Ατομική Διπλωματική Εργασία

Σύγκριση των γλωσσών προγραμματισμού των Μονάδων Επεξεργασίας Γραφικών : CUDA, OpenCL και HMPP Workbench

ΧΡΙΣΤΙΑΝΑ ΧΑΛΚΑ

ΠΑΝΕΠΙΣΤΗΜΙΟ ΚΥΠΡΟΥ

ΤΜΗΜΑ ΠΛΗΡΟΦΟΡΙΚΗΣ

ΜΑΙΟΣ 2012

ΠΑΝΕΠΙΣΤΗΜΙΟ ΚΥΠΡΟΥ

ΤΜΗΜΑ ΠΛΗΡΟΦΟΡΙΚΗΣ

Σύγκριση των γλωσσών προγραμματισμού των Μονάδων Επεξεργασίας Γραφικών : CUDA, OpenCL και HMPP Workbench

Χριστιάνα Χαλκά

Επιβλέπων Καθηγητής

Pedro Trancoso

Η Ατομική Διπλωματική Εργασία υποβλήθηκε προς μερική εκπλήρωση των απαιτήσεων απόκτησης του πτυχίου Πληροφορικής του Τμήματος Πληροφορικής του Πανεπιστημίου Κύπρου.

Μάιος 2012

Ευχαριστίες

Θα ήθελα να ευχαριστήσω ιδιαίτερα τον επιβλέποντα καθηγητή κ. Pedro Trancoso, που με επέλεξε και εμπιστεύτηκε για την εκπόνηση της διπλωματικής αυτής εργασίας. Τον ευχαριστώ ακόμη για την υπομονή αλλά και για την πολύτιμη καθοδήγηση που μου παρείχε.

Επίσης, θα ήθελα να ευχαριστήσω τους συνεργάτες και μέλη της ερευνητικής ομάδα C.A.S.P.E.R, την οποία συντονίζει ο κ. Pedro Trancoso, κ. Αντρέα Διαβαστό, κ. Κωνσταντίνο Χριστοφή και τον μεταπτυχιακό φοιτητή Γιώργο Νικολαΐδη, που με τη σειρά τους με βοήθησαν στο μέγιστο με τις ιδέες, την υπομονή αλλά και την καθοδήγησή τους.

Τέλος, ευχαριστώ όσους με την σειρά τους με στήριξαν και με βοήθησαν με το δικό τους τρόπο κατά τη διάρκεια της εκπόνησης της διπλωματικής αυτής εργασίας.

Περίληψη

Οι μονάδες επεξεργασίας γραφικών, ως γενικού σκοπού επεξεργαστές (GPGPUs) χρίζουν ραγδαίας εξέλιξης και έχουν υιοθετηθεί για τα καλά στις μέρες μας λόγω της υψηλού επιπέδου παράλληλης δομής τους, που τις καθιστά πιο αποτελεσματικές από την γενική χρήση των κεντρικών μονάδων επεξεργασίας (CPUs). Είναι εξειδικευμένες στην επίλυση αριθμητικά απαιτητικών αλγορίθμων, πέρα από την επεξεργασία γραφικών, εφόσον η επεξεργασία των μεγάλων μπλοκ δεδομένων γίνεται παράλληλα. Υπάρχουν διάφορες γλώσσες με τις οποίες μπορείς να προγραμματίσεις GPUs. Τρείς από αυτές είναι η CUDA της NVIDIA, η OpenCL της Khronos Group και η HMPP Workbench της Caps Entreprise.

Σε αυτή την διπλωματική εργασία θα εξετάσουμε και θα αξιολογήσουμε τις τρείς αυτές γλώσσες προγραμματισμού, ανάλογα με τις επιδόσεις τους, αλλά και του τρόπου προγραμματισμού σε κάθε μια από αυτές . Θα χρησιμοποιηθεί ένας αλγόριθμος, διαμορφωμένος κατάλληλα για κάθε ένα από τα πιο πάνω γλώσσες, που υλοποιεί τον πολλαπλασιασμό πινάκων, χρησιμοποιώντας παράλληλα την GPU και την CPU με την κάθε μονάδα να αναλαμβάνει το μέρος του προβλήματος στο οποίο είναι αποδοτικότερη.

Περιεχόμενα

Κεφάλαιο 1

Μονάδα επεξεργασίας γραφικών (GPU)

1.1 Εισαγωγή

Οι μονάδες επεξεργασίας γραφικών, σήμερα, μπορούν να χρησιμοποιηθούν και για γενικού σκοπού εφαρμογές πέρα από την επεξεργασία των γραφικών. Αυτή η νέα χρήση των μονάδων επεξεργασίας γραφικών, είναι εφικτή λόγω της εξέλιξης τόσο των μονάδων επεξεργασίας γραφικών όσο και των γλωσσών προγραμματισμού τους. Οι γλώσσες προγραμματισμού των μονάδων επεξεργασίας γραφικών, στην σημερινή τους μορφή είναι πολύ προσιτές προς τον προγραμματιστή. Αυτό επειδή, τους δίνουν την

δυνατότητα να χρησιμοποιούν ευρέως γνωστές γλώσσες σειριακού προγραμματισμού, με κάποιες επεκτάσεις, ως υψηλού επιπέδου γλώσσες προγραμματισμού.

Οι μονάδες επεξεργασίας γραφικών για γενικού σκοπού εφαρμογές και οι γλώσσες προγραμματισμού τους, χρίζουν ύψιστης σημασίας, όπου λόγω της υψηλού επιπέδου παράλληλης δομή στους και της υψηλής υπολογιστικής τους ισχύς, μας δίνουν την δυνατότητα να επιλύσουμε μια πληθώρα σημαντικών προβλημάτων, τα οποία παλαιοτέρα ήταν πολύ δύσκολο και χρονοβόρο να επιλυθούν. Μερικά παραδείγματα τέτοιων προβλημάτων αφορούν τους κλάδους της πυρηνικής φυσικής, της αστρονομίας καθώς επίσης και της τεχνητής ζωής και των υπολογιστικών συστημάτων μάθησης.

Εστί είναι πολύ σημαντική η μελέτη των γλωσσών προγραμματισμού των μονάδων επεξεργασίας γραφικών και των δυνατοτήτων τους. Σε αυτή την διπλωματική εργασία θα γίνει μελέτη και αξιολόγηση τριών τέτοιων γλωσσών, της CUDA, της OpenCL και της HMPP Workbench.

1.2 Σύγχρονες Μονάδες Επεξεργασίας Γραφικών (GPUs)

Μια από τις αρχιτεκτονικές που έχει πλέον συγκεντρώσει μεγάλο εμπορικό, ερευνητικό και επιστημονικό ενδιαφέρον είναι οι Μονάδες Επεξεργασίας Γραφικών (Graphics Processing Unit -- GPU) . Μια μονάδα επεξεργασίας γραφικών είναι ένας εξειδικευμένος μικροεπεξεργαστής που εμφανίστηκε σχεδόν ταυτόχρονα με τους σύγχρονους προσωπικούς υπολογιστές. Είναι σχεδιασμένη για να λαμβάνει το πολύγωνο μοντέλο μιας τρισδιάστατης εικόνας από τον κεντρικό επεξεργαστή, να το επεξεργάζεται εκτελώντας υπολογισμούς και να το μετατρέπει σε εικόνα στην οθόνη του υπολογιστή, μέσω μιας διαδικασίας αποτελούμενης από επιμέρους διακριτά βήματα, που ονομάζεται ροή πληροφοριών γραφικών(graphics pipeline), αποδεσμεύοντας έτσι την κεντρική μονάδα επεξεργασίας από τους συγκεκριμένους υπολογισμούς, λόγω της υψηλής δυνατότητας παραλληλισμού της [10].

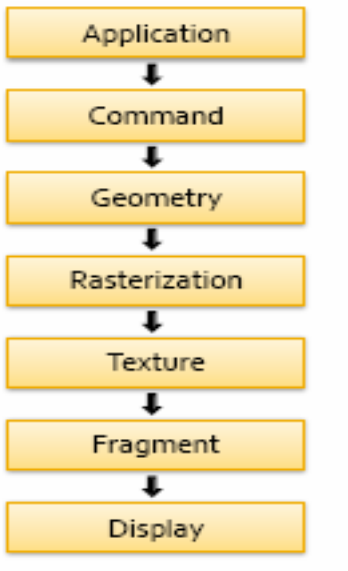
Τα τελευταία χρόνια λόγω της αύξησης της επεξεργαστικής ισχύος των μονάδων επεξεργασίας γραφικών και της παράλληλης δομής τους καθώς επίσης και της δυνατότητας προγραμματισμού τους, έχει ανοίξει ο δρόμος για την χρησιμοποίηση τους σε αριθμητικά απαιτητικούς αλγορίθμους πέρα από την επεξεργασία γραφικών[1]. Αυτή η νέα χρήση των καρτών γραφικών αποτελεί έναν πρόσφατο και ταχέως αναπτυσσόμενο κλάδο της Επιστήμης Υπολογιστών που έχει την ευρέως αποδεκτή ονομασία GPGPU(General-Purpose computations on GPUs) και θα αναλυθεί περαιτέρω σε αυτό το κεφάλαιο. Τέτοια μοντέλα καρτών με εξαιρετικές δυνατότητες έχουν εισαχθεί στην αγορά από τις δύο κυρίαρχες σε αυτό τον τομέα εταιρίες NVIDIA και ATI [2]. Σήμερα, υπάρχουν ενσωματωμένα GPUs στους επεξεργαστές των προσωπικών υπολογιστών, όπως τον i5 της Intel, καθώς επίσης και στην μητρικές κάρτες της Intel. Επίσης GPUs χρησιμοποιούνται και σε ένα μεγάλο αριθμό υπέρ-υπολογιστών της λίστας Top500, όπως ο Titan και US Jaguar[3] .Εκτός από τους προσωπικούς υπολογιστές και τους υπέρ-υπολογιστές, οι μονάδες επεξεργασίας γραφικών, χρησιμοποιούνται στα κινητά τηλέφωνα και αναλαμβάνουν τον ρόλο της επεξεργασίας των γραφικών. Χρησιμοποιούνται επίσης και σε servers, οι οποίοι εκμεταλλεύονται την επεξεργαστική τους ισχύ, για την επίτευξη πολύπλοκων υπολογισμών.

1.3 Εξέλιξη των Μονάδες Επεξεργασίας Γραφικών (GPUs)

Οι μονάδες επεξεργασίας γραφικών, αρχικά σχεδιάστηκαν για την επεξεργασία των γραφικών του υπολογιστή αλλά από τότε αντιμετωπίζουν ραγδαία ανάπτυξης, ειδικά τα τελευταία χρόνια. Σε αυτή την ενότητα θα παρουσιαστούν αρχικά τα στάδια, τα οποία ακολουθεί μια μονάδα επεξεργασίας γραφικών, για την επίλυση των γραφικών του υπολογιστή. Καθώς επίσης και πως αυτά εξελίχθηκαν με την πάροδο του χρόνου, δίνοντας σήμερα την δυνατότητα στις μονάδες επεξεργασίας γραφικών, να μπορούν να επεξεργάζονται και να επιλύουν πολύπλοκα προβλήματα πέρα από την επεξεργασία γραφικών.

Από τις αρχές τις δεκαετίας του 1980 μέχρι και τα τέλη της δεκαετίας του 1990, οι μονάδες επεξεργασίας γραφικών δεν παρουσίασαν σημαντικές εξελίξεις, στον τρόπο

επεξεργασίας των γραφικών. Χρησιμοποιούσαν σταθερές συναρτήσεις, για την ολοκλήρωση των σταδίων, που αναλύονται πιο κάτω, οι οποίες δεν ήταν προγραμματίσιμες. Την ίδια εποχή απέκτησαν δημοτικότητα διάφορες σημαντικές βιβλιοθήκες διασύνδεσης προγραμματισμού εφαρμογών γραφικών (APIs). Ένα API είναι ένα τυποποιημένο επίπεδο λογισμικού, δηλαδή μια συλλογή συναρτήσεων βιβλιοθήκης, το οποίο επιτρέπει στις εφαρμογές, να χρησιμοποιούν υπηρεσίες και λειτουργίες του λογισμικού ή του υλικού. Τέτοιου είδους διασύνδεση εφαρμογών για λειτουργίες πολυμέσων είναι το DirectX της Microsoft. Το Direct3D, ανήκει στην οικογένεια του DirectX, και παρέχει συναρτήσεις διασύνδεσης προς τις μονάδες επεξεργασίας των γραφικών, για την επεξεργασία τρισδιάστατων εικόνων. Είναι συμβατό και διαθέσιμο με τα διάφορα λειτουργικά συστήματα της οικογένειας Windows της Microsoft, και χρησιμοποιείται για την επεξεργασία των γραφικών, σε εφαρμογές με μεγάλες απαιτήσεις όπως τα ηλεκτρονικά παιχνίδια και η ψηφιακή επεξεργασία εικόνας και βίντεο. Μια άλλη διασύνδεση εφαρμογών είναι η OpenGL, η οποία υποστηρίζεται από διάφορα λειτουργικά συστήματα (ανοικτό πρότυπο), για την δημιουργία δισδιάστατων και τρισδιάστατων εικόνων. Αποτελείται από περισσότερες από 250 συναρτήσεις, δημιουργίας και σχεδίασης απλών μέχρι και ιδιαίτερα σύνθετων αντικειμένων και σκηνών. Το κάθε λειτουργικό υλοποιεί αυτές τις συναρτήσεις διαφορετικά, ακολουθώντας όμως τις αρχές τις οποίες ορίζουν οι προδιαγραφές της OpenGL [4]. Στο Σχήμα 1.1, φαίνονται τα κλασικά στάδια που ακολουθούνταν στην επεξεργασία των γραφικών, από τις τότε μονάδες επεξεργασίας, και πιο κάτω θα αναφερθεί, σε συντομία ο ρόλος του κάθε σταδίου.



Σχήμα 1.1 : Στάδια επεξεργασίας των γραφικών

Κάθε στάδιο είναι υπεύθυνο για τα εξής :

Εφαρμογή (Application) : Η εφαρμογή προετοιμάζει και φορτώνει στην μονάδα επεξεργασίας γραφικών τα δεδομένα προς επεξεργασία, και μέσω της υποστηριζόμενης διασύνδεσης (OpenGL,Direct3D) εκδίδονται οι εντολές.

Εντολές (Command) : Οι ενέργειες που λαμβάνουν χώρα κατά αυτό το στάδιο, έχουν να κάνουν με την τριγονωποίηση των πολυγώνων και την προετοιμασία της ροής των δεδομένων των κορυφών (vertex data streams).

Γεωμετρία (Geometry) : Στο στάδιο αυτό γίνεται η επεξεργασία των κορυφών, πραγματοποιώντας μετασχηματισμούς προοπτικής και οπτικής και κάποιους υπολογισμούς για τον φωτισμό.

Μετατροπή εικόνας σε εικονοστοιχεία (Rasterization) : Εδώ η εικόνα προς προβολή μετατρέπεται σε εικονοστοιχεία, μετά την επεξεργασία των τριγώνων και των κορυφών, που έχουν προκύψει από προηγούμενους υπολογισμούς.

Απεικόνιση υφής (Texture) : Σε αυτό το στάδιο επιτυγχάνεται η προσθήκη ρεαλισμού και λεπτομέρειας στη εικόνα προς απεικόνιση, με την χρήση της μνήμης και κατάλληλων φίλτρων. Αποτέλεσμα του σταδίου αυτού είναι τα fragments. Τα fragments είναι όλα τα στοιχεία και τα χαρακτηριστικά, του κάθε εικονοστοιχείου.

Επεξεργασία τμημάτων(Fragment) : Σε αυτό το στάδιο προστίθενται στα fragments πληροφορίες από το περιβάλλον, όπως αντανακλάσεις, εφαρμόζονται διάφορες υφές και χρησιμοποιούνται διάφορες τεχνικές για την προσθήκη περαιτέρω ρεαλισμού στην σκηνή. Υπολογιστικά είναι το πιο απαιτητικό στάδιο.

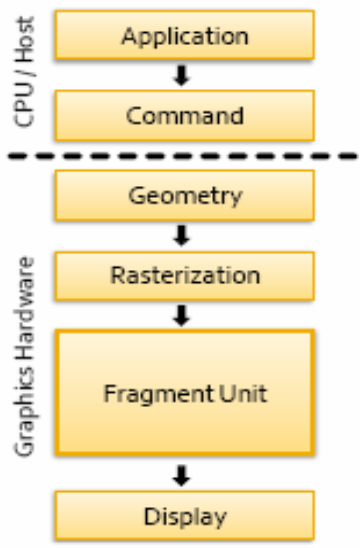
+ *Απεικόνιση (Display)* : Η Απεικόνιση είναι το τελικό στάδιο όπου όλα τα fragments μετατρέπονται σε εικονοστοιχεία και εκτελούνται κάποιες επιπλέον λειτουργίες, οι οποίες αφορούν το βάθος που θα εμφανιστούν τα διάφορα αντικείμενα. Τέλος, οι εικόνες που θα εμφανιστούν στην οθόνη γράφονται στο frame buffer.

Οι εποχές της εξέλιξης των μονάδων επεξεργασίας γαρφικών μπορούν να διακριθούν σε πέντε γενιές, όπου στην κάθε μια από αυτές παρατηρούνται διάφορες αλλαγές στα πιο πάνω στάδια, και θα μελετηθούν πιο κάτω.

CPUs πρώτης γενιάς. Το 1998 το στάδιο της απεικόνισης υφής και της επεξεργασίας των fragments έγιναν τα πρώτα προγραμματιζόμενα στάδια, χωρίς όμως να προσφέρουν πολλές δυνατότητες προγραμματισμού και ευελιξίας. Οι πρώτης γενεάς μονάδες επεξεργασίας γραφικών πρόσφεραν κάποιες συγκεκριμένες συναρτήσεις, οι οποίες μπορούσαν να μετατρέπουν προ-μετασχημετισμένα τρίγωνα σε εικονοστοιχεία, καθώς επίσης και να υποστηρίζουν την τεχνική του multi-texturing. Ενδεικτικά παραδείγματα CPUs, αυτής της γενιάς, είναι οι κάρτες TNT2 της NVIDIA, η Rage της ATI και η Voodoo της 3Dfx.

