ΠΑΝΕΠΙΣΤΗΜΙΟ ΘΕΣΣΑΛΙΑΣ

ΠΟΛΥΤΕΧΝΙΚΗ ΣΧΟΛΗ

Τμήμα Ηλεκτρολόγων Μηχανικών

& Μηχανικών Υπολογιστών

Σύστημα χρόνου εκτέλεσης για ετερογενείς υπολογιστικές πλατφόρμες

Runtime system for heterogenous computing platforms

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

Δήμτσας Αντώνιος

Επιβλέποντες καθηγητές: Χρήστος Δ.Αντωνόπουλος

Επίκουρος Καθηγητής

Νικόλαος Μπέλλας

Αναπληρωτής Καθηγητής

Βόλος ,Ιούνιος 2013

ΠΑΝΕΠΙΣΤΗΜΙΟ ΘΕΣΣΑΛΙΑΣ

ΠΟΛΥΤΕΧΝΙΚΗ ΣΧΟΛΗ

Τμήμα Ηλεκτρολόγων Μηχανικών & Μηχανικών Υπολογιστών

Σύστημα χρόνου εκτέλεσης για ετερογενείς υπολογιστικές πλατφόρμες

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

Δήμτσας Αντώνιος

Επιβλέποντες καθηγητές: Χρήστος Δ. Αντωνόπουλος

Επίκουρος Καθηγητής

Νικόλαος Μπέλλας

Αναπληρωτής Καθηγητής

Εγκρίθηκε από τη διμελής εξεταστική επιτροπή την 01/07/2013

…….	…….
Χρήστος .Δ Αντωνόπουλος	Νικόλαος Μπέλλας
Επίκουρος Καθηγητής	Αναπληρωτής Καθηγητής

Διπλωματική Εργασία για την απόκτηση του Διπλώματος του Μηχανικού Ηλεκτρονικών Υπολογιστών, Τηλεπικοινωνιών και Δικτύων του Πανεπιστημίου Θεσσαλίας, στα πλαίσια του Προγράμματος Προπτυχιακών Σπουδών του Τμήματος Μηχανικών Η/Υ, Τηλεπικοινωνιών και Δικτύων του Πανεπιστημίου Θεσσαλίας.

……..

Δήμτσας Αντώνιος

Διπλωματούχος Μηχανικός Τμήμα Ηλεκτρολόγων Μηχανικών και Μηχανικών Υπολογιστών
Πανεπιστημίου Θεσσαλίας

Στην οικογένεια και στους φίλους μου

Ευχαριστίες

Θέλω να ευχαριστήσω τους επιβλέποντες καθηγητές μου κ. Αντωνόπουλο και κ. Μπέλλα για την άψογη συνεργασία που είχαμε τα τελευταία 2 χρόνια των σπουδών μου. Για την εμπιστοσύνη που έδειξαν απέναντι στο πρόσωπο μου κατά τη διάρκεια της συνεργασίας μας και την πολύτιμη βοήθεια τους για την πρόοδο των σπουδών μου. Ευχαριστώ επίσης τα παιδιά του εργαστηρίου Β4 που με στήριξαν έμμεσα ή άμεσα κατά την εκπόνηση της διπλωματικής εργασίας μου, αλλά και για την συνεργασία που είχαμε μεταξύ μας τον τελευταίο χρόνο. Κυρίως θέλω να ευχαριστήσω το συμφοιτητή μου Σπύρου Μιχάλη για την στενή συνεργασία που είχαμε για την εκπόνηση της διπλωματικής μου εργασίας, για την καθοριστική βοήθεια και στήριξη του τα τελευταία 2 χρόνια για την ολοκλήρωση αυτής της εργασίας.

Ένα μεγάλο ευχαριστώ στους φίλους μου για τη στήριξη τους όλα αυτά τα χρόνια,μπροστά σε όλες τις δυσκολίες κατά τη διάρκεια των σπουδών μου. Χάρη σε αυτούς τα ατέλειωτα βράδια διαβάσματος, έγιναν ωραίες νύχτες μουσικής, παρέας, συζήτησης και στοχασμού. Μαζί τους οι λύπες μοιράζονταν και οι χαρές πολλαπλασιάζονταν.

Τέλος ευχαριστώ την οικογένεια μου, που από τότε που θυμάμαι τον εαυτό μου με στηρίζει. Ευχαριστώ τους γονείς μου που μου έμαθαν να παίρνω αποφάσεις στις ζωή μου,όσο δύσκολες και τολμηρές αν είναι. Ευχαριστώ για την στήριξη σας κατά τη διάρκεια των σπουδών μου, που χωρίς αυτή δεν ξέρω αν γραφόταν αυτές οι σελίδες σήμερα.

Αντώνιος Δήμτσας

Βόλος,2013

Περιεχόμενα

Κατάλογος Συντομογραφιών

API(Application Programming Interface)

CAL(Compute Abstract Level)

CPU(Central Process Unit)

DSP(Digital Signal Processing)

ELF(Executable Link File)

FPGA(Field Programmable Gate Array)

GPU(Graphic Process Unit)

IL(Intermediate Language)

ISA(Instruction Set Architecture)

JIT(Just In Time Compiler)

PE(Process Element)

RTS(Run Time System)

SIMD(Single Instruction Multiple Data)

Κεφάλαιο 1

Εισαγωγή

1.1 Περιγραφή Προβλήματος

Τα τελευταία χρόνια η συνεχής ανάπτυξη της τεχνολογίας στον τομέα των υπολογιστικών συστημάτων, όπως είναι η αύξηση της συχνότητας ρολογιού και η μείωση των διαστάσεων των transistors στους επεξεργαστές και γενικά στα ψηφιακά κυκλώματα, είχε ως συνέπεια να φτάσουμε σε ένα σημείο όπου η περαιτέρω βελτίωση των επιδόσεων των συστημάτων αυτών να καθίσταται πιο δύσκολη. Περάσαμε επομένως στα συστήματα με περισσότερους επεξεργαστές(multiprocessors), σε συστήματα με περισσότερους πυρήνες σε έναν επεξεργαστή (multicore) και γενικότερα επικράτησε τα τελευταία χρόνια στην αρχιτεκτονική των υπολογιστών η λογική του παραλληλισμού στα υπολογιστικά συστήματα. Ωστόσο οι μεγάλες υπολογιστικές ανάγκες που υπάρχουν σήμερα σε αρκετούς τομείς των επιστημών(βιοτεχνολογία, μετεωρολογία, πυρηνική φυσική κα) λόγω των υψηλών απαιτήσεων σε υπολογιστικές πράξεις(κυρίως με αριθμητική κινητής υποδιαστολής,floating point arithmetic) είχε ως αποτέλεσμα οι αρχιτέκτονες των υπολογιστικών συστημάτων να εντάξουν στα συστήματα αυτά και άλλες μη συμβατικές, μέχρι τότε, συσκευές όπως είναι οι κάρτες γραφικών(GPU's) οι οποίες κατά κύριο λόγω χρησιμοποιούνταν μέχρι τότε για την υποστήριξη του γραφικού περιβάλλοντος των υπολογιστικών συστημάτων.

Ο λόγος που επιλέχτηκαν οι GPUs για τον σκοπό αυτό είναι γιατί είναι αρκετά ισχυρές σε αριθμητική κινητής υποδιαστολής. Έχουμε λοιπόν ετερογενείς αρχιτεκτονικές υπολογιστικών συστημάτων, όπου συνυπάρχουν συμβατικοί επεξεργαστές μαζί με GPUs, με σκοπό να πετύχουμε μεγαλύτερη αύξηση της υπολογιστικής ισχύεις των συστημάτων αυτών, αξιοποιώντας τις διαφορετικές αρχιτεκτονικές των συστημάτων αυτών και του παραλληλισμού που αυτές μπορούν να μας προσφέρουν. Η ιδιαιτερότητα της αρχιτεκτονικής των GPUs ωστόσο κατέστησε δύσκολο την συγγραφή κώδικα από τους προγραμματιστές καθώς ο προγραμματιστής έπρεπε να γνωρίζει αρκετά καλά της αρχιτεκτονική της GPU για να μπορέσει να πετύχει την απόδοση που ήθελε. Αυτή η δυσκολία οδήγησε τους εκάστοτε κατασκευαστές GPUs να δημιουργήσουν το δικό τους προγραμματιστικό μοντέλο, για την διευκόλυνση των προγραμματιστών σε κάρτες γραφικών. Έτσι η NVIDIA ανέπτυξε την CUDA και η AMD το Stream. Το παράδοξο ωστόσο είναι πως αν ένας προγραμματιστής ήθελα να προγραμματίσει σε διαφορετική αρχιτεκτονική GPU θα

έπρεπε να ξέρει και το αντίστοιχο προγραμματιστικό μοντέλο του συγκεκριμένου κατασκευαστεί.

Για το λόγω αυτό το Kronos Group(με τη συνεργασία δεκάδων κατασκευαστών processors και GPUs) ανέπτυξε την OpenCL, ένα προγραμματιστικό μοντέλο το οποίο θα βοηθάει στον προγραμματισμό των GPUs(αλλά και τον processors) και θα είναι κοινό για όλες τις διαφορετικές αρχιτεκτονικές GPUs και επεξεργαστών. Το Kronos Group είναι μια μη κερδοσκοπική κοινοπραξία για τη δημιουργία ανοιχτών προτύπων, για τη συγγραφή και την επιτάχυνση των παράλληλων υπολογιστών, γραφικών, δυναμικά μέσα, όραση υπολογιστών και την επεξεργασία αισθητήρων σε μια ευρεία ποικιλία από πλατφόρμες και συσκευές. Όλα τα μέλη του Khronos είναι σε θέση να συμβάλουν στην ανάπτυξη των Khronos API, έχουν το δικαίωμα να ψηφίσουν σε διάφορα στάδια πριν από τη δημόσια αποστολή, και είναι σε θέση να επιταχύνει την υλοποίηση της αιχμής για 3D πλατφόρμες και τις εφαρμογές τους μέσω της έγκαιρης πρόσβασης σε σχέδια προδιαγραφών και της συμμόρφωσης δοκιμές Με το πρότυπο της OpenCL ωστόσο μπορούμε κάθε φορά να χρησιμοποιούμε έναν τύπο συσκευής κάθε φορά(GPU ή processor) χωρίς να καθίσταται δυνατή η πλήρης αξιοποίηση όλων των συσκευών του υπολογιστικού μας συστήματος.

1.2 Σκοπός της εργασίας

Σκοπός της διπλωματικής εργασίας είναι η δημιουργία ενός συστήματος χρόνου εκτέλεσης(Run Time System) με βάση το API της OpenCL, το οποίο θα αξιοποιεί όλες τις ετερογενείς συσκευές στο σύστημα μας(GPUs & processors) με σκοπό την καλύτερη αξιοποίηση του υπολογιστικού συστήματος. Πιο συγκεκριμένα αναλύουμε τον τρόπο με τον οποίο έχει δομηθεί το εν λόγω σύστημα, πως μπορούν δυο εντελώς διαφορετικές αρχιτεκτονικές, όπως είναι οι κάρτες γραφικών και οι επεξεργαστές, να συνυπάρχουν σε ένα κοινό υπολογιστικό σύστημα για την αύξηση της απόδοσης του, και τι προβλήματα-ζητήματα αντιμετωπίζουμε για την ανάπτυξη του εν λόγω συστήματος. Τέλος μέσα από τη διπλωματική παραθέτονται μια σειρά ζητήματα που πρέπει να αντιμετωπιστούν ώστε να έχουμε την καλύτερη δυνατή επίδοση του εν λόγω συστήματος. Το RTS αναπτύχθηκε για την υποστήριξη AMD GPU με βάση το API OpenCL 1.2.

1.3 Διάρθρωση της Διπλωματικής εργασίας

Το κεφάλαιο 2 αποτελεί μια σύντομη παρουσίαση του προτύπου της OpenCL. Συγκεκριμένα μέσα από το συγκεκριμένο κεφάλαιο παραθέτουμε τα βασικά χαρακτηριστικά του προτύπου της OpenCL,τον τρόπο με τον οποίο το συγκεκριμένο πρότυπο αντιλαμβάνεται την αρχιτεκτονική των GPU’s καθώς και θα παρατεθούν

ορισμένες ομοιότητες του προτύπου της OpenCL με την CUDA βλέποντας πως η OpenCL έχει κρατήσει αρκετά στοιχεία από την CUDA.

Το κεφάλαιο 3 αναλύει τη δομή του Run Time System. Συγκεκριμένα αναδεικνύει τα συστατικά μέρη του συστήματος, τον τρόπο με τον οποίο αυτά είναι δομημένα, πως επικοινωνούν μεταξύ τους. Επίσης παραθέτουμε τις δυσκολίες και τα προβλήματα που αντιμετωπίσαμε κατά την ανάπτυξη του εν λόγω συστήματος και τους λόγους για τους οποίους επιλέχτηκε να δομηθεί με τον τρόπο αυτό το σύστημα μας.

Το κεφάλαιο 4 ασχολείται με θέματα απόδοση του συστήματος μας. Κατά πόσο δηλαδή η απόδοση του συστήματος μας είναι σε μετρήσιμα επίπεδα με το πρότυπο της OpenCL, ποιοι παράγοντες είναι καθοριστικοί για την απόδοση του.

Το κεφάλαιο 5 κάνει μια παρουσίαση των κύριων σημείων της εργασίας και παραθέτει ορισμένες ιδέες για την μελλοντική ανάπτυξη της εργασίας.

