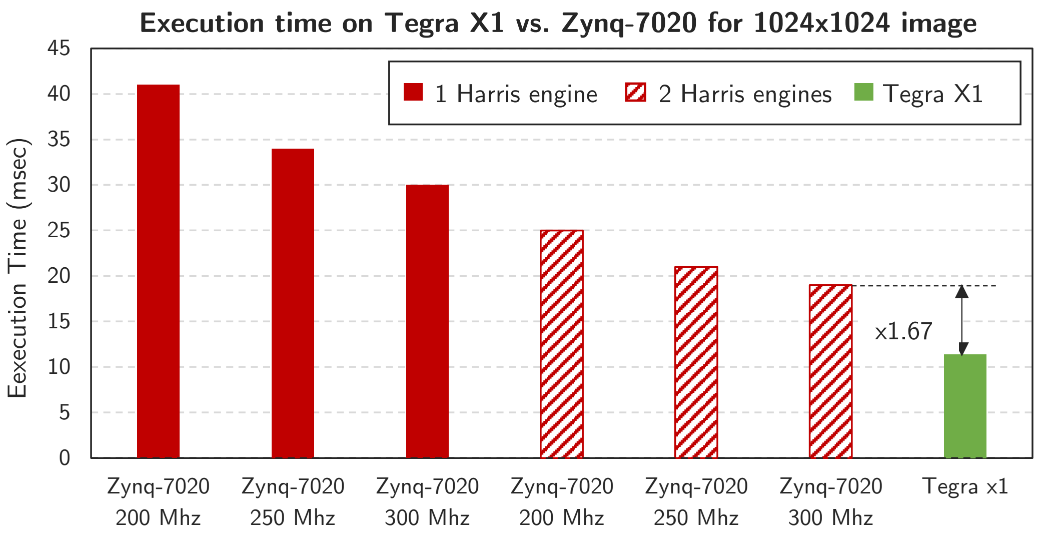
Εικόνα 19: Κατανάλωση ισχύος του Tegra x1 για διαφορετικά μεγέθη εικόνας

Σύγκριση των επιταχύνσεων του αλγορίθμου Harris σε NVIDIA Tegra x1 και ZC702

Όπως έχει ήδη αναφερθεί, σκοπός της παρούσας διπλωματικής είναι να συγκριθεί η επιτάχυνση που επιτύχαμε μέσω του Tegra x1 CPU-GPU-SoC, με τον ZC702 CPU-FPGA-SoC, προκειμένου να αποφανθούμε για την πιο αποδοτική πλατφόρμα που μπορεί να υποστηρίξει την εφαρμογή αυτή. Έτσι, στις Εικόνες 20 και 21 παρουσιάζουμε τους χρόνους εκτέλεσης της εφαρμογής για δύο ενσωματωμένες πλατφόρμες που ερευνούμε.



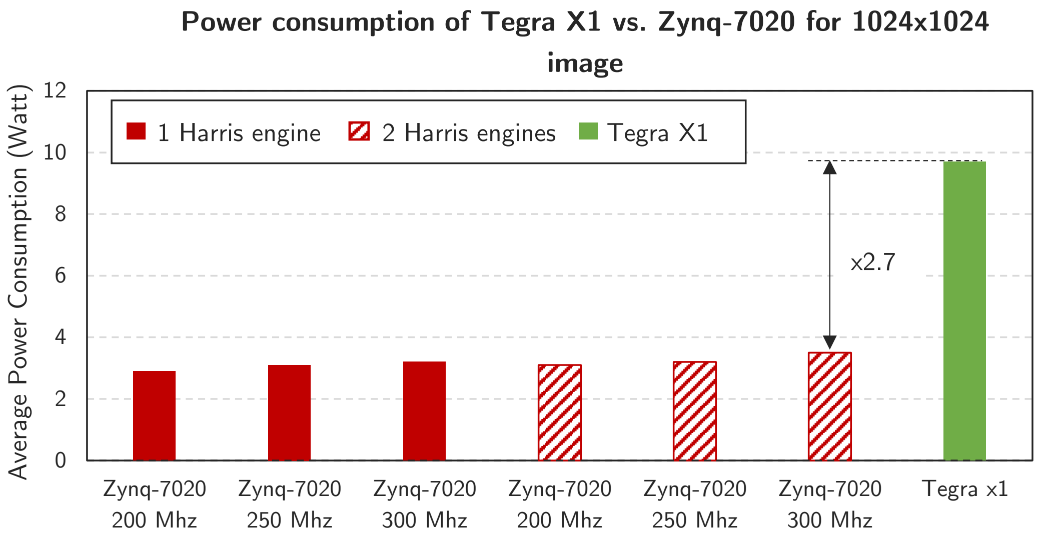
Εικόνα 20: Σύγκριση χρόνων εκτέλεσης των βελτιστοποιημένων αλγορίθμων Harris σε Tegra x1 και ZC702 για 512×384 pixels εικόνες



Εικόνα 21: Σύγκριση χρόνων εκτέλεσης των βελτιστοποιημένων αλγορίθμων Harris σε Tegra x1 και ZC702 για 1024×1024 pixels εικόνες

Από τις Εικόνες 20 και 21 φαίνεται πως η CPU-GPU υλοποίηση υπερέχει απο την καλύτερη υλοποίηση του CPU-FPGA (2 engines, 300 MHz) κατα $\times 2.1$ και $\times 1.67$ για εικόνες 512×384 και 1024×1024 pixels αντίστοιχα.

Όσον αφορά την κατανάλωση ισχύος, στην Εικόνα 22 παρουσιάζουμε τις τιμές σε Watt που μετρήθηκαν για τις δύο πλατφόρμες. Είναι φανερό πως η Tegra x1 καταναλώνει 2.7 περισσότερο ισχύ από το Zynq-7020.



Εικόνα 22: Σύγκριση κατανάλωσης ισχύος μεταξύ Tegra X1 και ZC702 για 1024x1024 pixels εικόνας

Chapter 1

Introduction

1.1 Image Processing on Embedded Systems

Image processing is a method to convert an image into digital form and perform some operations on it, in order to get an enhanced image or to extract some useful information from it. It is a type of signal dispensation in which input is image, like video frame or photograph and output may be image or characteristics associated with that image. Usually Image Processing system includes treating images as two dimensional signals while applying already set signal processing methods to them. It is among rapidly growing field today, with its applications in various aspects of technology. Image Processing forms core research area within engineering and computer science disciplines too.

The operations of Imagine Procdessing can be grouped according to the type of data that they process as presented in Figure 1.1.

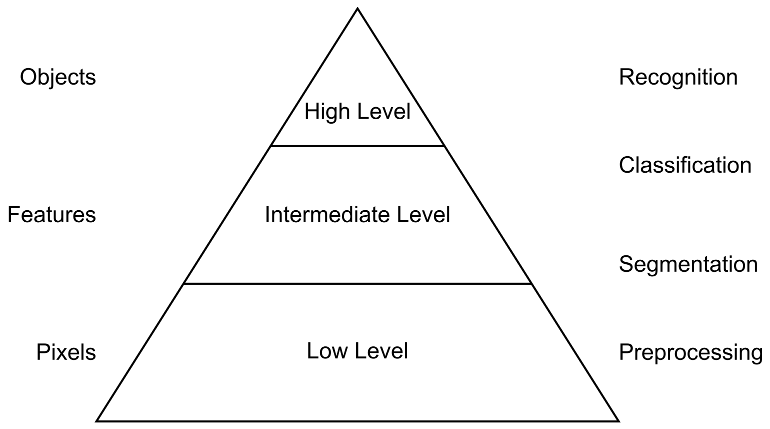


Figure 1.1: Image processing pyramid.

At the lowest level of the image processing pyramid are those operations which deal with the raw pixel values and can be though as prepossessing steps like distortion correction, contrast enhancement and filtering for noise reduction or edge detection. At the middle are the algorithms which utilize results from the low level processing and at the highest level are those methods which attempt to extract semantic meaning from the information provided by the lower levels.

An embedded system is some combination of computer hardware and software, either fixed in capability or programmable, that is designed for a specific function or for specific functions within a larger system. Industrial machines, agricultural and process industry devices, automobiles, medical equipment, cameras, household appliances, airplanes, vending machines and toys as well as mobile devices are all possible locations for an embedded system. They are getting more and more common in cars, cameras, and even on washing machines and fridges like auto-focus cameras, battery chargers, cell phones, temperature controllers etc.

Embedded systems are particularly well suited to process data streams at high speeds with fairly small programs, that is why recent years have witnessed a dramatic increase in the use of embedded systems to run image processing applications. Other than on CPUs, data can be processed in a highly parallel fashion, with high speeds and low power requirements.

1.2 GPUs in Image Processing

A graphics processing unit (GPU) is a specialized electronic circuit designed to rapidly manipulate and alter memory to accelerate the creation of images in a frame buffer intended for output to a display device. Modern GPUs are very efficient at manipulating computer graphics and image processing, and their highly parallel structure makes them more efficient than general-purpose CPUs for algorithms where the processing of large blocks of data is done in parallel. It is also becoming increasingly common to use a general purpose graphics processing unit (GPGPU) as a modified form of stream processor (or a vector processor), running compute kernels. This concept turns the massive computational power of a modern graphics accelerator's shader pipeline into general-purpose computing power, as opposed to being hard wired solely to do graphical operations. In certain applications requiring massive vector operations, this can yield several orders of magnitude higher performance than a conventional CPU.

GPUs are made of a large number of processing units which by themselves aren't very powerful, but become formidable when used in tandem. So, if you have processing to be done that is parallelizable, the GPU will be a great fit. With that in mind,it is almost obvious that image processing is a great fit for GPUs. A lot of image processing algorithms are data-parallel, meaning the same task/computation needs to be performed on many elements of the data. Lots of image processing algorithms either operate on pixels independantly or rely only on a neighborhood around pixels (like image filtering).

1.3 Thesis Goals and Organization

Nowdays many advanced algorithms have been adopted in present-day embedded systems targeting a variety of applications such as automotive, 3D navigation, surveillance, etc. However in real-time embedded systems, where latency and power play an important role, software-oriented implementations running on general purpose CPUs may not offer satisfactory solutions, because of the fact that CPUs have limited parallel processing capabilities to support the performance requirements of these applications and consume considerable power. In order to increase the performance per watt, various approaches have been proposed where dedicated accelarators is deployed alongside the CPU to accelerate the critical parts of the task or even the entire algorithm.

The goal of this thesis it to design and implement an image processing system for embedded applications and its deployment on a SoC GPU. As a case study the corner detection of an image is selected. Individual parts are implemented on GPU using CUDA C and the rest being developed as software components using the C programming language. The remainder of this thesis is organized as follows:

- Chapter 2 discusses about the related work that has examined the field of Image Processing on embedded devices.

- Chapter 3 presents Tegra X1, the CPU-GPU-SoC device we will use.
- Chapter 4 presents the Cuda Programming Model and what posibilities it can give developers.
- Chapter 5 presents some theoretical and technical background on key features of Harris algorithm and introduces the system implementation.
- Chapter 6 presents the experimental evaluation of our implementation of the Harris corner detection algorithm, as well as provides a comparison with a Zynq-7020 FPGA device.
- Chapter 7 concludes this thesis giving an overview of this thesis and provides some ideas for future work.

Chapter 2

Related work

2.1 Image Processing Algorithms on GPU SoCs

There are several works that have accelerated Image Processing Algorithms even with Compute Unified Device Architecture (CUDA) or OpenCL. Youngwan Lee et al. [7] present an implementation of Viola-Jones face detector algorithm on a mobile SoC GPU, Galaxy S5-LTEA smartphone. Using Cascade classifier, this work experiments with two datasets. Image of groups, where this work reach up to x3.3 gain and INHA FACE dataset where it reach up to x6.29 gain in comparison with a well-optimized OpenCV implementation on a CPU. The results of the two datasets are presented in the Figure 2.1 and in the Figure 2.2. For the best acceleration the authors propose CPU-GPU coordination and not only a GPU implementation to succeed the overlap between CPU and GPU operations.

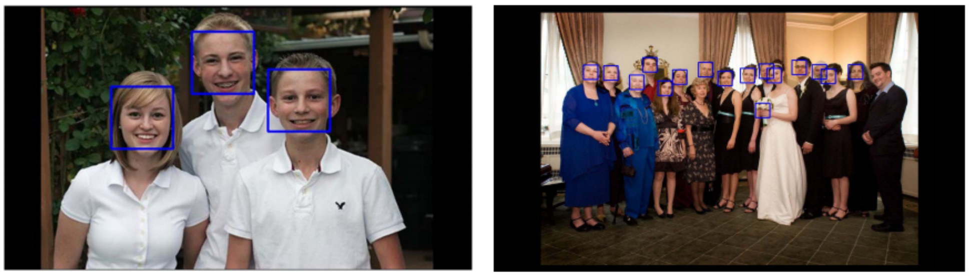


Figure 2.1: Results from the Images of the Groups dataset

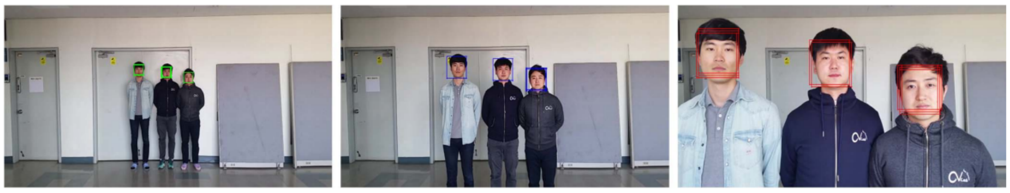


Figure 2.2: Results from the INHA FACE dataset

In addition, D. Hernandez-Juarez et al. [6] present an accelerated implementation of a computation of a stixel world as shown in Figure 2.3 using the embedded SoC GPU board NVIDIA Tegra X1. The authors' implementation using Dynamic Programming for the stixel estimation, reach a performance improvement up to 2 times, while the performance per watt ratio is 25 times better. Their proposal achieves real-time performance for realistic problem sizes, proving that the low-power envelope and remarkable performance of embedded CPU-GPU hybrid systems make them good target platforms for most real-time video processing tasks, paving the way for more complex and larger applications.

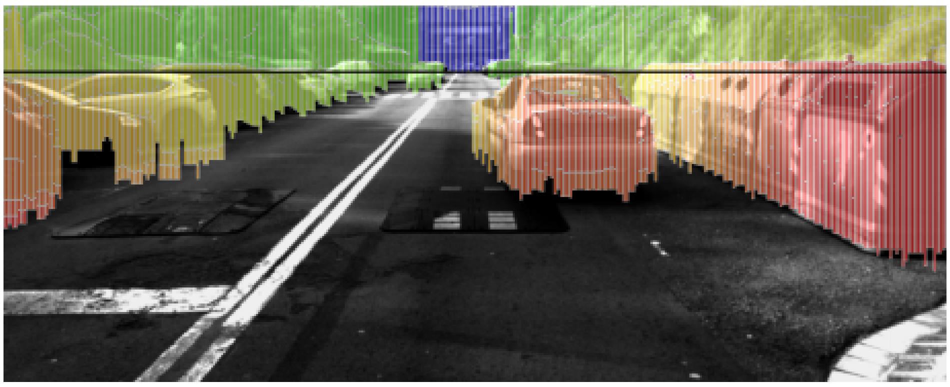


Figure 2.3: Example of Stixel World estimation. Sky stixels are represented on blue, object stixels are represented on green-to-red (read= close, green= far), and ground stixels are transparent.

Finally, Onur Ulusel et al. [11] present a power and performance evaluation of three low cost feature detection and description algorithms (Features from Accelerated Segment Test (FAST), FAST + Binary Robust Independent Elementary Features (BRIEF) and FAST + Binary Robust Invariant Scalable Keypoints (BRISK)) implemented on embedded CPUs, GPUs and FPGAs. More specifically, the devices are a MicroZED development board featuring a 28nm Zynq 7020 SoC, which integrates an Artix-7 FPGA with a dual-core ARM Cortex A9 CPU, and a 1GB DDR3 for embedded FPGA and a Tegra K1 SoC with an integrated Kepler GPU with 192 CUDA cores that run at 950MHz and a quadcore ARM Cortex A15 CPU that runs at 2.5GHz for embedded CPU and GPU. They show that FPGA implementations outperform the state-of-the-art embedded CPUs and GPUs in terms of both power and performance.

2.2 Implementation of Harris algorithm on embedded devices

As far as the specific algorithm will be implemented in this thesis concerned, the Harris Corner Detector algorithm, there are several studies that have accelerated it using embedded devices. Alexandru Amaricai et al. [2] using Xilinx Spartan6 and Xilinx Virtex-5 FPGA device propose a performance friendly solution for both the two SoC boards with significant less BRAM usage with respect to other approaches.