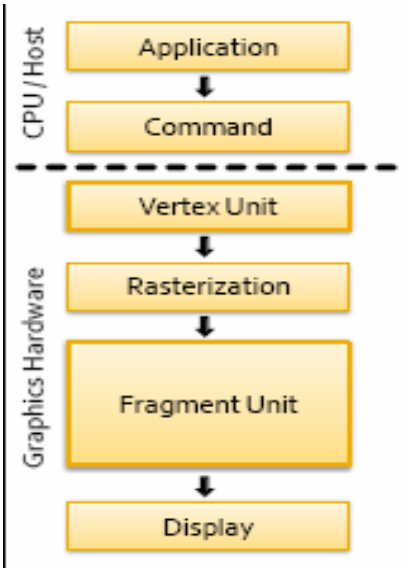
CPUs δεύτερης γενιάς. Το επόμενο βήμα έγινε γύρω στο 2000, όπου τα στάδια της απεικόνισης υφής και της επεξεργασίας των fragments, υλοποιούνταν σε μια ξεχωριστή προγραμματισζόμενη μονάδα, την Fragment Unit, όπως φαίνεται και στο Σχήμα 1.2. Οι προγραμματιστές, μπορούσαν πλέον να προγραμματίσουν την μονάδα αυτή, μέσω γλώσσας μηχανής. Έγινε επίσης εφικτή, η ανάγνωση από την μνήμη των

υφών μέσω μιας διαδικασίας που ονομάζεται αναζήτηση υφών(texture lookups). Το βήμα αυτό, έδωσε ιδιαίτερη ώθηση στην ποιότητα των γραφικών, ωστόσο δεν μπορούμε να πούμε ότι σε αυτή την γενιά οι μονάδες επεξεργασίας γραφικών ήταν αρκετά προγραμματίσιμες. Ενδεικτικά παραδείγματα CPUs, αυτής της γενιάς, είναι οι κάρτες GeForce256 και GeForce2 της NVIDIA, η Radeon 7500 της AMD και η Savage3D της S3 [5].



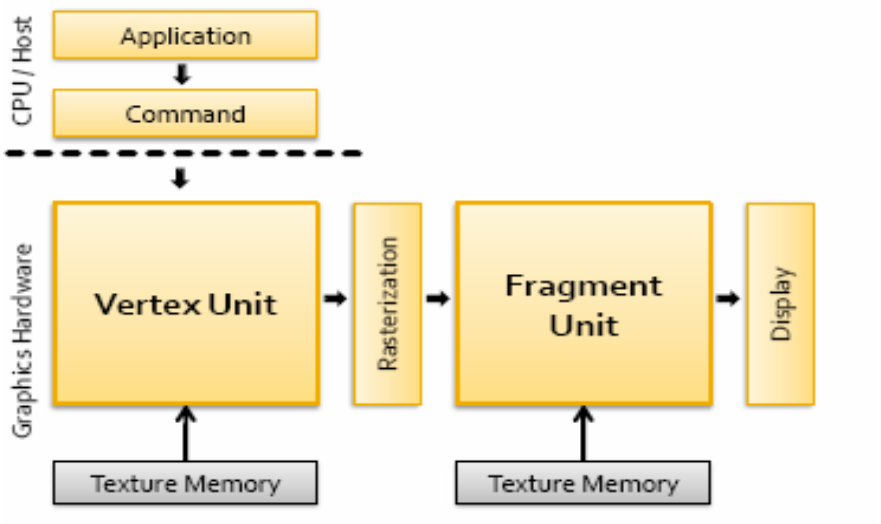
Σχήμα 1.2 : Στάδια επεξεργασίας των γραφικών. CPUs δεύτερης γενιάς

CPUs τρίτης γενιάς. Το 2001 οι προγραμματιστές είχαν την δυνατότητα προγραμματισμού του σταδίου Γεωμετρίας , μέσω γλώσσας μηχανής, αλλά δεν είχαν την δυνατότητα πρόσβασης στην μνήμη. Το μέγεθος των προγραμμάτων ήταν περιορισμένο, μέχρι 128 εντολές και δεν επιτρέπονταν οι διακλαδώσεις και οι βρόχοι. Το στάδιο της Γεωμετρίας το ανέλαβε η μονάδα κόμβων (Vertex Unit), όπως φαίνεται στο Σχήμα 1.3. Κατά το στάδιο αυτό άρχισαν να οι πρώτες σημαντικές προσπάθειες χρήσης των μονάδων επεξεργασίας γραφικών για γενικού σκοπού εφαρμογές. Αντιπροσωπευτικά μοντέλα CPUs αυτής της γενεάς είναι η GeForce3 και GeForce4 Ti της NVIDIA και Radeon 8500 της AMD [6].



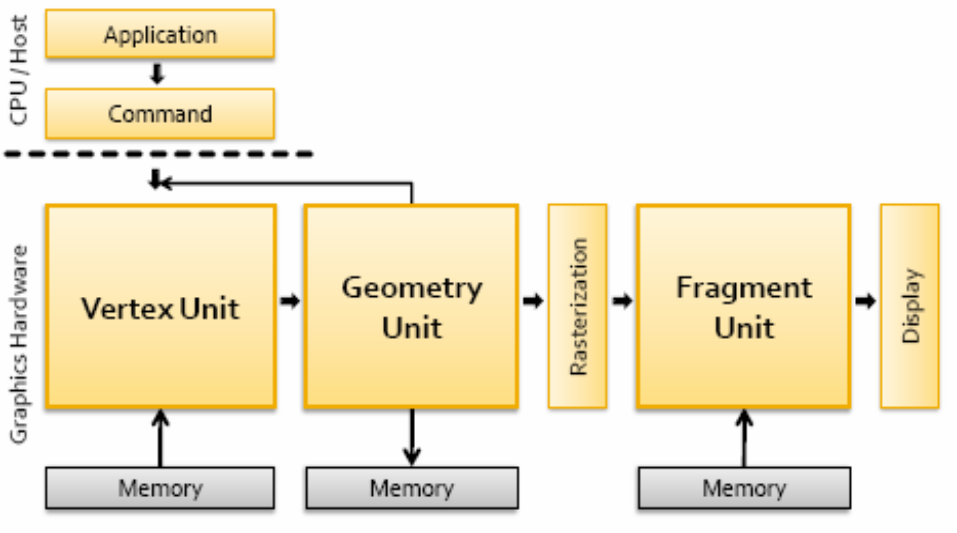
Σχήμα 1.3 : Στάδια επεξεργασίας των γραφικών. CPUs τρίτης γενιάς

CPUs τέταρτης γενιάς. Το 2003 έγιναν κάποιες σημαντικές αλλαγές. Πλέον, η μονάδα κόμβων μπορούσε να προβεί σε αναγνώσεις μνήμης, το μέγιστο μέγεθος προγραμματισμού αυξήθηκε, έγιναν εφικτές οι διακλαδώσεις, ενώ εμφανίστηκαν και γλώσσες υψηλού επιπέδου για προγραμματισμό των μονάδων επεξεργασίας γραφικών, όπως η HLSL – High Level Shading Language. Βέβαια η μονάδα κόμβων και η μονάδα fragment δεν μπορούσαν να γράψουν στην μνήμη, παρά μόνο στο frame buffer και δεν υποστήριζαν πράξεις μεταξύ ακεραίων αριθμών και πράξεις τελεστών μεταξύ των bit. Οι αλλαγές αυτές στα στάδια, φαίνονται στο Σχήμα 1.4. Τα σημαντικότερα μοντέλα CPUs αυτής της γενεάς είναι η GeForce FX και GeForce 6800 της NVIDIA και οι σειρές 9700, 9800 και X800 της AMD [5]. Η πρώτη GeForce 6 περιείχε 146 εκατομμύρια τρανζίστορ .



Σχήμα 1.4 : Στάδια επεξεργασίας των γραφικών. CPUs τέταρτης γενιάς

CPUs πέμπτης γενιάς. Το 2007 ήταν η εμφάνιση της τελευταίας γενιάς καρτών γραφικών. Είναι η πιο σύγχρονη γενιά γραφικών η οποία άλλαξε άρδην το σκηνικό στον προγραμματισμό των μονάδων επεξεργασίας γραφικών. Σε αυτή την γενιά δημιουργήθηκε η μονάδα Γεωμετρίας (Geometry Unit), όπως φαίνεται στο Σχήμα 1.5, η οποία μπορεί να εκτελέσει πρωτογενή γεωμετρικά στοιχεία και να γράφει στην μνήμη, κάτι που δεν είχε ξαναγίνει την ιστορία των μονάδων επεξεργασίας γραφικών. Ακόμη υπήρχαν και οι ενοποιημένες μονάδες επεξεργασίας [7] . Ο σημαντικότερος εκπρόσωπος αυτής της γενιάς είναι η όγδοη σειρά καρτών γραφικών της NVIDIA. Επίσης, αυτήν την περίοδο αναπτυχθήκαν και γλώσσες προγραμματισμού των μονάδων επεξεργασίας γραφικών, οι οποίες έδιναν την δυνατότητα στους προγραμματιστές να χρησιμοποιούν την γλώσσα προγραμματισμού C, σαν υψηλού επιπέδου γλώσσα προγραμματισμού, αντί την γλώσσα μηχανής. Τέτοιες γλώσσες προγραμματισμού είναι η CUDA και η OpenCL και η πιο πρόσφατη HMPP. Οι γλώσσες αυτές έκανα την αρχή για την χρήση των μονάδων επεξεργασίας γραφικών, για γενικού σκοπού εφαρμογές, GPGPUs, ένας κλάδος της Πληροφορικής που αναπτύσσεται ραγδαία τα τελευταία χρόνια [12].



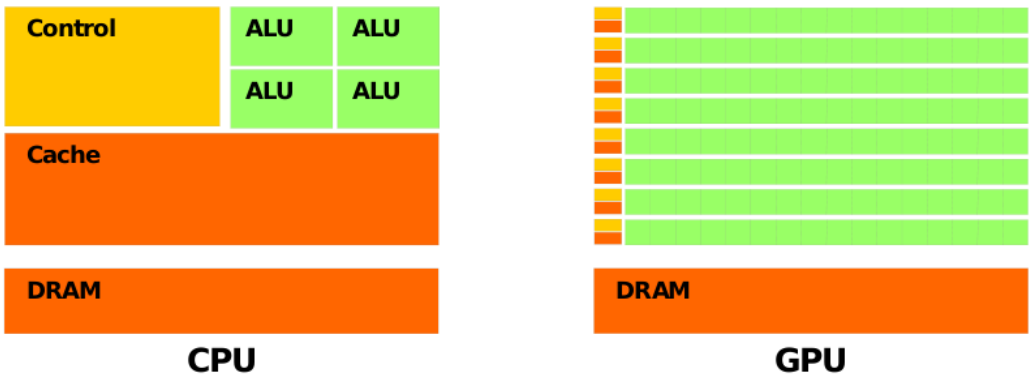
Σχήμα 1.5 : Στάδια επεξεργασίας των γραφικών. CPUs πέμπτης γενιάς

1.4 Γενικού Σκοπού Μονάδες Επεξεργασίας Γραφικών (GPGPUs)

Χρησιμοποιώντας τις πολυπύρηνες σχεδιάσεις και την δυνατότητα προγραμματισμού των rendering pipelines στις κάρτες γραφικών, οι μονάδες επεξεργασίας γραφικών (GPUs) έγιναν κατάλληλες για την επίλυση προβλημάτων πέρα από την επεξεργασία γραφικών, αναλαμβάνοντας ουσιαστικά τον ρόλο ενός παράλληλου επεξεργαστή, κατάλληλου για αλγορίθμους υψηλής παραλληλοποίησης και αριθμητικού φόρτου [12].

Σε αυτές τις σχεδιάσεις οι GPGPUs μπορούν να χρησιμοποιηθούν παράλληλα με τον κεντρικό επεξεργαστή, αναλαμβάνοντας το υπολογιστικό φόρτο για τέτοιου είδους αλγορίθμους που δεν είναι κατάλληλοι για την σειριακή λογική των μακρών pipelines των κεντρικών επεξεργαστών. Οι GPUs αποτελούνται από εκατοντάδες πυρήνες, οι οποίοι μπορούν να εκτελέσουν χιλιάδες νήματα παράλληλα. Κάθε ένας από αυτούς τους πυρήνες αποτελείται από αριθμητική και λογική μονάδα, που είναι υπεύθυνη για της αριθμητικές πράξεις, όπως πρόσθεση , αφαίρεση, πολλαπλασιασμό και διαίρεση. Την μονάδα κινητής υποδιαστολής , την μονάδα μετακίνησης και σύγκρισης καθώς

επίσης και από μια μονάδα branch [14]. Οι GPUs είναι σχεδιασμένες έτσι ώστε περισσότερα τρανζίστορ να αφιερώνονται στην επεξεργασία των δεδομένων και όχι στα δεδομένα προσωρινής αποθήκευσης και του έλεγχου ροής τους, όπως συμβαίνει στις CPUs. Οι δύο αρχιτεκτονικές φαίνονται στο Σχήμα 1.6 .



Σχήμα1.6 : Αρχιτεκτονική της CPU και GPU [13]

Λόγω αυτού του μεγάλου αριθμού πυρήνων, το overhead της δημιουργίας των νημάτων είναι σχεδόν μηδαμινό, και επιτυγχάνεται πιο μεγάλος βαθμός παραλληλισμού, σε αντίθεση με την CPU που αποτελείται από μικρότερο αριθμό πυρήνων. Στον πίνακα 1.1 παρουσιάζονται μερικά μοντέλα μονάδων επεξεργασίας γραφικών, και ο αριθμός των πυρήνων από τους οποίους αποτελούνται.

Κατασκευαστής	Μοντέλο	# πυρήνων
NVIDIA	GeForce 8800 GTX	128
AMD	Radeon 4670	320
NVIDIA	GeForce 670 GTX	1344
AMD	Radeon HD 7870	1280

Πίνακας 1.1 : Αριθμός πυρήνων διαφόρων μοντέλων μονάδων επεξεργασίας γραφικών [8]

Ένα ειδικά γραμμένο υποπρόγραμμα, ο kernel, εκτελείται ταυτόχρονα και ανεξάρτητα από όλους τους πυρήνες της GPU, με τον καθένα να επεξεργάζεται ένα συγκεκριμένο σετ από δεδομένα. Τα δεδομένα μεταφέρονται αρχικά από την κύρια μνήμη του υπολογιστή στην μνήμη της κάρτας γραφικών, επεξεργάζονται από την GPU και μεταφέρονται πάλι πίσω στην κύρια μνήμη. Αυτές οι αντιγραφές υποδηλώνουν την μεταφορά και επεξεργασία μεγάλων κομματιών μνήμης, αλλά κρύβονται τα επικοινωνιακά κόστη από την κατά πολύ ταχύτερη επεξεργασία στην GPU, καθώς επίσης και από διάφορες τεχνικές, όπως το double buffering.

Για την επίτευξη της μέγιστης δυνατής παραλληλοποίησης προσοχή πρέπει να δίνεται και στην κατάλληλη αποθήκευση των δεδομένων ώστε κάθε πυρήνας να επεξεργάζεται ένα ανεξάρτητο κομμάτι δεδομένων, αποθηκευμένο σε διαδοχικές θέσεις μνήμης για ταχύτερη προσπέλαση. Οι επιδόσεις από την χρήση των GPGPUs για αλγορίθμους που προσφέρονται για παράλληλη επεξεργασία, με κατάλληλα, σύμφωνα με τα πιο πάνω, προγράμματα είναι πολλές τάξεις μεγέθους πάνω από τις επιδόσεις της CPU σε τέτοια προβλήματα, προσφέροντας επιδόσεις που μέχρι πρότινος ήταν επιτεύξιμες μόνο με την χρήση ειδικών ακριβών αρχιτεκτονικών σε υπέρ-υπολογιστικά συστήματα [9].

Η πρώτη GPU, η οποία έδωσε τα έναυσμα για την τεράστια αυτή εξέλιξη των μονάδων επεξεργασίας γραφικών, για γενικού σκοπού εφαρμογές, είναι η DirectX 9 GPU. Ερευνητές οι οποίοι την μελετούσαν, παρατήρησαν την πορεία αύξησης των επιδόσεων που είχε, και άρχισαν να διερευνούν τη χρήση των GPUs και για επίλυση αλγορίθμων με μεγάλη υπολογιστική ένταση. Ωστόσο η DirectX 9 GPU, είχε σχεδιαστεί μόνο για να ταιριάζει τα χαρακτηριστικά που απαιτούνταν από τα APIs των γραφικών. Για να ήταν σε θέση κάποιους προγραμματιστής, να αποκτήσει πρόσβαση στους υπολογιστικούς της πόρους, έπρεπε να ανάγει τις εντολές του προβλήματος σε λειτουργίες των γραφικών, έτσι ώστε ο υπολογισμός τους να μπορούσε να δρομολογηθεί μέσω των OpenGL ή DirectX APIs κλήσεων, κάτι το οποίο ήταν πάρα πολύ δύσκολο. Για το λόγω αυτό γεννήθηκε η ανάγκη για την δημιουργία νέων προγραμματιστικών μοντέλων, τα οποία θα έκαναν τον προγραμματισμό των μονάδων

επεξεργασίας γραφικών, για γενικού σκοπού εφαρμογές, πιο εύκολο. Η αρχή έγινε από ένα ερευνητικό πρόγραμμα του πανεπιστημίου Stanford , όπου παρουσίασαν την γλώσσα προγραμματισμού BrookGPU [11], που είχε σαν στόχο να επιτρέπει στους προγραμματιστές που δεν έχουν άμεση σχέση με τα γραφικά, να μπορέσουν να γράψουν αποδοτικές εφαρμογές για GPUs, χρησιμοποιώντας την γλώσσα προγραμματισμού C με κάποιες επεκτάσεις. Αυτό το σκεπτικό ακολούθησε και η NVIDIA, όπου το 2006, με την παρουσίαση της CUDA, και του παράλληλου προγραμματιστικού της μοντέλο, έδωσε την ευχέρεια στους προγραμματιστές, να χρησιμοποιούν τις γλώσσες C και FORTRAN, σαν υψηλού επιδέσου γλώσσες προγραμματισμού των GPGPUs. Την NVIDIA ακολούθησαν και άλλες εταιρίες, όπως η Khrono's Group με την OpenCL και η Caps Enterprise με την HMPP. Με τις διάφορες επεκτάσεις που εισάγονται στα διάφορα αυτά μοντέλα προγραμματισμού επιτρέπουν σήμερα στους προγραμματιστές, να μπορούν να χειριστούν τον παραλληλισμό και τον προγραμματισμό των GPU, με αφηρημένες έννοιες.

1.5 Οργάνωση της Διπλωματικής Εργασίας

Στο Κεφάλαιο 2 παρουσιάζονται άλλες σχετικές εργασίες, οι οποίες κάνουν σύγκριση των γλωσσών προγραμματισμού των μονάδων επεξεργασίας γραφικών, CUDA και OpenCL , καθώς επίσης και σχολιασμός των αποτελεσμάτων τους.

Στη συνέχεια στο Κεφάλαιο 3 παρουσιάζονται τα βασικά χαρακτηριστικά της CUDA. Αρχικά γίνεται μια εισαγωγική αναφορά στην CUDA και στο προγραμματιστικό της περιβάλλον καθώς στο ότι υποστηρίζει ετερογενή προγράμματα στα οποία υπάρχει συν-επεξεργασία της CPU και της GPU. Στη συνέχεια παρουσιάζεται το προγραμματιστικό μοντέλο της CUDA με αναφορά στην συνάρτηση kernel, στην ιεραρχία των νημάτων και της μνήμης καθώς επίσης και της αρχιτεκτονικής της.