Κεφάλαιο 2

Background Issues

2.1 Εισαγωγή στο πρότυπο προγραμματισμού της OpenCL

Η OpenCL είναι ένα προτυποποιημένο, διασυστημικό(cross-platform) API παράλληλου προγραμματισμού που βασίζεται στη γλώσσα C. Είναι σχεδιασμένη με στόχο τη δυνατότητα ανάπτυξης φορητών παράλληλων εφαρμογών για συστήματα με ετερογενείς συσκευές υπολογισμού. Η ανάπτυξη της OpenCL γεννήθηκε από την ανάγκη μιας τυποποιημένης πλατφόρμας ανάπτυξης εφαρμογών υψηλής απόδοσης για την ταχέως αναπτυσσόμενη ποικιλία που παρουσιάζουν οι πλατφόρμες παράλληλης υπολογιστικής. Ειδικότερο υπερβαίνει σημαντικούς περιορισμούς των προηγούμενων μοντέλων προγραμματισμού σε ετερογενή συστήματα παράλληλης υπολογιστικής **[11].**

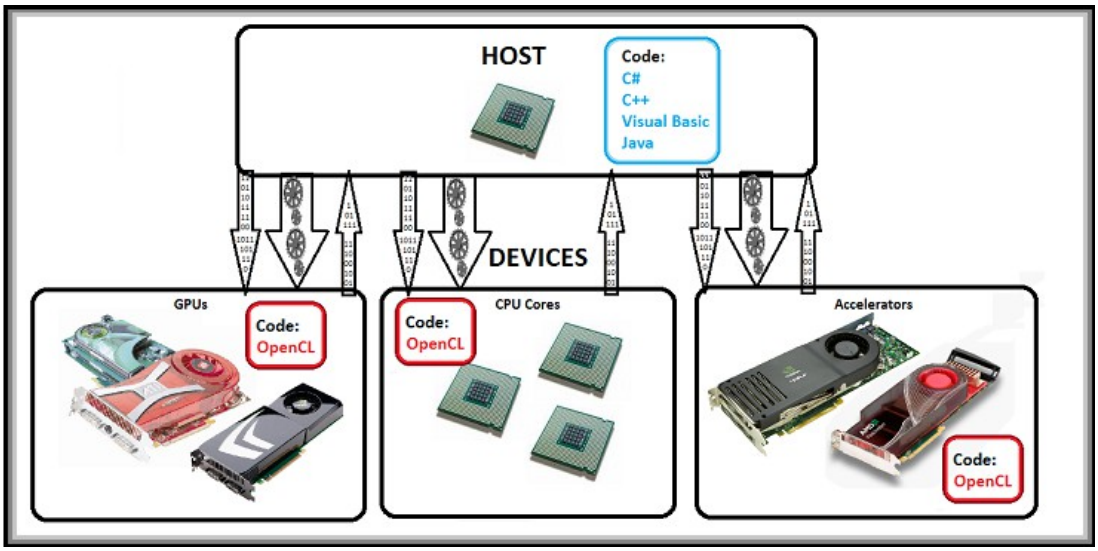
Τα μοντέλα παράλληλου προγραμματισμού για τις CPU τυπικά στηρίζονται σε πρότυπα όπως το OpenMP αλλά συνήθως δεν περιλαμβάνουν τη χρήση ειδικών τύπων μνήμης ή εκτέλεση SIMD από τους προγραμματιστές που παράγουν κώδικα υψηλών επιδόσεων. Συνδυασμένα μοντέλα ετερογενούς παράλληλου προγραμματισμού σε CPU/GPU όπως η CUDA χειρίζονται σύνθετες ιεραρχίες μνήμης και εκτέλεση SIMD αλλά συνήθως είναι εξειδικευμένα σε μια πλατφόρμα, έναν κατασκευαστή, ή συγκεκριμένο υλικό. Αυτό είναι και ένα από τα μειονεκτήματα των μοντέλων που είναι εξειδικευμένα σε συγκεκριμένες πλατφόρμες, καθώς οι προγραμματιστές είναι αναγκασμένοι να μαθαίνουν τα διαφορετικά προγραμματιστικά μοντέλα για κάθε διαφορετικό κατασκευαστή. Επίσης αυτοί οι περιορισμοί δεν επιτρέπουν σε έναν προγραμματιστή εφαρμογών την εύκολη πρόσβαση στην υπολογιστική ισχύ των CPU, των GPU και των άλλων τύπων μονάδων επεξεργασίας από μία μοναδική βάση πολυσυστημικού πηγαίου κώδικα **[11].**

Η OpenCL δανείζεται πολλά στοιχεία της CUDA όσο αφορά την υποστήριξη μίας μοναδικής βάσης κώδικα για ετερογενή παράλληλη υπολογιστική, παραλληλία δεδομένων, και σύνθετες ιεραρχίες μνήμης. Από την άλλη πλευρά η OpenCL διαθέτει ένα πιο σύνθετο μοντέλο διαχείρισης πλατφόρμας και συσκευών το οποίο αντανακλά την υποστήριξη που παρέχει στη φορητότητα μεταξύ συστημάτων και κατασκευαστών. Τα προγράμματα της OpenCL πρέπει να είναι προετοιμασμένα να χειριστούν πολύ μεγαλύτερη ποικιλία υλικού και επομένως παρουσιάζουν μεγαλύτερη πολυπλοκότητα. Επίσης πολλές λειτουργίες της OpenCL είναι προαιρετικές και μπορεί να μην υποστηρίζονται από όλες τις συσκευές, οπότε ένας φορητός κώδικας OpenCL πρέπει να αποφεύγει τη χρήση αυτών των προαιρετικών

λειτουργιών. Μερικές από αυτές τις προαιρετικές εντολές επιτυγχάνουν σημαντικά καλύτερη απόδοση σε συσκευές που τις υποστηρίζουν.

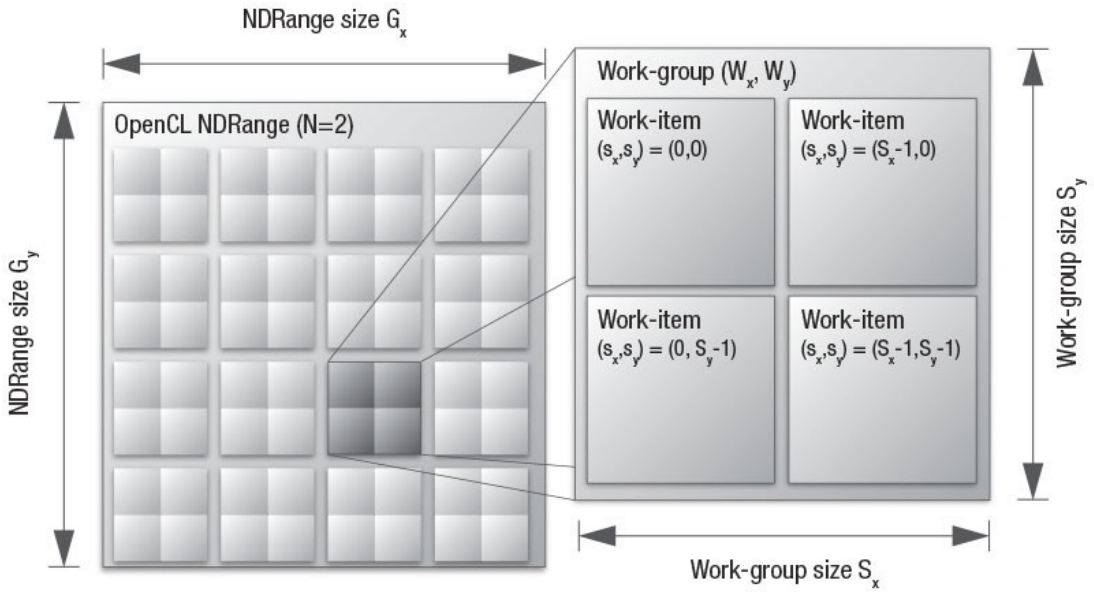
2.2 Μοντέλο Παραλληλίας Δεδομένων της OpenCL

Η OpenCL χρησιμοποιεί ένα μοντέλο παραλληλίας δεδομένων το οποίο παρουσιάζει άμεση αντιστοιχία με το μοντέλο παραλληλίας δεδομένων της CUDA. Ένα πρόγραμμα OpenCL αποτελείται από δυο μέρη: πυρήνες(kernels) που εκτελούνται σε μία ή περισσότερες συσκευές OpenCL και ένα πρόγραμμα υπηρεσίας(Host program) που διαχειρίζεται την εκτέλεση των πυρήνων. Ο τρόπος υποβολής εργασιών για παράλληλη εκτέλεση στην OpenCL είναι να ξεκινήσει το πρόγραμμα υπηρεσίας συναρτήσεις πυρήνα. Το παρακάτω σχήμα 2.1 δίνει μια αφαιρετική άποψη του μοντέλου της OpenCL



Σχήμα 2.1 Αφαιρετικό μοντέλο της OpenCL [13]

Όταν ξεκινάει μια συνάρτηση πυρήνα, ο κώδικας της εκτελείται από στοιχεία εργασίας(work items). Ένας χώρος αριθμοδεικτών(index space) ορίζει τα στοιχεία εργασίας και τον τρόπο αντιστοίχησης των δεδομένων στα στοιχεία αυτά, δηλαδή τα στοιχεία εργασίας της OpenCL προσδιορίζονται με εύρη αριθμοδεικτών καθολικής διάστασης(NDRanges). Τα στοιχεία εργασίας σχηματίζουν ομάδες εργασίας(work groups). Τα στοιχεία εργασίας της ίδιας ομάδας εργασίας μπορούν να συγχρονιστούν χρησιμοποιώντας φράγματα(barriers). Τα στοιχεία εργασίας διαφορετικών ομάδων εργασίας δεν μπορούν να συγχρονιστούν παρά μόνο τερματίζοντας τη συνάρτηση πυρήνα και ξεκινώντας μια νέα. Το σχήμα 2.2 δείχνει τη γενική άποψη του μοντέλου παράλληλης εκτέλεσης της OpenCL



Σχήμα 2.2 Μοντέλο παράλληλης εκτέλεσης της OpenCL [14]

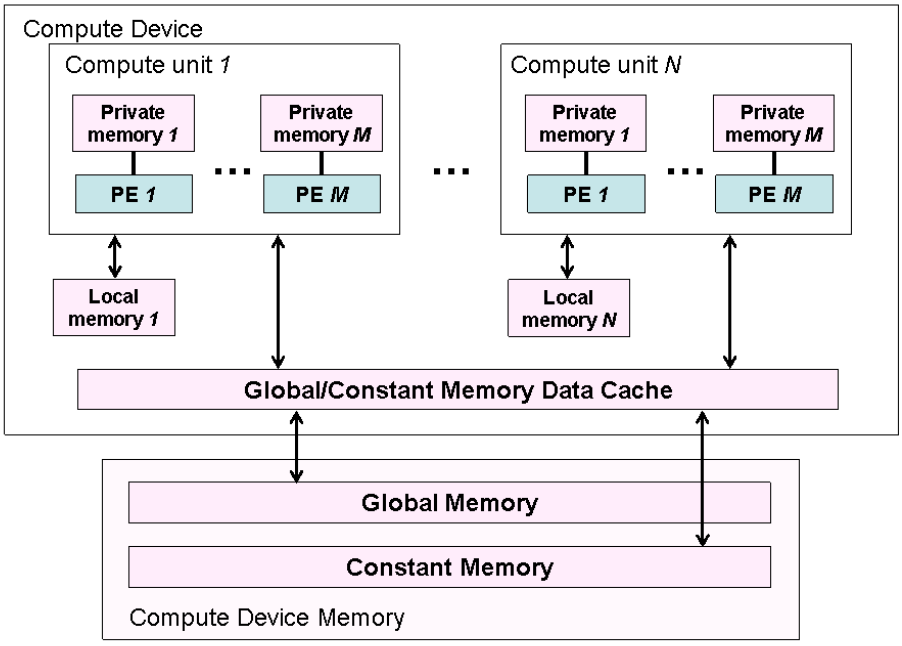
Όλα τα στοιχεία εργασίας έχουν τη δική τους μοναδική τιμή αριθμοδείκτη. Σε έναν πυρήνα OpenCL, ένα νήμα μπορεί να πάρει τις μοναδικές καθολικές τιμές αριθμοδεικτών του καλώντας μια συνάρτηση του API, *get_global_id,* με μια παράμετρο που προσδιορίζει τη διάσταση. Οι κλήσεις των get_global_id(0) και get_global_id(1) επιστρέφουν τις μοναδικές καθολικές τιμές αριθμοδεικτών νήματος στη διάσταση χ και y, αντίστοιχα. Η καθολική τιμή αριθμοδείκτη στη διάσταση χ είναι ισοδύναμη με το blockIdx.x *blockDim.x+threadIDx.x στην CUDA. Ένας πυρήνας OpenCL μπορεί επίσης να καλέσει τη συνάρτηση του API get_global_size() με μια παράμετρο που προσδιορίζει το μέγεθος διαστάσεων των NDRange. Οι κλήσεις get_global_size(0) και get_global_size(1) επιστρέφουν τον συνολικό αριθμό των στοιχείων εργασίας στη διάσταση x και y των NDRanges. Το ισοδύναμο της επιστρεφόμενης τιμής της get_global_size(0) στην CUDA θα ήταν gridDim.x * blockDim.x. Ακολουθεί ένας πίνακας που παραθέτει τις διαφορές ανάμεσα στο πρότυπο της OpenCL και της CUDA ώστε να γίνουν πιο αντιληπτές οι ομοιότητες και οι διαφορές των δυο μοντέλων.

Έννοια παραλληλίας στην OpenCL	*Ισοδύναμη έννοια στην CUDA*
Πυρήνας(Kernel)	*Πυρήνας(Kernel)*
Πρόγραμμα υπηρεσίας(Host Program)	*Πρόγραμμα υπηρεσίας(Host Program)*
NDRange(χώρος αριθμοδεικτών)	*Πλέγμα(Grid)*
Στοιχεία εργασίας(work item)	*Νήμα(Thread)*
Ομάδα εργασίας(work group)	*Μπλοκ(Block)*

Προγραμματισμός μαζικά παράλληλων επεξεργαστών David B.Kirk, Wen-mei [11].

2.3 Αρχιτεκτονική Συσκευής στο μοντέλο προγραμματισμού της OpenCL

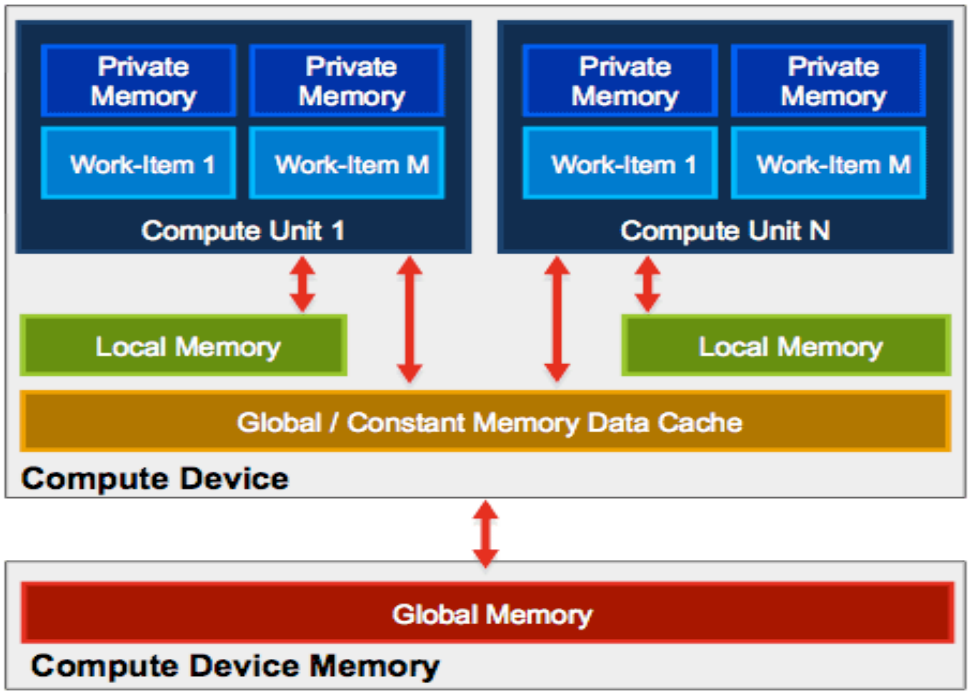
Η OpenCL μοντελοποιεί ένα ετερογενές σύστημα παράλληλης υπολογιστικής ως σύστημα υπηρεσίας και μία ή περισσότερες συσκευές OpenCL. Το σύστημα υπηρεσίας είναι μια παραδοσιακή CPU στην οποία εκτελείται το πρόγραμμα υπηρεσίας(Host Program). Κάθε συσκευή αποτελείται από μία ή περισσότερες μονάδες υπολογισμού(compute units-CU) που αντιστοιχούν στους πολυεπεξεργαστές συνεχούς ροής(streaming multiprocessors-SM) της CUDA. Όμως μια CU μπορεί επίσης να αντιστοιχεί σε πυρήνες CPU ή άλλους τύπους μονάδων εκτέλεσης επιταχυντών υπολογισμού(compute accelerators)όπως οι DSP και οι FPGA. Κάθε CU, με τη σειρά της αποτελείται από ένα ή περισσότερα στοιχεία επεξεργασίας(processing elements-PE), τα οποία αντιστοιχούν στους επεξεργαστές συνεχούς ροής(streaming process-SP) της CUDA. Οι υπολογισμοί σε μια συσκευή τελικά εκτελούνται στα μεμονωμένα PE(σχήμα 2.3)



Σχήμα 2.3 Εννοιολογική αρχιτεκτονική μιας συσκευής OpenCL[13]

Η OpenCL έχει μια ιεραρχία τύπων μνήμης που μπορούν να χρησιμοποιήσουν οι προγραμματιστές. Αυτοί οι τύποι μνήμης είναι: καθολική, σταθερών, τοπική και ιδιωτική. Η καθολική μνήμη μπορεί να εκχωρείται δυναμικά από το πρόγραμμα υπηρεσίας και υποστηρίζει προσπέλαση για ανάγνωση/εγγραφή τόσο από το σύστημα υπηρεσίας όσο και

από άλλες συσκευές. Η μνήμη σταθερών μπορεί να εκχωρείται δυναμικά από το σύστημα υπηρεσίας. Η μνήμη αυτή υποστηρίζει προσπέλαση για ανάγνωση/εγγραφή από το σύστημα υπηρεσίας και προσπέλαση μόνο για ανάγνωση από συσκευές. Η τοπική μνήμη μπορεί να εκχωρείται δυναμικά από το σύστημα υπηρεσίας και στατικά μέσα στον κώδικα της συσκευής. Η τοπική μνήμη δεν είναι προσπελάσιμη από το σύστημα υπηρεσίας ούτε υποστηρίζει κοινόχρηστη προσπέλαση ανάγνωσης/εγγραφής από όλα τα στοιχεία εργασίας μιας ομάδας εργασίας. Το σχήμα 2.4 που ακολουθεί δείχνει την ιεραρχία των μνημών που αναφέρθηκαν παραπάνω.