The same algorithm are trying to accelerate Tak Lon Chao et al. [3], using the Avnet Zedboard(FPGA: Xilinx xc7z020-clg484-1), an OmniVision OV7670 image sensor and a VGA monitor to show the result achieve up to 144 frames per second with an implementation based on using low budget of hardware (9485 6-bit LUTS and 4131 registers only without using any DSP resources). A frame of the output image of their application is shown in Figure 2.4

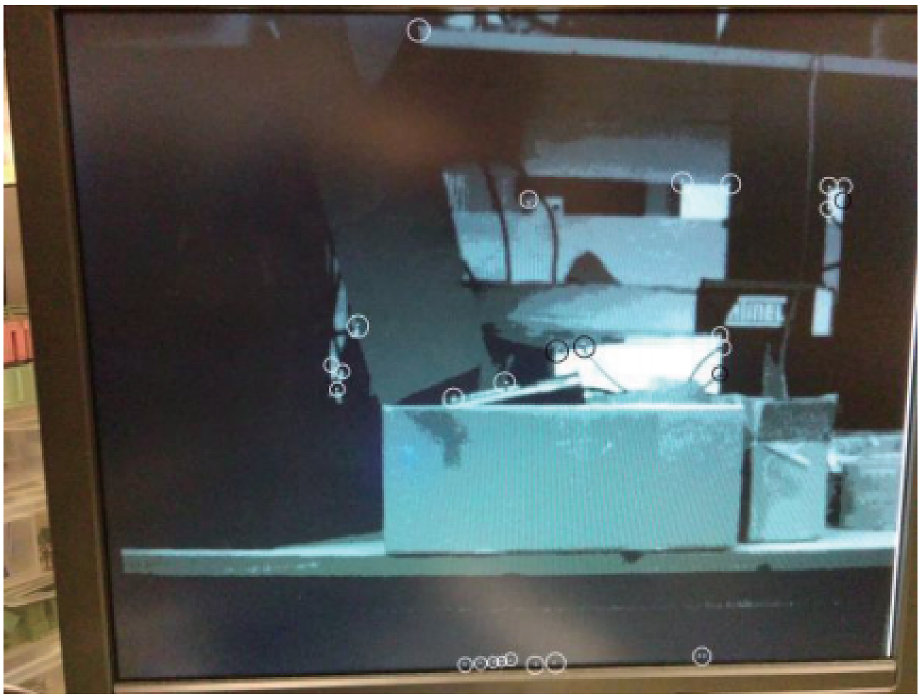


Figure 2.4: Harris Corner feature detected by their FPGA hardware method

Eventually, Ioannis Stratakos et al. [10] propose a design of an image processing system for embedded applications, included the Harris Corner detector, where using Xilinx's Zynq-7020 (ZC702 All Programmable SoC), he achieved to outperform the pure software implementations running on ARM and Intel processors achieving speedup of 71.6 and 2.5 respectively.

Chapter 3

Target Hardware Platform

3.1 Graphic Processing Units

A Graphics Processing Unit (GPU) is a single-chip processor that performs rapid mathematical calculations, primarily for the purpose of rendering images. In the early days of computing, the central processing unit (CPU) used to perform these calculations. As more graphics-intensive applications were developed, their demands degraded performance of CPU. GPU was introduced as a way to offload those tasks from the CPU, freeing up its processing power. Nvidia introduced the first GPU, GeForce 256, in 1999. This GPU model could process 10 million polygons per second and had more than 22 million transistors.

Nowadays, GPUs are widely used in embedded systems, mobile phones, personal computers, and game consoles. Modern GPUs are very efficient at manipulating graphics as well as in image processing. Furthermore, their highly parallel structure makes them more effective than general-purpose CPUs for algorithms where processing of large blocks of data is done in parallel. As a result, a large discrepancy in floating-point capability between the CPU and the GPU was emerged. The main reason is that GPUs are specialized for compute-intensive, highly parallel computation and therefore designed such as more transistors are devoted to data processing rather than data caching and flow control, as is the case for the CPU. More specifically, GPU is especially well-suited to address problems that can be expressed as data-parallel computations with high arithmetic intensity. Because the same program is executed in each data element, there is a lower requirement for sophisticated flow control. Memory access latency can be hidden with calculations instead of big data caches. GPUs can therefore be considered as general-purpose, high-performance, many-core processors capable of very high computation and memory throughput.

By the end of 2010, multicore processors had entered the mainstream of affordable computing. Nearly all new desktop computers used dual-core and even quad-core processors. Meanwhile, advances in semiconductor technology led GPUs to grow in sophistication and complexity. Those developments gave rise to heterogeneous computing systems. Heterogeneous architectures use more than one kind of processor (CPUs, GPUs, FPGAs etc.) and gain performance not just by adding cores, but also by incorporating specialized processing capabilities of each kind of processor to handle particular tasks.

GPUs are widely used in heterogeneous systems. A GPU is currently not a standalone platform but a co-processor to a CPU. Therefore, GPUs must operate in conjunction with a CPU-based host through a PCI-Express bus, as shown in Figure 3.1. That is why, in GPU computing terms, the CPU is called the host and the GPU is called the device.

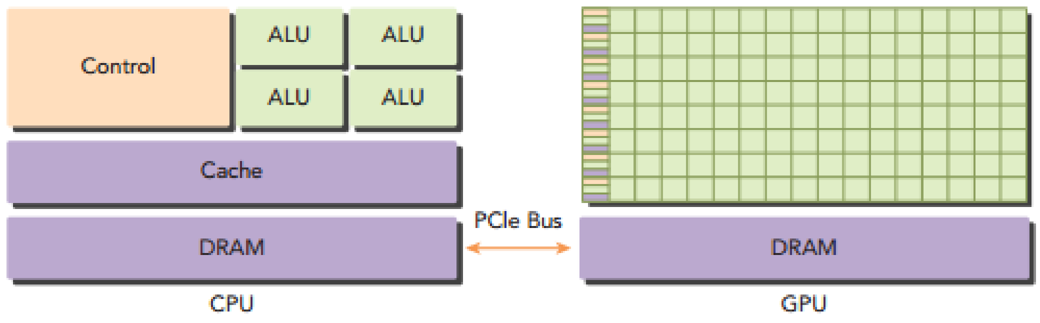


Figure 3.1: Heterogeneous architecture

An heterogeneous application consists of two parts:

- Host code
- Device Code

Host code runs on CPUs and device code runs on GPUs. An application executing on a heterogeneous platform is typically initialized by the CPU. The CPU code is responsible for managing the environment code and data for the device before loading compute-intensive tasks on the device.

With computational intensive applications, program sections often exhibit a rich amount of data parallelism. GPUs are used to accelerate the execution of this portion of data parallelism. When a hardware component that is physically separate from the CPU is used to accelerate computationally intensive sections of an application, it is referred to as a hardware accelerator. GPUs are arguably the most common example of a hardware accelerator.

3.2 NVIDIA Tegra x1 System-on-Chip

A system on chip (SoC) is an integrated circuit that integrates all components of a computer or other electronic systems. These components typically include a central processing unit (CPU), a hardware accelrator, memory, input/output ports and secondary storage – all on a single substrate. It may contain digital, analog, mixed-signal, and often radio-frequency functions, depending on the application. SoCs are very common in the mobile computing market because of their low power consumption. SoCs are commonly applied in the area of embedded systems.

Tegra, the system on chip developed by NVIDIA that we will use in this thesis, integrates an ARM architecture central processing unit (CPU), graphics processing unit (GPU)-sharing a common DRAM memory with CPU-, northbridge, southbridge, and memory controller onto one package. More specifically Nvidia's Tegra X1 (codenamed "Erista") features four ARM Cortex-A57 cores and four ARM Cortex-A53 cores (not to be accessed by the operating system and are used automatically in very low power scenarios), as well as a Maxwell-based graphics processing unit. Clearly the components of Tegra X1 are:

- CPU: ARMv8 ARM Cortex-A57 quad-core + ARM Cortex-A53 quad-core (64-bit)
- GPU: Maxwell-based 256 core GPU
- MPEG-4 HEVC and VP9 encoding/decoding support

- TSMC 20 nm process
- TDP (Thermal Design Power) 15 watts, with average power consumption less than 10 watts

Tegra X1 is NVIDIA's newest mobile processor, and includes NVIDIA's highest performing, and power efficient Maxwell GPU architecture. Utilizing a 256 CUDA Core Maxwell GPU, Tegra X1 delivers class-leading performance and incredible energy efficiency, while supporting all the modern graphics and compute APIs.

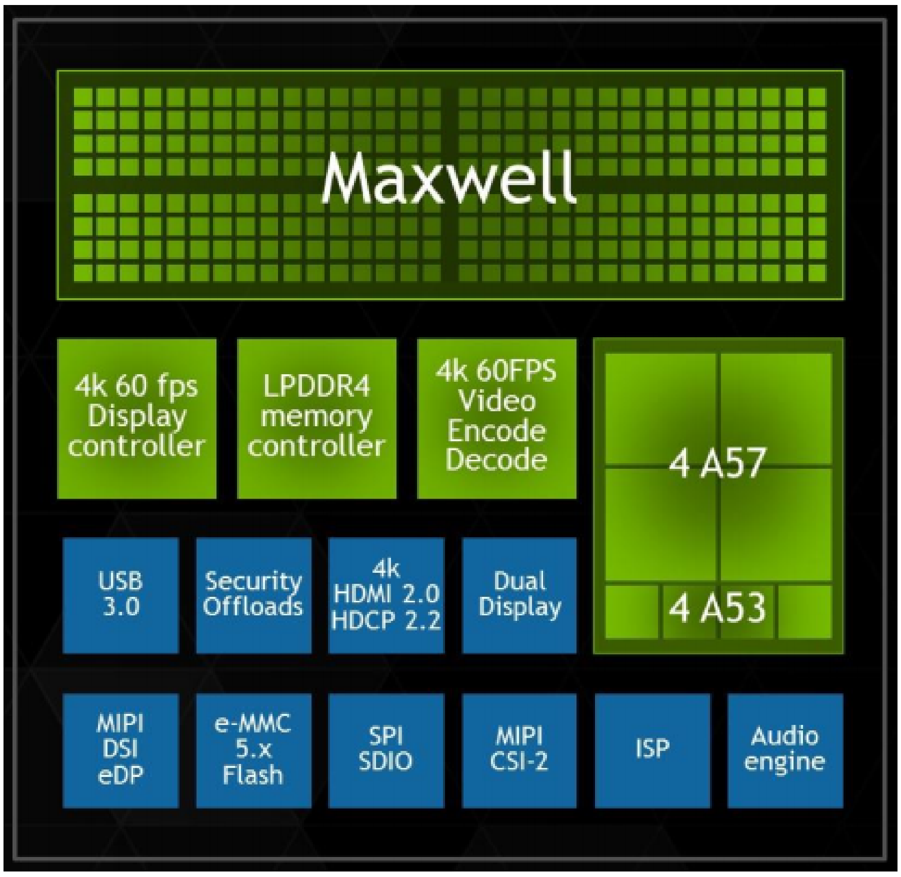


Figure 3.2: NVIDIA Tegra X1 Mobile Processor [8]

3.2.1 Tegra X1 CPU Architecture

The NVIDIA Tegra X1 CPU architecture uses four high performance ARM Cortex A57 cores in conjunction with four power-efficient ARM Cortex A53 cores. The Cortex A57 CPU complex on Tegra X1 shares a common 2MB L2 cache, and each of the four CPU cores has a 48KB L1 instruction cache and a 32KB L1 data cache. The lower performance, more power-efficient Cortex A53 CPU complex shares a common 512KB L2 cache, and each of its four CPU cores has its own 32KB L1 instruction cache and 32KB L1 data cache. Workloads that require high performance are processed by the A57 CPU cores, and lower performance workloads are processed by the energy-efficient A53 CPU cores. Intelligent algorithms analyze workloads presented by the operating system to dynamically switch between the high performance and low performance cores to deliver optimal performance and power efficiency.

3.2.2 Maxwell Graphics Architecture in Tegra X1

Maxwell GPU architecture is organized in Graphics Processing Clusters (GPC), Streaming Multiprocessors (SM), and memory controllers.The Maxwell GPU in Tegra X1 contains two SMs; each SM consists of fundamental compute cores called CUDA Cores, texture units, and a Polymorph engine. Each Maxwell SM (called SMM) includes 128 CUDA cores. The SM is the heart of our GPUs. Almost every operation flows through the SM at some point in the rendering pipeline. Maxwell GPUs feature a new SM that's been designed to provide dramatically improved performance per watt. Each SMM contains four warp schedulers, and each warp scheduler is capable of dispatching two instructions per warp every clock. The Maxwell SMM is partitioned into four distinct 32-CUDA core processing blocks (128 CUDA cores total per SM), each with its own dedicated resources for scheduling and instruction buffering as shown in Figure 3.3.

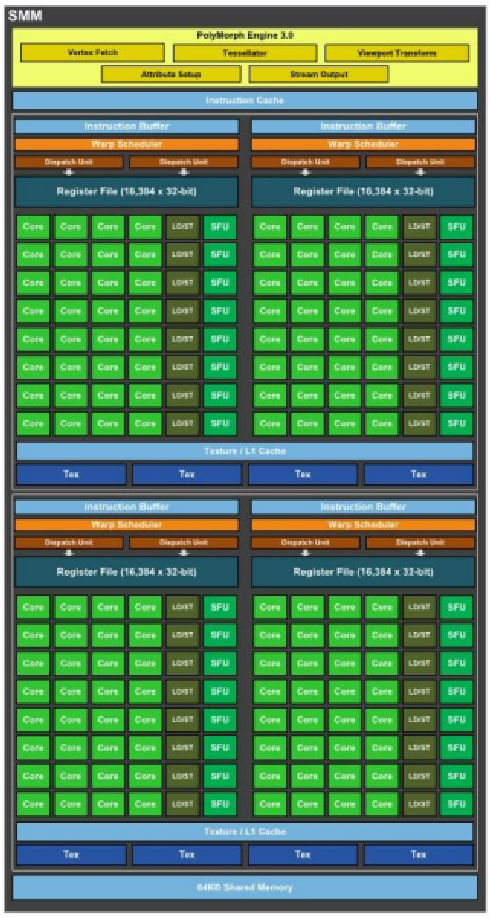


Figure 3.3: Maxwell SMM diagram [8]

Chapter 4

Cuda Programming Model

At first, we have to present the basic idea of how GPU works with CUDA. The CUDA programming model enables the programmer to execute applications on heterogeneous computing systems by simply annotating code with a small set of extensions to the C programming language. A heterogeneous environment consists of CPUs complemented by GPUs. Therefore, you should note the following distinction:

- Host: the CPU and its memory (host memory)
- Device: the GPU and its memory (device memory)

A key component of the CUDA programming model is the kernel — the code that runs on the GPU device. As developers we can express a kernel as a sequential program. Behind the scenes, CUDA manages scheduling programmer-written kernels on GPU threads. From the host, you define how your algorithm is mapped to the device based on application data and GPU device capability. The intent is to enable you to focus on the logic of your algorithm in a straightforward fashion (by writing sequential code) and not get bogged down with details of creating and managing thousands of GPU threads. A typical CUDA program consists of serial code complemented by parallel code. As shown in Figure 4.1, the serial code (as well as task parallel code) is executed on the host, while the parallel code is executed on the GPU device. The host code is written in ANSI C, and the device code is written using CUDA C.

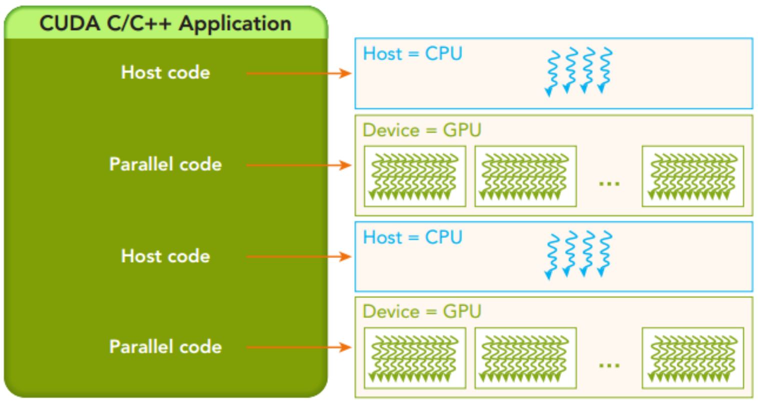


Figure 4.1: Separation between host and device

Apart from the implementation of the application, as developers we aim to the best performance that GPU can give us. There are many factors that affect the perofrmance of the implentation of an algorithm on the CPU-GPU system like thread/blocks organization, choise of data transfer way e.t.c. Some of the most important are:

In the rest of this Chapter we discuss the most important factors and techniques that CUDA provides us.

4.1 Organizing Threads/Blocks

When a kernel function is launched from the host side, execution is moved to a device where a large number of threads are generated and each thread executes the statements specified by the kernel function. Knowing how to organize threads is a critical part of CUDA programming. CUDA exposes a thread hierarchy abstraction to enable you to organize your threads. This is a two-level thread hierarchy decomposed into blocks of threads and grids of blocks, as shown in Figure 4.2. Grids and blocks can be organized on 1, 2 or 3 dimensions.