Ακολούθως στο Κεφάλαιο 4 γίνεται αναφορά στα βασικά χαρακτηριστικά της γλώσσα προγραμματισμού OpenCL. Αρχικά γίνεται μια εισαγωγή στην OpenCL και στη συνέχεια παρουσιάζεται το προγραμματιστικό της μοντέλο, με έμφαση στην συνάρτηση

kernel και στην ιεραρχία των νημάτων και της μνήμης. Και τέλος γίνεται αναφορά στο προγραμματιστικό μοντέλο της OpenCL από την πλευρά ενός CUDA προγραμματιστή.

Στο Κεφάλαιο 5 παρουσιάζεται η τελευταία γλώσσα προγραμματισμού των μονάδων επεξεργασίας γραφικών, η οποία μελετήθηκε, και είναι η HMPP Workbench. Αρχικά γίνεται μια εισαγωγή στην HMPP Workbench και ακολούθως παρουσιάζεται το προγραμματιστικό της μοντέλο και τα βασικά στοιχεία του codelet και callsite.

Στο Κεφάλαιο 6 αρχικά αναλύεται ο τρόπος με τον οποίο υλοποιείται το κλασικό πρόβλημα του πολλαπλασιασμού πινάκων στην CUDA, στην OpenCL και στην HMPP. Ακολούθως γίνεται αξιολόγηση των τριών αυτών γλωσσών και παρουσιάζονται τα αποτελέσματα.

Τέλος στο Κεφάλαιο 7 παρουσιάζονται τα συμπεράσματα καθώς επίσης και προτάσεις για μελλοντική εργασία.

Κεφάλαιο 2

Σχετικές Εργασίες

2.1 Σχετικές Εργασίες

Πολλοί ερευνητές τα τελευταία χρόνια έχουν ασχοληθεί με την μελέτη και την ανάλυση των δυνατοτήτων, των γλωσσών CUDA και OpenCL, οι οποίες χρησιμοποιούνται για τον προγραμματισμό των μονάδων επεξεργασίας γραφικών. Γενικότερα έχουν γίνει πάρα πολλές μελέτες για την αξιολόγηση των επιδόσεων τους. Σε αυτές τις μελέτες χρησιμοποιούνται διαφορετικοί αλγόριθμοι οι οποίοι υλοποιούνται σε CUDA και OpenCL, και στη συνέχεια γίνεται αξιολόγηση των επιδόσεων τους. Για αυτή την εργασία έχουν μελετηθεί διάφορα άρθρα για την σύγκριση των δύο πιο πάνω γλωσσών. Πιο κάτω θα αναφερθούν τα άρθρα τα οποία μελετήθηκαν και θα σχολιαστούν τα συμπεράσματα τους .

Ο Przemyslaw Plaszewski και οι συνεργάτες του [23] έκαναν μια μελέτη για την αξιολόγηση των επιδόσεων της CUDA και της OpenCL. Υλοποίησαν τον αλγόριθμο της μεθόδου των πεπερασμένων στοιχείων (FEM) σε CUDA και OpenCL για να πάρουν τα αποτελέσματα. Η FEM είναι μία τεχνική για την εξεύρεση , κατά προσέγγιση λύσεων, των μερικών διαφορικών εξισώσεων και των ολοκληρωμάτων. Η GPU που χρησιμοποιήθηκε ήταν η NVIDIA GTX 285. Τα αποτελέσματα τους έδειξαν ότι η CUDA έχει καλύτερες επιδόσεις από την OpenCL.

Επίσης, ο Peng Du και οι συνεργάτες του [24] έκανα μια έρευνα για την σύγκριση της CUDA και της OpenCL ανάλογα με τις επιδόσεις τους αλλά και του προγραμματιστικού τους μοντέλου. Για τα πειραματικά τους αποτελέσματα χρησιμοποίησαν τον αλγόριθμο πολλαπλασιασμού πινάκων DGEMM, για πραγματικούς αριθμούς με ακρίβεια δύο δεκαδικών ψηφίων, με πίνακες μεγέθους 4096 x 4096 και 10240 x 10240. Και στις δύο περιπτώσεις η CUDA είχε καλύτερες επιδόσεις από την OpenCL.

Επίσης σε μία άλλη έρευνα [25] που έγινε γίνεται η χρήση ενός CUDA και ενός OpenCL προγράμματος, τα οποία υλοποιούν την δισδιάστατη και τρισδιάστατη εγγραφή εικόνας. Όπου σε όλες τις περιπτώσεις η CUDA είχε καλύτερες επιδόσεις από την OpenCL.

Ο Jianbin Fang και οι συνεργάτες του [26], έχουν κάνει μια ανάλυση των επιδόσεων της CUDA και της OpenCL. Χρησιμοποίησαν πάρα πολλές εφαρμογές για να πάρουν τα αποτελέσματα τους καθώς επίσης και διαφορετικές GPUs. Επίσης παρουσίασαν και τις ομοιότητες της CUDA με την OpenCL. Στα αποτελέσματα τους έδειξαν ότι η CUDA έχει καλύτερες επιδόσεις από την OpenCL, Επισήμαναν επίσης ότι αυτές οι διαφορές στις επιδόσεις οφείλονται στις διαφορές των προγραμματιστικών μοντέλων, στις διαφορετικές βελτιστοποιήσεις των συναρτήσεων kernel, στις διαφορές των αρχιτεκτονικών καθώς επίσης και στις διαφορές των μεταγλωττιστών. Επίσης έχουν αποδείξει ότι οι επιδόσεις των δύο αυτών γλωσσών προγραμματισμού μπορούν να εξισωθούν με συστηματικές αλλαγές στον κώδικα.

Ο Kamran Karimi και οι συνεργάτες του [27] έκανα σύγκριση των επιδόσεων της CUDA και της OpenCL, χρησιμοποιώντας πολύπλοκες και παρόμοιες συναρτήσεις kernel. Έδειξαν ότι υπάρχουν ελάχιστες τροποποιήσεις που χρειάζεται να γίνουν για την μετατροπή ενός CUDA kernel σε ένα OpenCL kernel. Τα πειράματα τα οποία χρησιμοποίησαν μετρούσαν και σύγκριναν τον χρόνο μεταφοράς των δεδομένων από και προς τις μονάδες επεξεργασίας γραφικών , τον χρόνο εκτέλεσης της συνάρτησης kernel και τον χρόνο ολοκλήρωσης ολόκληρης της εφαρμογής. Χρησιμοποιήθηκε μόνο ένας αλγόριθμος για να πάρουν τις πιο πάνω μετρήσεις. Τα αποτελέσματα τους έδηξαν

ότι η CUDA έχει καλύτερες επιδόσεις από την OpenCL. Επισήμαναν επίσης ότι η επιλογή ανάμεσα στην χρήση της CUDA ή της OpenCL ,μπορεί να παρθεί από ένα προγραμματιστή, λαμβάνοντας υπόψη και άλλους σημαντικούς παράγοντες, όπως τα διαθέσιμα GPUs καθώς επίσης και αν η εφαρμογή θέλει να τρέχει και σε άλλα GPUs.

Στο [29], οι συγγραφείς αξιολογούν ποσοτικά την επίδοση προγραμμάτων σε CUDA και OpenCL, τα οποία αναπτύχθηκαν με σχεδόν τους ίδιους υπολογισμούς. Χρησιμοποιήθηκαν εφαρμογές όπως ο πολλαπλασιασμός πινάκων από SDK της CUDA καθώς επίσης και οι εφαρμογές MRI-Q και MRI-HD από Parboil benchmark. Τα αποτελέσματα τους δείχνουν ότι οι επιδόσεις της OpenCL είναι ανάλογες με αυτές της CUDA, με την CUDA να έχει ελαφρός καλύτερες επιδόσεις.

Κεφάλαιο 3

Βασικά χαρακτηριστικά της CUDA

3.1 Εισαγωγή

Η CUDA είναι μια γενικού σκοπού παράλληλη υπολογιστική αρχιτεκτονική της NVIDIA, που επιτρέπει την δραματική αύξηση των επιδόσεων ενός υπολογιστή από την εκμετάλλευση της δύναμης της μονάδας επεξεργασίας γραφικών (GPU). Κυκλοφόρησε για πρώτη φορά το 2006, και από τότε η CUDA έχει αναπτυχθεί ευρέως μέσα από χιλιάδες εφαρμογές και δημοσιεύσεις ερευνητικών εργασιών. Η CUDA μέχρι πρότινος υποστηριζόταν μόνο από NVIDIA GPUs, τον Δεκέμβριο όμως του 2011 η CUDA έγινε ανοικτό πρότυπο και υποστηρίζεται και από GPUs άλλον εταιρειών. Το προγραμματιστικό περιβάλλον της CUDA δίνει την δυνατότητα στους προγραμματιστές να χρησιμοποιούν ευρέως γνωστές γλώσσες σειριακού προγραμματισμού, με κάποιες

επεκτάσεις ως υψηλού επιπέδου γλώσσες προγραμματισμού. Οι γλώσσες οι οποίες υποστηρίζονται από την CUDA είναι η C, η FORTRAN, η OpenCL και η DIRECTCOMPUTE [13].

Ένα πρόγραμμα σε CUDA χωρίζεται σε δύο μέρη, το σειριακό και το παράλληλο. Ο σειριακός κώδικας (Host Code), εκτελείται στην κεντρική μονάδα επεξεργασίας (CPU), και ο παράλληλος κώδικας (Device Code), εκτελείται στην μονάδα επεξεργασίας γραφικών (GPU). Για λόγους συντομίας και λιτότητας στις εκφράσεις, στη συνέχεια του κειμένου, η κεντρική μονάδα επεξεργασίας θα αναφέρεται ως **host**, καθώς την μεταχειριζόμαστε ως την μητρική συσκευή από την οποία ξεκινά η εκτέλεση του προγράμματος, και η μονάδα επεξεργασίας γραφικών ως **device**.

Το host και το device έχουν το κάθε ένα την δική του ξεχωριστή μνήμη και τα δεδομένα τα οποία θα επεξεργαστούν και θα εκτελεστούν στο device πρέπει να αντιγράφονται από το host, και στην συνέχεια το αποτέλεσμα από το device να αντιγραφεί πίσω στο host.

3.2 Το Προγραμματιστικό Μοντέλο της CUDA

Σε αυτό το κεφάλαιο θα αναλυθούν οι βασικές έννοιες πίσω από το προγραμματιστικό μοντέλο της CUDA περιγράφοντας τον τρόπο με τον οποίο εκτείνεται στην γλώσσα προγραμματισμού C.

3.2.1 Συνάρτηση kernel

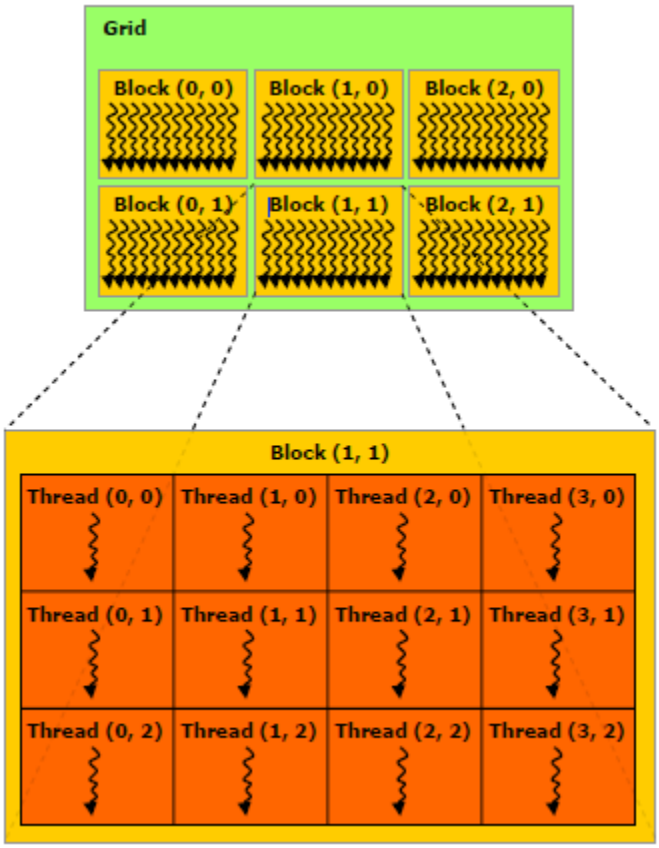
Η CUDA C έχει κάποιες προσθήκες στην γλωσσά προγραμματισμού C, οι οποίες δίνουν την δυνατότητα στους προγραμματιστές να ορίζουν συναρτήσεις C, που ονομάζονται kernel, οι οποίες όταν καλούνται, εκτελούνται N φορές παράλληλα από N διαφορετικά CUDA νήματα, σε αντίθεση με μόνο μια φορά που εκτελούνται οι καθιερωμένες συναρτήσεις της C, και αυτό επιτυγχάνεται με την προσθήκη κάποιων έτοιμων βιβλιοθηκών και δομών της CUDA. Μια συνάρτηση kernel ορίζεται στον device κώδικα, με την εντολή _global_ μπροστά από τον ορισμό της συνάρτηση, όπως

φαίνεται πιο κάτω, και όταν καλείται, στον host κώδικα, καθορίζεται ο αριθμός των CUDA νημάτων τα οποία θα την εκτελέσουν. Κάθε τέτοιο νήμα εκτελεί όλο το σώμα της συνάρτησης και όλα μαζί εκτελούνται παράλληλα στο device. Πριν από την κλήση του κώδικα πρέπει όλα τα απαραίτητα δεδομένα να μεταφερθούν από την μνήμη του host στην μνήμη του device.

Ορισμός Συνάρτησης kernel :
global kernel(…parameters…) { function body };
Κλήση συνάρτησης kernel:
Kernel<<< dimBlock,dimGrid >>>(…parameters…);
όπου με dimBlock ορίζεται ένα δισδιάστατο διάνυσμα για τον αριθμό των μπλοκ και με dimGrid ένα τρισδιάστατο διάνυσμα για τον αριθμό των νημάτων. Κάθε μπλοκ περιέχει dimGrid αριθμό νημάτων, και άρα ο συνολικός αριθμός των νημάτων που δημιουργούνται είναι dimGrid*dimBlock.

3.2.2 Ιεραρχία των Νημάτων

Οι βασικές μονάδες από τις οποίες αποτελείται η αρχιτεκτονική της CUDA είναι τα πλέγματα (grids), τα μπλοκ(blocks) και τα νήματα (threads),όπως φαίνεται στο Σχήμα 3.1. Τα πλέγματα αποτελούνται από μπλοκ και με κάθε κλήση από το host στο device δημιουργείται και ένα πλέγματα. Σε multi-GPU συστήματα γίνεται χρήση πολλαπλών πλεγμάτων για μέγιστη αποδοτικότητα. Όλα τα μπλοκ σε ένα πλέγμα εκτελούν το ίδιο κομμάτι κώδικα και κάθε ένα από αυτά έχει ένα μοναδικό αριθμό, το blockID. Τα μπλοκ αποτελούνται από ένα αριθμό συντονισμένων νημάτων, όπου και αυτά με την σειρά τους έχουν το καθένα ένα μοναδικό αριθμό, threadID. Ο μέγιστος αριθμός νημάτων που μπορεί να περιέχει ένα μπλοκ είναι 1024. Τα νήματα σε ένα μπλοκ μπορούν να συνεργάζονται μεταξύ τους ανταλλάσσοντας δεδομένα μέσω της κοινόχρηστης μνήμη που έχει κάθε μπλοκ, και μπορούν επίσης να συγχρονίζονται με την εντολή __syncthreads().



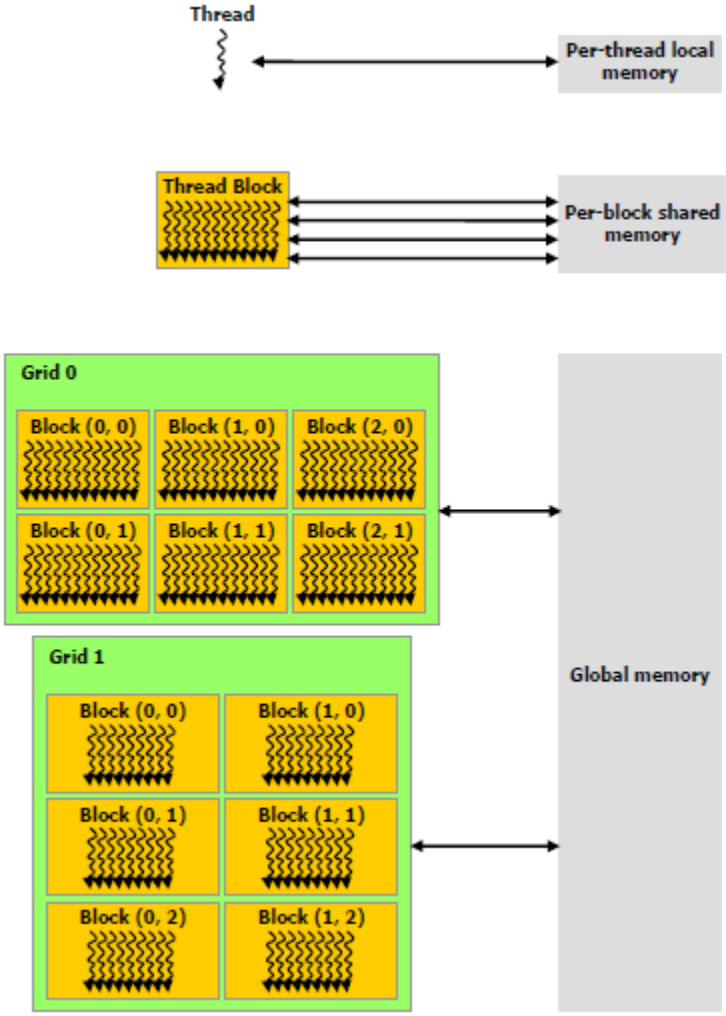
Σχήμα 3.1: Ένα grid το οποίο αποτελείται από μπλοκ, όπου το κάθε μπλοκ αποτελείται από νήματα.[13]

Αντίθετα νήματα διαφορετικών μπλοκ δεν μπορούν να συνεργάζονται μεταξύ τους, είναι όμως δυνατόν να συγχρονίζονται μεταξύ τους μόνο μέσω των λειτουργιών ατομικής μνήμης πάνω στην global μνήμη, που όμως κοστίζουν πολύ, και το προγραμματιστικό μοντέλο της εκάστοτε εφαρμογής πρέπει να τα αντιμετωπίζει ως ανεξάρτητες, παράλληλες υπολογιστικές μονάδες. Η χρονική σειρά εκτέλεσης των μπλοκ καθορίζεται αποκλειστικά από τις μονάδες ελέγχου του επεξεργαστή και δεν είναι δυνατόν να προσδιοριστεί από τον προγραμματιστή.

3.2.3 Ιεραρχία της μνήμης

Στην CUDA η μνήμη του υπολογιστή αναφέρεται ως host memory, ενώ η μνήμη της κάρτας γραφικών ως device memory. Ο κώδικας που βρίσκεται στο εσωτερικό ενός kernel μπορεί να επεξεργαστεί δεδομένα που βρίσκονται μόνο στην device memory, ενώ αντίθετα κώδικας έξω από τον kernel δεν μπορεί να συνεργαστεί απευθείας με αυτή. Για αυτόν τον λόγο υπάρχουν συναρτήσεις δέσμευσης, αποδέσμευσης, αντιγραφής και μεταφοράς δεδομένων ανάμεσα στην host και την device memory.