Σχήμα 2.4 Μοντέλο Μνήμης της OpenCL[15]

2.4 Συναρτήσεις πυρήνα στην OpenCL

Όλες οι δηλώσει πυρήνα στην OpenCL ξεκινούν με τη λέξη κλειδί *__kernel*. Ακολουθεί ένα απλό παράδειγμα ορισμού μιας συνάρτησης πυρήνα η οποία πραγματοποιεί το vector Add.

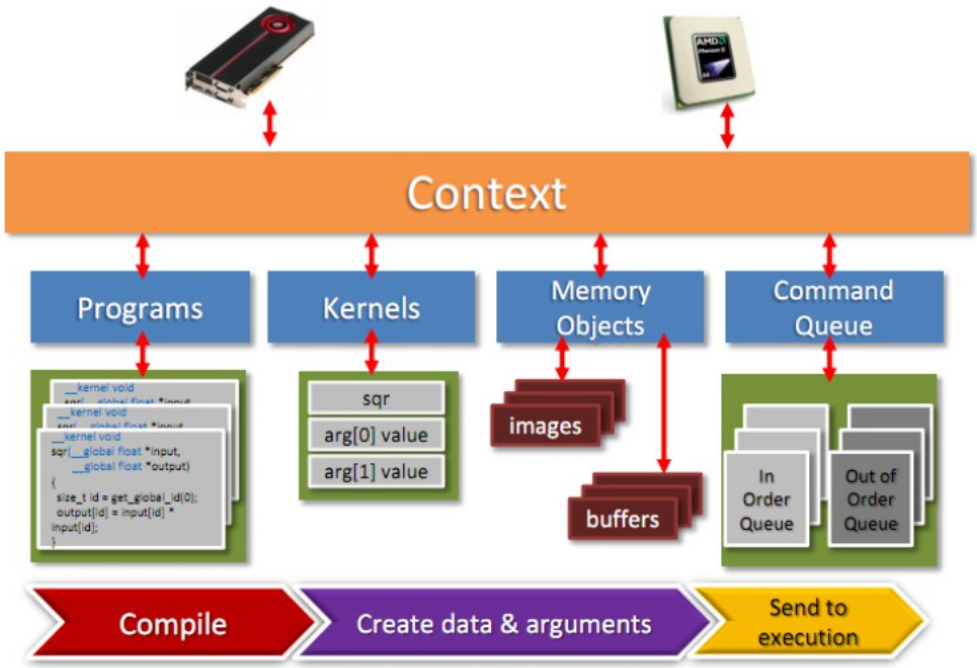
```
__kernel void vadd(__global const float *a ,__global const float *b, __global consta *result)
{
        int id = get_global_id(0);
        result[id] = a[id] + b[id];
}
```

Η εκτέλεση μιας συνάρτησης πυρήνα γίνεται από κάθε thread. Κάθε thread δηλαδή εκτελεί μια συνάρτηση πυρήνα. Οι δηλώσεις global στην κεφαλίδα της συνάρτησης δείχνουν ότι οι πίνακες εισόδου βρίσκονται όλοι στην καθολική μνήμη. Το σώμα της συνάρτησης πυρήνα γίνεται instantiate μια φορά για κάθε στοιχείο εργασίας από το run time system.

2.5 Διαχείριση συσκευών και εκκίνηση πυρήνων της OpenCL

Η OpenCL ορίζει ένα σύνθετο μοντέλο διαχείρισης συσκευών. Η πολυπλοκότητα αυτή προκύπτει από το γεγονός ότι η OpenCL υποστηρίζει πολλές πλατφόρμες υλικού. Στην OpenCL, η διαχείριση των συσκευών γίνεται μέσω θεματικών πλαισίων(contexts). Για να διαχειριστεί μία ή περισσότερες συσκευές του συστήματος, ο προγραμματιστής της OpenCL δημιουργεί πρώτα ένα θεματικό πλαίσιο που περιλαμβάνει αυτές τις συσκευές. Αυτό μπορεί να το κάνει καλώντας είτε τη clCreateContext() είτε τη clCreateContextFromType() του API της OpenCL. Τυπικά, μια εφαρμογή πρέπει να χρησιμοποιήσει τη συνάρτηση clGetDeviceIDs() του API για να προσδιορίσει τον αριθμό και τους τύπους των συσκευών που υπάρχουν σε ένα σύστημα και να μεταβιβάσει αυτές τις πληροφορίες στις συναρτήσεις clCreateContext().

Για να υποβάλει εργασία προς εκτέλεση σε μια συσκευή, το πρόγραμμα υπηρεσίας πρέπει πρώτα να δημιουργήσει μια ουρά διαταγών(command queue) για τη συσκευή. Αυτό μπορεί να γίνει με την κλήση της συνάρτησης clCreateCommandQueue() του API της OpenCL. Αφού δημιουργήσει μια ουρά διαταγών για μια συσκευή, ο κώδικας υπηρεσίας μπορεί να εκτελέσει μια αλληλουχία κλήσεων συναρτήσεων του API για να προσθέσει στην ουρά διαταγών έναν πυρήνα, μαζί με τις παραμέτρους διευθέτησης της εκτέλεσης του. Όταν η συσκευή είναι διαθέσιμη για εκτέλεση του επόμενου πυρήνα, αφαιρεί τον πυρήνα από την κεφαλή της ουράς ώστε αυτός να εκτελεστεί. Το παρακάτω σχήμα δίνει μια γενική άποψη της OpenCL πλατφόρμας.



Σχήμα 2.5 Γενική πλατφόρμα της OpenCL[13]

Ακολουθεί ένα κομμάτι προγράμματος του host για να γίνει πιο κατανοητή η διαδικασία διαχείρισης των συσκευών.

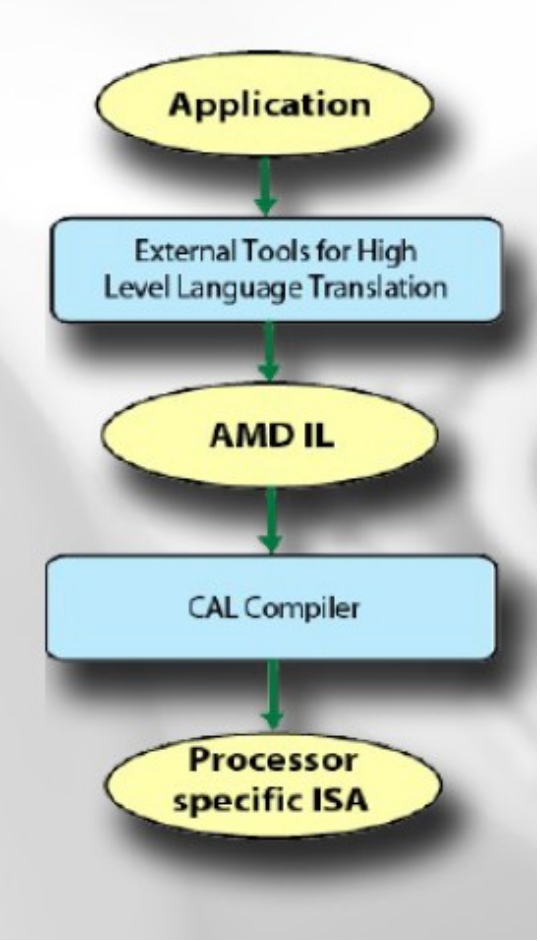
```
//error catch parameter
cl_int clerr = CL_SUCCESS;
//Create one context for All type of devices
 cl_context clctx = clCreateContextFromType( 0, CL_DEVICE_TYPE_ALL, NULL, NULL, &clerr);
size_t parmsz;
//Get info about the devices exists on the system
 clerr = clGetContextInfo( clctx, CL_CONTEXT_DEVICES, 0, NULL, &parmsz);
//Malloc memory for the devices on the system
 cl_device_id* cldevs = (cl_device_id*) malloc(parmsz);
//Create one command queue for each device
cl_command_queue clcmdq = clCreateCommandQueue( clctx, cldevs[0], 0, &clerr);
```

2.6 Εκτέλεση κώδικα στην AMD GPU

Η εκτέλεση μιας συνάρτησης πυρήνα σε μια AMD GPU αποτελεί ένα βασικό ζήτημα για την κατανόηση τεχνικών που χρησιμοποιήθηκαν για την ανάπτυξη του συστήματος χρόνου εκτέλεσης και οι οποίες θα αναλυθούν εκτενώς στο επόμενο κεφάλαιο. Για να μπορέσουμε να εκτελέσουμε μια συνάρτηση πυρήνα η οποία βρίσκεται μέσα σε μια εφαρμογή της OpenCL χρειάζεται η μεταγλώττιση της συνάρτησης αυτής και η μετατροπή της σε μια αναπαράσταση την οποία θα αναγνωρίζει η AMD GPU. Για να φτάσουμε στο σημείο η GPU μας να μπορεί να κατανοήσει των κώδικα της συνάρτησης πυρήνα πρέπει ο αρχικός κώδικας της συνάρτησης αυτής, ο οποίος έχει γραφτεί σε κάποια γλώσσα προγραμματισμού(C/C++), να περάσει από ορισμένα στάδια.

Αφού γράψουμε των κώδικα της συνάρτησης στη γλώσσα προγραμματισμού που θέλουμε, στην συνέχεια έχουμε την αναπαράσταση του σε μια ενδιάμεση ψευδό γλώσσα(pseudo-assembly), την AMD IL(intermediate language). Η αναπαράσταση του κώδικα μας στο σημείο αυτό είναι πιο κοντά στη γλώσσα μηχανής, ωστόσο είναι πιο υψηλού επιπέδου αναπαράσταση από αυτή που μπορεί να κατανοήσει μια AMD GPU. Με την ενδιάμεση αυτή αναπαράσταση δίνεται η δυνατότητα στους προγραμματιστές των εφαρμογών να βρίσκονται σε μια γλώσσα πιο κοντά σε αυτή της γλώσσας μηχανής που καταλαβαίνουν οι κάρτες γραφικών της AMD, ωστόσο να είναι και σχετικά υψηλού επιπέδου η αναπαράσταση της ώστε να μπορούν να την καταλαβαίνουν. Το επίπεδο στο οποίο βρίσκεται η ενδιάμεση αυτή ψευδό γλώσσα οι κατασκευαστές της AMD το έχουν ονομάσει ως CAL(Compute Abstraction Level). Το CAL παρέχει ένα API προς τον προγραμματιστή το οποίο είναι αρκετά κοντά στους driver των AMD GPU's και με αυτό τον τρόπο οι προγραμματιστές προσπερνούν ένα επίπεδο ανακατεύθυνσης του AMD Run Time.

Στη συνέχεια η ψευδό γλώσσα αυτή μετατρέπεται μέσω του CAL Compiler στο ISA(Instruction Set Architecture) της AMD GPU. Έχουμε δηλαδή τον αρχικό μας κώδικα στη μορφή που μπορεί να κατανοηθεί από τις κάρτες γραφικών της AMD και φυσικά να εκτελεστεί σε αυτές. Το API της OpenCL παρέχει συναρτήσεις για τη μετατροπή του αρχικού κώδικα στην AMD IL. Η συνάρτηση αυτή είναι η *clCreateProgramWithSource()* η οποία δέχεται τον κώδικα της συνάρτησης πυρήνα στην αρχική γλώσσα προγραμματισμού και επιστρέφει των κώδικα αυτό στην IL AMD. Στη συνέχεια για να πάρουμε τον τελικό εκτελέσιμο αρχείο του κώδικα, σε μορφή δηλαδή που καταλαβαίνει η AMD GPU, καλούμε από το API της OpenCL τη συνάρτηση *clBuildProgram()* η οποία δέχεται ως είσοδο τον AMD IL κώδικα και παράγει σαν έξοδο τον τελικό κώδικα σε μορφή κατανοητή για την κάρτα γραφικών. Η διαδικασία με την ενδιάμεση μορφή του κώδικα και η τελική του αναπαράσταση στο ISA της AMD GPU φαίνεται στο σχήμα 2.5



Σχήμα 2.6 [13]

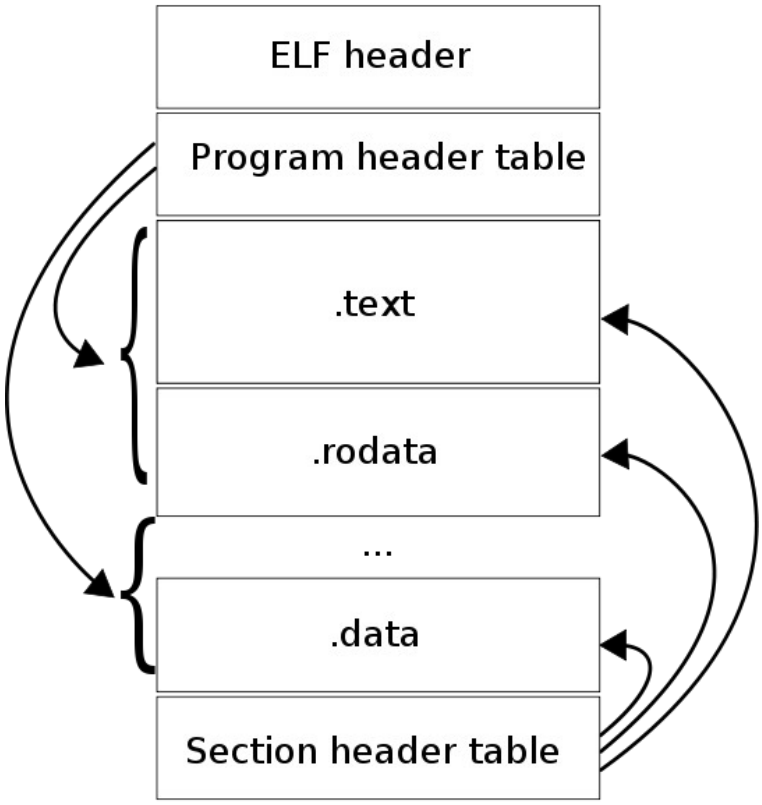
Ο τρόπος με τον οποίο πρέπει να γίνει η διαδικασία την οποία περιγράψαμε παραπάνω αποτέλεσε βασικό ζήτημα κατά το σχεδιασμό του συστήματος χρόνου εκτέλεσης και θα περιγραφεί με λεπτομέρεια στο επόμενο κεφάλαιο. Αυτό που αξίζει να αναφερθεί είναι το γεγονός ότι το πρότυπο της OpenCL κάνει just In Time Compile τα kernel functions γεγονός το οποίο επιβαρύνει σε ένα βαθμό την απόδοση του Run Time της OpenCL χωρίς φυσικά αυτό να σημαίνει ότι το κόστος αυτής της τεχνικής είναι ακριβό.