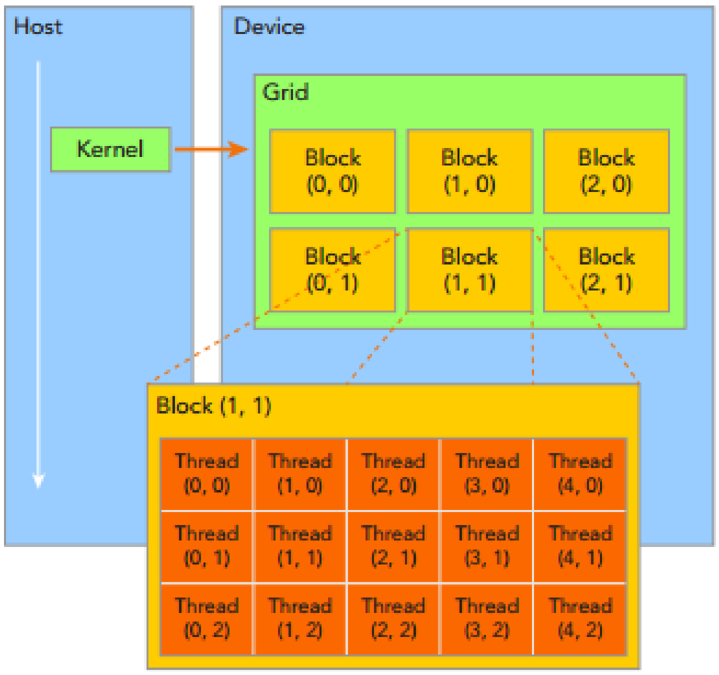


Figure 4.2: Two-level thread hierarchy decomposed into blocks of threads and grids of blocks [4]

All threads spawned by a single kernel launch are collectively called a grid. All threads in a grid share the same global memory space. A grid is made up of many thread blocks. A thread block is a group of threads that can cooperate with each other using:

- Block-local synchronization
- Block-local shared memory

Threads from different blocks cannot cooperate. Threads rely on the following two unique coordinates to distinguish themselves from each other:

- blockIdx (block index within a grid)
- threadIdx (thread index within a block

Depending on the compute capability of each GPU there are technical specifications that concern the organization of threads and blocks. For us, having a compute capability 5.3 in the Maxwell GPU of the Tegra x1, gives us the restriction of:

- Max threads per block: 1024
- Max thread dimensions: (1024, 1024, 64)
- Max grid dimensions: (2147483647, 65535, 65535)

A general rule for organization of threads is that every block has to contain multiples of 32 threads. This is because threads that execute to the GPU are divided into groups that are called “warps". All GPU generations have a warp size of 32 threads and all threads in the same warp execute the same instruction, typically on different data (Single Instruction Multiple Threads). Optimizing the workloads to fit within the boundaries of a warp (group of 32 threads) will generally lead to more efficient utilization of GPU compute resources.

4.2 Data transfer way

Memory management in CUDA programming is similar to C programming, with the added programmer responsibility of explicitly managing data movement between the host and device. There are various data transfer ways.

4.2.1 Classic Memory Transfer

The allocation of the device global memory on the host is implemented with the function:

```
cudaError_t cudaMalloc(void **devPtr, size_t count);
```

Once global memory is allocated, you can transfer data to the device from the host using the following function:

```
cudaError_t cudaMemcpy(void *dst, const void *src, size_t count, enum cudaMemcpyKind kin
```

This function copies count bytes from the memory location src to the memory location dst. The variable kind specifies the direction of the copy and can have the following values:

```
cudaMemcpyHostToHost
cudaMemcpyHostToDevice
cudaMemcpyDeviceToHost
cudaMemcpyDeviceToDevice
```

The data transfer from the host to the device is labeled HtoD, and from the device to the host DtoH.

4.2.2 Pinned Memory Transfer

Allocated host memory is by default pageable, that is, subject to page fault operations that move data in host virtual memory to different physical locations as directed by the operating system. Virtual memory offers the illusion of much more main memory than is physically available, just as the L1 cache offers the illusion of much more on-chip memory than is physically available.

The GPU cannot safely access data in pageable host memory because it has no control over when the host operating system may choose to physically move that data. When transferring data from pageable host memory to device memory, the CUDA driver fi rst allocates temporary page-locked or pinned host memory, copies the source host data to pinned memory, and then transfers the data from pinned memory to device memory, as illustrated on the left side of Figure 4.3.

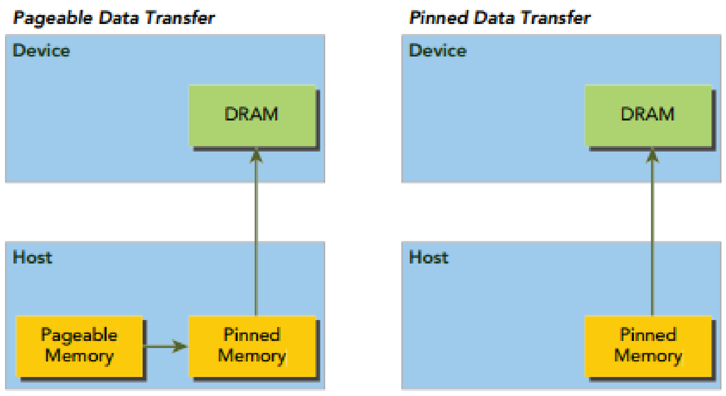


Figure 4.3: Transfer between host pinned and device memory [4]

The CUDA runtime allows you to directly allocate pinned host memory using:

```
cudaError_t cudaMallocHost(void **devPtr, size_t count);
```

This function allocates count bytes of host memory that is page-locked and accessible to the device. Since the pinned memory can be accessed directly by the device, it can be read and written with much higher bandwidth than pageable memory. However, allocating excessive amounts of pinned memory might degrade host system performance, since it reduces the amount of pageable memory available to the host system for storing virtual memory data.

4.2.3 Zero-Copy Memory Transfer

In general, the host cannot directly access device variables, and the device cannot directly access host variables. There is one exception to this rule: zero-copy memory. Both the host and device can access zero-copy memory. GPU threads can directly access zero-copy memory. There are several advantages to using zero-copy memory in CUDA kernels, such as: 1)Leveraging host memory when there is insufficient device memory 2)Avoiding explicit data transfer between the host and device 3)Improving PCIe transfer rates

Zero-copy memory is pinned (non-pageable) memory that is mapped into the device address space. You can create a mapped, pinned memory region with the following function:

```
cudaError_t cudaHostAlloc(void **pHost, size_t count, unsigned int flags);
```

You can obtain the device pointer for mapped pinned memory using the following function:

```
cudaError_t cudaHostGetDevicePointer(void **pDevice, void *pHost, unsigned int flags);
```

The advantage of this way is that the data storing in the Zero-Copy memory causes very low load and store throughput from the function kernels because this kind of memory does not use at all the cache data memory. This way of transfer data is recommended for applications that use simple-operations function-kernels and many data transfers between CPU-GPU because there is no need for data transfers.

4.2.4 Unified Virtual Addressing Memory Transfer

Devices with compute capability 2.0 and later support a special addressing mode called Unified Virtual Addressing (UVA). UVA, introduced in CUDA 4.0, is supported on 64-bit Linux systems. With UVA, host memory and device memory share a single virtual address space, as illustrated in Figure 4.4.

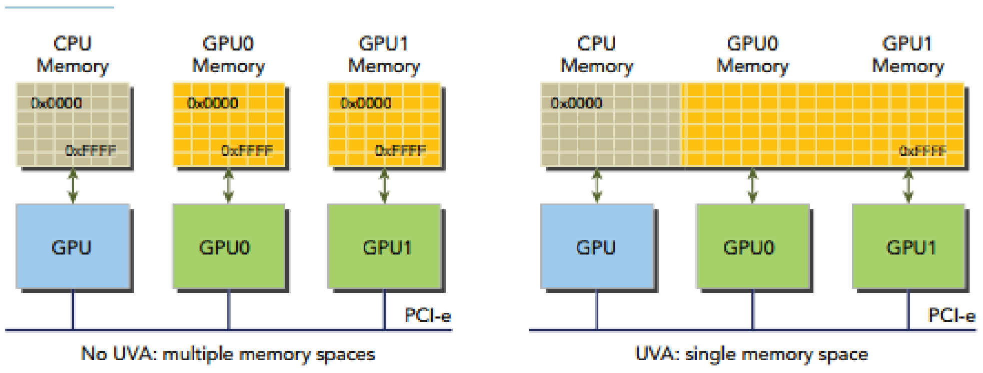


Figure 4.4: Unified Virtual Addressing Memory Transfer

The allocation of the data that will be executed by the host and the device is implemented with the function:

```
cudaHostAlloc((void **)&h_A, nBytes, cudaHostAllocMapped);
```

4.2.5 Unified Memory Transfer

With CUDA 6.0, a new feature called Unified Memory was introduced to simplify memory management in the CUDA programming model. Unified Memory creates a pool of managed memory, where each allocation from this memory pool is accessible on both the CPU and GPU with the same memory address (that is, pointer). The underlying system automatically migrates data in the unified memory space between the host and device. This data movement is transparent to the application, greatly simplifying the application code. You can also allocate managed memory dynamically using the following CUDA runtime function:

```
cudaError_t cudaMallocManaged(void **devPtr, size_t size, unsigned int flags=0);
```

This function allocates size bytes of managed memory and returns a pointer in devPtr. The pointer is valid on all devices and the host.

4.2.6 Pitch Transfer

CUDA provides the cudaMallocPitch function to "pad" 2D matrix rows with extra bytes so to achieve the desired alignment. Assuming that we want to allocate a 2D padded array of floating point (single precision) elements, the syntax for cudaMallocPitch is the following:

```
cudaMallocPitch(&devPtr, &devPitch, Ncols * sizeof(float), Nrows);
```

CUDA provides also the cudaMemcpy2D function to copy data from/to host memory space to/from device memory space allocated with cudaMallocPitch.

```
cudaMemcpy2D(devPtr, devPitch, hostPtr, hostPitch, Ncols * sizeof(float), Nrows, cudaMem
```

When accessing 2D arrays in CUDA, theoretically memory transactions are much faster if each row is properly aligned, but we need to make experiments to ensure that that this proposal.

4.2.7 Comparison

There is not simple answer to the question what is the better way of transfer. It depends on the application and the hardware we are working for. To have an idea of the behavior of each transfer way on NVIDIA Tegra X1, we implement a 2d arrays multiplication using the beforementioned ways for transfering the arrays from host to device and vice versa. In the Figures 4.5, 4.6, 4.7, 4.8, 4.9, 4.10, 4.11, 4.12 we present the memory operations and the execution times of Matrix Multiplication for various sizes of arrays.