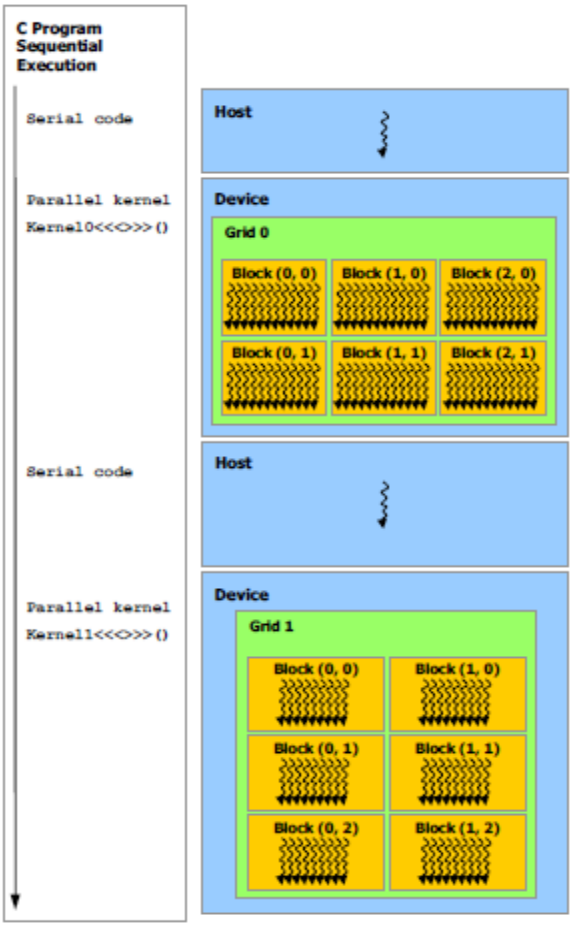
Τα νήματα έχουν πρόσβαση σε δεδομένα από διαφορετικούς χώρους μνήμης της κάρτας γραφικών. Σε κάθε νήμα αντιστοιχεί ένα σύνολο από registers προσβάσιμους μόνο από αυτό. Το συνολικό μέγεθος αυτού του registers file για κάθε πυρήνα είναι 32-64KB και αποτελεί την πιο γρήγορη μνήμη που είναι διαθέσιμη. Επίσης όλα τα νήματα ενός μπλοκ έχουν πρόσβαση σε έναν κοινό χώρο μνήμης μεγέθους 16KB ανά πυρήνα, με διάρκεια ζωής όσο και αυτή του μπλοκ. Και οι δύο αυτοί τύποι μνήμης βρίσκονται πάνω σε chips και άρα επιτυγχάνουν πολύ υψηλούς ρυθμούς διαμεταγωγής, αλλά αν δεν υπάρχει ανάγκη για επικοινωνία δεδομένων ανάμεσα στα νήματα πρέπει να προτιμάται η χρήση του registers file καθώς είναι γρηγορότερο. Τέλος υπάρχει και η global ή device memory της κάρτας γραφικών που χρησιμοποιείται για την μεταφορά και αποθήκευση δεδομένων από και προς την host memory, ενώ δύο ειδικοί χώροι μνήμης της είναι οι constant και texture memory που είναι βελτιστοποιημένες για κάποιες ειδικές χρήσεις (data filtering κτλ).



Σχήμα 3.2: Χώροι μνήμης στην device memory [13]

3.2.4 Υβριδικός Προγραμματισμός

Το προγραμματιστικό μοντέλο της CUDA, υποδηλώνει ότι τα CUDA νήματα εκτελούνται σε μια ξεχωριστή συσκευή που λειτουργεί ως συνεπεξεργαστής με τον Host που εκτελεί το σειριακό πρόγραμμα σε C, όπως φαίνεται και στο Σχήμα 3.3. Άρα έχουμε ένα υβριδικό πρόγραμμα, όπου οι kernel συναρτήσεις εκτελούνται στην GPU, και το υπόλοιπο πρόγραμμα C εκτελείται στην CPU.



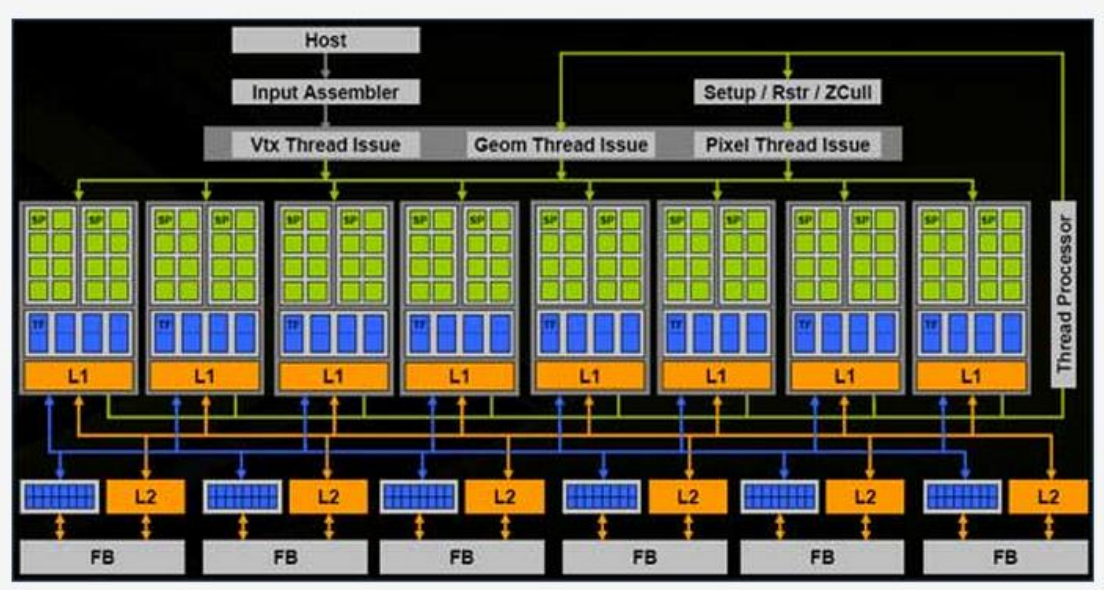
Σχήμα 3.3: Υβριδικό Πρόγραμμα. GPU ως συνεπεξεργαστής της CPU [13]

3.3 Αρχιτεκτονική της CUDA

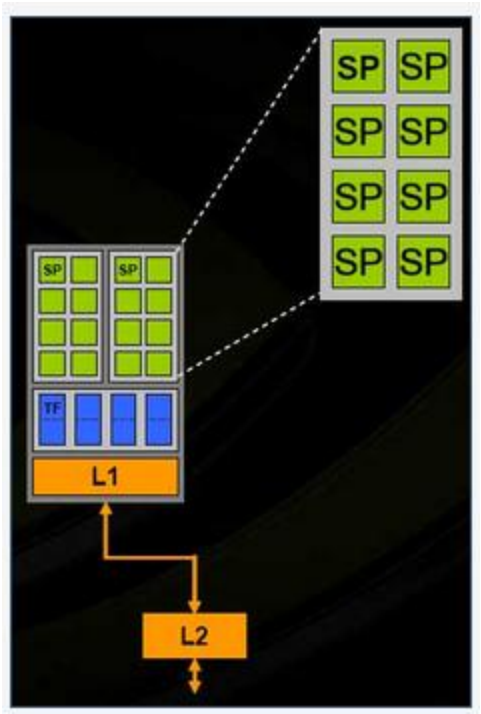
Στο Σχήμα 3.4 φαίνεται η αρχιτεκτονική ενός τυπικού CUDA GPU. Είναι οργανωμένο σε μια σειρά από υψηλού επιπέδου νημάτων, πολυεπεξεργαστές ροής (SMs). Όπως φαίνεται στο Σχήμα 3.5, σε κάθε μπλοκ υπάρχουν δύο SMs, αλλά ο αριθμός τους μπορεί να ποικίλει από μια γενιά CUDA GPU σε άλλη. Κάθε SM αποτελείται από μια σειρά από επεξεργαστές ροής (SPs), οι οποίοι μοιράζονται τον έλεγχο στο χώρο της cache μνήμης. Κάθε SM μπορεί ανεξάρτητα να δημιουργήσει και να εκτελέσει νήματα, τα οποία είναι οργανωμένα σε μπλοκ, και εκτελούν τις ίδιες οδηγίες έναντι διαφορετικών συγχρονισμένων δεδομένων (SIMD). Κάθε GPU σήμερα περιέχει μέχρι και 4 giga bytes (GDDR) DRAM, που αναφέρεται ως καθολική μνήμη στο σχήμα.

Αυτές οι (GDDR) DRAM διαφέρουν από τις DRAM της CPU, με την έννοια ότι ουσιαστικά η μνήμη buffer χρησιμοποιείται για τα γραφικά.

Η αρχιτεκτονική G80 της NVIDIA, η οποία χρησιμοποιήθηκε σε αυτήν την εργασία, έχει 86,4 GB/s εύρος ζώνης της μνήμης καθώς επίσης και 8 GB/s εύρος ζώνης για επικοινωνία με την CPU. Μια εφαρμογή σε CUDA μπορεί να μεταφέρει δεδομένα από την μνήμη του συστήματος σε 4 GB/s στην μνήμη της συσκευής, και ταυτόχρονα να μεταφέρει δεδομένα από την μνήμη της συσκευής στην μνήμη του συστήματος σε 4 GB/s. Το μαζικά παράλληλο G80 τσιπ αποτελείται από 128 SPs, 16 SMs όπου ο κάθε ένας αποτελείται από 8 SPs. Κάθε SPs έχει μια επιπλέον μονάδα για πολλαπλασιασμό και πρόσθεση (MAD) και μια επιπλέον μονάδα για πολλαπλασιασμό μόνο. Με 128 SPs, έχουμε ένα σύνολο με πάνω από 500 gigaflops. Ένας Intel επεξεργαστής υποστηρίζει 2 ή 4 νήματα, ανάλογα με το μοντέλο της μηχανής ανά πυρήνα, ενώ το G80 τσιπ υποστηρίζει έως και 768 νήματα σε κάθε SM, άρα συνολικά υποστηρίζει περίπου 12000 νήματα.



Σχήμα 3.4 : Αρχιτεκτονική των NVIDIA GPUs[16]



Σχήμα 3.5 : Πολυεπεξεργαστής ροής (SM)

Κεφάλαιο 4

Βασικά χαρακτηριστικά της OpenCL

4.1 Εισαγωγή

Η OpenCL είναι μια τυποποιημένη, πολλαπλή πλατφόρμα, και διαθέτει API για παράλληλους υπολογισμούς, με βάση την γλώσσα προγραμματισμού C. Έχει σχεδιαστεί για να επιτρέπει την ανάπτυξη παράλληλων φορητών εφαρμογών, για συστήματα με ετερογενής συσκευές υπολογισμών.

Η ανάπτυξης της OpenCL, υπαγορεύθηκε από την ανάγκη για μια τυποποιημένη πλατφόρμα, υψηλής απόδοσης ανάπτυξη εφαρμογών, για την ταχέως αναπτυσσόμενη ποικιλία των παράλληλων υπολογιστικών πλατφόρμων. Συγκεκριμένα αντιμετωπίζει σημαντικούς περιορισμούς που είχαν προηγούμενα μοντέλα προγραμματισμού για ετερογενή συστήματα παράλληλων υπολογισμών.

Τα παράλληλα μοντέλα προγραμματισμού που εκτελούνται στην CPU, τυπικά βασίζονται σε πρότυπα όπως η OpenMP, αλλά συνήθως δεν περιλαμβάνουν τη χρήση ειδικών τύπων μνήμης ή εκτελέσεις SIMD από υψηλής επίδοσης προγραμματιστές. Έτσι ο περιορισμός αυτός καθιστούσε δύσκολη την πρόσβαση στην υπολογιστική ισχύ των επεξεργαστών, GPUs, από τους προγραμματιστές.

Η ανάπτυξη της OpenCL ξεκίνησε από την Apple και αναπτύχθηκε από την Khronos Group, που είναι η ίδια ομάδα που διαχειρίζεται την OpenGL. Από την μία πλευρά, η OpenCL βασίζεται σε μεγάλο βαθμό στην CUDA , συγκεκριμένα στον τομέα του παραλληλισμού των δεδομένων, καθώς επίσης και στις ιεραρχίες της μνήμης. Από την άλλη πλευρά, η OpenCL, έχει μια πιο σύνθετη πλατφόρμα και έχει πιο σύνθετο μοντέλο διαχείρισης, που αντανακλά στην υποστήριξη των πολλαπλών πλατφόρμων. Επίσης η OpenCL υποστηρίζει κώδικα για ετερογενή παράλληλους υπολογισμούς, δηλαδή αν δεν υπάρχει διαθέσιμη συσκευή στην μονάδα επεξεργασίας γραφικών, τότε το πρόγραμμα θα εκτελεστεί εξ ολοκλήρου από την κεντρική μονάδα επεξεργασίας. Υπάρχουν ήδη OpenCL εφαρμογές σε AMD, ATI και NVIDIA GPUs καθώς επίσης και σε x86 CPUs. Ως μελλοντική προοπτική των OpenCL εφαρμογών, είναι η υποστήριξη και άλλων τύπων συσκευών, όπως ψηφιακούς επεξεργαστές σήματος (DSPs) [22] και προγραμματιζόμενες συστοιχίες πυλών (FPGAs) [21]. Το OpenCL πρότυπο έχει σχεδιαστεί για να υποστηρίζει την φορητότητα του κώδικά, σε συσκευές που παράγονται από διαφορετικούς προμηθευτές.

Πολλά χαρακτηριστικά της OpenCL είναι προαιρετικά και μπορεί να μην υποστηρίζονται από όλες τις συσκευές που αναφέρονται πιο πάνω, έτσι, σε ένα φορητό OpenCL κώδικας πρέπει να αποφεύγεται η χρήση αυτών των προαιρετικών λειτουργιών. Μερικές από αυτές τις προαιρετικές δυνατότητες, όμως αν χρησιμοποιηθούν, στις συσκευές που τις υποστηρίζουν, επιτυγχάνονται σημαντικά μεγαλύτερες επιδόσεις.

4.2 Το Προγραμματιστικό Μοντέλο της OpenCL

Το προγραμματιστικό μοντέλο της OpenCL είναι αντίστοιχο με το προγραμματιστικό μοντέλο της CUDA, και βασίζεται στην γλώσσα προγραμματισμού C. Σε ένα OpenCL πρόγραμμα πρέπει αρχικά να καθοριστεί το OpenCL περιβάλλον στον host κώδικα, το οποίο επιτυγχάνεται με την δημιουργία ενός OpenCL context. Ένα OpenCL context, ένα αντικείμενο τύπου cl_context, και δημιουργείται από την πιο κάτω ειδικά διαμορφωμένη συνάρτηση της OpenCL [15].

```
// Returs the context
cl_context clCreateContext (const cl_context_properties *properties,
        cl_uint num_devices,   // Number of devices
        const cl_device_id *devices,   // Pointer to the devices object
        void (*pfn_notify)(const char *errinfo, const void *private_info,
                size_t cb, void *user_data),
        void *user_data,
        cl_int *errcode_ret)   // error code result
```

Επίσης πρέπει να δημιουργηθεί και μια ουρά εντολών (OpenCL command queue), η οποία περιέχει όλες τις εντολές που πρέπει να εκτελεστούν από το device. Η δημιουργία μιας ουρά εντολών επιτυγχάνεται από την κλήση τις πιο κάτω OpenCL συνάρτηση στον host κώδικα [15]:

```
cl_command_queue clCreateCommandQueue (cl_context context,
            cl_device_id device,
            cl_command_queue_properties properties,

                        cl_int *errcode_ret)   // error code result
```

4.2.1 Συνάρτηση kernel

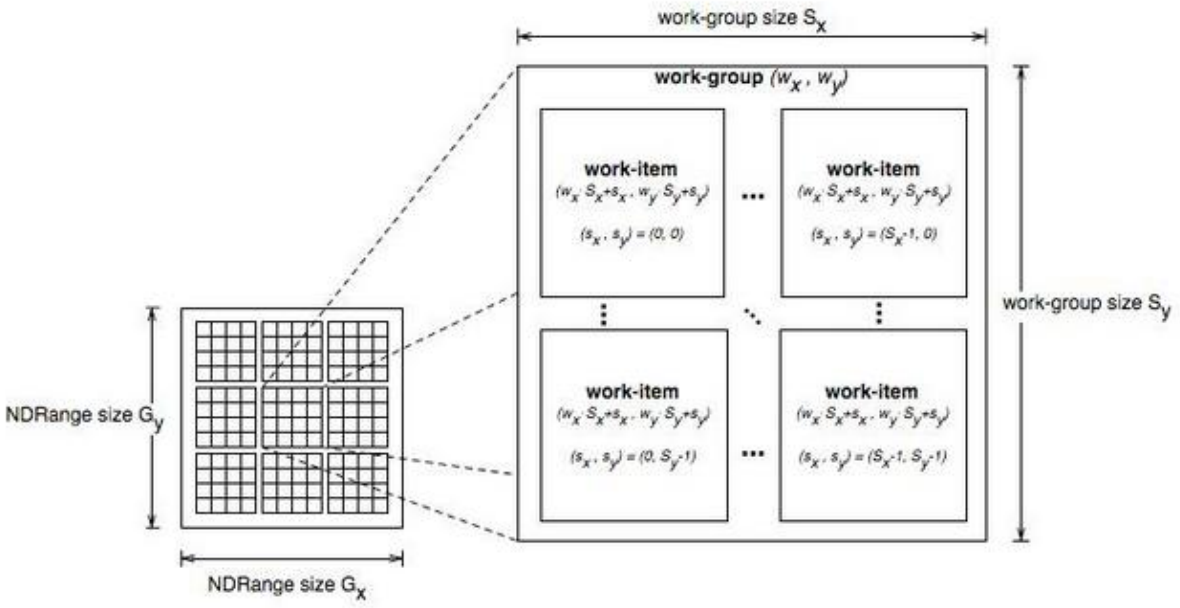
Ένα OpenCL πρόγραμμα αποτελείται από δύο μέρη : συναρτήσεις kernel, οι οποίες εκτελούνται σε μία ή περισσότερες OpenCL συσκευές και το πρόγραμμα host, το οποίο διαχειρίζεται την εκτέλεση των συναρτήσεων αυτών. Όπως και στην CUDA, ο τρόπος για να καθορίσεις συγκεκριμένους υπολογισμούς, οι οποία θα εκτελούνται παράλληλα, γίνεται από το host πρόγραμμα όπου καλείται η συνάρτηση kernel, και εκτελείται N φορές από N διαφορετικά νήματα παράλληλα . Η δήλωση για μια συνάρτηση kernel στην OpenCL αρχίζει με την λέξη κλειδί _ kernel [17].