2.7 ELF Files Technology

Μια μέθοδος που χρησιμοποιείται στο RTS προκειμένου να μπορέσουμε να υποστηρίξουμε πολλαπλά εκτελέσιμα αρχεία από διαφορετικές συναρτήσεις πυρήνα, είναι τα fat binaries, τα οποία θα αναλύσουμε στο επόμενο κεφάλαιο. Η μέθοδος αυτή ωστόσο στηρίζεται στην τεχνολογία των elf files και για το λόγο αυτό η ενότητα αυτή έχει σκοπό να καλύψει τα βασικότερα θέματα της τεχνολογίας αυτής. Τα elf files είναι common standard file format το για τα εκτελέσιμα(executable), object code, shared libraries και core

dumps(Wikipedia). Ο βασικός λόγος που χρησιμοποιείται η τεχνολογία αυτή στο RTS είναι γιατί μας δίνουν τη δυνατότητα με δυναμικό τρόπο να βρίσκουμε με δυναμικό τρόπο το binary κώδικα που βρίσκεται μέσα σε ένα binary file και να κάνουμε το linking αντίστοιχη συσκευή.

Η δομή ενός elf file είναι η παρακάτω. Αποτελείται από δύο μέρη όπου το πρώτο είναι το program header όπου μας δείχνει τα τμήματα που χρησιμοποιούνται στο run-time. Το δεύτερο διακριτό μέρος είναι το Section Header το οποίο απαριθμεί το σύνολο των τμημάτων(sections) των binaries. Οι compilers, assemblers και linkers μεταχειρίζονται το elf file σαν ένα σύνολο από logical sections, το οποίο περιγράφεται από το Section header. Από την άλλη ο system loader μεταχειρίζεται το elf file σαν ένα σύνολο από segments, το οποίο περιγράφεται στο Program header. Το σχήμα 2.6 διαφωτίζει καλύτερα τη δομή ενός elf file.



Σχήμα 2.7 Δομή Elf file [16]

Κεφάλαιο 3

SOpenCL Run Time System

3.1 Εισαγωγή στο σχεδιασμό του Συστήματος χρόνου Εκτέλεσης

Μεγάλη έμφαση κατά το σχεδιασμό και την ανάπτυξη του συστήματος χρόνου εκτέλεσης δόθηκε στον τρόπο με τον οποίο έπρεπε να δομηθεί το σύστημα. Βασική προϋπόθεση του συστήματος είναι να κρατήσει τα χαρακτηριστικά του προτύπου της OpenCL, όποτε ο παράγοντας αυτός συνέβαλε στη δόμηση του. Επίσης το γεγονός ότι υποστηρίζει την εκτέλεση κώδικα τόσο σε συμβατικούς επεξεργαστές όσο και σε κάρτες γραφικών(GPU's) της AMD, έκανε απαραίτητη τη δόμηση και αρχιτεκτονική του συστήματος ώστε να μην έχει μεγάλη πολυπλοκότητα, το οποίο θα αναλυθεί εκτενώς σε παρακάτω ενότητα. Τέλος η απόδοση του RTS πρέπει να είναι συγκρίσιμη με τη Run Time library της OpenCL, το οποίο φυσικά απαιτούσε τον κατάλληλο σχεδιασμό ως προς τη δομή του συστήματος.

Το RTS δημιουργεί έναν αριθμό από kernel-level threads τα οποία χρησιμοποιούνται είτε σαν βοηθητικά(helper threads) είτε για την εκτέλεση κώδικα(execution threads). Έχουμε επίσης τον main thread το οποίο εκτελεί τον κώδικα της OpenCL εφαρμογής. Επίσης κατά την αρχικοποίηση του RTS δημιουργούνται τα working threads τα οποία είναι υπεύθυνα για την εκτέλεση της κύριας υπολογιστικής δουλειάς της εφαρμογής. Για την ακρίβεια δημιουργούνται τόσα working threads όσα είναι και τα devices τα οποία υπάρχουν στο υπολογιστικό μας σύστημα. Τέλος όσον αφορά τη διαχείριση(management) και παρακολούθηση(monitoring) των εργασιών έχουμε το helper thread του οποίου οι αρμοδιότητες και η υλοποίηση εξαρτώνται από την αρχιτεκτονική της συσκευής.

Ένα βασικό ζήτημα το οποίο έπρεπε να λυθεί κατά το σχεδιασμό ήταν οι ασύγχρονες εντολές οι οποίες έπρεπε να εκτελεστούν στην CPU. Αρχικά να αναφέρουμε ότι οι εκτέλεση ασύγχρονων εντολών στην GPU δεν αποτελεί πρόβλημα για το σχεδιασμό μας διότι υπάρχει Hardware μηχανισμός στις συσκευές αυτές ο οποίος διαχειρίζεται τέτοιου είδους προβλήματα χωρίς να χρειάζεται η άμεση παρέμβαση του προγραμματιστή. Ωστόσο κάτι ανάλογο δεν υπάρχει και για τους συμβατικούς επεξεργαστές, όπου ο προγραμματιστής θα πρέπει να διαχειριστεί τέτοιες ενέργειες με την άμεση παρέμβαση του. Για το σκοπό αυτό δημιουργήθηκε ένας software μηχανισμός για την διαχείριση αυτών των εντολών. Συγκεκριμένα έχουμε στη δομή του συστήματος μας μια ουρά στην οποία τοποθετούνται οι

ασύγχρονες εντολές(Asynchronous command queue) και τη δημιουργία ενός ακόμα thread, του asynchronous thread το οποίο είναι υπεύθυνο για να διαχειρίζεται τις εργασίες στην

ουρά των ασύγχρονων εργασιών και όταν τελειώνουν την εκτέλεση τους να ειδοποιεί το σύστημα με κατάλληλο τρόπο. Φαίνεται από εδώ η πρώτη διαφορά ανάμεσα στον τρόπο που το RTS διαχειρίζεται τέτοιου είδους ενέργειες για τις CPU's και πως τις διαχειρίζεται για τις GPU's, το οποίο προκύπτει από το γεγονός ότι οι δυο αυτές συσκευές έχουν διαφορετική αρχιτεκτονική. Στην επόμενη ενότητα του κεφαλαίου αυτού γίνεται ανάλυση του τρόπου με τον οποίο διαχειριζόμαστε τις εργασίες.

3.2 Διαχείριση εργασιών

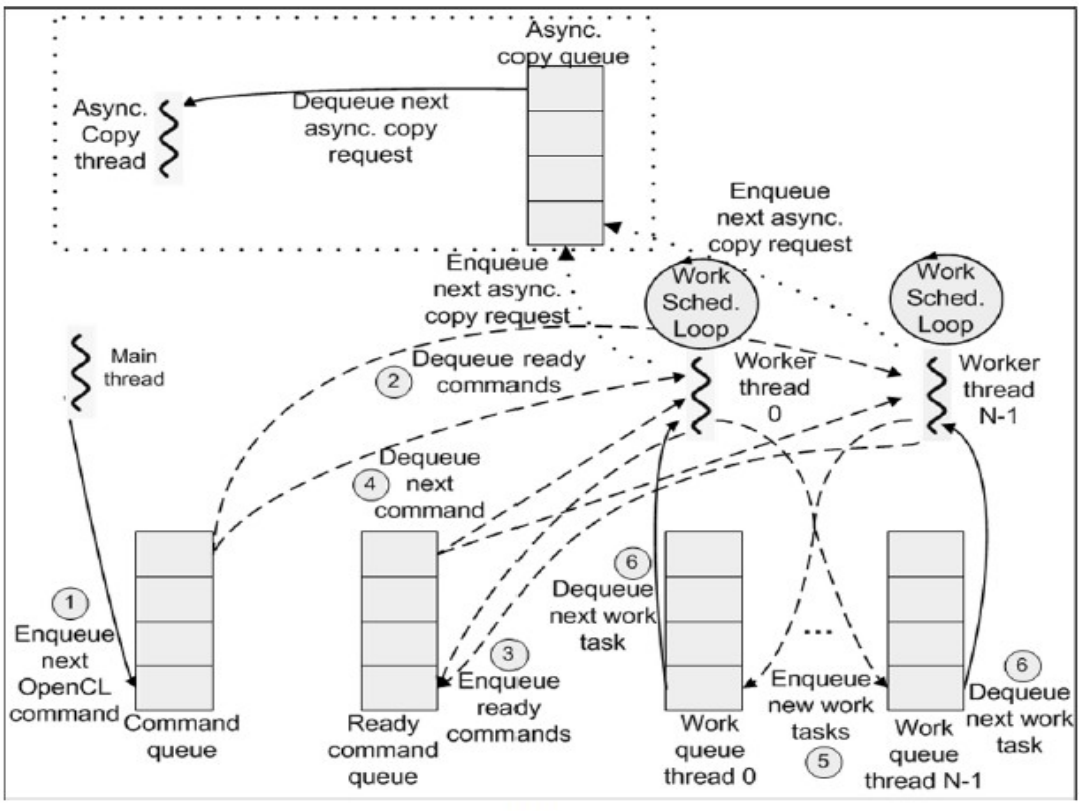
Για κάθε εργασία που υπάρχει στο πρόγραμμα υπηρεσίας(host program) το main thread δημιουργεί μια περιγραφή της εντολής(command description) και στη συνέχεια την τοποθετεί στην command queue του RTS. Ο αμοιβαίος αποκλεισμός στην πρόσβαση της ουράς υλοποιείται με την χρήση futexes **[12]**. Στη συνέχεια οι commands μεταφέρονται από την command queue στην ready queue όταν είναι έτοιμες για εκτέλεση. Μια εντολή μπορεί να μην είναι έτοιμη για εκτέλεση διότι περιμένει να λάβει ορισμένα ορίσματα από κάποια προηγούμενη της εντολή ή γιατί έχει κάποιον δομικό περιορισμό από παραπάνω εντολές της. Υπάρχουν δυο τρόποι με τους οποίους μπορούν να υλοποιηθούν οι command queues. Είτε in-order-execution είτε out-of-order execution. Στην πρώτη περίπτωση μια εντολή μεταφέρεται από την command queue στην ready queue αν βρίσκεται στην κορυφή της πρώτης και όλες οι προηγούμενες εντολές της έχουν τελειώσει την εκτέλεση τους.

Ωστόσο η υλοποίηση που επιλέχτηκε για το RTS είναι out-of-order execution και θα γίνει κατανοητός ο λόγος στη συνέχεια. Στην υλοποίηση με out-of-order execution απαιτείται ένα είδους scheduling scheme. Κάθε εντολή που βρίσκεται στην command queue μπορεί να έχει εξαρτήσεις από μια ή παραπάνω προηγούμενες της εντολές. Κάθε εντολή που τελειώνει λοιπόν την εκτέλεση της ανανεώνει τις εξαρτήσεις που έχουν προς αυτή άλλες εντολές που βρίσκονται στην command queue. Τη στιγμή που μια εντολή στην command queue δεν είναι εξαρτημένη από άλλες εντολές είναι έτοιμη να μπει στην ready queue. Τη στιγμή που μια εντολή, η οποία σχετίζεται με την εκτέλεση κάποιου πυρήνα(kernel execution), θα μπει στην ready queue, δημιουργούνται τα work tasks. Ουσιαστικά το work task αναπαριστά τον υπολογισμό που πρέπει να γίνει σε ένα work-group και ο αριθμός αυτών εξαρτάται από το διαχωρισμό(portioning) που έχει γίνει στο global space της εφαρμογής. Σε περίπτωση που ο προγραμματιστής της OpenCL εφαρμογής δεν έχει ορίσει το global space το RTS εκτελεί στατικά το διαχωρισμό του global space.

Τα worker threads συνεχώς εκτελούν ένα scheduling loop όπου σε κάθε επανάληψη λαμβάνουν το επόμενο διαθέσιμο work-task από την κορυφή της work-queue και εκτελούν

την αντίστοιχη συνάρτηση πυρήνα(kernel function).Οι work-queues είναι οι ουρές από τις οποίες τα worker-thread παίρνουν τις εργασίες που πρόκειται να εκτελέσουν. Τόσο τα worker threads της CPU όσο και της GPU έχουν άμεση πρόσβαση στις work queues. Όταν ένα worker-thread τελειώσει την εργασία του, επικοινωνεί με το helper thread για να πάρει το επόμενο work-task. Αν υπάρχει διαθέσιμη δουλειά τότε το helper-thread ειδοποιεί το worker-thread. Σε αντίθετη περίπτωση αν δεν υπάρχει εργασία προς εκτέλεση το helper thread θέτει το αντίστοιχο worker thread ως idle μέχρι να έχει ένα work-task να του δώσει.

Η επιλογή να υπάρχει ένα work-queue για κάθε work-thread έγινε γιατί με αυτό τον τρόπο μειώνεται το κόστος του συγχρονισμού(overhead synchronization). Επίσης ενισχύουμε την τοπικότητα(locality) και βελτιώνεται το scalability του RTS. Για να μπορέσουμε να εξισορροπήσουμε το load-balance επιτρέπουμε το work-stealing από τις work-queues. Κάθε worker-thread που είναι idle προσπαθεί να κλέψει task από το τέλος της ουράς ενός άλλου worker-thread. Ο λόγος που επιλέγουμε να κλέψουμε από το τέλος και όχι από την αρχή της ουράς έχει να κάνει με το γεγονός ότι οι εργασίες στην κορυφή της ουράς είναι πιο σημαντικές από αυτές που βρίσκονται στο τέλος της. Το παρακάτω σχήμα δίνει την γενική άποψη της λειτουργίας του συστήματος.



Σχήμα 3.1 Διαχείριση εργασιών [2]

3.3 Δυναμικό κάλεσμα της συνάρτησης πυρήνα

Βασικό ζήτημα το οποίο απασχόλησε την υλοποίηση του συστήματος μας είναι ο τρόπος με τον οποίο θα καλούμε της συναρτήσεις πυρήνα για εκτέλεση. Για το σκοπό αυτό ήταν απαραίτητο να βρεθεί μια τεχνική ώστε δυναμικά, την ώρα που τρέχει το πρόγραμμα, το run-time να καλεί τη συνάρτηση πυρήνα που θέλουμε να εκτελεστεί. Η προσέγγιση που χρησιμοποιείται για το σκοπό αυτό είναι το δυναμικό κάλεσμα της συνάρτησης πυρήνα μέσω ενός δείκτη συνάρτησης(function pointer). Αυτή η προσέγγιση φυσικά προϋποθέτει ότι είναι γνωστές οι διευθύνσεις των συναρτήσεων όπως επίσης και ο αριθμός των ορισμάτων που αυτές χρειάζονται. Μια εφαρμογή OpenCL μπορεί να καλέσει πολλές συναρτήσεις πυρήνα κατά την εκτέλεση της, τις οποίες και προσδιορίζει ρητά δίνοντας το string του ονόματος των συναρτήσεων.