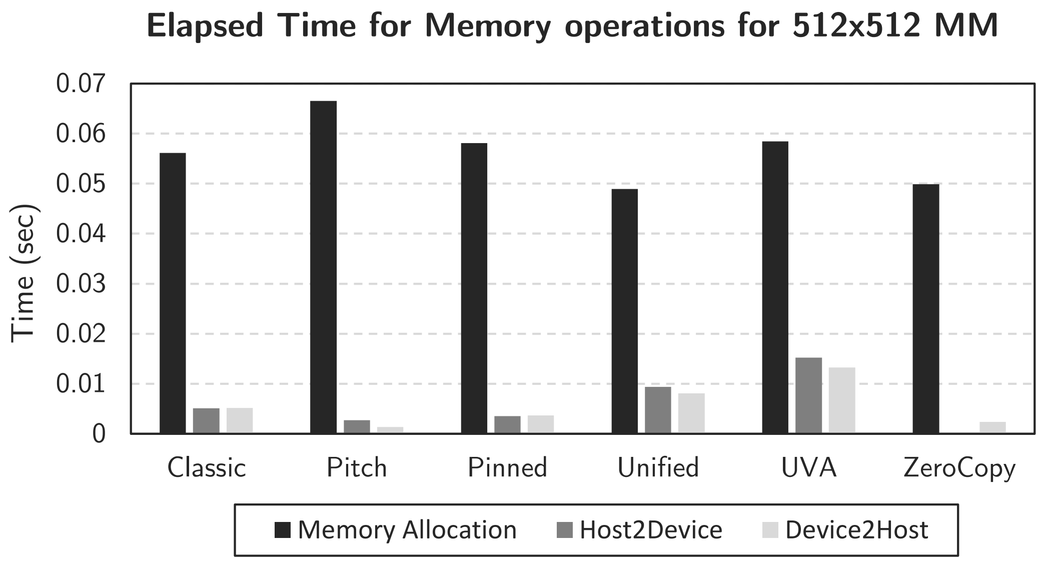


Figure 4.5: Memory operations time for 512 × 512 MM

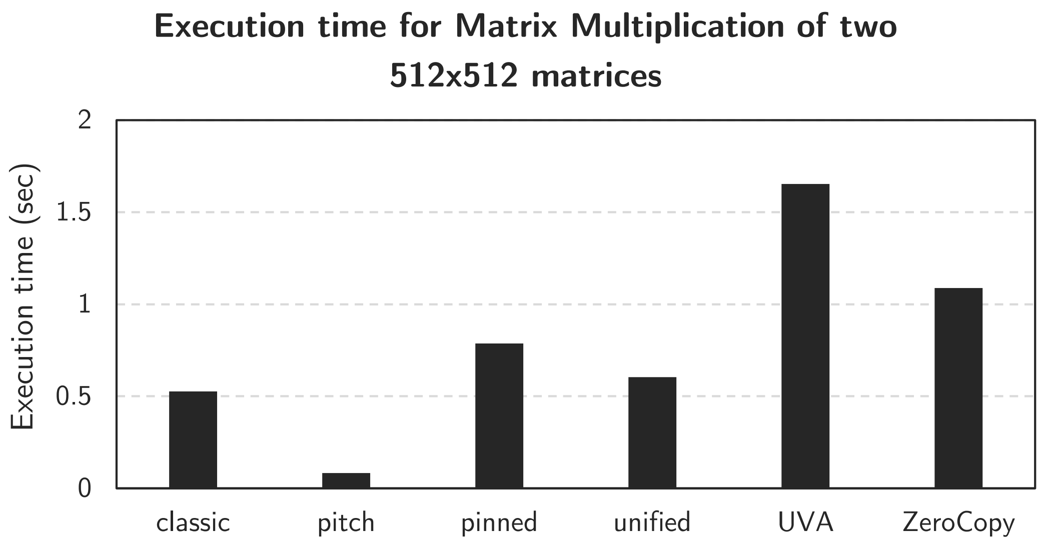


Figure 4.6: Execution time for 512 × 512 MM

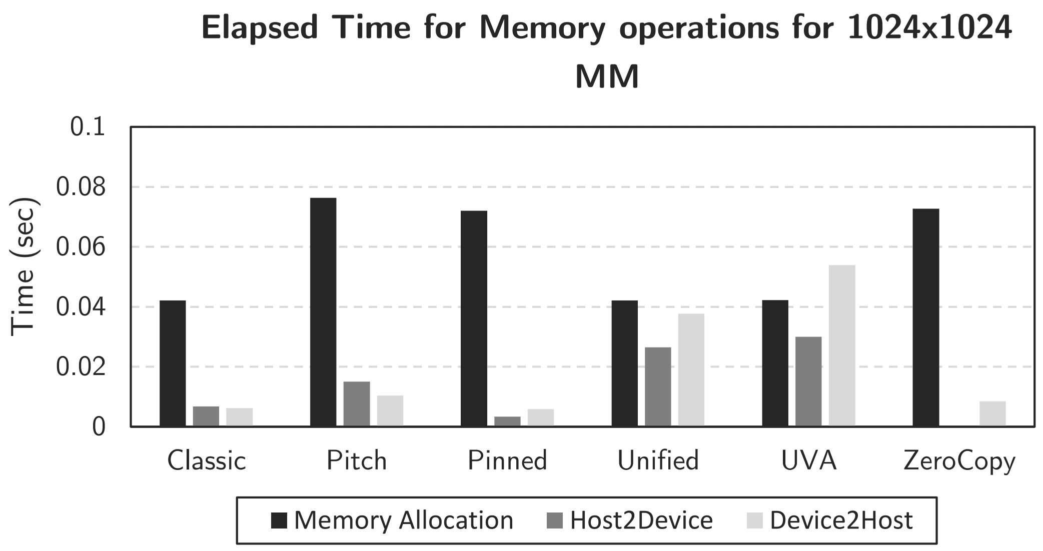


Figure 4.7: Memory operations time for 1024 × 1024 MM

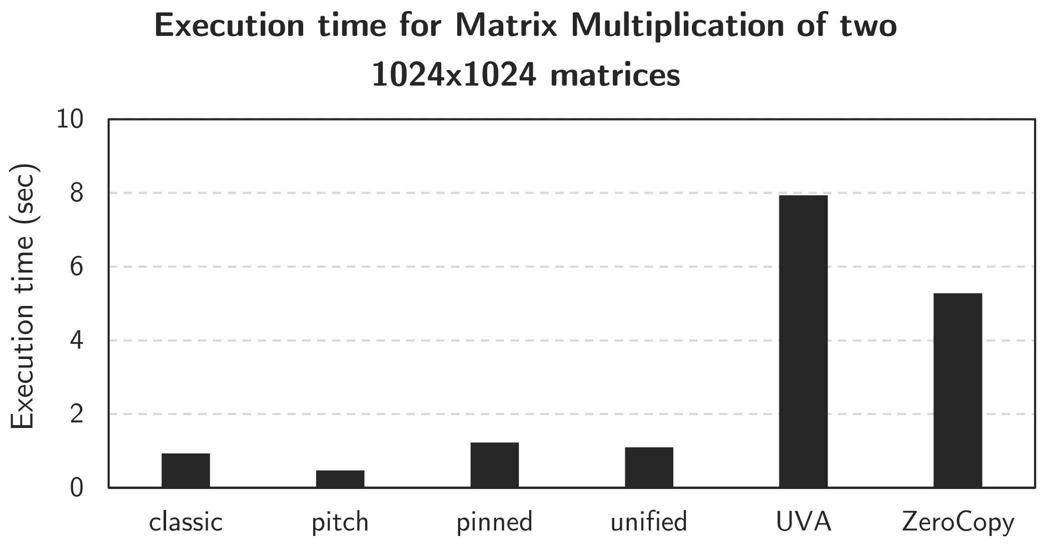


Figure 4.8: Execution time for 1024 × 1024 MM

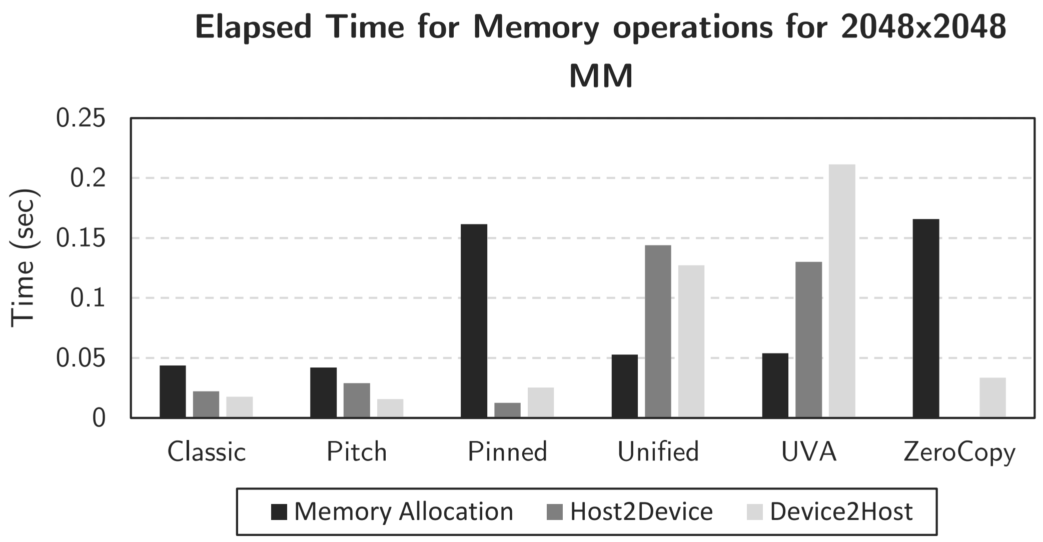


Figure 4.9: Memory operations time for 2048 × 2048 MM

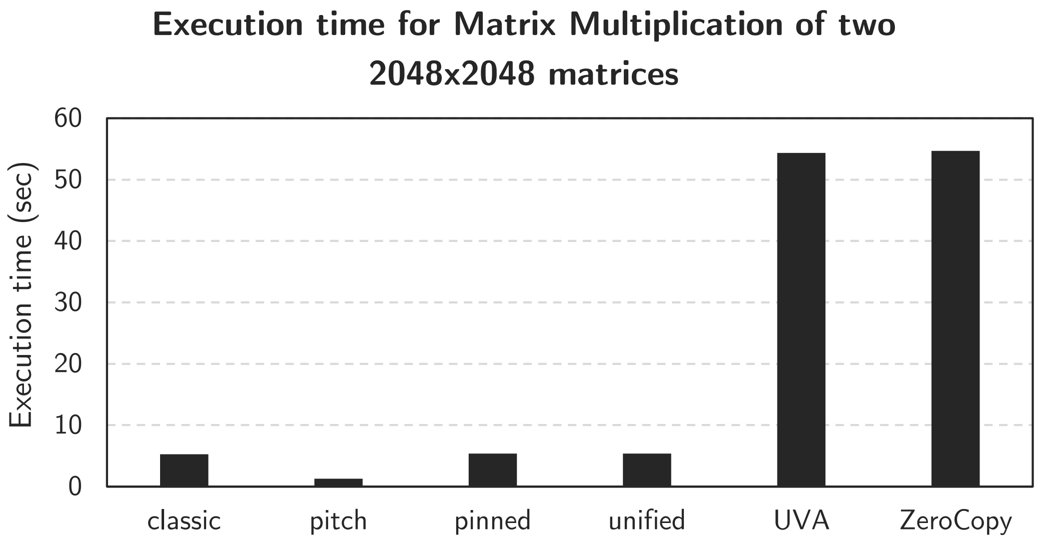


Figure 4.10: Execution time for 2048 × 2048 MM

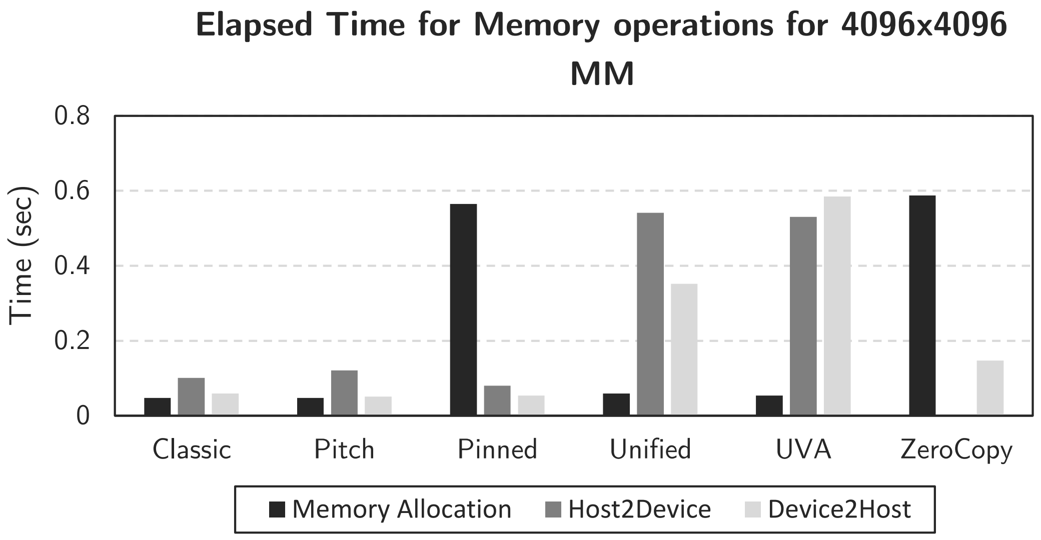


Figure 4.11: Memory operations time for 4096 × 4096 MM

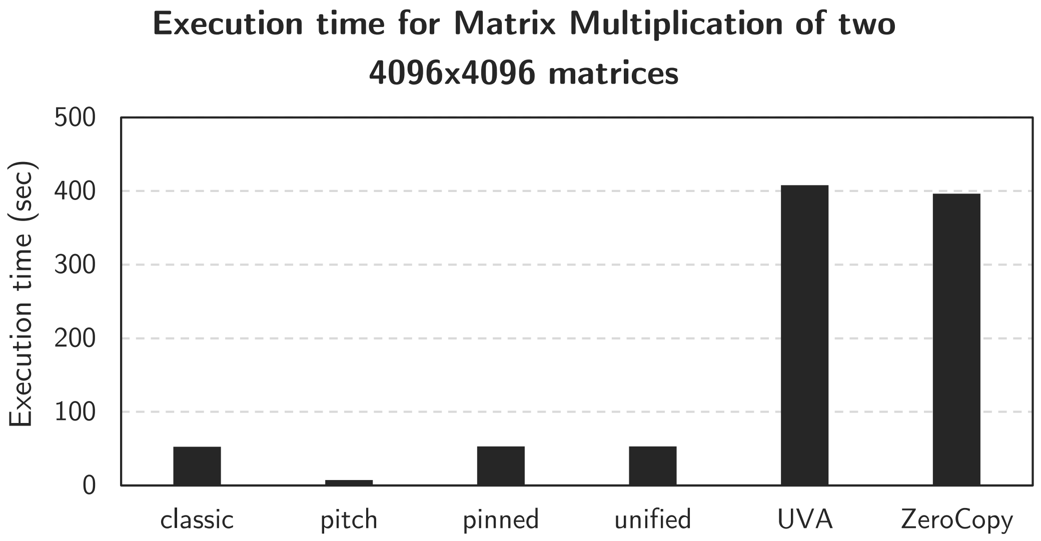


Figure 4.12: Execution time for 4096 × 4096 MM

From the Figures above, we can conclude that transfering data time is not the most important issue as those ways are concerned. We see that execution kernel times varies from way to way. UVA and ZeroCopy transfer way cause high-latencies function kernels beacuse of their cache unfriendly behavior. These conclusions are very useful to us for the full exploration of transfer ways and other programming and algorithmic techniques that will be presented in the next Chapter.

4.3 Shared Memory

GPUs are equipped with two types of memory:

- On-board memory
- On-chip memory

Global memory is large, on-board memory and is characterized by relatively high latencies. Shared memory is smaller, low-latency on-chip memory that offers much higher bandwidth than global memory. It as a program-managed cache that is generally useful as:

- An intra-block thread communication channel
- A program-managed cache for global memory data
- Scratch pad memory for transforming data to improve global memory access patterns

Shared memory (SMEM) is one of the key components of the GPU. Physically, each SM contains a small low-latency memory pool shared by all threads in the thread block currently executing on that SM. Shared memory enables threads within the same thread block to cooperate, facilitates reuse of on-chip data, and can greatly reduce the global memory bandwidth needed by kernels. Because the contents of shared memory are explicitly managed by the application, it is often described as a program-managed cache. As illustrated in Figure 4.13 , all load and store requests to global memory go through the L2 cache, which is the primary point of data unifi cation between SM units. Note that shared memory and L1 cache are physically closer to the SM than both the L2 cache and global memory. As a result, shared memory latency is roughly 20 to 30 times lower than global memory, and bandwidth is nearly 10 times higher.

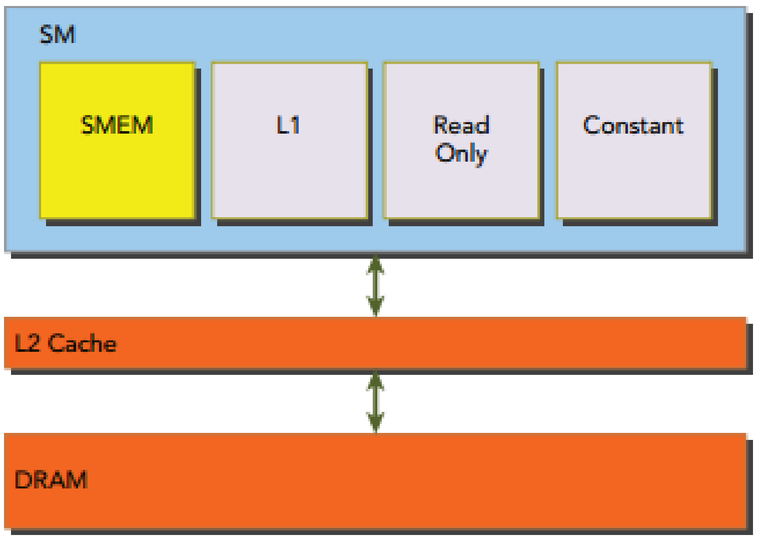


Figure 4.13: Shared Memory

There are several ways to allocate or declare shared memory variables depending on your application requirements. You can allocate shared memory variables either statically or dynamically. Shared memory can also be declared as either local to a CUDA kernel or globally in a CUDA source code fi le. CUDA supports declaration of 1D, 2D, and 3D shared memory arrays.

A shared memory variable is declared with the following qualifi er:

```
__shared__
```

The following code segment statically declares a shared memory 2D float array. If declared inside a kernel function, the scope of this variable is local to the kernel. If declared outside of any kernels in a file, the scope of this variable is global to all kernels.

```
__shared__ float tile[size_y][size_x];
```

If the size of shared memory is unknown at compile time, you can declare an un-sized array with the extern keyword. For example, the following code segment declares a shared memory 1D un-sized int array. This declaration can be made either inside a kernel or outside of all kernels.

```
extern __shared__ int tile[];
```

Because the size of this array is unknown at compile-time, you need to dynamically allocate shared memory at each kernel invocation by specifying the desired size in bytes as a third argument inside the triple angled brackets, as follows:

```
kernel<<<grid, block, isize * sizeof(int)>>>(...)
```

Note that we can only declare 1D arrays dynamically.

4.4 Constant and Texture Memory

Constant memory is a special-purpose memory used for data that is read-only and accessed uniformly by threads in a warp. While constant memory is read-only from kernel codes, it is both readable and writable from the host. It resides in device DRAM (like global memory) and has a dedicated on-chip cache. Like the L1 cache and shared memory, reading from the per-SM constant cache has a much lower latency than reading directly from constant memory. There is 64 KB limit on the size of constant memory cache per SM. Constant memory has a different optimal access pattern than any of the other types of memory mentioned already. It is best if all threads in a warp access the same location in constant memory. Accesses to different addresses by threads within a warp are serialized. Thus, the cost of a constant memory read scales linearly with the number of unique addresses read by threads within a warp.

In addition, Texture memory resides in device memory and is cached in a per-SM, read-only cache. Texture memory is a type of global memory that is accessed through a dedicated read-only cache. The readonly cache includes support for hardware fi ltering, which can perform fl oating-point interpolation as part of the read process. Texture memory is optimized for 2D spatial locality, so threads in a warp that use texture memory to access 2D data will achieve the best performance. For some applications, this is ideal and provides a performance advantage due to the cache and the filtering hardware. However, for other applications using texture memory can be slower than global memory.

Chapter 5

The Harris Corner Detector

5.1 Feature Detection

In computer vision and image processing feature detection includes methods for computing abstractions of image information and making local decisions at every image point whether there is an image feature of a given type at that point or not. The resulting features will be subsets of the image domain, often in the form of isolated points, continuous curves or connected regions. Features can be categorized as follows:

- **Edges** : Edges are points where there is a boundary between two image regions. In general, an edge can be of almost arbitrary shape, and may include junctions. In practice, edges are usually defined as sets of points in the image. Edges have a one-dimensional structure.
- **Corners** : The terms corners or interest points refer to point-like features in an image, which have a local two dimensional structure.
- **Blobs** : Blobs provide a complementary description of image structures in terms of regions, as opposed to corners that are more point-like.
- **Ridges** : From a practical viewpoint, a ridge can be thought of as a one-dimensional curve that represents an axis of symmetry, and in addition has an attribute of local ridge width associated with each ridge point.

The interest of this thesis is on corner detection. The corner detection, or interest point detection, is a useful method that used of an image to extract the feature or infer the context. It is predominantly applied in many aspects such as image mosaicing, tracking and recognizing.

In image processing, a point can be considered as a corner if there is a intersection of two edges. A good indicator that determines the quality of a corner detection algorithm is to see if it can detect the same corner under multiple circumstances, in other words, different similar pictures that had done some other image processing such as rotation, darken, etc. The most frequently used algorithm for corner detection is proposed by Harris and Stephens [5], which is a further work on a method developed by Moravec [9] and was first published in 1988. In this thesis, the Harris Corner Detector is chosen and implemented as an accelerator.

5.2 Theoretical Background

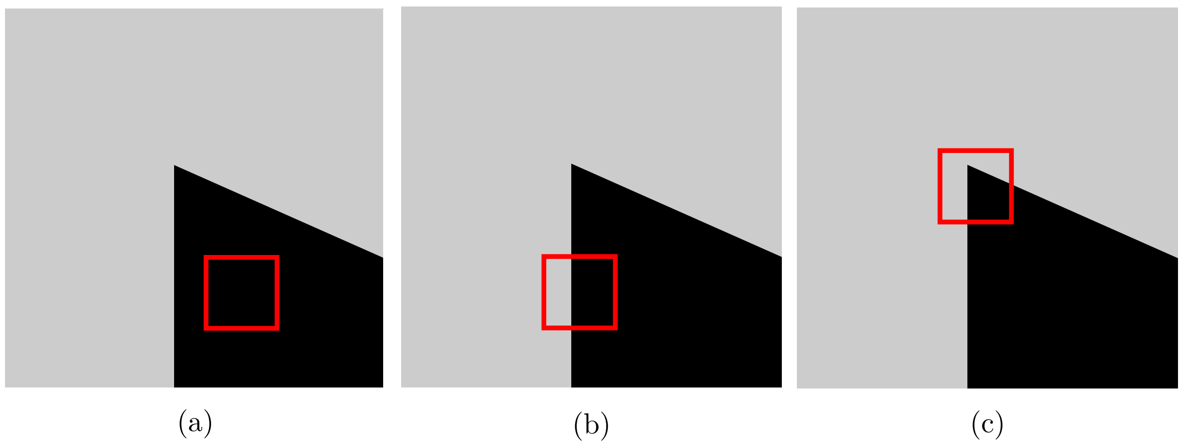


Figure 5.1: An example of (a) a flat region, (b) an edge and (c) a corner.

Harris Corner Detector was first introduced by Chris Harris and Mike Stephens in 1988 upon the improvement of Moravec's corner detector. Compared to the previous one, Harris' corner detector takes the differential of the corner score into account with reference to direction directly, instead of using shifting patches for every 45 degree angles, and has been proved to be more accurate in distinguishing between edges and corners.Since then, it has been improved and adopted in many algorithms to preprocess images for subsequent applications.