4.2.2 Ιεραρχία των Νημάτων

Όταν η συνάρτηση kernel ξεκινήσει, ο κώδικας εκτελείται από τα work-items, τα οποία αντιστοιχούν με τα CUDA νήματα, και το κάθε work-item εκτελεί το ίδιο κομμάτι κώδικα. Τα work-item ομαδοποιούνται στα work-groups, τα οποία αντιστοιχούν με τα CUDA μπλοκ. Τα work-groups επιτρέπουν την επικοινωνία και την συνεργασία μεταξύ των work-items που τα αποτελούν [17].

Όλα τα work-items έχουν την δική τους μοναδική καθολική τιμή. Υπάρχει μια μικρή διαφορά μεταξύ της CUDA και της OpenCL, στον τρόπο που διαχειρίζονται τις τιμές αυτές. Στην CUDA, σε κάθε νήμα αντιστοιχούν δύο τιμές, η blockIdx και η threadIdx. Οι δύο αυτές τιμές συνδυάζονται και σχηματίζουν την μοναδική καθολική τιμή του νήματος, η οποία προσδιορίζεται από την μεταβλητή threadid. Σε ένα OpenCL kernel, ένα νήμα μπορεί να πάρει την μοναδική καθολική του τιμή από την κλήση της API συνάρτησης get_global_id(), όπου παίρνει σαν παράμετρο μια μεταβλητή η οποία καθορίζει την διάσταση. Για παράδειγμα η κλήση της συνάρτησης με παράμετρο 0, get_global_id(0) , αναφέρεται στην x διάσταση, με παράμετρο 1, get_global_id(1), στην y διάσταση και με παράμετρο 2, get_global_id(2), στην z διάσταση. Ένας OpenCL kernel μπορεί επίσης να καλέσει την API συνάρτησης get_global_size(), όπου με τις αντίστοιχες παραμέτρους που αναφέρονται πιο πάνω (0 – διάσταση x, 1 – διάσταση y

και 2 – διάσταση z) σου επιστρέφει τον συνολικό αριθμό των work-items για την συγκεκριμένη ND Range διάσταση.



Σχήμα 4.1 : Οργάνωση ενός ND Range [6]

4.2.3 Ιεραρχία της μνήμης

Η OpenCL μοντελοποιεί ένα ετερογενή παράλληλο υπολογιστικό σύστημα, από τον host και μια ή περισσότερες OpenCL συσκευές. Κάθε συσκευή περιέχει μια ή περισσότερες υπολογιστικές μονάδες (CU), που αντιστοιχούν στους πολυεπεξεργαστές της CUDA (SMs). Επίσης μια CU, μπορεί να αντιστοιχεί και με πυρήνες της CPU ή άλλους τύπους μονάδων εκτέλεσης όπως DSPs και FPGAs.

Κάθε CU, με την σειρά της, αποτελείται από ένα ή περισσότερα στοιχεία επεξεργασίας(PEs), τα οποία αντιστοιχούν με τους επεξεργαστές SP, της CUDA. Οι υπολογισμοί σε μια συσκευή γίνονται από μεμονωμένα PE.

Στην OpenCL υπάρχει ιεραρχία μεταξύ των διαφόρων τύπων μνήμης, που μπορούν να χρησιμοποιηθούν από τους προγραμματιστές. Οι διάφοροι τύποι της μνήμης είναι η καθολική, η διαρκή, η τοπική και η ιδιωτική. Η καθολική μνήμη της OpenCL αντιστοιχεί στην καθολική μνήμη της CUDA, και μπορεί να δεσμευτεί δυναμικά από το host πρόγραμμα και υποστηρίζει πρόσβαση για διάβασμα και γράψιμο, τόσο από το host όσο και από το device. Η διαρκή μνήμη μπορεί να δεσμευτεί δυναμικά από το host, και υποστηρίζει πρόσβαση για διάβασμα και γράψιμο από το host, και μόνο διάβασμα από το device. Η OpenCL για να είναι σε θέση να υποστηρίζει πολλαπλές πλατφόρμες, δεν έχει περιορισμό για το μέγεθος της διαρκής μνήμης σε 64KB, όπως η CUDA. Αντί για αυτό, υπάρχει μια εντολή όπου σου επιστρέφει το σταθερό μέγεθος μνήμης που υποστηρίζει η συγκεκριμένη συσκευή.

Η τοπική μνήμη της OpenCL είναι κάτι ανάλογο με την κοινόχρηστη μνήμη της CUDA. Μπορεί να δεσμευτεί δυναμικά από τον host και στατικά από τον device κώδικα. Η τοπική μνήμη της OpenCL δεν μπορεί να προσπελαστεί από τον host και παρέχει πρόσβαση για διάβασμα και γράψιμο από όλα τα work-items ενός work-group.Η ιδιωτική μνήμη της OpenCL αντιστοιχεί στην τοπική μνήμη της CUDA.

4.3 Προγραμματιστικό μοντέλο της OpenCL για την αρχιτεκτονική CUDA

Ένας πολυεπεξεργαστής αντιστοιχεί σε μια OpenCL υπολογιστική μονάδα. Ένας πολυεπεξεργαστής εκτελεί ένα CUDA νήμα για κάθε OpenCL work-item, και ένα μπλοκ από νήματα για κάθε ένα OpenCL work-group. Η συνάρτηση kernel εκτελείται από ένα OpenCL NDRange από ένα πλέγμα από μπλοκ με νήματα. Κάθε μπλοκ από νήματα που εκτελείται από την συνάρτηση kernel, είναι μοναδικά καθορισμένο από το work-group ID που το αντιπροσωπεύει, και κάθε νήμα από το καθολικό του ID, ή από ένα συνδυασμό από το τοπικό του ID και το work-group ID στο οποίο ανήκει.

Σε κάθε νήμα δίνεται επίσης ένα threadID που αναγνωρίζεται μέσα στο μπλοκ στο οποίο βρίσκεται. Για ένα μονοδιάστατο μπλοκ το threadID ενός νήματος παραμένει ως έχει, για ένα δισδιάστατο μπλοκ μεγέθους (Dx,Dy), το threadID ενός νήματος με δείκτη (x,y) είναι (x+yDx), και για ένα τρισδιάστατο μπλοκ μεγέθους (Dx,Dy,Dz), το threadID ενός νήματος με δείκτη (x,y,z) είναι (x+yDx+zDxDy).

Όταν ένα OpenCL πρόγραμμα στον Host περιλαμβάνει συνάρτηση kernel, τα work-groups απαριθμούνται και διανέμονται σε μπλοκ από νήματα στους πολυεπεξεργαστές, οι οποίοι έχουν διαθέσιμη ικανότητα εκτέλεσης. Τα νήματα, ενός μπλοκ από νήματα, εκτελούνται ταυτόχρονα σε ένα πολυεπεξεργαστή. Όταν τα νήματα του μπλοκ τερματίσουν την εκτέλεση τους, ένα νέο μπλοκ ξεκινά την εκτέλεση του στον εκάστοτε πολυεπεξεργαστή [17].Ένας πολυεπεξεργαστή είναι σχεδιασμένος να εκτελεί εκατοντάδες νήματα ταυτόχρονα..

Κεφάλαιο 5

Βασικά χαρακτηριστικά της HMPP Workbench

5.1 Εισαγωγή

Η HMPP Workbench είναι ένα νέο πλαίσιο επιταχυντή της Caps Enterprise, που επιτρέπει τον προγραμματισμό του επιταχυντή υλικού, δίνοντας την δυνατότητα στους προγραμματιστές να μπορούν να τον προγραμματίσουν έχοντας μόνο ελάχιστες γνώσεις για το υλικό. Το πλαίσιο αυτό βασίζεται σε οδηγίες, οι οποίες επιταχύνουν τον σειριακό κώδικα, από επεξεργαστές πολλαπλών πυρήνων, τόσο από GPUs όσο και από συσκευές που μπορούν να προγραμματιστούν με CUDA και OpenCL. Η ιδέα του πλαισίου αυτού είναι η προσθήκη πληροφοριών με την μορφή οδηγιών, για συγκεκριμένα τμήματα κώδικα τα οποία θα εκτελεστούν στον επιταχυντή του υλικού. Η HMPP Workbench βασίζεται στις γλώσσες προγραμματισμού C και FORTRAN. Σε αυτή την εργασία θα μελετηθούν οι επεκτάσεις στην γλώσσα προγραμματισμού C [18].

5.2 Το Προγραμματιστικό Μοντέλο της HMPP Workbench

Όπως αναφέρεται και παραπάνω το HMPP Workbench πλαίσιο βασίζεται σε οδηγίες. Οι οδηγίες αυτές στην γλώσσα προγραμματισμού C διακρίνονται μεταξύ των #pragma hmpp και #pragma hmppcp οι οποίες θα αναλυθούν πιο κάτω [19].

5.2.1Codelet/Callsite

Η HMPP Workbench βασίζεται στην έννοια των codelets. Τα codelets είναι διακεκριμένα κομμάτια κώδικα, συγκεκριμένα συναρτήσεις, που μαρκάρονται για αυτόματη μετάφραση σε κώδικα επιταχυντή υλικού ούτως ώστε να εκτελεστούν σε αυτόν. Ένα codelet είναι μια συνάρτηση void, δεν περιλαμβάνει static δηλώσεις μεταβλητών, δεν είναι αναδρομική συνάρτηση, δεν καλείται στο σώμα της συνάρτησης άλλη συνάρτηση codelet και τέλος δεν περιέχει κλήση συναρτήσεων όπως συναρτήσεις βιβλιοθηκών malloc, printf κτλ. Όλοι αυτοί οι περιορισμοί ελέγχονται από τον HMPP μεταγλωττιστή. Ένα codelet μπορεί να κληθεί από ένα callsite.

Οι οδηγίες #pragma hmpp δίνουν τον έλεγχο επί του επιταχυντή. Χρησιμοποιούνται για τον ορισμό των codelets , και των δεδομένων τα οποία θα μεταφερθούν από την κύρια μνήμη στην μνήμη του επιταχυντή του υλικού και αντίσροφα. Από την άλλη οι οδηγίες #pragma hmppcp μπορούν να χρησιμοποιηθούν για περαιτέρω ανάθεση παραλληλισμού στους βρόχους εντός ενός codelet, ανάλογα με τον επιταχυντή του υλικού. Τα codelets μπορούν να ομαδοποιηθούν για να μοιράζονται κοινά δεδομένα. Σε αυτή την περίπτωση πρέπει να τοποθετηθούν στο ίδιο αρχείο πηγαίου κώδικα. Κατά τον ορισμό ενός codelet καθορίζεται η αρχιτεκτονική στόχος και ο τρόπος μεταφοράς των δεδομένων, αυτόματα ή από τον προγραμματιστή, όπως φαίνεται πιο κάτω:

```
#pragma hmpp temp1 codelet, target = CUDA, args[*].transfer=auto;
#pragma hmpp temp2 codelet, target = OPENCL, args[*].transfer=manual;
```

Ορισμός του callsite γίνεται με την πιο κάτω εντολή :

```
#pragma hmpp temp1 callsite;
temp1(…parameters…);
```

Κεφάλαιο 6

Αξιολόγηση – Αποτελέσματα

6.1.1 Πολλαπλασιασμός Πινάκων σε CUDA

Το πρόγραμμα το οποίο χρησιμοποιήθηκε σε CUDA, υλοποιεί τον πολλαπλασιασμό πινάκων. Πιο κάτω θα αναλυθούν οι βασικές εντολές του προγράμματος που χρησιμοποιήθηκε, καθώς επίσης και ο τρόπος που επιλύεται το πιο πάνω πρόβλημα. Αρχικά θα αναφερθούν τα βασικά βήματα που ακολουθούνται στον κώδικα για τον host , και στην συνέχεια θα αναλυθεί η συνάρτηση kernel που ορίζεται στον κώδικα για το device.

Στο Σχήμα 6.1 αναπαριστάται ένα απόσπασμα του κώδικα για τον host. Πρώτα δεσμεύεται μνήμη για τους πίνακες A και B στην μνήμη του host [γραμμή 2 μέχρι 8] και ακολούθως στην μνήμη του device [γραμμή 14 μέχρι 18]. Η συνάρτηση στην γραμμή 11 και 12 δίνει τυχαίες τιμές στα στοιχεία του πίνακα A και B, αντίστοιχα. Στη συνέχεια

αντιγράφονται οι πίνακες αυτοί από την μνήμη του host στην μνήμη του device, με την συνάρτηση της CUDA, cudaMemcpy() [γραμμή 21 και 22], και την αντίστοιχη εντολή cudaMemcpyHostToDevice. Έπειτα δεσμεύεται μνήμη για τον πίνακα C, στην μνήμη του device [γραμμή 25 μέχρι 28] και στην μνήμη του host [γραμμή 31], και στη συνέχεια καθορίζεται ο αριθμός των νημάτων σε ένα μπλοκ [γραμμή 35], και ο αριθμός των μπλοκ σε ένα πλέγμα [γραμμή 36]. Αφού ολοκληρωθούν τα πιο πάνω βήματα καλείται η συνάρτηση kernel [γραμμή 47], η οποία θα εκτελεστεί στο device. Η συνάρτηση της CUDA cudaThreadSynchronize() [γραμμή 40], υποδηλώνει ότι τα νήματα θα εκτελούνται συγχρονισμένα. Τέλος όταν τερματίσει η συνάρτηση kernel, το αποτέλεσμα, αντιγράφεται από την μνήμη του device στην μνήμη του host [γραμμή 56].

Στο Σχήμα 6.2 αναπαριστάται η συνάρτηση kernel στον κώδικα για το device. Αρχικά καθορίζονται οι ενδείξεις για τα μπλοκ [γραμμή 5 και 6], τα οποία είναι δισδιάστατα, και στη συνέχεια οι ενδείξεις για τα νήματα [γραμμή 9 και 10], όπου το threadIdx.x αναφέρεται στην διάσταση x του μπλοκ, και το threadIdx.y στην διάσταση y του μπλοκ. Ακολούθως δημιουργούνται οι υποπίνακες A και B [γραμμή 12 με 21]. Η φυλοσογία πίσω από τον τρόπο που υπολογίζεται η τιμή του στοιχείου C[i][j], του πίνακα C, γενικά, είναι ότι πολλαπλασιάζεται η γραμμή i του πίνακα A με την στήλη j του πίνακα B. Πιο συγκεκριμένα πολλαπλασιάζει το στοιχείο κ της γραμμή i του πίνακα A, με το στοιχείο κ της στήλη j του πίνακα B, και προστήθονται όλα τα αποτελέσματα μαζί και προκύπτει η τιμή του στοιχείου C[i][j]. Στο παράδειγμα μας κάθε στοιχείο του πίνακα A και B αντιπροσωπεύεται από ένα νήμα, και οι πιο πάνω πράξεις για να βρεθεί η τιμή κάθε στοιχείου του πίνακα C, που αντιπροσωπεύεται από την μεταβλητή Csub, γίνονται παράλληλα και προκύπτει ο τελικός πίνακας C.

```
...
// allocate host memory for matrices A and B
    unsigned int size_A = uiWA * uiHA;
    unsigned int mem_size_A = sizeof(float) * size_A;
    float* h_A = (float*)malloc(mem_size_A);
    unsigned int size_B = uiWB * uiHB;
    unsigned int mem_size_B = sizeof(float) * size_B;
    float* h_B = (float*)malloc(mem_size_B);

    // initialize host memory
    randomInit(h_A, size_A);
    randomInit(h_B, size_B);

    // allocate device memory
    float* d_A;
    cutilSafeCall(cudaMalloc((void**) &d_A, mem_size_A));
    float* d_B;
    cutilSafeCall(cudaMalloc((void**) &d_B, mem_size_B));

    // copy host memory to device
    cutilSafeCall(cudaMemcpy(d_A, h_A, mem_size_A,cudaMemcpyHostToDevice) );
    cutilSafeCall(cudaMemcpy(d_B, h_B, mem_size_B,cudaMemcpyHostToDevice) );

    // allocate device memory for result
    unsigned int size_C = uiWC * uiHC;
    unsigned int mem_size_C = sizeof(float) * size_C;
    float* d_C;
    cutilSafeCall(cudaMalloc((void**) &d_C, mem_size_C));

  // allocate host memory for the result
    float* h_C = (float*) malloc(mem_size_C);
...

 // setup execution parameters
    dim3 threads(BLOCK_SIZE, BLOCK_SIZE);
    dim3 grid(uiWC / threads.x, uiHC / threads.y);

 // kernel warmup
matrixMul<<< grid, threads >>>(d_C, d_A, d_B, uiWA, uiWB);
 cudaThreadSynchronize();
...

 // execute the kernel
    int nIter = 30;
    for (int j = 0; j < nIter; j++)
      {
          matrixMul<<< grid, threads >>>(d_C, d_A, d_B, uiWA, uiWB);
       }

...

  cudaThreadSynchronize();

...
// copy result from device to host
 cutilSafeCall(cudaMemcpy(h_C, d_C, mem_size_C,cudaMemcpyDeviceToHost) );
```

Σχήμα 6.1 Host κώδικας σε CUDA, για την υλοποίηση του προβλήματος του πολλαπλασιασμού πινάκων [20].

```
1  __global__ void
2  matrixMul( float* C, float* A, float* B, int wA, int wB)
3  {
4      // Block index
5      int bx = blockIdx.x;
6      int by = blockIdx.y;
7
8      // Thread index
9      int tx = threadIdx.x;
10     int ty = threadIdx.y;
11
12     // Index of the first sub-matrix of A processed by the block
13     int aBegin = wA * BLOCK_SIZE * by;
14     // Index of the last sub-matrix of A processed by the block
15     int aEnd   = aBegin + wA - 1;
16     // Step size used to iterate through the sub-matrices of A
17     int aStep  = BLOCK_SIZE;
18     // Index of the first sub-matrix of B processed by the block
19     int bBegin = BLOCK_SIZE * bx;
20     // Step size used to iterate through the sub-matrices of B
21     int bStep  = BLOCK_SIZE * wB;
22
23     // Csub is used to store the element of the block sub-matrix that is computed by the thread
24     float Csub = 0;
25
26     // Loop over all the sub-matrices of A and B required to compute the block sub-matrix
27     for (int a = aBegin, b = bBegin;
28              a <= aEnd;
29              a += aStep, b += bStep) {
30
31      // Declaration of the shared memory array As used to store the sub-matrix of A
32        __shared__ float As[BLOCK_SIZE][BLOCK_SIZE];
33
34      // Declaration of the shared memory array Bs used to store the sub-matrix of B
35        __shared__ float Bs[BLOCK_SIZE][BLOCK_SIZE];
36
37      // Load the matrices from device memory to shared memory; each thread loads
38      // one element of each matrix
39        AS(ty, tx) = A[a + wA * ty + tx];
40        BS(ty, tx) = B[b + wB * ty + tx];
41
42      // Synchronize to make sure the matrices are loaded
43        __syncthreads();
44
45      // Multiply the two matrices together;each thread computes one element
46      // of the block sub-matrix
47        for (int k = 0; k < BLOCK_SIZE; ++k)
48           Csub += AS(ty, k) * BS(k, tx);
49
50        // Synchronize to make sure that the preceding computation is done before loading two
51        // new sub-matrices of A and B in the next iteration
52        __syncthreads();
53     }
54
55     // Write the block sub-matrix to device memory;
56     // each thread writes one element
57     int c = wB * BLOCK_SIZE * by + BLOCK_SIZE * bx;
58     C[c + wB * ty + tx] = Csub;
59 }
```

Σχήμα 6.2 Device κώδικας σε CUDA, υλοποίηση της συνάρτησης Kernel [20]

6.1.2 Πολλαπλασιασμός Πινάκων σε OpenCL

Σε αυτή την ενότητα παρουσιάζεται το OpenCL πρόγραμμα, που υλοποίει τον πολλαπλασιασμό πινάκων. Το πρόγραμμα αυτό χρησιμοποιεί την πλατφόρμα της NVIDIA, και έχει πολλές ομοιότητες με το CUDA πρόγραμμα, σχετικά με τα βήματα που ακολουθούνται. Πιο κάτω θα αναλυθούν τα βήματα αυτά, καθώς επίσης και ο τρόπος που υλοποιούνται στην OpenCL..