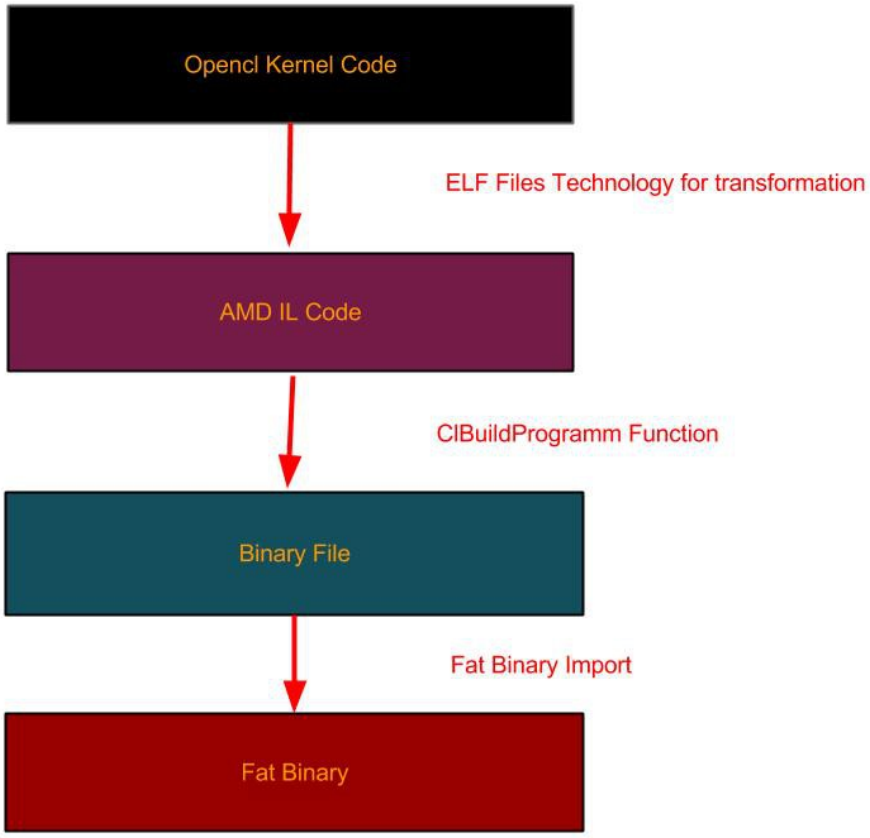
Στο σημείο αυτό έχουμε μια ουσιαστική διαφορά στον τρόπο με τον οποίο καλούμε τις συναρτήσεις πυρήνα όταν αυτές πρόκειται να εκτελεστούν σε CPU και όταν πρόκειται να εκτελεστούν σε μια GPU. Το βασικό θέμα που απασχόλησε τη συγκεκριμένη διπλωματική εργασία είναι η εκτέλεση στην AMD GPU και όσο τόσο στη CPU για το λόγο αυτό θα γίνει εκτενής αναφορά στην εκτέλεση συνάρτησης πυρήνα στην AMD GPU. Ωστόσο αξίζει να αναφερθεί ότι προκειμένου να μπορέσουμε να υποστηρίξουμε δυναμικό κάλεσμα συνάρτησης πυρήνα σε μια CPU, με διαφορετικό αριθμό και τύπων ορισμάτων, φτιάχτηκε ένα κατάλληλο interface το οποίο συνεργάζεται στενά με τον SOpenCL compiler. Ο συγκεκριμένος compiler δημιουργεί μία wrapper function για κάθε συνάρτηση πυρήνα της εφαρμογής μας. Κάθε wrapper function έχει ένα μοναδικό όρισμα, έναν πίνακα από void pointers, και είναι υπεύθυνη για να κάνει την εξαγωγή των ορισμάτων της συνάρτησης πυρήνα. Με αυτό τον τρόπο ένα work-task χρειάζεται να ξέρει τη διεύθυνση της αντίστοιχης wrapper function και να έχει έναν δείκτη προς τα αντίστοιχα ορίσματα της συνάρτησης.

Βασικό ζήτημα που απασχόλησε την εκτέλεση κώδικα για την GPU ήταν το γεγονός ότι έπρεπε να μετατρέψουμε τον αρχικό κώδικα μιας συνάρτησης πυρήνα στο ISA της AMD GPU ωστόσο όπως είδαμε στο κεφάλαιο 2 αυτό γίνεται σε δύο στάδια. Στο πρώτο στάδιο έχουμε την αναπαράσταση του αρχικού κώδικα σε μια ψευδό γλώσσα, την AMD IL και στο δεύτερο στάδιο έχουμε την μετατροπή της ενδιάμεσης αυτής αναπαράστασης στο ISA της AMD. Ωστόσο θέλουμε η διαδικασία αυτή να γίνεται με έναν συστηματικό τρόπο, το οποίο να είναι διαφανές από τον προγραμματιστή χωρίς την άμεση παρέμβαση του. Η μέθοδος που χρησιμοποιείται για την επίλυση αυτού του ζητήματος είναι τα binary files σε συνεργασία με τα elf files. Μέσω 2 υποστηρικτικών μηχανισμών, που θα περιγραφούν στην επόμενη ενότητα, το σύστημα μας εκτελεί την κατάλληλη συνάρτηση πυρήνα φορτώνοντας τη από το fat binary το οποίο έχουμε δημιουργήσει(η δημιουργία αυτού του αρχείου περιγράφεται στην επόμενη ενότητα). Η τεχνική των fat binaries μας βοηθάει να εκτελούμε διαφορετικές συναρτήσεις πυρήνα στη CPU και στη GPU με παράλληλο τρόπο.

Φορτώνοντας λοιπόν το binary file της συνάρτησης που θέλουμε να εκτελέσουμε από το fat binary αρχείο στην GPU πετυχαίνουμε την εκτέλεση του κώδικα που μας ενδιαφέρει.

3.4 Μηχανισμοί υποστήριξης του RTS-Fat binaries

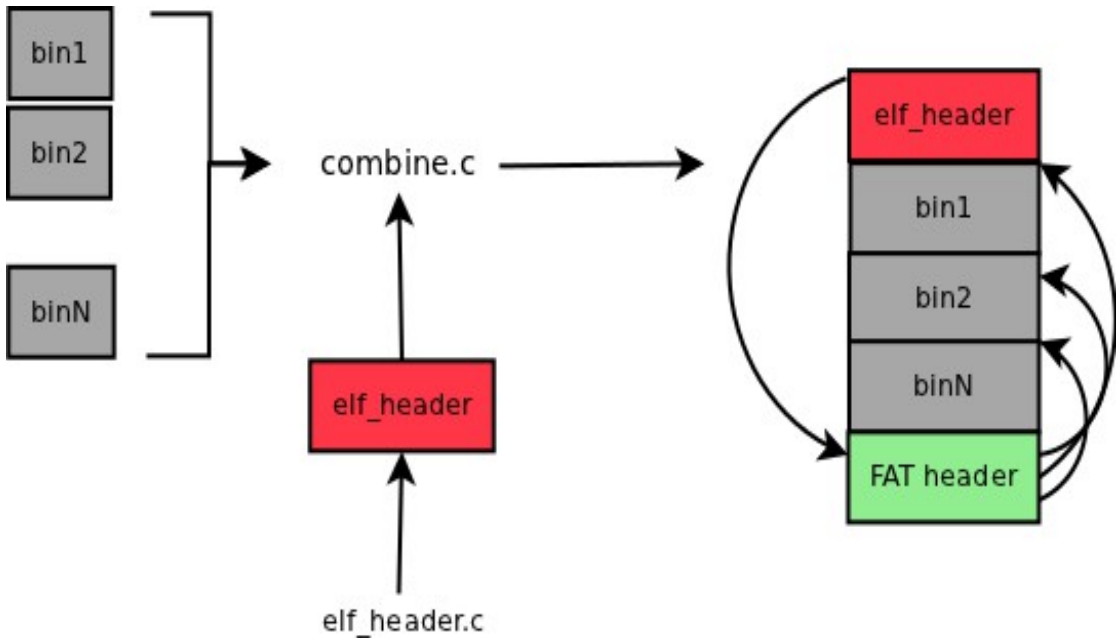
Η αναπαράσταση του αρχικού κώδικα της συνάρτησης πυρήνα στην ενδιάμεση ψευδογλώσσα, AMD IL, είναι το πρώτο βήμα ώστε να μπορέσει ο κώδικας μας να τρέξει στην GPU. Η μετατροπή του αρχικού κώδικα στο ενδιάμεσο αυτό επίπεδο μας δίνει στη συνέχεια τη δυνατότητα να εξάγουμε το binary κώδικα αυτού. Χρησιμοποιώντας την τεχνολογία των elf files εξάγουμε τον IL κώδικα της συνάρτησης πυρήνα που θέλουμε να εκτελέσουμε. Με την μετατροπή αυτή στην συνέχεια μπορούμε να εξάγουμε το binary file χρησιμοποιώντας τη συνάρτηση *clBuildProgram()* από το API της OpenCL. Το παρακάτω σχήμα δείχνει τα στάδια που περνάει μια συνάρτηση πυρήνα μέχρι να τοποθετηθεί στο fat binary αρχείο.



Σχήμα 3.2

Η διαδικασία που περιγράψαμε παραπάνω για την τελική παραγωγή του binary file γίνεται offline και όχι κατά την διάρκεια εκτέλεσης του συστήματος μας, για να μην υπάρχει επιβάρυνση στο χρόνο εκτέλεσης της εφαρμογής που θέλουμε να εκτελέσουμε. Το γεγονός αυτό έχει ληφθεί υπόψη κατά το σχεδιασμό του συστήματος μας και για το λόγο αυτό η διαδικασία παραγωγής του binary file από τον αρχικό κώδικα γίνεται με αυτοματοποιημένο και γρήγορο τρόπο, χωρίς να χρειάζεται ο χρήστης να εμπλέκεται με τεχνικές λεπτομέρειες υλοποίησης του μηχανισμού αυτού. Ο Μηχανισμός αυτός αν και δεν εντάσσεται άμεσα στη δομή του run-time ωστόσο είναι σημαντικός για τη λειτουργία του και για αυτό το λόγο έχει συμπεριληφθεί στο πακέτο του συστήματος μας.

Ο δεύτερος υποστηρικτικός μηχανισμός που χρησιμοποιείται είναι αυτός της παραγωγής του fat binary. Η τεχνική των fat binary αρχείων χρησιμοποιείται αρκετά στα Unix συστήματα, στα λειτουργικά συστήματα και όπου υπάρχει άμεσα ή έμμεσα το Just In Time Compilation. Ένα fat binary αρχείο είναι ένα αρχείο όπου εμπεριέχει διαφορετικά binary files από συσκευές με διαφορετική αρχιτεκτονική η μία από την άλλη(CPU's, GPU's ή άλλου τύπου accelerators) με σκοπό να μπορούν να επιλεγούν τα binary file που θέλουμε για να εκτελεστεί ο αντίστοιχος κώδικας. Στην υλοποίηση τη δική μας σκοπός μας είναι να μπορούμε να επιλέγουμε στο run-time κομμάτια κώδικα που θέλουμε να εκτελέσουμε, είτε στη CPU είτε στη GPU είτε παράλληλα και στις 2 συσκευές και αυτό να γίνεται με τρόπο γρήγορο και αποτελεσματικό. Η δομή των binary files είναι η εξής: Το πρώτο πεδίο είναι το όνομα της συνάρτησης πυρήνα που θέλουμε να εκτελέσουμε. Το δεύτερο πεδίο είναι το μέγεθος του binary file. Το τρίτο πεδίο είναι το offset που υπάρχει μεταξύ του συγκεκριμένου binary file και του επομένου του. Το τέταρτο πεδίο αφορά τον τύπο της συσκευής για το οποίο είναι προορισμένο το binary file, αν είναι δηλαδή για CPU ή για GPU, ενώ το πέμπτο και τελευταίο πεδίο αφορά το όνομα του fat binary. Ξέροντας λοιπόν το όνομα της συνάρτησης πυρήνα που θέλουμε να εκτελέσουμε μπορούμε με δυναμικό τρόπο στο run-time να φορτώσουμε το αντίστοιχο binary file ψάχνοντας κατάλληλα μέσα στο fat binary. Το σχήμα 3.1 διαφωτίζει τη διαδικασία που περιγράψαμε



Σχήμα 3.3 Παραγωγή Binary File

<u>Ακολουθεί ο κώδικας ο οποίος κάνει ανάγνωση ενός fat binary αρχείου</u>

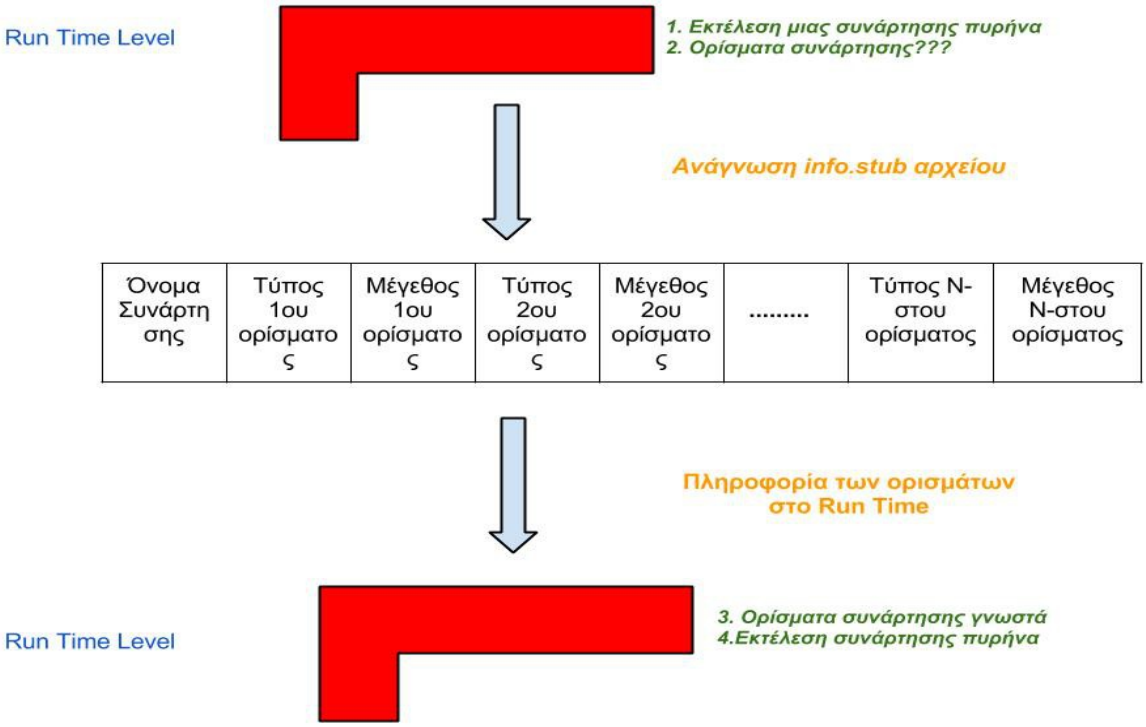
```
/* Main loop for scanning fat binary. */
  while ( 1 ) {

      /* When we reach the end of the fatbinary header stop */
      if ( strncmp( (char *)(binaries[j] + curr_offset),
      ".FATBIN_END", 11) == 0 )
        break;
/* Get the first binary info */
      memcpy( header_info->name_len, binaries[j] + curr_offset, 5);
      curr_offset += 5;
      memcpy( header_info->size, binaries[j] + curr_offset, 10);
      curr_offset += 10;
      memcpy( header_info->offset, binaries[j] + curr_offset, 10);
```

```
curr_offset += 10;
memcpy( header_info->type, binaries[j] + curr_offset, 3);
curr_offset += 3;
k = strtoul( header_info->name_len, NULL, 16);
temp_buffer = (char *) malloc( sizeof( char ) * k );
if ( temp_buffer == NULL ) {
  *errcode_ret = CL_OUT_OF_HOST_MEMORY;
   return ( NULL );
}
```