Assuming a 2-dimensional image, whose intensity is denoted as I, Moravec starts with a window W centered at the pixel $p(x, y)$ and moves (shifts) this window in the neighborhood of p. If the movement is (u, v) the changes of the intensity are measured with the help of the auto-correlation function as

$$E(u,v) = \sum_{x,y} w(x,y)[I(x+u,y+v) - I(x,y)]^2 \tag{5.1}$$

where :

- $w(x, y)$ a window function equal to 1 inside the window W and 0 outside
- $I(x+u, y+v)$ the shifted intensity, where the shifts are $(u, v) = (1, 0), (1, 1), (0, 1), (-1, 1)$
- $I(x, y)$ the intensity of the image at position (x, y)

Small changes will appear in all directions for a constant intensity in the neighborhood of $p(x, y)$, which means there is a flat region (Figure 5.1a). Small changes in only one direction can be found for an edge (Figure 5.1b) whereas the direction of nearly no changes resembles the direction of the edge and finally big changes in all directions will be observed for a corner (Figure 5.1c).

Harris and Stephens [5] improved upon Moravec's corner detector by considering the differential of the corner score with respect to direction directly, instead of using shifts. To eliminate Moravec's algorithm shortcomings, which were noisy response due to the binary window function and anisotropic response due to the shifts used, first they applied a Gaussian window function

$$w(x,y) = e^{\left(-\frac{(x^2+y^2)}{2\sigma^2}\right)} \tag{5.2}$$

and secondly they approximated $I(x + u, y + v)$, by a Taylor expansion to consider all small shifts, as

$$I(x+u,y+v) \approx I(x,y) + I_x(x,y)u + I_y(x,y)v \tag{5.3}$$

where I_x, I_y partial derivatives of I. Based on this approximation the expression 5.1 becomes

$$E(u,v) \approx \sum_{x,y} w(x,y)[I_x(x,y)u + I_y(x,y)v]^2 \tag{5.4}$$

or

$$E(u,v) \approx \begin{bmatrix} u & v \end{bmatrix} A \begin{bmatrix} u \\ v \end{bmatrix} \tag{5.5}$$

where A is a 2x2 matrix computed from the image derivatives

$$A = \sum_{x,y} w(x,y) \begin{bmatrix} I_x^2 & I_x I_y \\ I_x I_y & I_y^2 \end{bmatrix} \tag{5.6}$$

The eigenvalues λ_1 and λ_2 of this matrix describe the changes inside the window similar to the moving window of Moravec. The way to compute the exact value of eigenvalues is computationally expensive and complex, in which the calculation of square root is needed, so Harris and Stephens developed another approach to "measure" the eigenvalues or in other words the corner response ("cornerness") by means of the function

$$R = \det A - \kappa(\operatorname{Tr} A)^2 \tag{5.7}$$

where $\det A = \mathrm{I}_x^2\mathrm{I}_y^2 - (\mathrm{I}_x\mathrm{I}_y)^2 = \lambda_1\lambda_2$, $\operatorname{Tr} A = \lambda_1 + \lambda_2$ and $\kappa = 0.04 - 0.06$. Based on the value of R the following cases are considered:

- when $|R|$ is small, which happens when λ_1 and λ_2 are small, the region is flat.
- when $R < 0$, which happens when $\lambda_1 \gg \lambda_2$ or vice versa, the region is an edge.
- when R is large, which happens when λ_1, λ_2 are large and $\lambda_1 \sim \lambda_2$, the region is a corner.

Figure 5.2 shows a visual representation of the classification of image points into corners, edges and flat regions based on Harris Corner Detector.

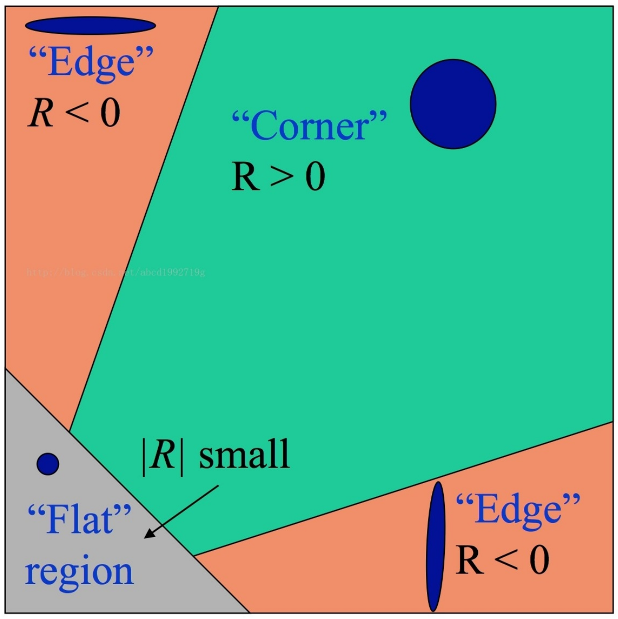


Figure 5.2: Classification of image points based on their corner response.

5.3 Hardware Implementation

As presented earlier, from an algorithmic point of view, for each pixel, Harris calculates a "cornerness" strength according to the formula $R = \mathrm{I}_x^2\mathrm{I}_y^2 - (\mathrm{I}_x\mathrm{I}_y)^2 - 0.04 \cdot (\mathrm{I}_x^2 + \mathrm{I}_y^2)^2$, where $\mathrm{I}_x^2, \mathrm{I}_y^2$ and $\mathrm{I}_x\mathrm{I}_y$ denote the Gaussian-smoothed products of the image derivatives, which are themselves computed via a Sobel operator. More specifically, the steps of the implementation of the algorithm are presented below:

- Construction of $\mathrm{I}_x, \mathrm{I}_y$ by convolving the image with a 5×5 tap kernel
- Construction of $\mathrm{I}_x^2, \mathrm{I}_y^2, \mathrm{I}_x\mathrm{I}_y$ given the $\mathrm{I}_x, \mathrm{I}_y$.
- Convolution of $\mathrm{I}_x^2, \mathrm{I}_y^2$ and $\mathrm{I}_x\mathrm{I}_y$ with a 7×7 tap kernel
- Calculation of $\mathrm{R} = \mathrm{I}_x^2, \mathrm{I}_y^2 - (\mathrm{I}_x\mathrm{I}_y)^2 - 0.04 \cdot (\mathrm{I}_x^2 + \mathrm{I}_y^2)^2$, given the new $\mathrm{I}_x^2, \mathrm{I}_y^2, \mathrm{I}_x\mathrm{I}_y$.
- Selection of pixels as corners that they have the value R as local maximum

In the Figure 5.3 we show the Harris execution latency percentage per step implemented on the ARM Cortex A57 single-thread, for different resolutions of images.

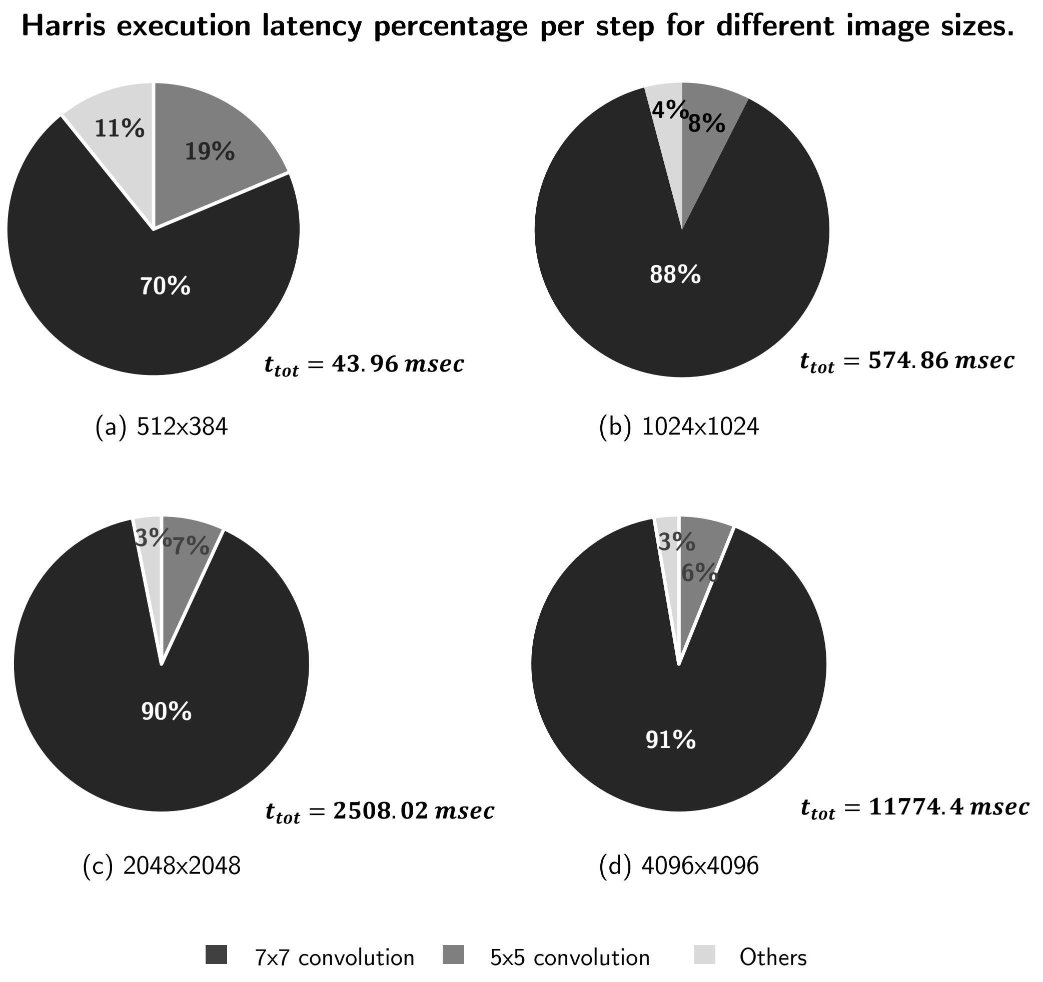


Figure 5.3: Harris execution latency percentage per step for different resolutions of image

It is already clear that the steps with the biggest execution latency are those that concern the 2D convolutions and especially the 7 × 7 convolution. It is obvious that the bigger the tap kernel with whom we convolve the image with is, the bigger the execution latency will be, because of the many accesses to the memory and operations. These are the pieces of code that have the priority of acceleration with the GPU. A first though would be that it is enough to accelerate just the 7×7 convolution, but with this implementation, after this step we had to transfer the result back to CPU to continue the processing. To avoid these transfers that would cause more latency to the whole application, we prefer to exeute all steps on the GPU.

We have already mentioned some techniques that CUDA provides us, to achieve the best performance. Following the idea of taking the advantage of parallelism and keeping the operations of a single thread of the GPU simple -due to the lack of branch predictors and complex ALUs-, every thread is corresponded to a pixel of the image. That means that the number of the total threads we will launch the compute kernel has to be equal with the total pixels of the image that we input to the system. Apart from the programming techniques we mentioned in the Chapter 4 and we will use them in the Chapter 6 there is an interesting suggested algorithmic technique to achieve better performance for applications that concern 2D convolutions. The technique of separability promises smaller implementation latency of a 2D convolution especially when we have big filter kernels.

Filtering an M×N image with a P×Q filter kernel requires roughly M×N×P×Q multiplies and adds (assuming we aren't using an implementation based on the FFT). If the kernel is separable, you can filter in two steps. The first step requires about M×N×P multiplies and adds. The second requires about M×N×Q multiplies and adds, for a total of (M×N)(P + Q). Thus, the computational advantage of separable convolution versus nonseparable convolution is $(P \times Q)/(P + Q)$. From a theoritical point of view, that means that the separability is suggested for convolutions with filter-kernels bigger than 2×2.

The proof of separable convolution 2D is presented below.

Proof. By the definition of Convolution 2D:

$$y[m,n] = x[m,n] * h[m,n] = \sum_{j=-\infty}^{\infty} \sum_{i=-\infty}^{\infty} x[i,j]h[m-i,n-j]$$

Since convolution is commutative

$$(x[n] * y[n] = y[n] * x[n])$$

Thus, we can swap the order of convolution. So

$$y[m,n] = h[m,n] * x[m,n] = \sum_{j=-\infty}^{\infty} \sum_{i=-\infty}^{\infty} h[i,j]x[m-i,n-j]$$

And, if $h[m,n]$ is separable to (M× 1) and (1×N) then

$$h[m,n] = h_1[m]h_2[n]$$

Therefore, by substituting $h[m,n]$ into the equation, we get:

$$y[m,n] = h[m,n] * x[m,n]] = \sum_{j=-\infty}^{\infty} \sum_{i=-\infty}^{\infty} h[i,j]x[m-i,n-j] =$$

$$= \sum_{j=-\infty}^{\infty} \sum_{i=-\infty}^{\infty} h_1[i]h_2[j]x[m-i,n-j] =$$

$$= \sum_{j=-\infty}^{\infty} h_2[j][\sum_{i=-\infty}^{\infty} h_1[i]x[m-i,n-j]]$$

Since the definition of convolution 1D is

$$y[n] = x[n] * h[n] = \sum_{k=-\infty}^{\infty} x[k]h[n-k]$$

it is convolving with input and h_1, then convolve once again with the result of previous convolution and h_2. Therefore, the separable 2D convolution is performing twice of 1D convolution in horizontal and vertical direction.

$$y[m,n] = (h_1[m]h_2[n]) * x[m,n] = h_2[n] * (h_1[m] * x[m,n]) = h_1[m] * (h_2[n] * x[m,n])$$

$\square$

Finally, in addition to the techniques already mentioned for the improvement of performance, we used Constant and Texture 2D Memory too. Unfortunately, they did not offer us better efficiency and for this reason we do not record the experimental results we extracted.

Chapter 6

Experimental results

6.1 Experimental setup

Our target evaluation System-on-Chip, as mentioned in the Chapter 3, is NVIDIA SoC Tegra x1. It includes two distinct ARM multi-core processors, Cortex A57 and Cortex A53, coupled together with a 256-core GPU. Moreover, it includes a video accelerator (for H.265 and VP9 compression), display controller, memory controller (for external DDR4), audio engine, and other peripherals for interfacing via USB, SDIO, eMMC, HDMI, etc. The two Cortex processors of Tegra X1 are based on the latest 64-bit ARMv8-A architecture with floating-point units and SIMD extensions (NEON engine). Each one is equipped with 4 processing cores (8 in total) that operate on frequencies of up to 1.9 GHz. The differences between A53 and A57 are mainly in their pipeline (in-order dual-issue on A53 versus out-of-order multi-issue on A57), their cache size (512KB shared L2 on A53 versus 2MB shared L2 on A57), and energy efficiency approach (A57 is more power-hungry). In total, A53 can achieve up to 2.3 DMIPS/MHz/core, whereas A57 can achieve up to 4.7 DMIPS/MHz/core.

The GPU included in Tegra X1 utilizes 256 processing cores arranged in a Maxwell architecture (with its conventional CUDA cores, Streaming multiprocessors (SMs), Polymorph engines, Warp schedulers, Texture caches, etc.) and operates at frequencies of up to 1 GHz. More specifically, X1 includes two SMs with 128 CUDA cores per SM and 4 warp schedulers per SM (practically, the CUDA core can be viewed as an ALU-FPU pair). The total L2 cache is 256KB. Tegra X1 can achieve up to 512 GFLOPs peak performance (single-precision FP32). Overall, the Tegra X1 architecture focuses on power efficiency and targets embedded applications. To this direction, its power consumption is only 10 Watts, however it features 3x-12x less performance compared to the latest desktop GPUs, which consume 90-250 Watts of power. The entire SoC is supervised by an Linux Operating System (Ubuntu 16.04), while supporting CUDA 8.0.

For the scope of this thesis, the conducted experiments focus on the Harris algorithm Corner Detector application, using as input images with resolution of 512×384, 1024×1024, 2048×2048 and 4096×4096 pixels. In order to create an unbiased input dataset, random images were acquired from the world wide web. In addition, we compare the accelerated performances between CPU-GPU-SoC (Tegra x1) and CPU-FPGA-SoC (ZC702), according to the results presented in [10]. In the following Section 6.2 we present the techniques and the steps for building the GPU accelerated version of the the Harris-Corner detection on Tegra x1. In Section 6.3 we present the comparison between the results of our work and of an acceleration implementation on a CPU-FPGA-SoC [10].

6.2 Acceleration on NVIDIA Tegra x1

6.2.1 Naive implementation

In the computer science context the term naive implementation, refers to a straightforward, initial design with limited or no optimizations. For the rest of the experiments, we use the naive implementation of our target application as a baseline, without using any of the special techniques mentioned in the Chapter 4. More precisely, we merely launch a kernel with total number of threads as many as the pixels of the image, so that every thread processes a pixel, thus achieving elementary parallelism.

In the naive implementation, despite the fact that the parallelization strategy is trivial, there is still the open design choice with respect to the way that data will be exchanged between the CPU and GPU of the SoC. As presented in Section 4.2, there is a variety of different options for the implementation of this communication, and the impact of each one of these choices to the execution time of the target application is the focus of our first experiment. Figure 6.1 summarizes the Finding Corners execution time (See Section 5.3) all examined communication was for an image size of 512 × 384 pixels, based on the naive implementation.