Στο Σχήμα 6.3 αναπαριστάται ένα απόσπασμα του κώδικα για τον host. Αρχικά αρχικοποιείται η πλατφόρμα με την πλατφόρμα της NVIDIA [γραμμή 3], και στη συνέχεια αποκτάτε η λίστα με τις διαθέσιμες συσκευές της πλατφόρμας [γραμμή 6 με 8]. Επιλέγονται οι συσκευές που θα χρησιμοποιηθούν [γραμμή 17 και 36] και δημιουργούνται οι ουρές εντολών για κάθε συσκευή [γραμμή 19 και 42]. Στη συνέχεια δημιουργείται το context [γραμμή 11], και δεσμεύεται η μνήμη για τους πίνακες A, B και C στην μνήμη του host [γραμμή 46 με 63]. Έπειτα δημιουργείται το OpenCL πρόγραμμα [γραμμή 71], γίνεται build [γραμμή 75] και δημιουργείται ο OpenCL kernel [γραμμή 78 με 85]. Ακολούθως δεσμεύεται η μνήμη για τους πίνακες A και B στο device[γραμμή 88 και 89], αντιγράφονται στην μνήμη του [γραμμή 92] και δεσμεύεται η μνήμη για τον πίνακα C [γραμμή 100]. Στην συνέχεια καθορίζονται οι παράμετροι του OpenCL kernel [γραμμή 103 με 109], και στην συνέχεια καλέιται για εκτέλεση στο device [γραμμή 111 με 137]. Όταν ο kernel τερματίσει αντιγράφεται το αποτέλεσμα από την μνήμη του device στην μνήμη του host [γραμμή 141 με 147].

Η συνάρτηση kernel του OpenCL προγράμματος είναι ακριβώς η ίδια με αυτή του CUDA προγράμματος για αυτό δεν θα γίνει κάποια αναφορά σε αυτή.

```
...
//Get the NVIDIA platform
 ciErrNum = oclGetPlatformID(&cpPlatform);
...
//Get the devices
ciErrNum = clGetDeviceIDs(cpPlatform, CL_DEVICE_TYPE_GPU, 0, NULL, &ciDeviceCount);
cdDevices = (cl_device_id *)malloc(ciDeviceCount * sizeof(cl_device_id) );
ciErrNum = clGetDeviceIDs(cpPlatform, CL_DEVICE_TYPE_GPU, ciDeviceCount, cdDevices, NULL);
...
//Create the context
cxGPUContext = clCreateContext(0, ciDeviceCount, cdDevices, NULL, NULL, &ciErrNum);
...
if(shrCheckCmdLineFlag(argc, (const char**)argv, "device"))
 {
            ...
            // get and print the device for this queue
            cl_device_id device = oclGetDev(cxGPUContext, atoi(deviceStr));
            // create command queue
            commandQueue[ciDeviceCount] = clCreateCommandQueue(cxGPUContext, device,
            CL_QUEUE_PROFILING_ENABLE, &ciErrNum);
            ...
}
else
{
      ...
      // Find out how many GPU's to compute on all available GPUs
      size_t nDeviceBytes;
      ciErrNum |= clGetContextInfo(cxGPUContext, CL_CONTEXT_DEVICES, 0, NULL,
               &nDeviceBytes);
      ciDeviceCount = (cl_uint)nDeviceBytes/sizeof(cl_device_id);

       // create command-queues
       for(unsigned int i = 0; i < ciDeviceCount; ++i)
       {
          // get and print the device for this queue
          cl_device_id device = oclGetDev(cxGPUContext, i);
          shrLog("Device %d: ", i);
          oclPrintDevName(LOGBOTH, device);
          shrLog("\n");

          // create command queue
          commandQueue[i] = clCreateCommandQueue(cxGPUContext, device,
          CL_QUEUE_PROFILING_ENABLE, &ciErrNum);
      }
}
// allocate host memory for matrices A and B
    unsigned int size_A = uiWA * uiHA;
    unsigned int mem_size_A = sizeof(float) * size_A;
    float* h_A_data = (float*)malloc(mem_size_A);
    unsigned int size_B = uiWB * uiHB;
    unsigned int mem_size_B = sizeof(float) * size_B;
    float* h_B_data = (float*)malloc(mem_size_B);

// initialize host memory
    srand(2006);
    shrFillArray(h_A_data, size_A);
    shrFillArray(h_B_data, size_B);
```

Σχήμα 6.3 Host κώδικας σε OpenCL, για την υλοποίηση του προβλήματος του πολλαπλασιασμού πινάκων [20].

```
// allocate host memory for result
    unsigned int size_C = uiWC * uiHC;
    unsigned int mem_size_C = sizeof(float) * size_C;
    float* h_C = (float*) malloc(mem_size_C);

// create OpenCL buffer pointing to the host memory
cl_mem h_A = clCreateBuffer(cxGPUContext, CL_MEM_READ_ONLY | CL_MEM_USE_HOST_PTR,
                                    mem_size_A, h_A_data, &ciErrNum);
…
// create the program
cl_program cpProgram = clCreateProgramWithSource(cxGPUContext, 1, (const char **)&source,
                                        &program_length, &ciErrNum);
…
// build the program
ciErrNum = clBuildProgram(cpProgram, 0, NULL, "-cl-fast-relaxed-math", NULL, NULL);
…
// Create Kernel
for(unsigned int i = 0; i < ciDeviceCount; ++i) {
       multiplicationKernel[i] = clCreateKernel(cpProgram, "matrixMul", &ciErrNum);
// Run multiplication on 1..deviceCount GPUs to compare improvement
 shrLog("\nRunning Computations on 1 - %d GPU's...\n\n", ciDeviceCount);
 for(unsigned int k = 1; k <= ciDeviceCount; ++k)
   {
      matrixMulGPU(k, h_A, h_B_data, mem_size_B, h_C);
   }
…
// Input buffer
workSize[i] = (i != (ciDeviceCount - 1)) ? sizePerGPU : (uiHA - workOffset[i]);
d_A[i] = clCreateBuffer(cxGPUContext, CL_MEM_READ_ONLY, workSize[i] * sizeof(float) * uiWA,
NULL,NULL);
// Copy only assigned rows from host to device
clEnqueueCopyBuffer(commandQueue[i], h_A, d_A[i], workOffset[i] * sizeof(float) * uiWA,
                        0, workSize[i] * sizeof(float) * uiWA, 0, NULL, NULL);

// create OpenCL buffer on device that will be initiatlize from the host memory on first use on device
d_B[i] = clCreateBuffer(cxGPUContext, CL_MEM_READ_ONLY | CL_MEM_COPY_HOST_PTR,
                           mem_size_B, h_B_data, NULL);
…
// Output buffer
 d_C[i] = clCreateBuffer(cxGPUContext, CL_MEM_WRITE_ONLY,  workSize[i] * uiWC * sizeof(float),
NULL,NULL);´
//set the args values
clSetKernelArg(multiplicationKernel[i], 0, sizeof(cl_mem), (void *) &d_C[i]);
clSetKernelArg(multiplicationKernel[i], 1, sizeof(cl_mem), (void *) &d_A[i]);
clSetKernelArg(multiplicationKernel[i], 2, sizeof(cl_mem), (void *) &d_B[i]);
clSetKernelArg(multiplicationKernel[i], 3, sizeof(float) * BLOCK_SIZE *BLOCK_SIZE, 0 );
clSetKernelArg(multiplicationKernel[i], 4, sizeof(float) * BLOCK_SIZE *BLOCK_SIZE, 0 );
clSetKernelArg(multiplicationKernel[i], 5, sizeof(cl_int), (void *) &uiWA);
clSetKernelArg(multiplicationKernel[i], 6, sizeof(cl_int), (void *) &uiWB);
// Execute Multiplication on all GPUs in parallel
size_t localWorkSize[] = {BLOCK_SIZE, BLOCK_SIZE};
size_t globalWorkSize[] = {shrRoundUp(BLOCK_SIZE, uiWC), shrRoundUp(BLOCK_SIZE, workSize[0])};
```

Σχήμα 6.3 Host κώδικας σε OpenCL, για την υλοποίηση του προβλήματος του πολλαπλασιασμού πινάκων [20]

```
117  // Launch kernels on devices
118  #ifdef GPU_PROFILING
119              int nIter = 30;
120      for (int j = -1; j < nIter; j++)
121      {
122          // Sync all queues to host and start timer first time through loop
123          if(j == 0){
124              for(unsigned int i = 0; i < ciDeviceCount; i++)
125              {
126                  clFinish(commandQueue[i]);
127              }
128              shrDeltaT(0);
129          }
130  #endif
130     for(unsigned int i = 0; i < ciDeviceCount; i++)
132      {
133        // Multiplication - non-blocking execution:  launch and push to device(s)
134        globalWorkSize[1] = shrRoundUp(BLOCK_SIZE, workSize[i]);
135        clEnqueueNDRangeKernel(commandQueue[i], multiplicationKernel[i], 2, 0, globalWorkSize,
136  localWorkSize,0, NULL, &GPUExecution[i]);
137              clFlush(commandQueue[i]);
138          }
139  #ifdef GPU_PROFILING
140  //COPY RESULTS FROM DEVICE MEMORY
141  for(unsigned int i = 0; i < ciDeviceCount; i++)
142      {
143          // Non-blocking copy of result from device to host
144          clEnqueueReadBuffer(commandQueue[i], d_C[i], CL_FALSE, 0, uiWC * sizeof(float) *
145  workSize[i],
146                              h_C + workOffset[i] * uiWC, 0, NULL, &GPUDone[i]);
147      }
148
149
150
```

Σχήμα 6.3 Host κώδικας σε OpenCL, για την υλοποίηση του προβλήματος του πολλαπλασιασμού πινάκων [20]

6.1.3 Πολλαπλασιασμός Πινάκων σε HMPP

Σχήμα 6.4 παρουσιάζεται μια υλοποίηση του πολλαπλασιασμού πινάκων σε HMPP με αρχιτεκτονική στόχο την CUDA.

```
1   ...
2   #pragma hmpp sgemm1 codelet, target=CUDA, args[vout].io=inout
3   extern void sgemm( int m, int n, int k, float alpha, const float vin1[n][n], const float vin2[n][n],
4   float beta, float vout[n][n] );
5   ...
6
7
8   int main(int argc, char **argv) {
9   ...
10              #pragma hmpp sgemm1 callsite
11                      sgemm( size, size, size, alpha, vin1, vin2, beta, vout );
12  return 0;
13  }
14  ...
15
16  #pragma hmpp sgemm1 codelet, target=CUDA, args[vout].io=inout
17  void sgemm( int m, int n, int l, float alpha, const float vin1[n][n], const float vin2[n][n], float beta,
18  float vout[n][n] ) {
19    int j;
20    for( j = 0 ; j < n ; j++ ) {
21      int i;
22      for( i = 0 ; i < n ; i++ ) {
23        int k;
24        float prod_I0_J0 = 0.0f;
25       for( k = 0 ; k < n ; k++ ) {
26          prod_I0_J0 += vin1[k][i] * vin2[j][k];
28        }
29        vout[j][i] = alpha * prod_I0_J0 + beta * vout[j][i];
30      }
31    }
32  }
33  ...
```

Σχήμα 6.4 Υλοποίηση πολλαπλασιασμού πινάκων σε HMPP (πρόγραμμα από την HMPP Workbench)

Στην γραμμή 2 ορίζεται ένα codelet για το segmm1, και αυτό υποδηλώνει ότι ο κώδικας μεταξύ της γραμμής 17 και της γραμμής 32 σημειώνεται για να εκτελεστεί στον επιταχυντή του υλικού. Η εντολή η οποία χρησιμοποιείται είναι

#pragma hmpp sgemm1 codelet, target=CUDA, args[vout].io=inout

Η αρχιτεκτονική στόχος του codelet καθορίζεται από την εντολή target=CUDA, όπου στην προκειμένη περίπτωση είναι η CUDA. Η εντολή args[vout].io=inout, καθορίζει τα

δεδομένα τα οποία θα μεταφερθούν από την κύρια μνήμη στην μνήμη της συσκεύς και αντίστροφα, λόγω του ορίσματος inout.

Η HMPP οδηγία για την κλήση ενός codelet είναι

```
#pragma hmpp sgemm1 callsite
        sgemm( size, size, size, alpha, vin1, vin2, beta, vout );
```

η οποία καθορίζει μια αντιστοίχηση μεταξύ της κλήσης της συνάρτησης στην επόμενη γραμμή και του codelet που ορίζεται στην γραμμή 2.
Αν ο CUDA επιταχυντής υλικού είναι διαθέσιμος τότε μεταφέρονται τα δεδομένα και το codelet εκτελείται συγχρονισμένα. Σε περίπτωση που ο προγραμματιστής δεν επιθυμεί το codelet να εκτελεστεί συγχρονισμένα υπάρχουν ειδικές οδηγίες οι οποίες το εκτελούν ασύγχρονα. Αν ο επιταχυντής υλικού δεν είναι διαθέσιμος ή η εκτέλεση αποτύχει, τότε το συγκεκριμένο κομμάτι του κώδικα εκτελείται στην CPU.

6.1.4 Αξιολόγηση

Η CUDA και η OpenCL έχουν ένα παρόμοιο μοντέλο προγραμματισμού. Ο προγραμματιστής για να είναι σε θέση να γράψει ένα πρόγραμμα σε CUDA ή OpenCL, πρέπει να γνωρίζει πολύ καλά τις βιβλιοθήκες και τις συναρτήσεις, που παρέχει η κάθε μια, καθώς επίσης και τον τρόπο που γράφεται ο κώδικας του host και ο κώδικας του device. Στον host κώδικα πρέπει να ακολουθούνται τα βήματα που αναφέρονται πιο πάνω, και ο προγραμματιστής πρέπει να είναι σε θέση να τα υλοποίει, χρησιμοποιώντας τις διάφορες βιβλιοθήκες και συναρτήσεις τις κάθε μιας, ανάλογα με το πρόβλημα που έχει να αντιμετωπίσει. Στον κώδικα για το device, ορίζεται η συνάρτηση kernel, όπου ο προγραμματιστής πρέπει να γνωρίζει πώς ορίζονται οι διαστάσεις ενός μπλοκ, με ποιο τρόπο γίνεται η διαχείριση των νημάτων ενός μπλοκ, και με ποιο τρόπο πρέπει να χωρίζονται τα δεδομένα στα μπλοκ ώστε να λυθεί το πρόβλημα σωστά, καθώς επίσης και τις διάφορες λέξεις κλειδιά. Στον πιο κάτω πίνακα 6.3 επισημαίνονται οι διαφορές και οι ομοιότητας, για τον κώδικα του host σε CUDA και OpenCL.

Το προγραμματιστικό μοντέλο της HMPP, διαφέρει κατά πολύ από αυτό της CUDA και της OpenCL. Ο προγραμματιστής, πρέπει να είναι σε θέση να αντιλαμβάνεται, ποια σημεία του σειριακού κώδικα χρειάζεται να εκτελεστούν παράλληλα και με την καλή γνώση απλών οδηγιών της HMPP είναι σε θέση να γράψει ένα HMPP πρόγραμμα.

Στο θέμα της συγγραφής κώδικα, για την υλοποίηση ενός αριθμητικά απαιτητικού υβριδικού αλγορίθμου, η HMPP είναι η πιο προσυτή προς τον προγραμματιστή. Πιο κάτω παρουσιάζονται κάποιες ομοιότητες και κάποιες διαφορές στους τύπους μνήμης της CUDA και της OpenCL, και των βασικών χαρακτηριστικών των προγραμματιστικών τους μοντέλων

CUDA	OpenCL
Συνάρτηση kernel	Συνάρτηση kernel
Κώδικας για Host	Κώδικας για Host
Grid	ND Range
Thread	work - items
Block	work- group

Πίνακας 6.1 Βασικά χαρακτηριστικά προγραμματιστικού μονέλο σε CUDA και OpenCL

CUDA Memory Types	OpenCL Memory Types
Global memory	Global memory
Constant memory	Constant memory
Shared memory	Local memory
Local memory	Private memory

Πίνακας 6.2 Τύποι μνήμης της CUDA και αντίστοιχοι τύποι μνήμης της OpenCL

CUDA	OPENCL
ΡΥΘΜΙΣΕΙΣ	
	Αρχικοποίηση Πλατφόρμας Λίστα με διαθέσιμα devices της πλατφόρμας Επιλογή των devices Δημιουργία Context Δημιουργία της Command Queue
ΡΥΘΜΙΣΕΙΣ ΣΤΗΝ HOST ΚΑΙ DEVICE ΜΝΗΜΗ	
Δέσμευση της μνήμης του host Δέσμευση της μνήμης του devices για είσοδο Αντιγραφή δεδομένων της μνήμη του host στη μνήμη του device Δέσμευση της μνήμης του devices για έξοδο	Δέσμευση της μνήμης του host Δέσμευση της μνήμης του devices για είσοδο Αντιγραφή δεδομένων της μνήμη του host στη μνήμη του device Δέσμευση της μνήμης του devices για έξοδο
ΑΡΧΙΚΟΠΟΙΗΣΗ ΤΟΥ KERNEL	
	Φόρτωση του κώδικα του kernel Δημιουργία αντικειμένου του προγράμματος Κατασκεύασε το πρόγραμμα Δημιουργία του αντικειμένου για kernel και Δέσμευση του στην συνάρτηση του kernel
ΕΚΤΕΛΕΣΗ ΤΟΥ KERNEL	
	Καθορισμός των παραμέτρων του kernel
Ρύθμιση της διαμόρφωσης του device Κλήση του kernel	Ρύθμιση της διαμόρφωσης του device Κλήση του kernel
ΑΝΤΙΓΡΑΦΗ ΑΠΟΤΕΛΕΣΜΑΤΟΣ ΣΤΗΝ ΜΝΗΜΗ ΤΟΥ HOST	
Αντιγραφή δεδομένων της μνήμη του device στη μνήμη του host	Αντιγραφή δεδομένων της μνήμη του device στη μνήμη του host
ΕΚΚΑΘΑΡΙΣΗ	
Εκκαθάριση όλων των παραπάνω που έχουν συσταθεί	Εκκαθάριση όλων των παραπάνω που έχουν συσταθεί

Πίνακα 6.3 Κώδικας για host, της CUDA και OPENCL

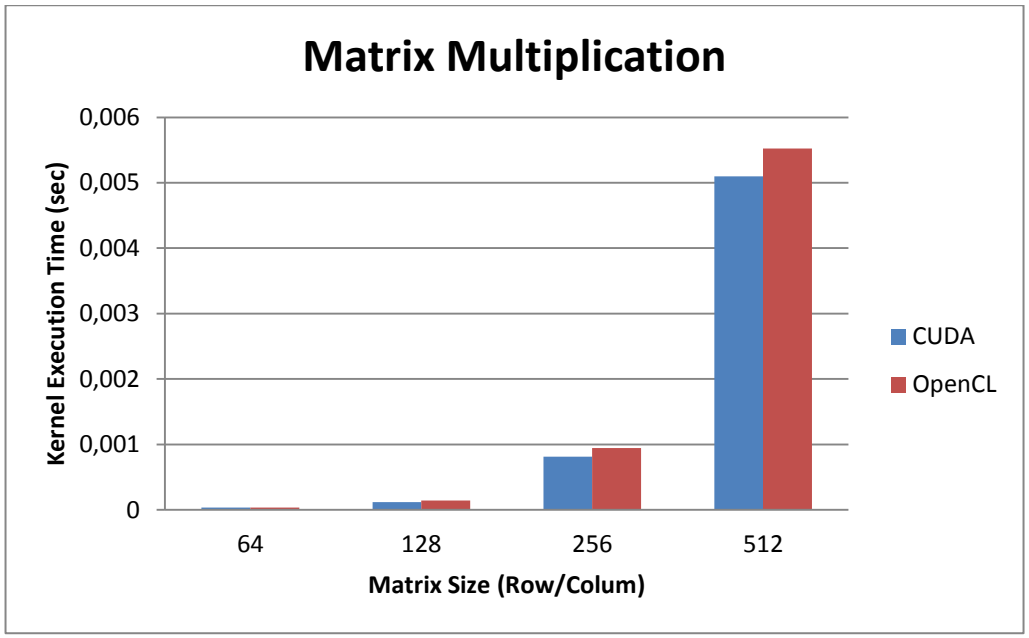
6.2 Άλλοι τομείς αξιολόγηση

Σε αυτήν την ενότητα θα παρουσιαστούν τα χαρακτηριστικά της κάθε γλώσσας, σε κάθε μια από τις πιο κάτω κατηγορίες.