3.5 Δυναμικός προσδιορισμός παραμέτρων συνάρτησης πυρήνα για τη GPU

Βασικό ζήτημα για την υλοποίηση του συστήματος χρόνου εκτέλεσης αποτέλεσε ο τρόπος με τον οποίο δυναμικά, κατά τη διάρκεια του χρόνου εκτέλεσης της εφαρμογής, θα μπορέσουμε να προσδιορίσουμε τον τύπο και τον αριθμό των ορισμάτων της συνάρτησης πυρήνα. Δε θα ήταν υλοποιήσιμα σωστό να δίνουμε με κάποιο στατικό τρόπο των αριθμό και τον τύπο τον ορισμάτων διότι θα χάναμε τη διαφάνεια που πρέπει να έχει το σύστημα χρόνου εκτέλεσης από τον προγραμματιστή και δεύτερον αυτό θα οδηγούσε και σε ένα άσχημο δομικό σχεδιασμό του συστήματος με αρκετές εξαρτήσεις από τους προγραμματιστές. Θέλουμε λοιπόν να παίρνουμε την πληροφορία αυτή κατά τη διάρκεια του χρόνου εκτέλεσης της εφαρμογής μας.

Για το σκοπό αυτό δημιουργούμε ένα αρχείο με το όνομα info.stub.txt το οποίο ο προγραμματιστής της συνάρτησης πυρήνα επεξεργάζεται. Το αρχείο αυτό έχει την παρακάτω δομή. Στην αρχή του αρχείου υπάρχει το όνομα της συνάρτησης πυρήνα την οποία θέλουμε να εκτελέσουμε. Στη συνέχεια ακολουθεί η πληροφορία για τον τύπο των ορισμάτων της συνάρτησης. Συγκεκριμένα για κάθε τύπο ορίσματος γράφουμε με συγκεκριμένο τρόπο αν το όρισμα είναι global(βάζοντας 0) ή local(βάζοντας 3). Στη συνέχεια δίπλα από τον τύπο του ορίσματος γράφουμε το μέγεθος(σε bytes) του συγκεκριμένου ορίσματος. Το αρχείο info.stub.txt διαβάζεται κατά τη διάρκεια του χρόνου εκτέλεσης τη στιγμή που χρειάζεται η πληροφορία για τον τύπο και αριθμό ορισμάτων της συνάρτησης. Ακολουθεί ο κώδικας του kernel για τη συνάρτηση Vector Add και το αντίστοιχο info.stub.txt αρχείο. Η παρακάτω εικόνα δείχνει τη διαδικασία που περιγράψαμε προηγουμένως.



Σχήμα 3.4 Δυναμικός προσδιορισμός παραμέτρων

```
__kernel void VectorAdd(__global const float* a, __global const float* b, __global float* c, int iNumElements)
{
    // get index into global data array
    int iGID = get_global_id(0);

    // bound check (equivalent to the limit on a 'for' loop for standard/serial C code
    if (iGID >= iNumElements)
    {
        return;
    }
    // add the vector elements
    c[iGID] = a[iGID] + b[iGID];
```

}

info.stub.txt: 01 VectorAdd 04 0 0 0 0 0 0 3 4

3.6 Δυναμικό φόρτωμα συναρτήσεων της OpenCL Run Time Library

Για τη λειτουργία του RTS είναι απαραίτητο να μπορέσουμε να φορτώσουμε με δυναμικό τρόπο κατά τη λειτουργία του συστήματος τις συναρτήσεις από το API της OpenCL. Αυτό πετυχαίνεται συνδέοντας με δυναμικό τρόπο(dynamic linking) τις συναρτήσεις που χρειαζόμαστε από τη Run Time Library της OpenCL. Για το σκοπό αυτό γίνεται χρήση της βιβλιοθήκης <dlfcn.h> η οποία μας παρέχει τις κατάλληλες συναρτήσεις για το σκοπό μας. Συγκεκριμένα χρησιμοποιώντας τη συνάρτηση dlopen() της συγκεκριμένης βιβλιοθήκης εξετάζουμε αν οι συσκευές του συστήματος μας υποστηρίζουν το πρότυπο της OpenCL. Αυτό γίνεται προσπαθώντας να κάνουμε το dynamic linking με τη συγκεκριμένη δυναμική βιβλιοθήκη. Για τους σκοπούς της AMD GPU πρέπει να βρεθεί η δυναμική βιβλιοθήκη amd64OpenCL.so και να γίνει το αντίστοιχο link με αυτή κατά το χρόνο εκτέλεσης του συστήματος μας. Στη συνέχεια χρησιμοποιώντας την dlsym κάνουμε το αντίστοιχο link με τις συναρτήσεις της βιβλιοθήκης που θέλουμε να χρησιμοποιήσουμε. Συγκεκριμένα η συνάρτηση αυτή κοιτά το symbol table της δυναμικής βιβλιοθήκης για να μπορέσει να βρει τη συνάρτηση με το όνομα που ζητάμε. Αν βρεθεί η συγκεκριμένη συνάρτηση τότε επιστρέφει επιτυχώς και γίνεται το κατάλληλο link με τη συνάρτηση της βιβλιοθήκης που επιθυμούμε. Σε αντίθετη περίπτωση έχουμε επιστροφή σφάλματος από την εκτέλεση της συνάρτησης. Η μέθοδος του δυναμικού φορτώματος των συναρτήσεων από τη run time library της OpenCL δεν έχει επιβάρυνση στην απόδοση του συστήματος μας, όπως επίσης η μέθοδος είναι scalable αφού η προσθήκη νέων συναρτήσεων γίνεται με απλό τρόπο και χωρίς να χρειάζεται το σύστημα μας να επιβαρυνθεί ή να αλλάξει τη δομή του. Ωστόσο η σύνδεση με τη run time library της OpenCL δεν αποτελεί τη μόνη ιδέα που υπήρξε κατά το σχεδιασμό του συστήματος μας. Ο παρακάτω κώδικας διαφωτίζει τη διαδικασία που περιγράφτηκε παραπάνω

```
1. dlhandle = dlopen("/usr/lib/fglrx/libamdocl64.so", RTLD_LAZY | RTLD_GLOBAL );

2. amd_ocl = ( amd_runtime_functions *) malloc ( sizeof ( amd_runtime_functions ) );

3. amd_ocl->_clGetPlatformIDs = dlsym(dlhandle, "clGetPlatformIDs");
```

```
4. amd_ocl->_clGetPlatformInfo = dlsym(dlhandle,
"clGetPlatformInfo");

5. amd_ocl->_clGetDeviceIDs = dlsym(dlhandle, "clGetDeviceIDs");

6. amd_ocl->_clCreateContext = dlsym(dlhandle, "clCreateContext");

                    ...
```

Μια δεύτερη λύση για το σχεδιασμό του συγκεκριμένου συστήματος ήταν να κάνουμε dynamic link με την CAL library της AMD η οποία είναι σε χαμηλότερο επίπεδο από τη library της OpenCL και φυσικά πιο κοντά στους driver της GPU. Ένα πλεονέκτημα που υπάρχει στη λύση αυτή είναι το γεγονός ότι το σύστημα μας προσπερνάει ένα επίπεδο στο run time της OpenCL γεγονός το οποίο κάνει ελάχιστα πιο αποδοτικό όσον αφορά το χρόνο εκτέλεσης του συστήματος μας. Ωστόσο η διαφορά είναι σχεδόν μηδαμινή με βάση τις μετρήσεις που έγιναν. Το γεγονός ότι η βιβλιοθήκη αυτή είναι σε χαμηλότερο επίπεδο έκανε όμως πιο περίπλοκη τη διαδικασία υλοποίησης του συστήματος μας. Χρειάζεται ο προγραμματιστής του συστήματος να χρησιμοποιήσει αρκετά περίπλοκο API για να καταφέρει να πετύχει τη λειτουργικότητα που επιθυμεί, όπως επίσης σε ορισμένες περιπτώσεις χρειάζεται να επικοινωνήσεις άμεσα με τους driver της συσκευής. Το γεγονός αυτό θα έκανε πρώτον πιο περίπλοκο το σύστημα μας όσον αφορά τον τρόπο δομής του, με πιο περίπλοκα κομμάτια κώδικα, το οποίο θα έκανε αρκετά πιο δύσκολη την υλοποίηση του. Τέλος η εξάρτηση του από την επικοινωνία με τους driver του συστήματος θα είχαν ως αποτέλεσμα το RTS να μην είναι αρκετά scalable. Για τους λόγους αυτούς απορρίφθηκε η υλοποίηση του συστήματος με τη μέθοδο αυτή.

3.7 Δομικά μέρη του RTS

Για την βαθύτερη κατανόηση του συστήματος χρόνου εκτέλεσης και της λειτουργίας του στην ενότητα αυτή θα γίνει μια αναφορά στα βασικά δομικά μέρη του συστήματος. Το πρώτο εξ αυτών είναι οι ουρές εντολών(command queues) στις οποίες γίνεται η εισαγωγή των εντολών από το πρόγραμμα υπηρεσίας. Οι ready queues είναι ο δεύτερος τύπος ουρών που υποστηρίζει το σύστημα μας. Στις ουρές αυτές μπαίνουν οι εντολές οι οποίες είναι έτοιμες προς εκτέλεση. Ο τρίτος τύπος ουράς είναι οι work-queues οι οποίες είναι οι ουρές που έχουν τα task προς εκτέλεση για το συγκεκριμένο work-thread. Τέταρτος και τελευταίος τύπος ουράς είναι η asynchronous queue στην οποία μπαίνουν οι εντολές προς εκτέλεση για την CPU και οι οποίες έχουν ασύγχρονο τρόπο εκτέλεσης. Στη συνέχεια έχουμε τους memory buffers. Τα συγκεκριμένα δομικά στοιχεία είναι υπεύθυνα για την ανταλλαγή των δεδομένων ανάμεσα στο host και το αντίστοιχο device. Δομικά στοιχεία του συστήματος μας επίσης θεωρούνται και τα ίδια τα devices για τα οποία κρατάμε πληροφορίες που χρειάζεται το σύστημα μας, όπως αν είναι διαθέσιμες για εκτέλεση, πόσες εντολές έχει στην command queue, τον τύπο της συσκευής.

Για την αναπαράσταση ορισμένων λειτουργιών της OpenCL χρησιμοποιούνται ανάλογες struct. Μια από αυτές είναι η αναπαράσταση ενός kernel της OpenCL. Για το σκοπό αυτό υπάρχει αντίστοιχη struct η οποία κρατάει έχει ως πεδία το όνομα του kernel, τη διεύθυνση της συνάρτησης του πυρήνα που πρέπει να εκτελέσει, έναν pointer προς τα αντίστοιχα ορίσματα της συνάρτησης πυρήνα. Η OpenCL για να χειριστεί ασύγχρονες εντολές αλλά και για να μπορέσει να συντονίσει την εκτέλεση διαφορετικών συναρτήσεων πυρήνα που βρίσκονται στην ίδια εφαρμογή χρησιμοποιεί τα events. Η αναπαράσταση αυτών στο σύστημα μας γίνεται μέσω μιας struct. Τέλος για τη διαχείριση ορισμένων πράξεων με 2-D και 3-D γραφικά το σύστημα μας υποστηρίζει ένα ξεχωριστό τύπο μνήμης τις texture.

3.8 Έναρξη και Τερματισμός του RTS

Σε όλα τα Unix συστήματα υπάρχει μια συγκεκριμένη διαδικασία με την οποία ξεκινάει την εκτέλεση του ένας κώδικας και συγκεκριμένη διαδικασία με την οποία τερματίζεται αυτός. Πριν ένας κώδικας σε μια γλώσσα C για παράδειγμα, ξεκινήσει την εκτέλεση του μπαίνοντας στη συνάρτηση main, καλείται μια άλλη συνάρτηση στο παρασκήνιο αρχικά, χωρίς να γίνεται αντιληπτό στους προγραμματιστές. Η συνάρτηση αυτή ονομάζεται `constructor` και είναι υπεύθυνη να φορτώσει ορισμένες βιβλιοθήκες χρόνου εκτέλεσης και να αρχικοποιεί ορισμένες λειτουργίες που χρειάζονται για την εκτέλεση του προγράμματος. Ωστόσο αν ένας προγραμματιστής γράψει μια εφαρμογή η οποία ορίζει στον κώδικα της και τη συνάρτηση του `constructor` τότε αυτόματα το σύστημα μας πριν την εκτέλεση του καλεί τη συνάρτηση αυτή(και όχι την default του συστήματος) και εκτελεί τις πράξεις που είναι ορισμένες μέσα σε αυτή. Όσον αφορά τον τερματισμό τώρα, μετά την επιστροφή(return) της συνάρτησης main και τον τερματισμό της, καλείται πάλι μια συνάρτηση της οποίας η εκτέλεση δεν είναι φανερή στον προγραμματιστή και ονομάζεται `destructor`. Η συνάρτηση αυτή είναι υπεύθυνη για να απελευθερώσει ότι έχει φορτώσει κατά τη διάρκεια της συνάρτησης `constructor` και να απελευθερώσει θέσεις μνήμης οι οποίες δεν χρησιμοποιούνται πλέον. Το παρακάτω παράδειγμα αναδεικνύει τον τρόπο που δουλεύει η συνάρτηση `constructor`.

main.c

```c
#include <stdio.h>

/*
 * Simply "main"
 */
int main(){
        printf("main\n");
        return 0;
}
```

obj.c

```c
#include <stdio.h>

/*
 * This attribute lead gcc/ld to
 * exec this function
 * before the "main".
 */
__attribute__ ((__constructor__))
void pre_func(void) {
        printf("pre_func\n");
}
```

Execute it:

```
$ gcc main.c

$ ./a.out

main


$ gcc main.c obj.c

$ ./a.out

pre_func
main
```

Google:Experiments with c constructors

Για την αρχικοποίηση του συστήματος χρόνου εκτέλεσης έχει υλοποιηθεί η συνάρτηση `constructor` η οποία εκτελεί όλες τις πράξεις οι οποίες είναι απαραίτητες για την αρχικοποίηση του. Συγκεκριμένα αρχικοποιούνται όλες οι ουρές οι οποίες χρειάζονται στο σύστημα μας. Στη συνέχεια γίνεται probing του συστήματος μας για τον αριθμό των συσκευών που είναι στο σύστημα μας. Η διαδικασία αυτή ωστόσο έχει αρκετά στάδια και για το σκοπό αυτό η λειτουργικότητα της θα αναφερθεί σε επόμενη ενότητα. Αφού έχουμε την πληροφορία με τον αριθμό των συσκευών στο σύστημα μας βρίσκουμε τον τύπο της κάθε συσκευής και στη συνέχεια δημιουργούνται τα work-threads. Τέλος αρχικοποιείται ένα global context με σκοπό να κρατάει την πληροφορία για όλα τα context που θα δημιουργηθούν κατά τη διάρκεια εκτέλεσης του συστήματος μας.