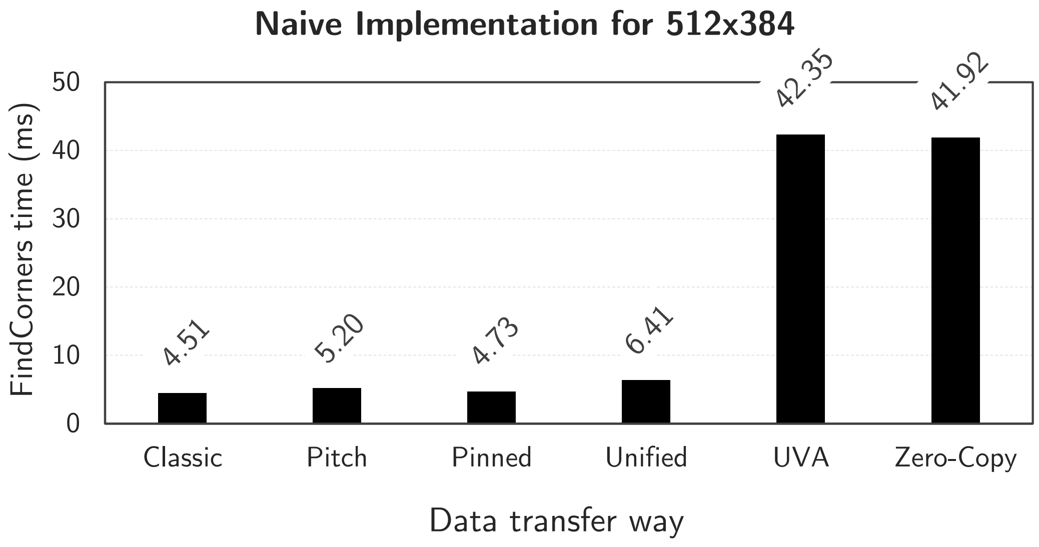


Figure 6.1: Finding Corners time for various ways of communication for 512x384 pixels image

According to the measured results, we reject the UVA and the Zero-Copy transfer ways as candidate solutions for our optimal configuration. We observe that despite the fact that the rejected ways avoid transfer latencies (see Subsections 4.2.3 and 4.2.4), the total execution latency of the application remains very high, because of the low load and store throughput. That means that data being stored in the global device memory from UVA or Zero-Copy memory is not read and written efficiently from the kernel in comparison with the other ways. However, the gain we achieved using the classic, pitch, pinned and unified ways is quite impressive, if we taking into consideration that the total time of finding the corners in a single-thread software implementation on an ARM Cortex A57 CPU is 43.96 ms (see Figure 5.3.)

The experiment is repeated for scaled image size, excluding the already rejected the UVA and the Zero-Copy memory communication alternative. The examined image resolutions are:

- 512x384 (0.196 MP)
- 1024x1024 (1.0 MP)
- 2048x2048 (4.2 MP)

- 4096x4096 (16.8 MP)

In Figure 6.2 we present the Finding Corners execution time of various ways of communications for different resolution of the input image.

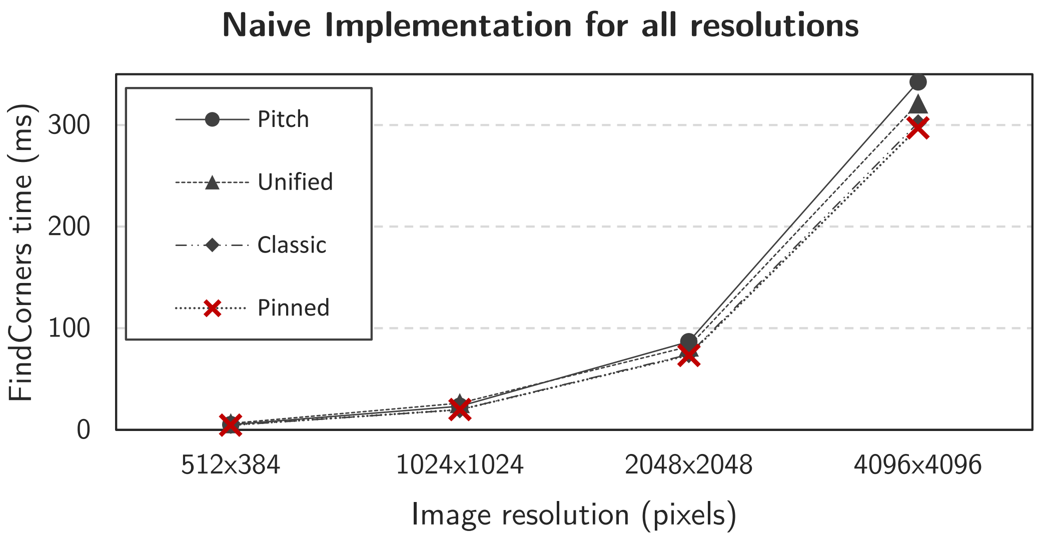


Figure 6.2: Finding Corners time for various ways of communication for various resolutions

From the Figure 6.2, it is obvious that for both small and big images the classic and the pinned ways are more efficient, achieving almost similar results. In general, although it is not clearly shown in the Figure 6.2, pinned memory communication outperforms the classic one for large image sizes and this will be thoroughly presented in the following Sections. In total, having already reached a gain of up to $\times 9.74$ for 512×384 pixels image, we are going to explore other programming techniques to gain a better performance.

6.2.2 Implementation with Shared Memory

In the effort to optimize the aforementioned naive implementation, we proceed to a version of the application where the computations are performed using shared memory. As analyzed in Section 5.3, the vast majority of the computations is dedicated to the execution of the 2D convolutions with 5×5 and the 7×7 tap kernels. Therefore, we will focus our optimization efforts only on these two kernels. As mentioned in Section 4.3, using shared memory as a managed cached memory causes lower latency, thus leading to higher application execution efficiency.

The implementation is as follows. To begin with, we load blocks of the image into arrays in shared memory, do a point-wise multiplication of a filter-size portion of the block, and then write this sum into the output image in device memory as presented in Figure 6.3. Each thread block processes a block of the image, stored in a shared memory array, accessible from all threads in a block. Each thread generates a single output pixel.

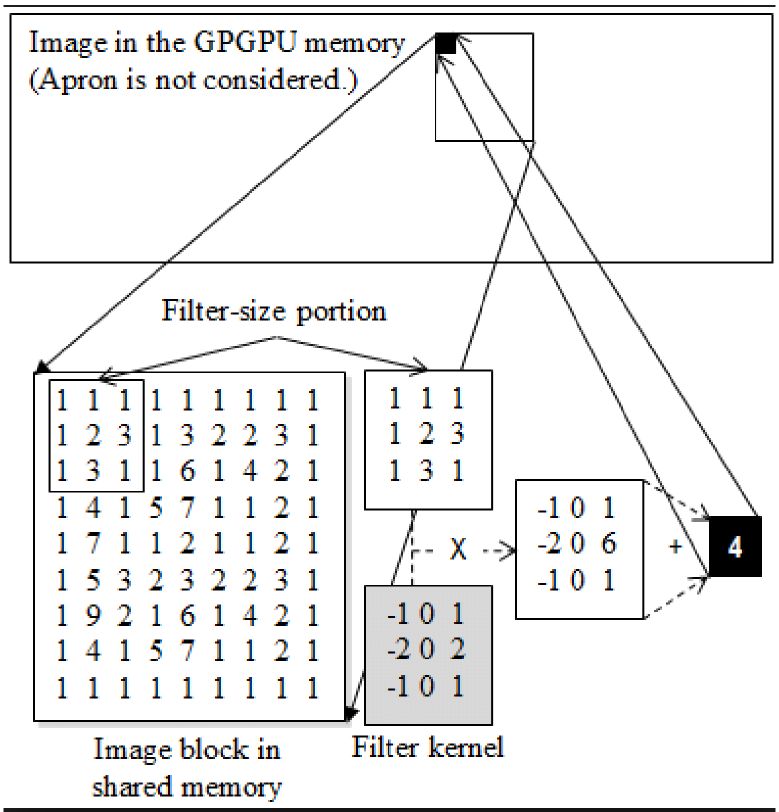


Figure 6.3: Convolution implementation between image and kernel matrix

The problem is that the the processing of pixels at the edge of the shared memory array depends on pixels non existent the in shared memory. The solution of this problem is for each thread block to load into shared memory the pixels to be filtered and the apron pixels. The apron pixels are the pixels needed at the edge of each image block. This means that if I choose a 28×28 image block, my thread block needs to contain $(28+2a) \times (28+2a)$ threads, where a is the radius of the tap kernel. For our application the image blocks will be $(28+4) \times (28+4)$ for the 5×5 and $(28+6) \times (28+6)$ for the 7×7 convolution.

The challenge here is that we need to choose image blocks that allow us to use thread blocks which contain multiple of 32 threads for the best performance as presented in the Section 4.1. With the extra restriction that the maximum number of threads in a block must be 1024, we designate in Table 6.1 that the possible dimensions of image blocks are:

Table 6.1: Possible dimensions of blocks-arrays in shared memory

5x5 tap kernel	**7x7 tap kernel**
4x4	2x2
12x12	10x10
28x28	26x26
28x12	26x10
12x28	10x26
28x4	26x2
4x28	2x26
12x4	10x2
4x12	2x10

In Figures 6.4 and 6.5 we present the execution latency of the implementation of the 5×5 and 7×7 convolution, with the respect to the possible shared block dimensions.

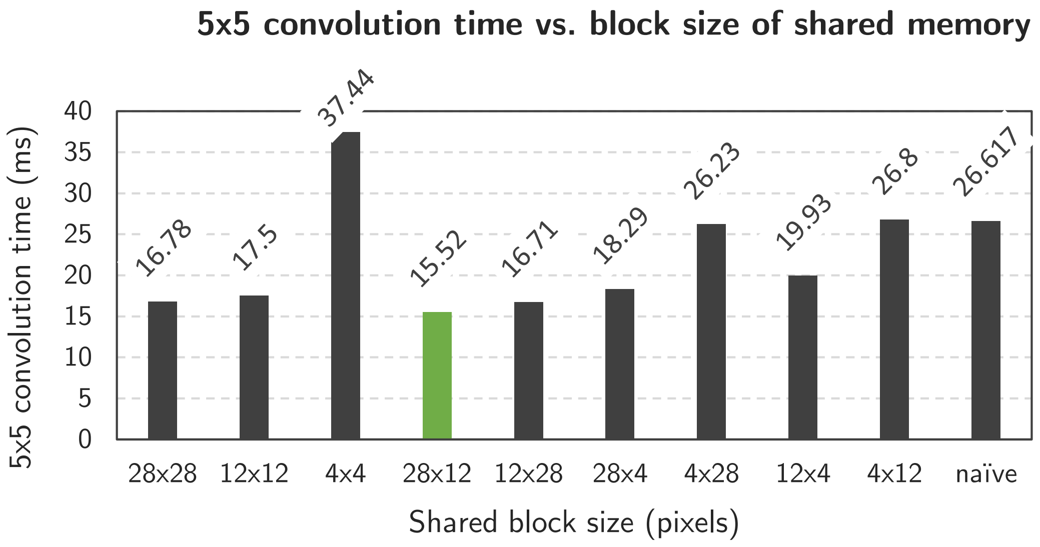


Figure 6.4: Implementation of 5 × 5 convolution between naive and shared way

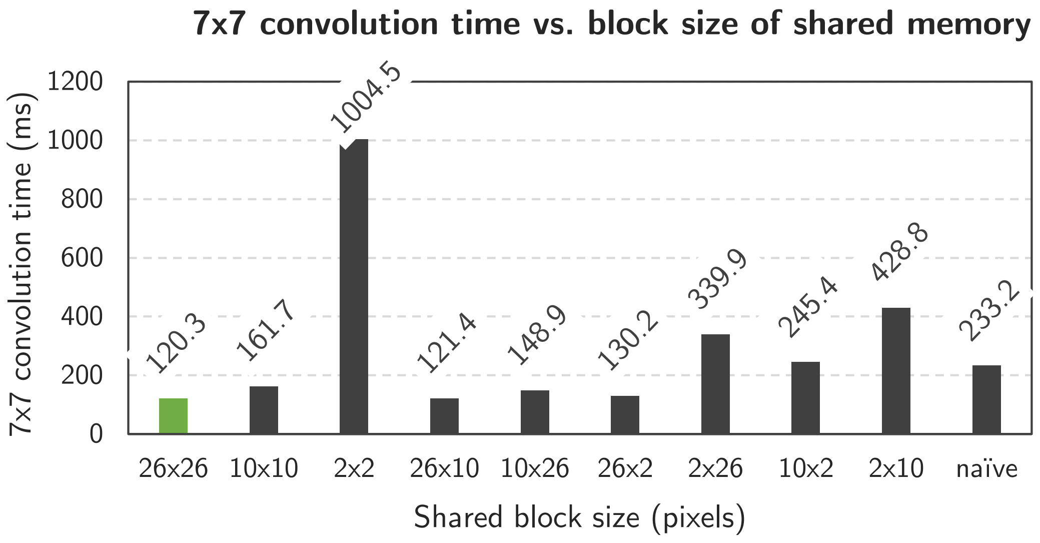


Figure 6.5: Implementation of 7 × 7 convolution between naive and shared way

From Figures 6.4 and 6.5, we conclude that the ideal image blocks are 28×12 for the 5×5 and 26×26 for the 7×7 tap kernel. This happens because the ideal organization for the compute kernels that convolve the image are $(28+4) \times (12+4) = 32 \times 16$ and $(26+6) \times (26+6) = 32 \times 32$ threads in a block for the 5×5 and 7×7 convolutions, respectively. In addition, both number of threads are multiple of 32, as it is required. An interesting observation that stems from the above Figures is the fact that the dimensions a×b result in different execution time from the dimensions b×a. More specifically, we observe smaller latencies when we have the big dimension in the horizontal dimension of the image. This is explained by contemplating the GPU architecture and the way it works. To explain it, as 2D arrays are stored in DRAM memory as 1d arrays, that means that they are stored in row and not in column. Having this in mind, a very important characteristic that we need to taking into about GPUs is the coalesced access of memory. Memory coalescing is a technique which allows optimal usage of the global memory bandwidth. That is, when parallel threads run the same instruction, they access consecutive locations in the global memory as presented in the Figure 6.6. In this way, the most favorable access pattern is achieved. Thus, by operating on an image block where the width is larger than the height, we make use of the coalescing abilities, leading to better performance. This is why block dimensions like 26×2 offer a higher efficiency than 2×26.

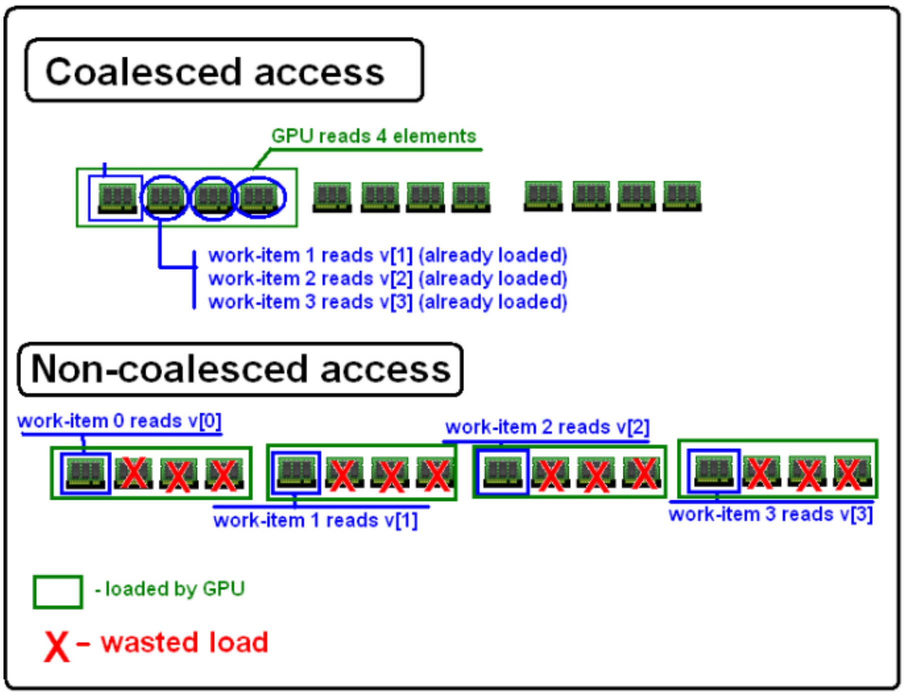


Figure 6.6: Implementation of coalesced and non coalesced memory

Figures 6.7 and 6.8 illustrate a comparison of the naive and the shared way -with ideal image blocks selected in the previous Figure- of implementation for the 5×5 and 7×7 2D convolutions:

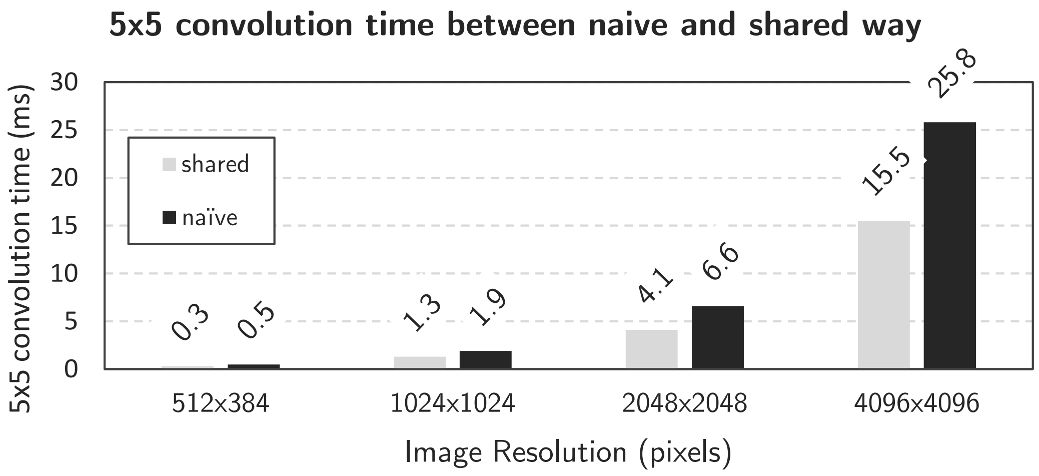


Figure 6.7: Implementation of 5×5 convolution with naive and shared memory way for various image resolutions