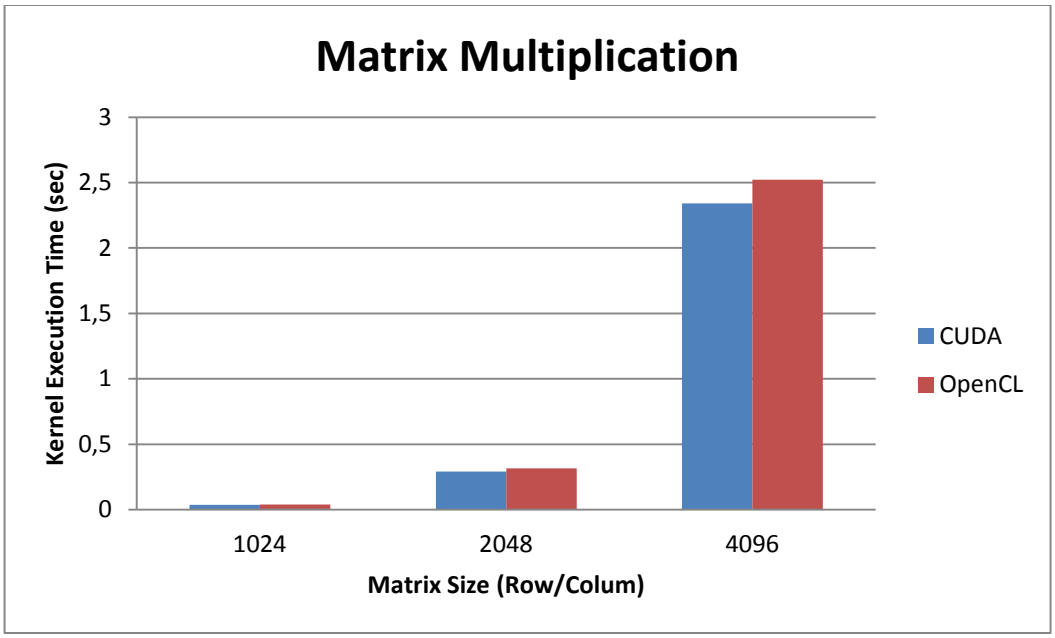
Επίδοση

Για την αξιολόγηση των επιδόσεων της CUDA και της OpenCL έχουν χρησιμοποιηθεί τέσσερις αλγόριθμοι, οι οποίοι θα παρουσιαστούν πιο κάτω.

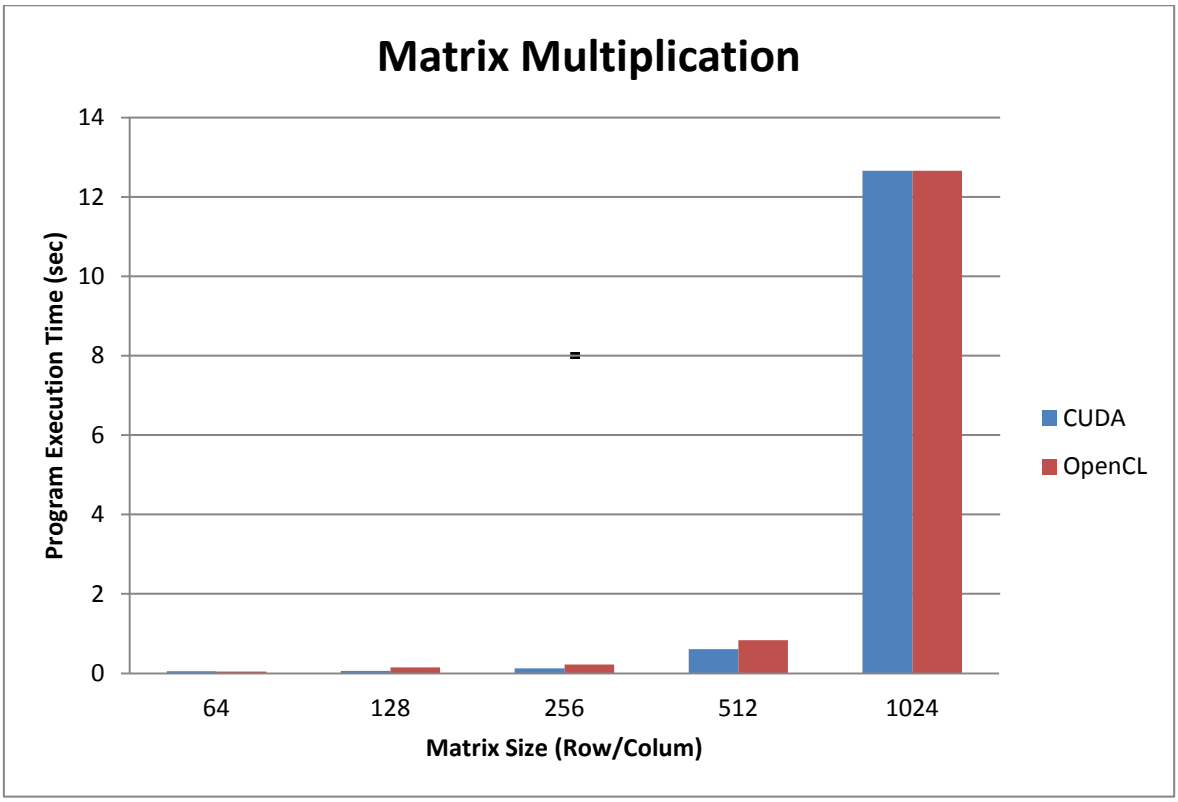
1. **Πολλαπλασιασμός Πινάκων** : Το κλασικό πρόβλημα του πολλαπλασιασμού πινάκων το οποίο αναλύεται στην ενότητα 5.1



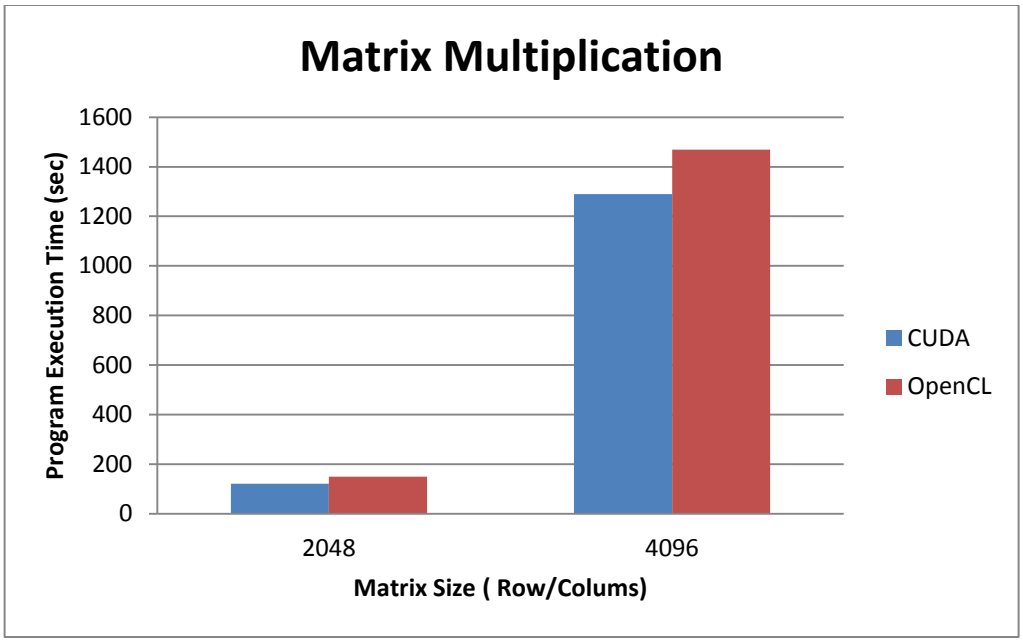
Χρόνοι εκτέλεσης της συνάρτησης kernel



Χρόνοι εκτέλεσης της συνάρτησης kernel

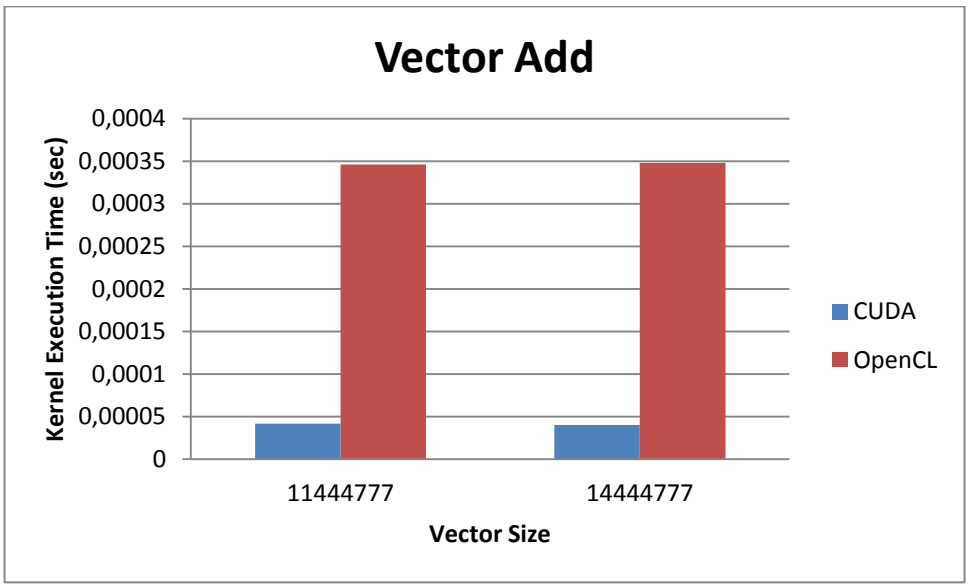


Χρόνοι εκτέλεσης του όλου προγράμματος - CPU και GPU

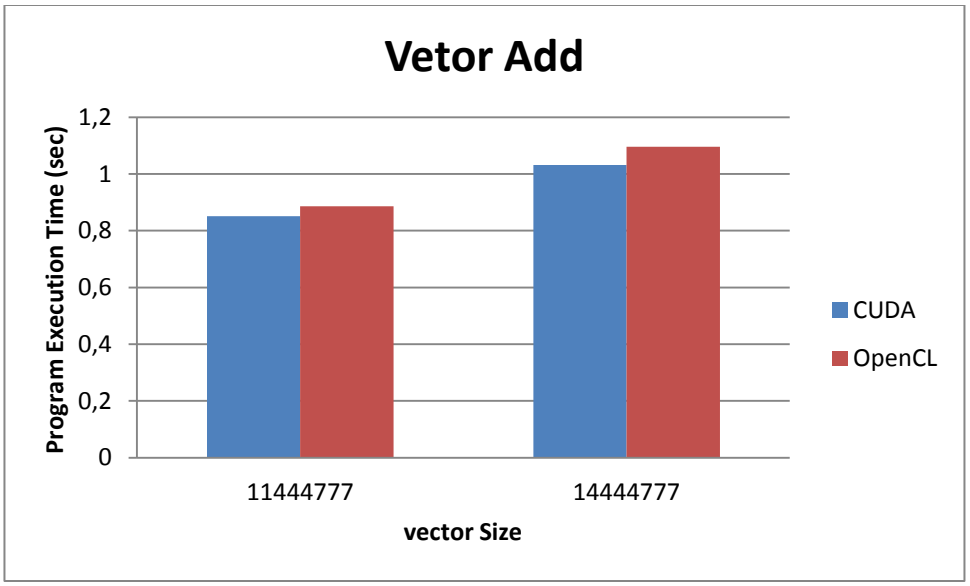


Χρόνοι εκτέλεσης του όλου προγράμματος - CPU και GPU

2. **Πρόσθεση Διανυσμάτων** : Στο παράδειγμα αυτό γίνεται πρόσθεση δύο διανυσμάτων. C = A+B , όπου τα C , A και B είναι διανύσματα [20].

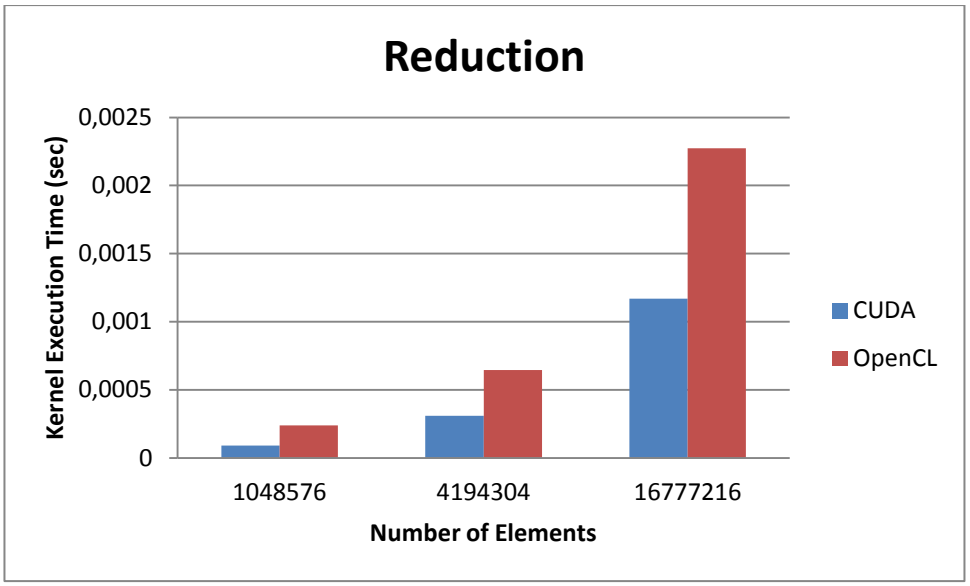


Χρόνοι εκτέλεσης της συνάρτησης kernel

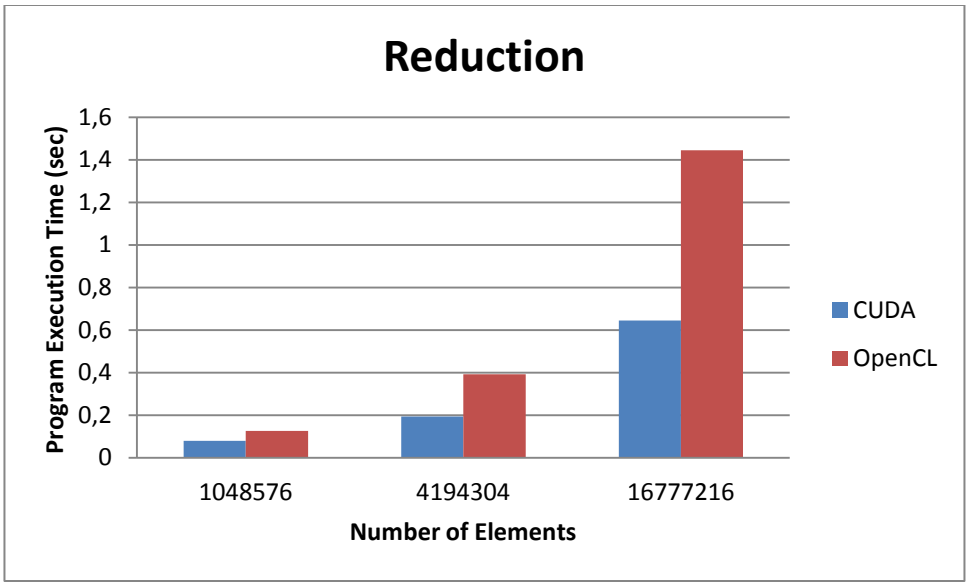


Χρόνοι εκτέλεσης του όλου προγράμματος - CPU και GPU

3. **Reduction** : Στο παράδειγμα αυτό χρησιμοποιείται ένα μονοδιάστατος πίνακας και γίνεται πρόσθεση όλων των στοιχείων του και στο τέλος επιστρέφεται αυτή το άθροισμα αυτό [30].