Για τον τερματισμό του συστήματος χρόνου εκτέλεσης υλοποιήθηκε η συνάρτηση `destructor` η οποία αποδεσμεύει τη μνήμη που δεσμεύτηκε κατά την αρχικοποίηση του

συστήματος και να κάνει τις απαραίτητες πράξεις για τον τερματισμό των συσκευών του συστήματος μας. Συγκεκριμένα τερματίζει όλα τα work-threads τα οποία έχουν δημιουργηθεί, αφού πρώτα εξασφαλίσει ότι έχουν τελειώσει τη δουλειά τους. Στη συνέχεια αποδεσμεύει τη μνήμη που έχει δεσμεύσει για τις ουρές των συσκευών και τέλος τερματίζει την εκτέλεση των συσκευών του συστήματος.

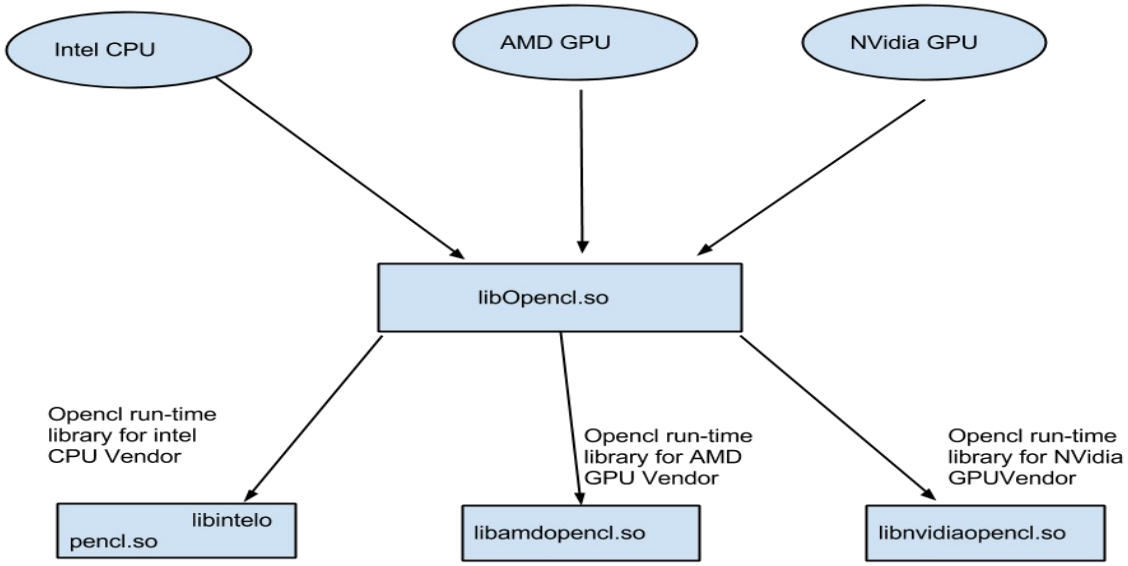
3.9 Probe Device

Όπως αναφέρθηκε στην προηγούμενη ενότητα η διαδικασία ανίχνευσης των συσκευών αποτελείται από ορισμένα στάδια τα οποία θα αναλυθούν στην ενότητα αυτή. Για να ανιχνεύσουμε τις GPU συσκευές που υπάρχουν στο σύστημα μας αρκεί να χρησιμοποιήσουμε από το API της OpenCL τη συνάρτηση `clGetPlatformIDs`. Η συνάρτηση αυτή επιστρέφει τον αριθμό των AMD GPUs που υπάρχουν στο σύστημα μας. Στην συνέχεια για κάθε AMD GPU που έχει ανιχνευθεί στο σύστημα μας είναι απαραίτητη η αρχικοποίηση της. Συγκεκριμένα γίνεται η διαδικασία δυναμικής φόρτωσης των συναρτήσεων από το run time της OpenCL. Στη συνέχεια για κάθε GPU device αρχικοποιείται η αντίστοιχη command queue. Με τον τρόπο αυτό γίνονται όλες οι απαραίτητες ενέργειες ώστε μια GPU συσκευή να είναι έτοιμη για χρήση στη συνέχεια.

3.10 Extensibility & Transparency

Δύο βασικά χαρακτηριστικά που πρέπει να έχει ένα σύστημα χρόνου εκτέλεσης είναι η επεκτασιμότητα(extensibility) και η διαφάνεια του(transparency) προς το χρήστη. Στην ενότητα αυτή θα αναφερθούν ορισμένα χαρακτηριστικά του συστήματος χρόνου εκτέλεσης τα οποία χαρακτηρίζουν το σύστημα μας scalable και transparent. Μια βασική προϋπόθεση για να μπορεί να χαρακτηρίζεται το σύστημα μας extensible είναι η εύκολη ενσωμάτωση νέων αρχιτεκτονικών συσκευών GPU στο σύστημα μας. Για να γίνει κατανοητός ο αρχικός λόγος για τον οποίο το σύστημα μας είναι scalable θα γίνει μια αναφορά αρχικά στον τρόπο που η OpenCL Run-time library λειτουργεί με διαφορετικούς vendor(π.χ AMD, Nvidia). Στο path /usr/lib βρίσκεται η βιβλιοθήκη libOpenCL.so η οποία είναι η έκδοση του τελευταίου vendor ο οποίος εγκατέστησε στο υπολογιστικό μας σύστημα το SDK της OpenCL. Παράδειγμα αν το σύστημα μας έχει ένα SDK της OpenCL για τις Nvidia GPUs και ένα SDK OpenCL για τις AMD GPUs και η AMD εγκατέστησε τελευταία το SDK της τότε στο /usr/lib θα βρίσκεται η libOpenCL.so της AMD. Τι γίνεται όμως όταν θέλουμε να τρέξουμε μια εφαρμογή OpenCL από το SDK της Nvidia; Αυτό που γίνεται στην πραγματικότητα είναι ότι η βιβλιοθήκη libOpenCL.so στο /usr/lib κάθεται πάνω από όλες τις run-time libraries της OpenCL των vendors που υπάρχουν στο σύστημα μας(AMD GPU, Intel CPU, Nvidia GPU). Κάθε φορά λοιπόν που που ένας vendor καλεί τη libOpenCL.so η βιβλιοθήκη αυτή αναγνωρίζει τον τύπο του vendor και καλεί την αντίστοιχη βιβλιοθήκη της

OpenCL για το συγκεκριμένο vendor(η οποία βρίσκεται σε συγκεκριμένο path για κάθε vendor). Το σχήμα 4.1 δείχνει την παραπάνω διαδικασία με αφαιρετικό τρόπο.



Σχήμα 4.1

Ο λόγος για το οποίο το σύστημα μας είναι extensible είναι αποτέλεσμα του τρόπου με τον οποίο καλούμε τη run-time library της OpenCL μέσα στο δικό μας run-time. Συγκεκριμένα για τον AMD vendor καλούμε απευθείας τη βιβλιοθήκη libamd64OpenCL.so, η οποία σε διαφορετική περίπτωση θα καλούνταν έμμεσα από την libOpenCL.so(όταν ο vendor είναι AMD GPU). Αν θέλουμε στην αρχιτεκτονική του συστήματος μας να προσθέσουμε τις Nvdia GPUs αρκεί να φτιάξουμε ένα wrapper file, με τον ίδιο τρόπο που είναι το wrapper file για την AMD GPU, με όλες τις function που καλεί το OpenCL run-time της Nvidia και να κάνουμε link με την αντίστοιχη βιβλιοθήκη του vendor. Βλέπουμε ότι δε χρειάζεται να γράψουμε μεγάλη ποσότητα κώδικα, αλλά ούτε να αλλάξουμε την δομή και αρχιτεκτονική του συστήματος μας για να προσθέσουμε νέα αρχιτεκτονική συσκευής.

Επίσης για να μπορέσουμε να τρέξουμε κώδικα με διαφορετικό ISA από της AMD δεν χρειάζεται να αλλάξουμε τίποτα στη δομή του συστήματος μας. Ο κώδικας που θα αναγνωρίζεται από άλλου τύπου αρχιτεκτονικής θα συμπεριλαμβάνεται μέσα στο fat binary αρχείο, με την ανάγνωση αυτού να γίνεται με τον ίδιο τρόπο από το RTS. Βλέπουμε λοιπόν ότι ούτε ο τρόπος με τον οποίο θέλουμε να εκτελούμε κώδικα διαφορετικής αρχιτεκτονικής επηρεάζει τη δομή του συστήματος μας αλλά αντιθέτως η ενσωμάτωση της διαδικασίας αυτής στο σύστημα μας γίνεται με εύκολο τρόπο.

Όσον αφορά το transparency θα αναφέρουμε ορισμένες ενέργειες οι οποίες εκτελούνται στο παρασκήνιο και ο χρήστης δε γνωρίζει για αυτές. Αρχικά να αναφέρουμε το σύστημα μας αρχικοποιείται αλλά και τερματίζει χωρίς ο χρήστης να αντιλαμβάνεται τις ενέργειες που πραγματοποιούνται κατά τις δύο αυτές φάσεις. Συγκεκριμένα στην αρχικοποίηση του συστήματος κρύβεται το γεγονός ότι κάνουμε probe device, αρχικοποιούνται τα command queues για τα devices αυτά, καθώς επίσης δημιουργούνται όλα τα kernel threads που χρειάζονται για τη λειτουργία του συστήματος. Με το ίδιο τρόπο κατά τον τερματισμό του συστήματος ο προγραμματιστής δεν αντιλαμβάνεται τις ενέργειες που αφορούν την αποδέσμευση θέσεων μνήμης που έγιναν κατά την αρχικοποίηση του συστήματος, τον τερματισμό των συσκευών του συστήματος και όλων των kernel threads που σχετίζονται με αυτές αλλά και αυτών που είναι βοηθητικά για τη λειτουργία του RTS.

Επίσης ο τρόπος με τον οποίο γίνεται η εκτέλεση του kernel και οι ενέργειες που απαιτούνται για να επιτευχθεί αυτό, είναι τελείως άγνωστες στον προγραμματιστή της OpenCL εφαρμογής. Συγκεκριμένα το μόνο που έχει να κάνει ο προγραμματιστής είναι να καλέσει ορισμένες συναρτήσεις από το API της OpenCL. Ωστόσο κατά τη διάρκεια του χρόνου εκτέλεσης της εφαρμογής το σύστημα βρίσκει τη γεωμετρία του προβλήματος(αν ο προγραμματιστής δεν την έχει δώσει από το πρόγραμμα υπηρεσίας). Με δυναμικό τρόπο εντοπίζει τον αριθμό και τον τύπο των ορισμάτων χωρίς την παρέμβαση του προγραμματιστή. Επίσης ο τρόπος με τον οποίο το σύστημα μας διαβάζει το fat binary και τρέχει τον επιθυμητό kernel στη GPU είναι μια διαδικασία άγνωστη προς τον προγραμματιστή. Το μόνο που έχει να κάνει ο προγραμματιστής στην περίπτωση αυτή είναι να δημιουργήσει το fat binary αρχείο.

Ο τρόπος με τον οποίο μια εργασία χωρίζεται σε work-tasks τα οποία στη συνέχεια εκτελούνται από τα worker-threads είναι κρυφός προς τον προγραμματιστή της εφαρμογής. Δε γίνεται φανερός ο τρόπος με τον οποίο μια εργασία που δίνεται από το πρόγραμμα υπηρεσίας μετά την εισαγωγή της στη command queue ακολουθεί και άλλα στάδια. Ειδικότερα ο τρόπος εισαγωγής από την command queue στη ready queue, με τον out-of-order scheduling μηχανισμό, δεν είναι φανερός όπως επίσης φανερό δε γίνεται το γεγονός ότι αν μια συσκευή δεν έχει εργασία προς εκτέλεση χρησιμοποιεί τη μέθοδο του work-stealing. Τέλος ο τρόπος με τον οποίο γίνεται η ανάθεση των εργασιών στα devices από τη worker function είναι κρυφός από τον προγραμματιστή.

3.11 Start-up Overhead

Μια βασική μετρική για την απόδοση ενός συστήματος χρόνου εκτέλεσης είναι ο χρόνος που χρειάζεται για την εκκίνηση του(start-up overhead). Όπως έχει αναφερθεί στο προηγούμενο κεφάλαιο κατά την αρχικοποίηση του συστήματος γίνεται ανίχνευση των

διαθέσιμων συσκευών στο σύστημα μας, η κατάλληλη αρχικοποίηση αυτών, η δημιουργία ορισμένων kernel-level-threads και τέλος η δημιουργία ενός global context. Να αναφερθεί αρχικά ότι όλες αυτές οι ενέργειες είναι απαραίτητες κατά την αρχικοποίηση του συστήματος και δεν είναι δυνατόν ορισμένες από αυτές να γίνουν κατά τη διάρκεια εκτέλεσης του. Για τη μέτρηση του χρόνου έναρξης του συστήματος μας χρησιμοποιήθηκε η συνάρτηση time(). Παρατηρήσαμε ότι το μεγαλύτερο ποσοστό του χρόνου αρχικοποίησης αναλώνεται στην αρχικοποίηση των συσκευών. Αυτό είναι λογικό αφού η αρχικοποίηση των συσκευών απαιτεί αρχικά την ανίχνευση αυτών(probe device). Μετά την ανίχνευση των συσκευών χρειάζεται να η αρχικοποίηση των αντίστοιχων command queues. Επίσης κατά την αρχικοποίηση των συσκευών έχουμε το dynamic linking με τη run-time library της OpenCL. Η δημιουργία των threads κατά την αρχικοποίηση του συστήματος αποτελεί ένα μικρό ποσοστό του χρόνου αρχικοποίησης.

Συγκρίνοντας ωστόσο το χρόνο αρχικοποίησης του δικού μας συστήματος χρόνου εκτέλεσης με το αντίστοιχο σύστημα χρόνου εκτέλεσης της OpenCL παρατηρήσαμε ότι ο χρόνος του δεύτερου είναι ίδιος με το δικό μας. Το αποτέλεσμα αυτό είναι λογικό καθώς η αρχικοποίηση του συστήματος της OpenCL χρειάζεται να ανιχνεύσει και να αρχικοποιήσει τα devices καθώς επίσης δημιουργεί threads για τη λειτουργία του συστήματος, ενέργειες οι οποίες είναι κοινές και για τα 2 συστήματα. Το μόνο πλεονέκτημα το οποίο έχει το run-time της OpenCL είναι ότι δεν χρειάζεται κατά την αρχικοποίηση των συσκευών να κάνει dynamic link. Ωστόσο η επιβάρυνση της πράξης αυτής στο σύστημα μας είναι της τάξης μερικών ns, χρόνος ο οποίος δικαιολογεί και το ίδιο ποσό χρόνο που απαιτείται για την αρχικοποίηση των δυο διαφορετικών συστημάτων.