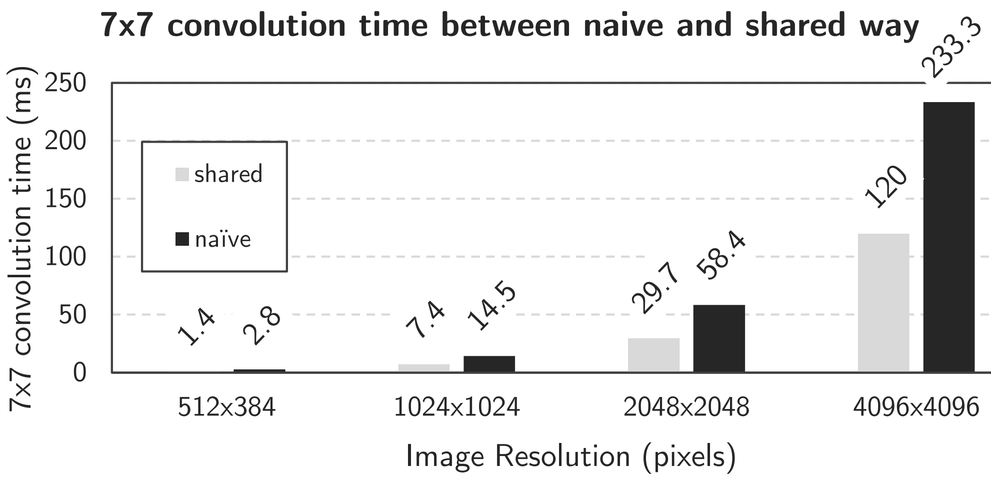


Figure 6.8: Implementation of 7×7 convolution with naive and shared memory way for various image resolutions

From Figures 6.7 and 6.8 it becomes clear, that the shared memory programming technique leads to the higher efficiency in the execution of the both 2D convolutions. This was expected since the computations of the 2D convolutions make use of the same data many times. Consequently, by storing these data in shared memory we avoid many high-latency accesses to global memory. The achieved gains for both convolution kernels are similar, however the 7×7 exhibits higher gain, since it is a bigger tap kernel and for its computation more accesses to the memory are needed.

6.2.3 Separable Implementation

Image convolutions as convolutions between 2D arrays require many accesses to the global memory, causing big latencies in our application. That is why, convolutions are the highest latency steps of the application. To achieve a better performance we have to reduce these memory accesses. As we showed in the Section 5.3, separable convolution might be an effective solution to our problem.

Let us express a convolution as $y = x * k$ where y is the output image, x is the input image, and k is the kernel. They are all 2D arrays. Let us assume that k can be calculated as $k = k1 \times k2$ where k1 is a collumn and k2 is a row. If tap kernel k could be written at this form, then we have the ability to implement the separable convolution. Instead of doing a 2D convolution with k, we could get to the same result by doing 2 1D convolutions with k1 and k2 as it has been proven in the Section 5.3. As we proved there, the theoritical computational advantage of separable convolution versus nonseparable convolution is $(P \times Q)/(P + Q)$. That means that the separability is suggested for convolutions with the biggest possible filter kernels as it helps us to achieve to reduce many global memory accesses.

Figures 6.9 and 6.10 summarize the 5×5 and the 7×7 convolution execution time, implemented in a naive and separable way:

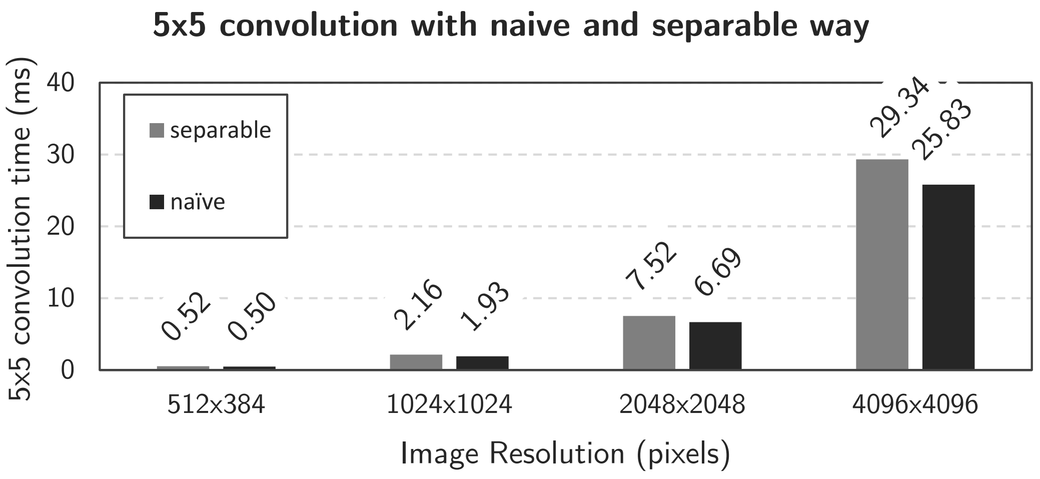


Figure 6.9: Comparison between naive and separable way for the 5 × 5 convolution

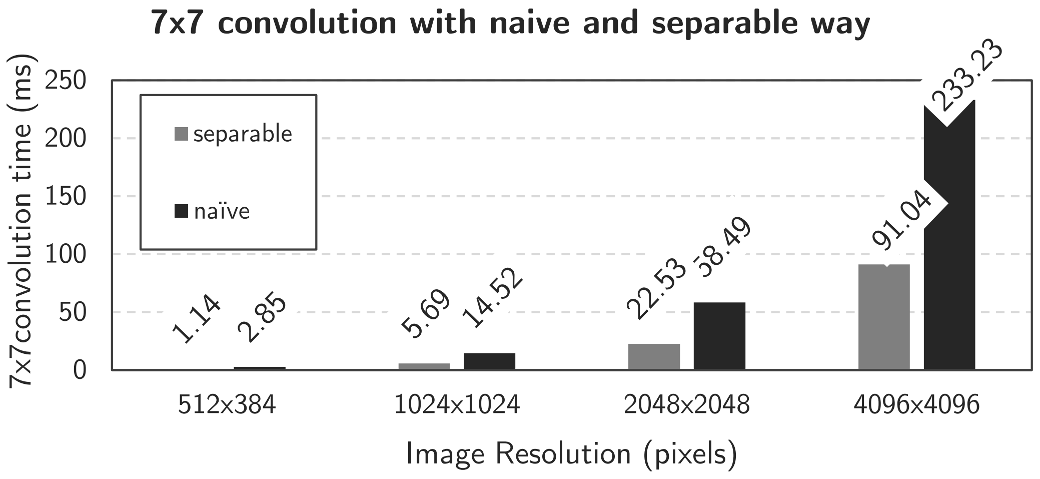


Figure 6.10: Comparison between naive and separable way for the 7 × 7 convolution

By examining Figure 6.9 we reach the conclusion, that the separable implementation of the 5×5 convolution does not provide any gains in the execution latency. Conversely, as shown in Figure 6.10, in the 7×7 convolution, the separable implementation leads to significant gains of up to $\times 2.5$ for all the examined resolutions of image with respect to the naive implemntation. These results are aligned with the theoretical expectations, since the separable implementation is not advised in the case of small tap kernels.

6.2.4 Optimum configuration

Given these results, it is up to our comparison to find the best accelerated implementation for our application. In order to achieve it, we have to combine the more efficient implementations of the 2D convolutions, and follow the naive one for the rest of the compute kernels (functions implemented in parallel on GPU). In the Figure 6.11 and Figure 6.12, we present a comparison between naive, separable and shared memory way of implementation as concern the 2d convolutions, the heaviest and the most high-latency pieces of Harris application.

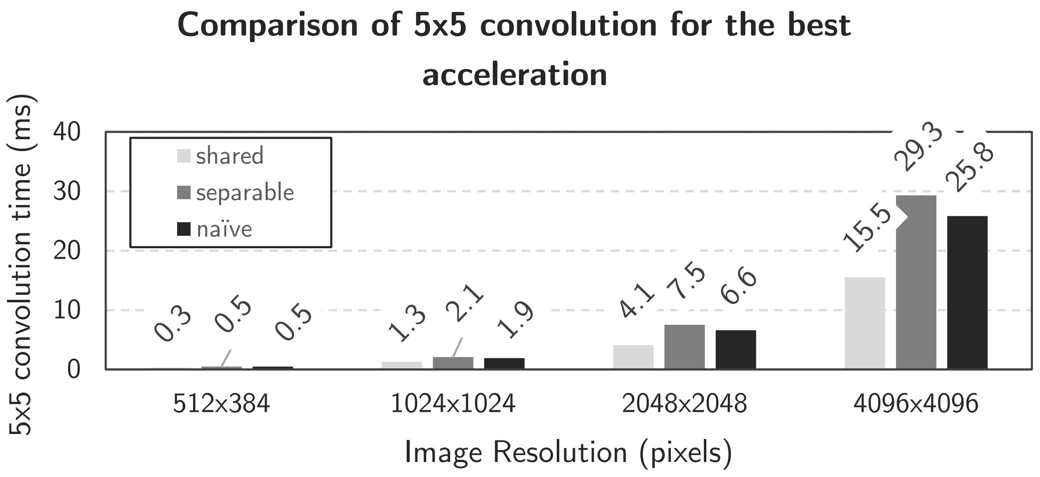


Figure 6.11: Comparison of 5×5 convolution between naive, shared and separable way

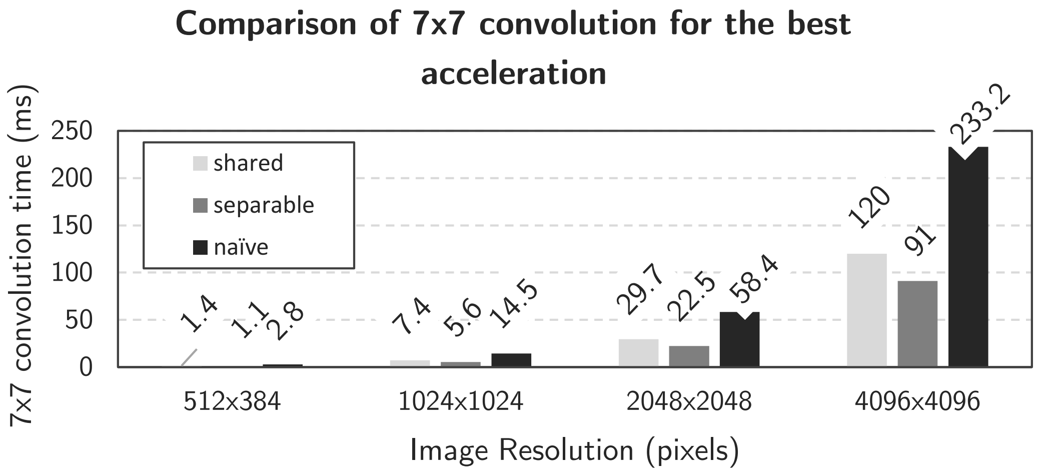


Figure 6.12: Comparison of 7×7 convolution between naive, shared and separable way

From the Figures 6.11 and 6.12 we conclude that for the 5×5 convolution is required the shared memory implementation and for the 7×7 convolution is required the separable implementation. To achieve the best performance and to end up to our optimum configuration, we have to explore the best way of CPU-GPU communication for the implementation we proposed. As we discovered in the subsection 6.2.1, the classic and the pinned ways of transfer are suggested for our application. In the Figure 6.13 we present the comparison for the latency of the complete application between classic and pinned ways for various image resolutions.

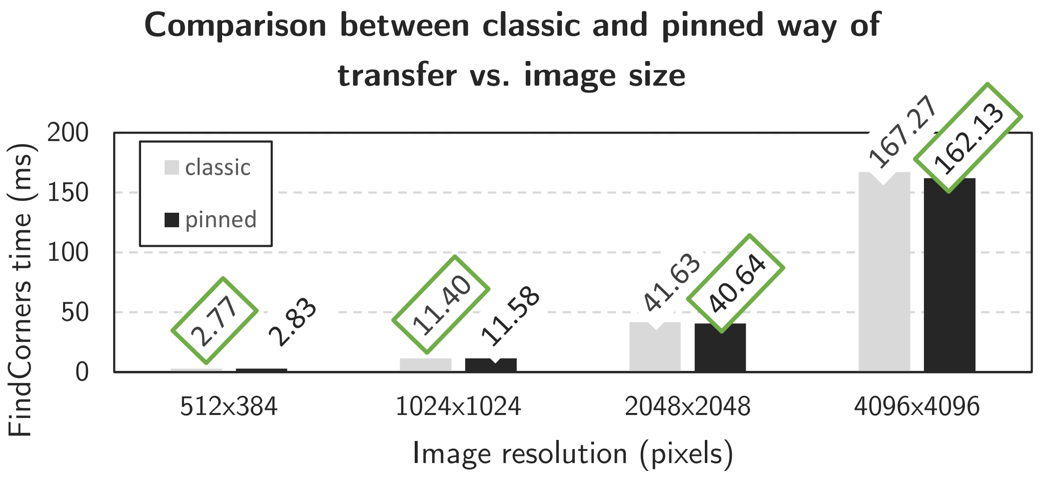


Figure 6.13: Comparison of classic and pinned way of transfer for the optimum configuration

As we can find out from the Figure 6.13, for images smaller than 2048×2048 pixels (4 MP) the classic way of transfer is recommended and for images bigger than 2048×2048 pixels (4 MP) the pinned way is recommended. Generally, pinned memory is recommended for transferring large amounts of data, between CPU and GPU, so these results are the expected ones.

In the Figure 6.14 we present the speedup gain we achieved with Tegra x1 acceleration compared with a single-thread software implementation on an ARM Cortex A57 CPU.

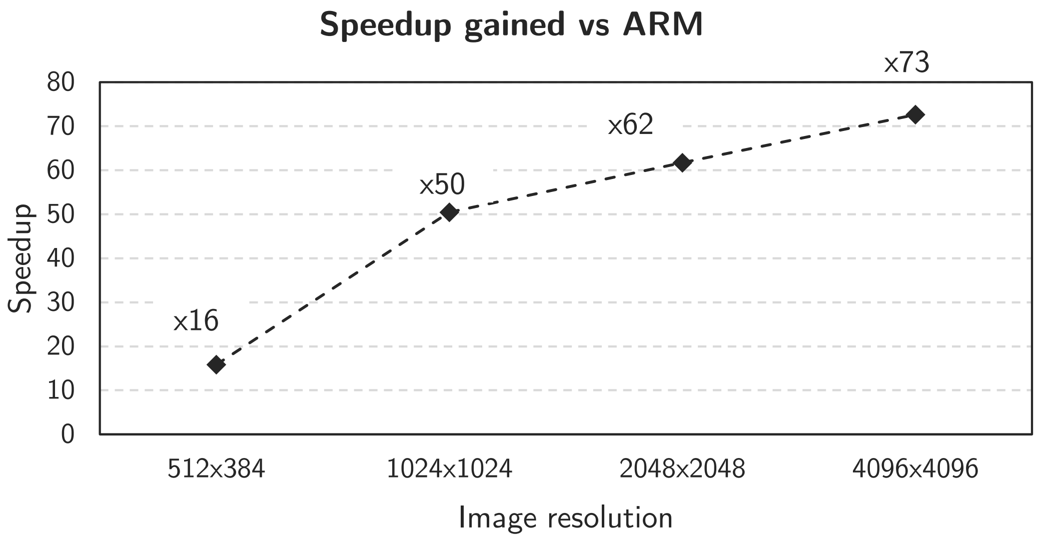


Figure 6.14: Achieved speedup gain with respect to a single-threaded software implementation on an ARM Cortex A57 CPU

6.2.5 Power and Energy Consumption

In embedded systems power plays an important role to the evaluation of the performance. Thus, in the 6.15 we present the power consumption of our optimum configuration of Harris algorithm as it is shown and described in the Subsection 6.2.4 for various resolutions of image.