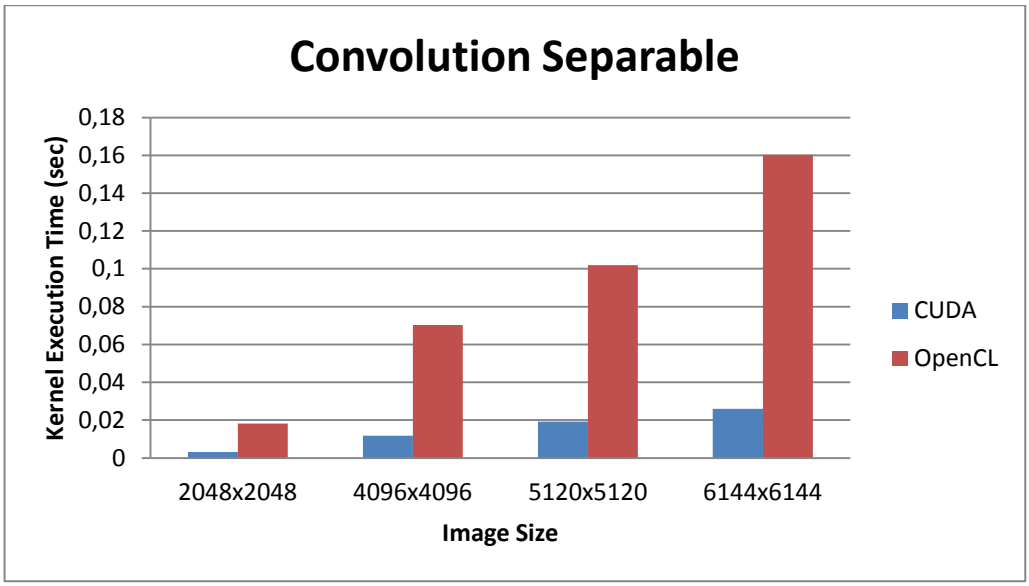


Χρόνοι εκτέλεσης της συνάρτησης kernel

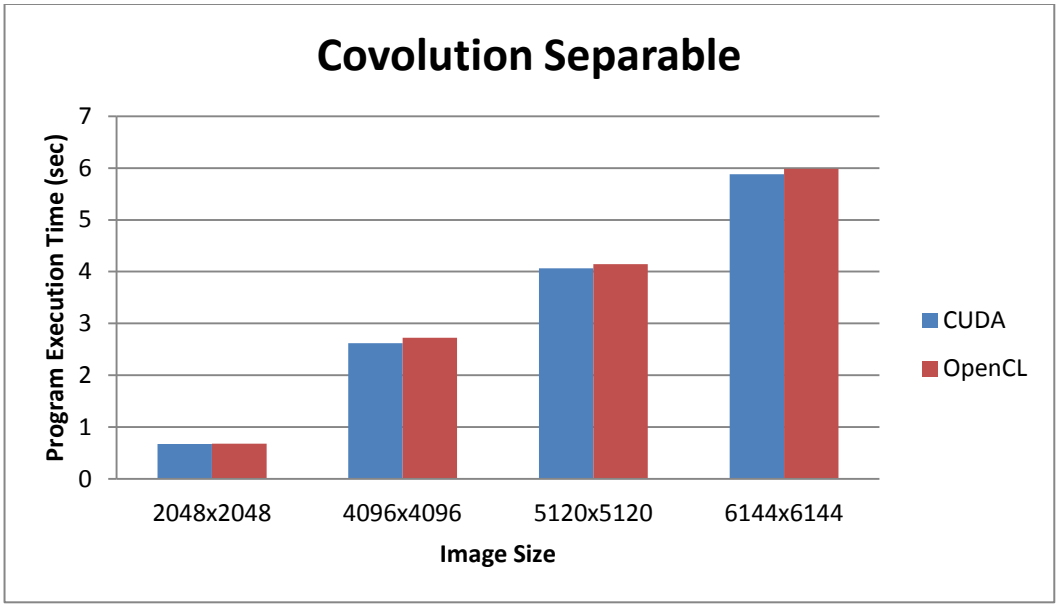


Χρόνοι εκτέλεσης του όλου προγράμματος - CPU και GPU

4. **Convolution Separable** : Στο παράδειγμα αυτό γίνεται συνέλιξη σε μια εικόνα με δομικό στοιχείο ένα πίνακα 3x3 [29].



Χρόνοι εκτέλεσης της συνάρτησης kernel



Χρόνοι εκτέλεσης του όλου προγράμματος - CPU και GPU

Η κάρτα γραφικών που χρησιμοποιήθηκε για να πάρουμε τις πιο πάνω μετρήσεις είναι η NVIDIA GeForce 8800 GTS. Από τις πιο πάνω γραφικές παραστάσεις, παρατηρούμε ότι η CUDA και η OpenCL έχουν παρόμοιους χρόνους εκτέλεσης τόσο της συνάρτησης kernel όσο και όλου του προγράμματος. Η CUDA όμως σε όλες τις πιο πάνω περιπτώσεις επιτυγχάνει ελαφρώς καλύτερες επιδόσεις από την OpenCL. Σε άλλες έρευνες οι οποίες έχουν γίνει για την σύγκριση των επιδόσεων της CUDA και της OpenCL, όπου μερικές από αυτές παρουσιάζονται στο Κεφάλαιο 2, έχουν βρει τα ίδια αποτελέσματα. Ότι δηλαδή η CUDA έχει καλύτερες επιδόσεις από την OpenCL.

Κοινωνία

Η CUDA και η OpenCL είναι ήδη καθιερωμένες και δημοφιλής για τις υψηλές επιδόσεις που παρέχουν. Έχουν και οι δύο μια μεγάλη και επαρκή κοινότητα χρηστών, όπου οι χρήστες μπορούν να βρουν απαντήσεις σε οποιαδήποτε ερωτήματα τους απασχολούν, τόσο σε προγραμματιστικό επίπεδο όσο και θέματα αρχιτεκτονικής, μέσα από τα online φόρουμ. Όσον αφορά την HMPP, λόγω του ότι βρίσκεται πιο μικρό χρονικό διάστημα στην αγορά από τις άλλες δύο, δεν είναι και τόσο διαδεδομένη και η κοινότητα των χρηστών είναι κατά πολύ μικρότερη. Η HMPP δεν έχει φόρουμ, έτσι οι χρήστες δεν έχουν την δυνατότητα να συνομιλούν με άλλους χρήστες και να βρίσκουν απαντήσεις σε διάφορα ερωτήματα τους. Με αυτή την προσέγγιση η HMPP δείχνει ότι η κοινότητα χρηστών, στην οποία στοχεύει είναι οι επιχειρήσεις.

Υποστήριξη

Η CUDA και η OpenCL διατίθενται δωρεάν, και για τον λόγω αυτό δεν παρέχεται πλήρη υποστήριξη στους χρήστες. Η HMPP από την άλλη παρέχει μια πλήρη υποστήριξη στους χρήστες της, και η ομάδα υποστήριξης της είναι πάντα διαθέσιμη για να βοηθήσει με τον καλύτερο δυνατό τρόπο τους χρήστες, στα διάφορα προβλήματα που αντιμετωπίζουν.

Τεκμηρίωση

Η CUDA έχει τις καλύτερες διαθέσιμες δημοσιεύσεις και έρχεται με ένα ευρύ φάσμα από δείγματα σε κώδικες (GPU Computing SDK). Αυτό δίνει την δυνατότητα στους χρήστες της για καλύτερη και ποιο βαθιά κατανόηση των τεράστιων δυνατοτήτων της CUDA. Η OpenCL έχει και αυτή με τη σειρά της αρκετές διαθέσιμες δημοσιεύσεις, αλλά όχι σε τέτοιο βαθμό όπως η CUDA. Η HMPP από την άλλη, εξελίσσεται με πολύ γρήγορους ρυθμούς. Οι διαθέσιμές δημοσιεύσεις για αυτή, καθώς επίσης και τα παραδείγματα

κώδικα τα οποία σου παρέχει μπορεί να μην είναι πολλά σε όγκο, σε σχέση και με τις άλλες δύο, αλλά είναι πάντα επίκαιρα και χρήσιμα.

Φορητότητα

Η OpenCL και η HMPP Workbench έχουν τις περισσότερες δυνατότητες φορητότητας αφού υποστηρίζεται από AMD, ATI και NVIDIA GPUs. Η HMPP Workbench σήμερα υποστηρίζει τις αρχιτεκτονικές CUDA και OpenCL, και ανακοίνωσε ότι θα συνεργαστεί με την Intel, για υποστήριξη και από Pthreads. Η CUDA από τον Δεκέμβριο του 2011 έγινε και αυτή ανοικτό πρότυπο και δεν υποστηρίζεται πλέον μόνο από NVIDIA GPUs. Λόγω του ότι έχουν περάσει μόνο λίγοι μήνες από την αλλαγή αυτή της CUDA, δεν έχουν γίνει αρκετές έρευνες για να μπορούμε να αναφερθούμε στα θετικά και στα αρνητικά της σε ότι αφορά την φορητότητα και αν είναι καλύτερη από τις άλλες δύο.

Ευελιξία

Χρησιμοποιώντας την CUDA και την OpenCL, έχεις την ελευθερία να γράψεις κώδικα για τον χειρισμό του επιταχυντή του υλικού, και επίσης μπορείς να έχεις τον πλήρη έλεγχο του. Η HMPP Workbench υλοποιεί μια εξελιγμένη έννοια της διαχείρισης των δεδομένων.

Μελλοντική Διαθεσιμότητα

Μια αξιόπιστη πρόγνωση δεν είναι δυνατόν να γίνει. Αλλά οι μελλοντικές προοπτικές και για τρείς αυτές γλώσσες προγραμματισμού είναι καλές. Η CUDA και η OpenCL είναι ήδη πολύ καλά καθιερωμένες για τις υψηλές επιδόσεις που παρέχουν. Η HMPP Workbench ήδη σχεδιάζει να εξελιχτεί και με την συνεργασία της με την Intel, και ασχολείται ολοένα και με περισσότερες μελέτες.

	CUDA	OpenCL	HMPP
Επίδοση	+		
Κοινωνία	++	++	-
Υποστήριξη	-	-	+
Τεκμηρίωση	++	+	-
Φορητότητα	-	++	++
Ευελιξία	++	++	-
Μελλοντική Διαθεσημότητα	+	+	+

Πίνακας αξιολόγησης των γλωσσών προγραμματισμού

Κεφάλαιο 7

Συμπεράσματα

7.1 Συμπεράσματα Διπλωματικής Εργασίας

Οι γλώσσες προγραμματισμού των μονάδων επεξεργασίας γραφικών, στην σημερινή τους μορφή είναι πολύ προσιτές προς τον προγραμματιστή. Αυτό επειδή, τους δίνουν την δυνατότητα να χρησιμοποιούν ευρέως γνωστές γλώσσες σειριακού προγραμματισμού, με κάποιες επεκτάσεις, ως υψηλού επιπέδου γλώσσες προγραμματισμού, κάτι που παλαιότερα δεν ίσχυε.

Οι μονάδες επεξεργασίας γραφικών, για γενικού σκοπού εφαρμογές, και οι γλώσσες προγραμματισμού τους, χρίζουν ύψιστης σημασίας, λόγω της υψηλής υπολογιστικής τους ισχύς, που μας δίνουν την δυνατότητα να επιλύσουμε μια πληθώρα σημαντικών προβλημάτων. Για παράδειγμα, πολλές μαθηματικές πράξεις που από την φύση τους έχουν το στοιχείο παραλληλισμού, ιδανικές για την επίλυση τους από τους πολλούς πυρήνες των μονάδων επεξεργασίας γραφικών, επιλύονται ολοένα και πιο γρήγορα. Αυτό έχει σαν αποτέλεσμα πιο γρήγορη εξόρυξη δεδομένων, για προβλήματα

τα οποία βασίζονται στην επίλυση μαθηματικών εξισώσεων, όπως για παράδειγμα στον σχεδιασμό φαρμάκων, στην πυρηνική επιστήμη, την αστρονομία κ.α.

Επίσης, εκτός από τις χρήσεις που αναφέρονται πιο πάνω, ακόμη και σε προσωπικούς υπολογιστές, χρησιμοποιούνται για τον υπολογισμό πολλών πράξεων που παραδοσιακά εκτελούνταν από την κεντρική μονάδα επεξεργασίας. Αυτό επιτρέπει στους επεξεργαστές να μπορούν να λειτουργούν σε πιο χαμηλή συχνότητα, και γενικά να έχουν μειωμένο φόρτο εργασίας, όταν οι GPUs είναι διαθέσιμες.

Δεν μπορούμε να απαντήσουμε με σιγουριά στην ερώτηση ποια από τις γλώσσες προγραμματισμού για GPUs είναι η καλύτερη. Στα παραδείγματα που μελετήθηκαν στα πλαίσια αυτής της διπλωματικής εργασίας, δηλαδή το πρόβλημα πολλαπλασιασμού πινάκων, της πρόσθεσης διανυσμάτων, της πρόσθεσης των στοιχείων ενός μονοδιάστατου πίνακα και της συνέλιξης μιας εικόνας, σε όλες τις περιπτώσεις η CUDA είχε καλύτερες επιδόσεις από την OpenCL, αλλά δεν σημαίνει ότι είναι η καλύτερη για όλες τις εφαρμογές. Ανάλογα με το είδος του προβλήματος που έχει ένας προγραμματιστής να επιλύσει, με τον αν θέλει η εφαρμογή του να είναι φορητή και να εκτελείται σε διάφορα GPUs, αλλά και τις γνώσεις που έχει για κάθε μια από τις γλώσσες, θα επιλέξει την κατάλληλη.

7.2 Προτάσεις για Μελλοντική Εργασία

Λόγω της συνεχούς ανάπτυξης των μονάδων επεξεργασίας γραφικών, και αναπόφευκτα της εξέλιξη των γλωσσών προγραμματισμού τους, θα πρέπει οι συγκεκριμένες γλώσσες να τυγχάνουν συνεχής μελέτη και αξιολόγησης.

Όπως αναφέρεται και στο κεφάλαιο 5, στην ενότητα 5.2, όσο αφορά την συνεργασία της HMPP Workbench με την Intel, για την υποστήριξη των Pthreads, θα πρέπει να γίνει μια ανάλυση των δυνατοτήτων της και σύγκριση της με τις άλλες γλώσσες.

Επίσης, η CUDA πλέον προσφέρεται και σε ανοικτό πρότυπο, και υποστηρίζεται και από GPUs άλλων κατασκευαστών, πέρα από αυτούς της NVIDIA. Οπότε χρειάζεται να γίνει και μια μελέτη, όσο αφορά τον τρόπο προγραμματισμού της καθώς και οι επιδόσεις της σε σχέση με τα GPUs της NVIDIA, που εξετάστηκαν σε αυτή την διπλωματική εργασία, και με αυτά των άλλων κατασκευαστών.

Βιβλιογραφία

[1] WIKIPEDIA - Graphics Processing unit (http://en.wikipedia.org/wiki/Graphics_processing_unit#Hybrid_solutions)

[2] ΒΙΚΙΠΑΙΔΕΙΑ - ΚΑΡΤΑ ΓΡΑΦΙΚΩΝ http://el.wikipedia.org/wiki/%CE%9A%CE%AC%CF%81%CF%84%CE%B1_%CE%B3%CF%81%CE%B1%CF%86%CE%B9%CE%BA%CF%8E%CE%BD

[3] Top500 list of world's faster supercomputers (http://www.top500.org/list/2011/11/100)

[4] T. S. Crow, "Evolution of the Graphical Processing Unit," M.S. thesis, University of Nevada, Reno, 2004

[5] P. Lilly, "From Voodoo to GeForce: The Awesome History of 3D Graphics", May 2009, [Online].Available: http://www.maximumpc.com/article/features/graphics_extravaganza_ultimate_gpu_retrospective

[6] J. Nickolls and W.J Dally, "The GPU Computing Era" , Proceeding of the IEEE Computer society, Vol.30, 2010

[7] Whitepaper, "NVIDIA's Next Generation CUDA Compute Architecture" Fermi , 2010

[8] "NVIDIA GeForce 8800 GPU Archtecture Overview", Technical Brief, NVIDIA, November 2006.

[9] WIKIPEDIA-GPGPU (http://en.wikipedia.org/wiki/GPGPU)

[10] J.D. Owens, M. Houston, D. Luebke, S. Green, J.E. Stone, and J.C. Phillips, “GPU Computing”, in *Proceeding of the IEEE*, Vol.96, Issue: 5, May 2008

[11] I. Buck, T. Foley, D. Horn, J. Sugerman, K. Fatahalian, M. Houston and P. Hanrahan, “Brook for GPUs: Stream Computing on Graphics Hardware”, Stanford University

[12] JJohn D. Owens, David Luebke, Naga Govindaraju, Mark Harris, Jens Krüger, Aaron E. Lefohn, and Timothy J. Purcell. “A Survey of General-Purpose Computation on Graphics Hardware.” In *Eurographics 2005, State of the Art Reports*, August 2005, pp. 21-51. Eurographics 2005

[13] NVIDIA CUDA C Programming Guide, Version 3.2

[14] Antonio Tumeo, “Massively Parallel Computing with CUDA”, NVIDIA

[15] OPENCL Tutorials 1 - Quick start

[16] Anand Lal Shimpi and Deker Wilson , “NVIDIA’s GeForce 8800(G80): GPUs Re-archritected for DirectX 10” , 2006, [Online].Available: http://www.anandtech.com/show/2116

[17] OpenCL Programming Guide for the CUDA Architecture, Version 3.1, 2010

[18] Romain Dolbeau, Stephane Bihan and Francois Bodin “HMPP : A Hybrid Multi-core Parallel Programming Enviroment,” CAPS enterprise.

[19] “Code Acceleration with HMPP,” Irish Centre for High-End Computing (ICHEC), Outcomes from HMPP training

[20] NVIDIA GPU Computing SDK

[21] R. Merritt, OpenCL gets upgrade, Altera tips FPGA tool

[22] “MotionDSP Taps AMD and OpenCL Industry Standard for Fast Processing of Ikena ISR Video Software,” *AMD heterogeneous computing technology enables real-time “CSI-style” video reconstruction to support public safety,* SUNNYVALE, Calif. —10/20/2011 .

[23] P. Plaszewski, K. Banas and P. Maciol, “Higher order FEM numerical integration on GPUs with OpenCL”,in *Proceeding of the IMCSIT*, Vol. 5, 2010

[24] P. Du, P. Luszczek and J. Dongarra, “OpenCL Evaluation for Numerical Linear Algebra Library Development”, University of Tennessee Innovative Computing Laboratory, Oak Ridge National Laboratory, University of Manchester, Tech. Rep.

[25] R. Membarth, F. Hannig, J. Teich, M. Korner and W. Eckert, “Frameworks for GPU Accelerators:A Comprehensive Evaluation using 2D/3D Image Registration”, in *Proceedings of the 9th IEEE Symposium on Application Specific Processors (SASP)*, San Diego, CA, USA, 5-6 June, 2011

[26] J. Fang, A. L. Varbanescu and H. Sips, “A Comprehensive Performance Comparison of CUDA and OpenCL”, Parallel and Distriduted Systems Group, Delft University of Technology, Tech. Rep.

[27] K. Karimi, N. G. Dickson and F. Hamze, “A Performance Comparison of CUDA and OpenCL”, May 2010

[28] K. Komatsu1, K. Sato, Y. Arai, K. Koyama, H. Takizawa, and H. Kobayashi1, “Evaluating Performance and Portability of OpenCL Programs,” in *Proceedings of the Fifth international Workshop on Automatic Performance Tuning(iWAPT2010)*, (Berkeley, USA), June 2010

[29] V. Podlozhnyuk, “Image Convolution with CUDA”, NVIDIA GPU Computing SDK, June 2007

[30] M. Harris, “Optimizing Parallel Reduction in CUDA”, NVIDIA GPU Computing SDK, June 2007