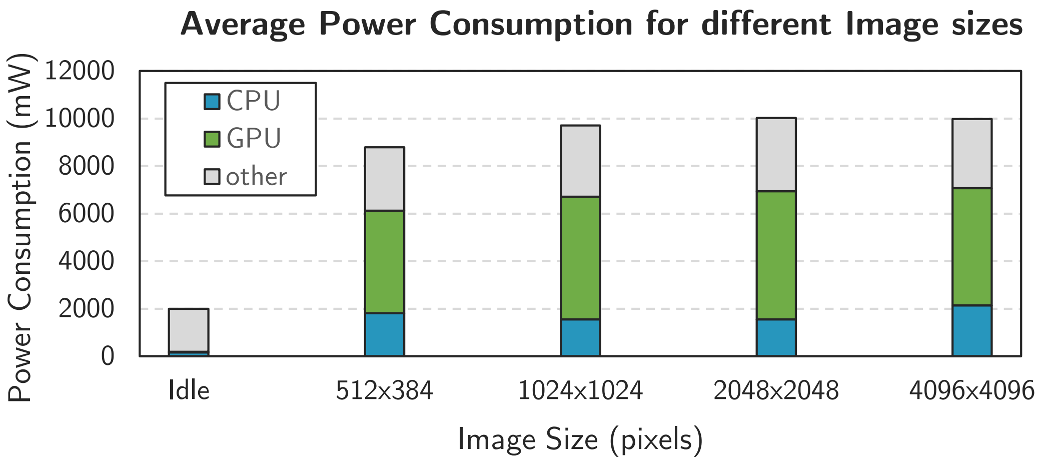


Figure 6.15: Power consumption of Tegra x1 SoC for various image resoutions

The Idle bar of this figure refers to the base power consumption of the board when no operation is executed by a user. Therefore, Tegra X1 consumes approximately 2 Watts for its basic operation, which mainly consists of the activities performed by the operating system. As far as our implementation is concerned, from this figure we can conclude that the maximum power that can be consumed is 10 Watt for the most high-quality image of our experiment, i.e. the 4096 × 4096 pixels image, which is expected

6.3 Comparison between NVIDIA Tegra x1 and ZC702

As mentioned in the Section 2.2, Ioannis Stratakos et al. [10] have implemented an FPGA-based acceleration of the Harris corner detector for 512 × 384 and 1024 × 1024 image resolutions. The accelerated kernel has been evaluated on a ZC702 board which consists of a Zynq-7000 SoC and more specifically, Zynq-7020. On the Processing System (PS) side, Zynq-7020 features a Dual-core ARM Cortex-A9 MPCore, with maximum frequency of up to 866 Mhz. On the Programmable Logic (PL) side, Zynq-7020 features 85K logic cells, 53200 Look-Up Tables (LUT), 4.9Mb of Block RAM (BRAM), 220 DSP Slices and a maximum of 200 I/O Pins. Figure 6.16 show a block diagram of the Zynq-7000 SoC family.

We compare our implementation with two different deployments on the FPGA, i.e. 1 Harris engine and 2 Harris engines. 1 Harris engine refers to an implementation where the whole input image is processed by a single engine on the PL side, whereas 2 Harris engines refers to an implementation where the input image is divided in the middle and then processed by two distinct engines. So for example, in case of a 1024x1024 image, 1 Harris engine processes the whole image. On the other hand, 2 Harris engines implementation separates the image in two images of 512x512 pixels each and then, each engine process each sub-image in parallel. Tables 6.2 and 6.3 depict the resources utilization of the FPGA for each one of the aforementioned implementations for a 1024x1024 pixels image.

Table 6.2: Resources Utilization of ZC-702 for 1 Harris engine

Resources	**Utilization**	**Available**	**Utilization (%)**
LUT	11620	53200	21.84
LUTRAM	657	17400	3.78
FF	15884	106400	14.93
BRAM	92	140	65.71
DSP	54	220	24.55

Table 6.3: Resources Utilization of ZC-702 for 2 Harris engines

Resources	**Utilization**	**Available**	**Utilization (%)**
LUT	19156	53200	36.00
LUTRAM	1016	17400	5.84
FF	27383	106400	25.74
BRAM	98.5	140	70.36
DSP	108	220	49.09

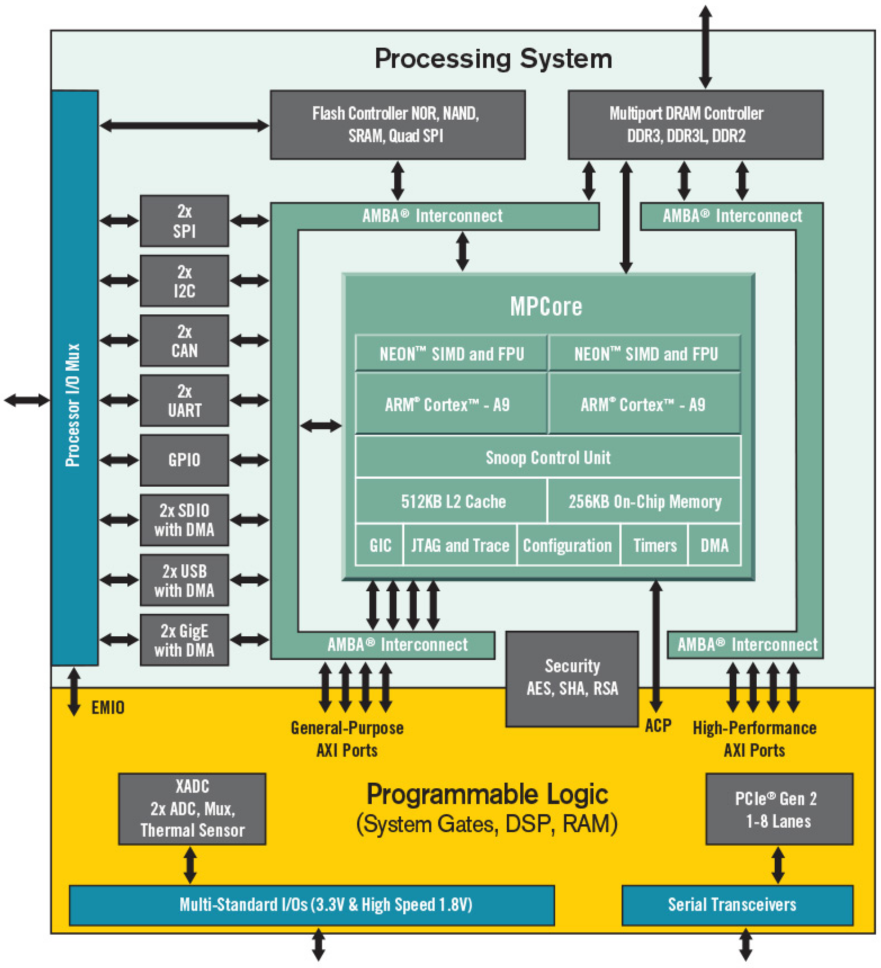


Figure 6.16: Zynq-7000 Block diagram [1]

First, we compare our implementation with the one presented in [10], using 1 Harris engine for a 512x284 image size. Figure 6.17 shows the execution time of corner detection algorithm. This measurement refers to the time for executing the corner detection algorithm and does not take into account the time needed for transferring the data from/to the FPGA/GPU. In addition, the accelerated FPGA algorithm has been designed using three different clock frequencies, i.e. 200Mhz, 250Mhz and 300Mhz. As shown in the chart, the GPU implementation achieves $x2.1$ speedup compared to the best implementation on the FPGA device.

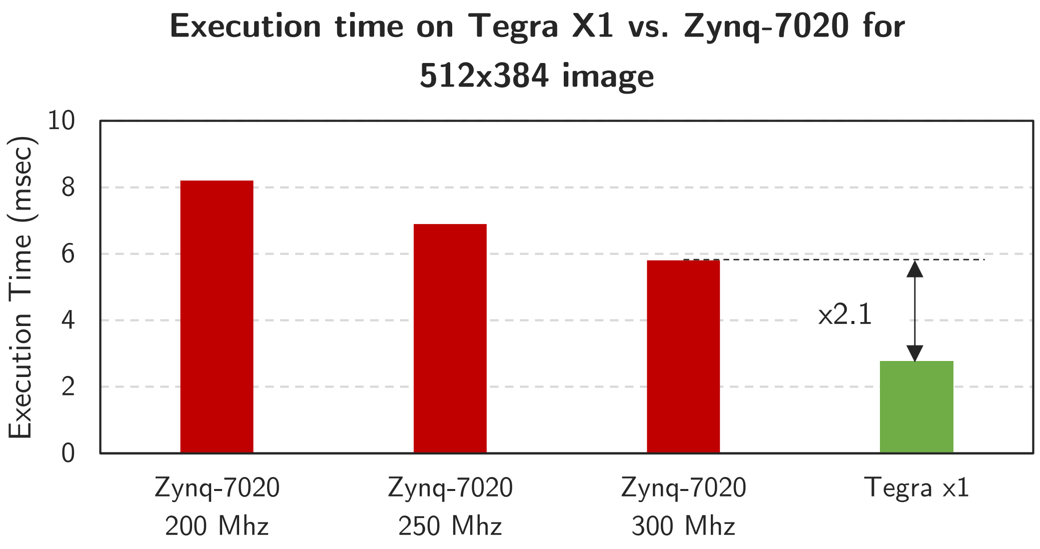


Figure 6.17: Execution time comparison between Tegra x1 and ZC702 for 512 × 384 image resolution

For the 1024x1024 image, Tegra still outperforms Zynq, but the speed-up is reduced to x1.67 compared to a 2 Harris engines implementation using a 300 Mhz clock frequency (6.19. Here, it is worth mentioning that the resources on the fpga are not 100% utilized. Therefore, a possible implementation with 3 Harris engines might reach even better performance close to the one that Tegra achieves.

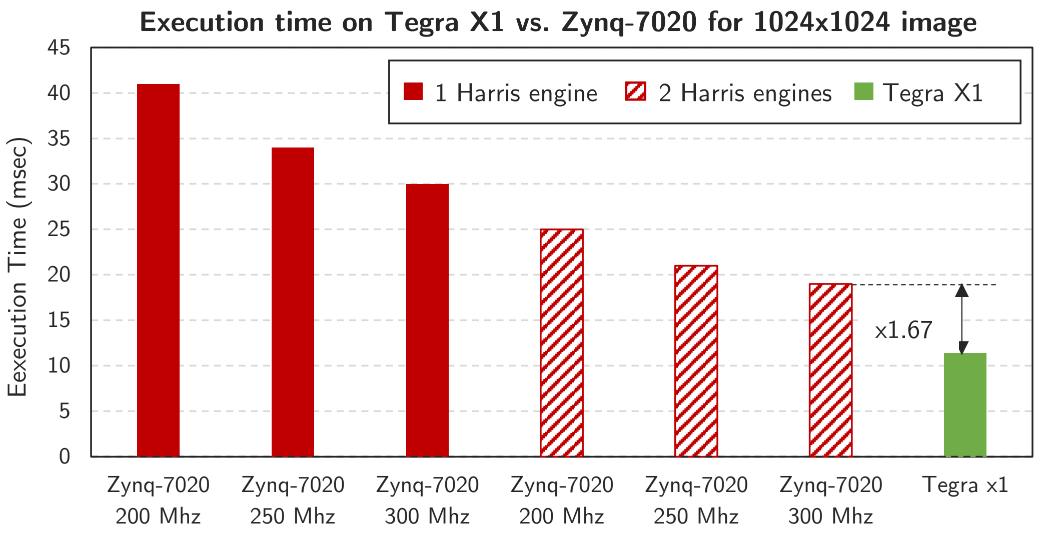


Figure 6.18: Execution time comparison between Tegra x1 and ZC702 for 1024 × 1024 image resolution

Concerning the power that the 2 SoC boards consume, in the Figure 6.19 we present the Power consumption comparison between Tegra x1 and ZC702 for 1024 × 1024 image resolution.

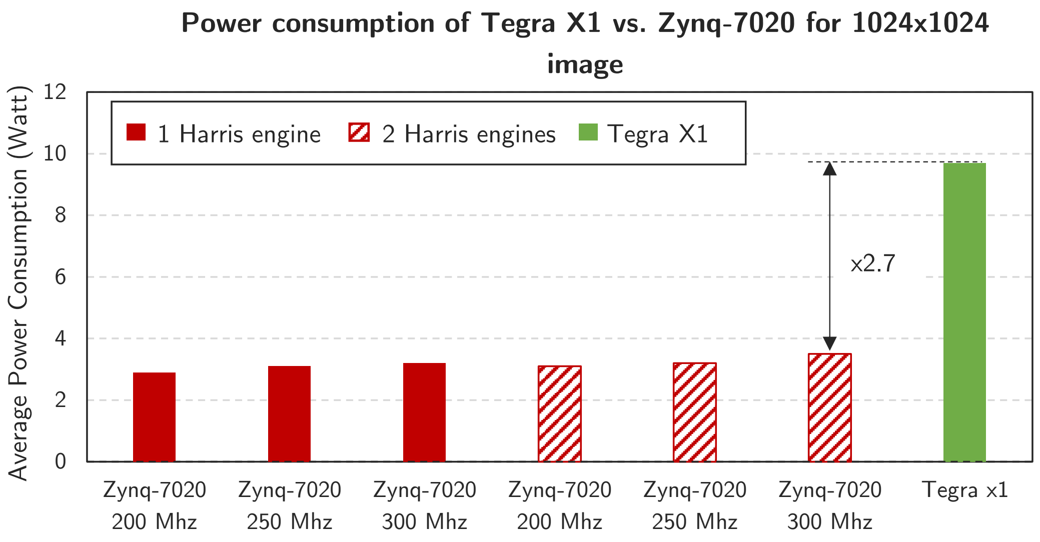


Figure 6.19: Power consumption comparison between Tegra x1 and ZC702 for 1024×1024 image resolution

We can clearly see the trade-off between the two devices, as Tegra achieves better performance whereas the Zynq device achieves better power consumption.

Chapter 7

Conclusions

7.1 Thesis summary

Image processing is a very demanding field, where both scientific community and industries try to provide new clever solutions to efficiently deploy image processing algorithms. Especially when targeting embedded systems the strict restrictions that these systems impose make the development of embedded image processing applications even more challenging.

This thesis dealt with image processing from a practical point of view. The main goal was to design and implement an image processing system and deploy it to a SoC CPU-GPU platform, specifically NVIDIA Tegra x1. The case study of our acceleration is Harris Corner detector. After explorating various programming techniques that take the advantage of the GPU architecture we achieved a final acceleration using CUDA C up to $\times 73$ compared to a pure software implementation on ARM Cortex A57 and $\times 2.1$ compared to an implementation of the same algorithm on ZC702 (Zynq 7020 FPGA). Through our exploration, we reached the conclusion that there is a trade-off between power consumption and execution time with regard to the dilemma of using GPU or FPGA as an accelerator, that we have to exploit as smarter as possible in order to create the device wich will be a panacea in the field of embedded systems.

7.2 Future Work

In applications where frames per second (fps) are not that important and we care most about keeping power consumption low, we can use other programming techniques like different data transfer ways, for instance UVA or Zero-Copy memory which consume low power but cost time. It could be a good idea to create a controller with the ability to choose the ideal programming technique depending on the consumption and time needs of the real time application or user.

In addition, taking into consideration the fact that FPGA appears lower power consumption than GPU, it could be very promising approach to enrich the aforementioned controller with the capability of cooperation GPU-FPGA in terms of an embedded SoC platform. This approach could also be enhanced with a thermal management controller, in order to avoid heat sinks on an embedded device.

Bibliography

[1] https://www.xilinx.com.

[2] A. Amaricai, C.-E. Gavriliu, and O. Boncalo. An fpga sliding window-based architecture harris corner detector. In *Field Programmable Logic and Applications (FPL), 2014 24th International Conference on*, pages 1–4. IEEE, 2014.

[3] T. L. Chao and K. H. Wong. An efficient fpga implementation of the harris corner feature detector. In *Machine Vision Applications (MVA), 2015 14th IAPR International Conference on*, pages 89–93. IEEE, 2015.

[4] J. Cheng, M. Grossman, and T. McKercher. *Professional Cuda C Programming*. John Wiley & Sons, 2014.

[5] C. Harris and M. Stephens. A combined corner and edge detector. In *Alvey vision conference*, volume 15, page 50. Citeseer, 1988.

[6] D. Hernandez-Juarez, A. Espinosa, J. C. Moure, D. Vázquez, and A. M. López. Gpu-accelerated real-time stixel computation. In *Applications of Computer Vision (WACV), 2017 IEEE Winter Conference on*, pages 1054–1062. IEEE, 2017.

[7] Y. Lee, C. Jang, and H. Kim. Accelerating a computer vision algorithm on a mobile soc using cpu-gpu co-processing-a case study on face detection. In *Mobile Software Engineering and Systems (MOBILESoft), 2016 IEEE/ACM International Conference on*, pages 70–76. IEEE, 2016.

[8] I. Lütkebohle. BWorld Robot Control Software. `https://international.download.nvidia.com/pdf/tegra/Tegra-X1-whitepaper-v1.0.pdf`, 2008. [Online; accessed 19-July-2008].

[9] H. P. Moravec. Obstacle avoidance and navigation in the real world by a seeing robot rover. Technical report, DTIC Document, 1980.

[10] I. Stratakos, D. Reisis, G. Lentaris, K. Maragos, and D. Soudris. A co-design approach for rapid prototyping of image processing on soc fpgas. In *Proceedings of the 20th Pan-Hellenic Conference on Informatics*, PCI '16, pages 57:1–57:6, New York, NY, USA, 2016. ACM.

[11] O. Ulusel, C. Picardo, C. B. Harris, S. Reda, and R. I. Bahar. Hardware acceleration of feature detection and description algorithms on low-power embedded platforms. In *Field Programmable Logic and Applications (FPL), 2016 26th International Conference on*, pages 1–9. IEEE, 2016.