3.10 Host Program-Παράδειγμα εφαρμογής Vector Add

Στη συνέχεια θα αναλύσουμε την εφαρμογή της OpenCL η οποία κάνει Vector Add για να κατανοηθεί μέσα από αυτό το παράδειγμα ο ρόλος του προγράμματος υπηρεσίας(host program).

```
55      //Get an OpenCL platform
56      ciErr1 = clGetPlatformIDs(1, &cpPlatform, NULL);
57
58      printf("clGetPlatformID...\n");
59      if (ciErr1 != CL_SUCCESS)
60      {
61
62          Cleanup(EXIT_FAILURE);
63      }
64
65      //Get the devices
66      ciErr1 = clGetDeviceIDs(cpPlatform, CL_DEVICE_TYPE_GPU, 1, &cdDevice, NULL);
67      printf("clGetDeviceIDs...\n");
68      if (ciErr1 != CL_SUCCESS)
69      {
70
71          Cleanup(EXIT_FAILURE);
72      }
73
74      //Create the context
75      cxGPUContext = clCreateContext(0, 1, &cdDevice, NULL, NULL, &ciErr1);
76      printf("clCreateContext...\n");
77      if (ciErr1 != CL_SUCCESS)
78      {
79
80          Cleanup(EXIT_FAILURE);
81      }
```

Για να μπορέσει να τρέξει μια εφαρμογή της OpenCL προϋπόθεση είναι να δημιουργηθεί μια πλατφόρμα η οποία θα περιέχει τις συσκευές οι οποίες είναι διαθέσιμες στο σύστημα μας. Η δημιουργία της πλατφόρμας αυτή γίνεται με την κλήση της συνάρτησης στη γραμμή 56. Στη συνέχεια στη γραμμή 66 κοιτάμε να δούμε πόσες διαθέσιμες GPUs έχει το σύστημα μας. Η πληροφορία αυτή αποθηκεύεται στον πίνακα cdDevice, ο οποίος κρατάει σε κάθε θέση του πίνακα την GPU με τις αντίστοιχες πληροφορίες αυτής. Στη συνέχεια όπως αναφέραμε και στο κεφάλαιο 2 για την OpenCL χρειάζεται να δημιουργήσουμε ένα πλαίσιο(context) ώστε να μπορέσει να εκτελεστεί η εφαρμογή μας. Αυτί γίνεται στη γραμμή 75 του κώδικα, όπου το πλαίσιο αποθηκεύεται στο cxGPUContext.

```
82
83      // Create a command-queue
84      cqCommandQueue = clCreateCommandQueue(cxGPUContext, cdDevice, 0, &ciErr1);
85      printf("clCreateCommandQueue...\n");
86      if (ciErr1 != CL_SUCCESS)
87      {
88
89          Cleanup(EXIT_FAILURE);
90      }
91
92      // Allocate the OpenCL buffer memory objects for source and result on the device GMEM
93      cmDevSrcA = clCreateBuffer(cxGPUContext, CL_MEM_READ_ONLY, sizeof(cl_float) * szGlobalWorkSize, NULL, &ciErr1);
94      cmDevSrcB = clCreateBuffer(cxGPUContext, CL_MEM_READ_ONLY, sizeof(cl_float) * szGlobalWorkSize, NULL, &ciErr2);
95      ciErr1 |= ciErr2;
96      cmDevDst = clCreateBuffer(cxGPUContext, CL_MEM_WRITE_ONLY, sizeof(cl_float) * szGlobalWorkSize, NULL, &ciErr2);
97      ciErr1 |= ciErr2;
98      printf("clCreateBuffer...\n");
99      if (ciErr1 != CL_SUCCESS)
100     {
101
102         Cleanup(EXIT_FAILURE);
103     }
```

Στη συνέχεια για κάθε συσκευή η οποία βρίσκεται στο σύστημα μας (πληροφορία στον πίνακα cdDevice) πρέπει να δημιουργηθεί και να αρχικοποιηθεί η αντίστοιχη command queue. Αυτό γίνεται στη γραμμή 84 του κώδικα μας, όπου η συνάρτηση clCreateCommandQueue υλοποιεί αυτή τη λειτουργικότητα. Στη συνέχεια πρέπει να δημιουργηθούν οι memory buffers οι οποίοι είναι υπεύθυνοι για τη μεταφορά των δεδομένων από το host στα devices και το αντίστροφο. Αυτό γίνεται στις γραμμές 93 εως 96, όπου για κάθε πίνακα δημιουργείται και ο αντίστοιχος buffer.

```
105   printf("...loading VectorAdd.fatbin from file\n");
106   FILE* fp = fopen("oclVectorAdd.fatbin", "r");
107   if (!fp) {
108      exit(1);
109
110   }
111   fseek (fp , 0 , SEEK_END);
112   const size_t lSize = ftell(fp);
113   rewind(fp);
114   unsigned char* buffer;
115   buffer = (unsigned char*) malloc (lSize);
116   fread(buffer, 1, lSize, fp);
117   fclose(fp);
118   cl_int status;
119
120   printf("...creating add program\n");
121   cpProgram = clCreateProgramWithBinary(cxGPUContext,1, (const cl_device_id *)cdDevice,&lSize, (const unsigned char**)&buffer, &status, &ciErr1
122     if ( ciErr1 == CL_SUCCESS )
123      printf("SUCCESS!\n\n");
124
125     else {
126      printf("Error clCreateProgramWithBinary(): %d\n",ciErr1);
127      Cleanup(EXIT_FAILURE);
128   }
129
130      ciErr1 = clBuildProgram(cpProgram, 0, NULL, NULL, NULL, NULL);
131    printf("clBuildProgram...\n");
132    if (ciErr1 != CL_SUCCESS)
133    {
134
135        Cleanup(EXIT_FAILURE);
136    }
```

Στη συνέχεια στις γραμμές 105 έως 117 διαβάζουμε το fat binary που περιέχει τη συνάρτηση πυρήνα που θέλουμε να εκτελέσουμε. Στη γραμμή 121 δημιουργείται η αντίστοιχη δομή προγράμματος που χρησιμοποιείται στο πρότυπο της OpenCL ενώ στη γραμμή 130 η clBuildProgram μετατρέπει τον αρχικό κώδικα της συνάρτησης σε μορφή κατανοητή για την AMD GPU.

```
138     // Create the kernel
139     ckKernel = clCreateKernel(cpProgram, "VectorAdd", &ciErr1);
140     printf("clCreateKernel (VectorAdd)...\n");
141     if (ciErr1 != CL_SUCCESS)
142     {
143
144         Cleanup(EXIT_FAILURE);
145     }
146
147     // Set the Argument values
148     ciErr1 = clSetKernelArg(ckKernel, 0, sizeof(cl_mem), (void*)&cmDevSrcA);
149     ciErr1 |= clSetKernelArg(ckKernel, 1, sizeof(cl_mem), (void*)&cmDevSrcB);
150     ciErr1 |= clSetKernelArg(ckKernel, 2, sizeof(cl_mem), (void*)&cmDevDst);
151     ciErr1 |= clSetKernelArg(ckKernel, 3, sizeof(cl_int), (void*)&iNumElements);
152     printf("clSetKernelArg 0 - 3...\n\n");
153     if (ciErr1 != CL_SUCCESS)
154     {
155
156         Cleanup(EXIT_FAILURE);
157     }
158
159     // --------------------------------------------------------
160     // Start Core sequence... copy input data to GPU, compute, copy results back
161
162     // Asynchronous write of data to GPU device
163     ciErr1 = clEnqueueWriteBuffer(cqCommandQueue, cmDevSrcA, CL_FALSE, 0, sizeof(cl_float) * szGlobalWorkSize, srcA, 0, NULL, NULL);
164     ciErr1 |= clEnqueueWriteBuffer(cqCommandQueue, cmDevSrcB, CL_FALSE, 0, sizeof(cl_float) * szGlobalWorkSize, srcB, 0, NULL, NULL);
165     printf("clEnqueueWriteBuffer (SrcA and SrcB)...\n");
166     if (ciErr1 != CL_SUCCESS)
167     {
168
169         Cleanup(EXIT_FAILURE);
```

Στη γραμμή 139 δημιουργούμε τον kernel, διαβάζοντας με δυναμικό τρόπο στο run-time το αρχείο info.stub.txt ενέργεια η οποία είναι διαφανές από το χρήστη. Στο σημείο αυτό το run-time βρίσκει τη γεωμετρία του προβλήματος μας. Στη συνέχεια στις γραμμές 148 έως 151 θέτουμε τα ορίσματα της kernel function. Στις γραμμές 163 και 164 μεταφέρουμε/αντιγράφουμε τα δεδομένα της εφαρμογής μας στη GPU. Αυτό γίνεται χρησιμοποιώντας τους memory buffers που ορίσαμε πιο πάνω. Τα βήματα που περιγράψαμε μέχρι στιγμής είναι συγκεκριμένα και τα εφαρμόζουμε σε κάθε εφαρμογή OpenCL την οποία θέλουμε να εκτελέσουμε.

```
171     // Launch kernel
172     ciErr1 = clEnqueueNDRangeKernel(cqCommandQueue, ckKernel, 1, NULL, &szGlobalWorkSize, &szLocalWorkSize, 0, NULL, NULL);
173     printf("clEnqueueNDRangeKernel (VectorAdd)...\n");
174     if (ciErr1 != CL_SUCCESS)
175     {
176
177         Cleanup(EXIT_FAILURE);
178     }
179
180     // Synchronous/blocking read of results, and check accumulated errors
181     ciErr1 = clEnqueueReadBuffer(cqCommandQueue, cmDevDst, CL_TRUE, 0, sizeof(cl_float) * szGlobalWorkSize, dst, 0, NULL, NULL);
182     printf("clEnqueueReadBuffer (Dst)...\n\n");
183     if (ciErr1 != CL_SUCCESS)
184     {
185
186         Cleanup(EXIT_FAILURE);
187     }
188     //-----------------------------------------------------
189
190          for (o=0; o < 40 ; o++)
191        printf("%d Apotelesma: %f\n",o,dst[o]);
192
193     // Cleanup and leave
194     Cleanup (EXIT_SUCCESS);
195 }
196
```

Η εφαρμογή μας είναι τώρα έτοιμη να εκτελέσει την συνάρτηση πυρήνα. Αυτό γίνεται στη γραμμή 172 όπου καλούμε τη συνάρτηση του πυρήνα μέσω της clEnqueueNDRangeKernel. Τέλος στη γραμμή 180 παίρνουμε τα αποτελέσματα από τη GPU και τα αντιγράφουμε πίσω στη CPU για να τα διαβάσουμε στη συνέχεια. Η μεταφορά των δεδομένων από τη GPU προς τη CPU γίνεται μέσω της συνάρτησης clEnqueueReadBuffer, όπου χρησιμοποιούμε για άλλη μια φορά τους memory buffers για τη μεταφορά των δεδομένων. Η ανάγνωση του αποτελέσματος γίνεται στο for loop στη γραμμή 190 και 191 του κώδικα μας.

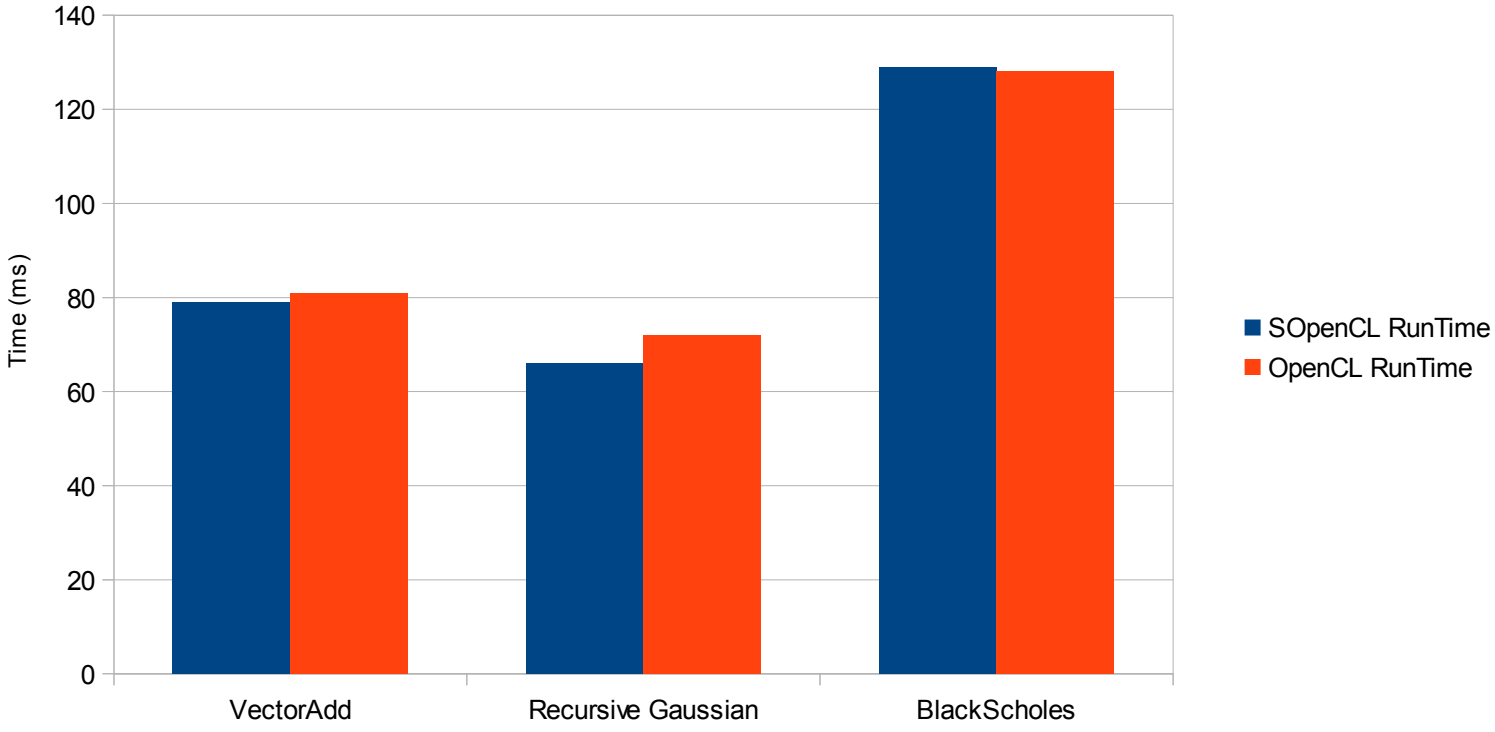
Το παράδειγμα αυτό μας δείχνει ένα γενικό τρόπο με τον οποίο ένας προγραμματιστής μπορεί να εκτελεί εφαρμογές σε OpenCL χρησιμοποιώντας το σύστημα χρόνου εκτέλεσης μου αναπτύχθηκε στα πλαίσια της διπλωματικής εργασίας. Βλέπουμε ότι υπάρχει πλήρης διαφάνεια σχετικά με τις ενέργειες που εκτελεί το RTS στο παρασκήνιο. Ο προγραμματιστής το μόνο που χρειάζεται να κάνει είναι να καλεί τις αντίστοιχες συναρτήσεις που η βιβλιοθήκη του συστήματος χρόνου εκτέλεσης του παρέχει.

Κεφάλαιο 4

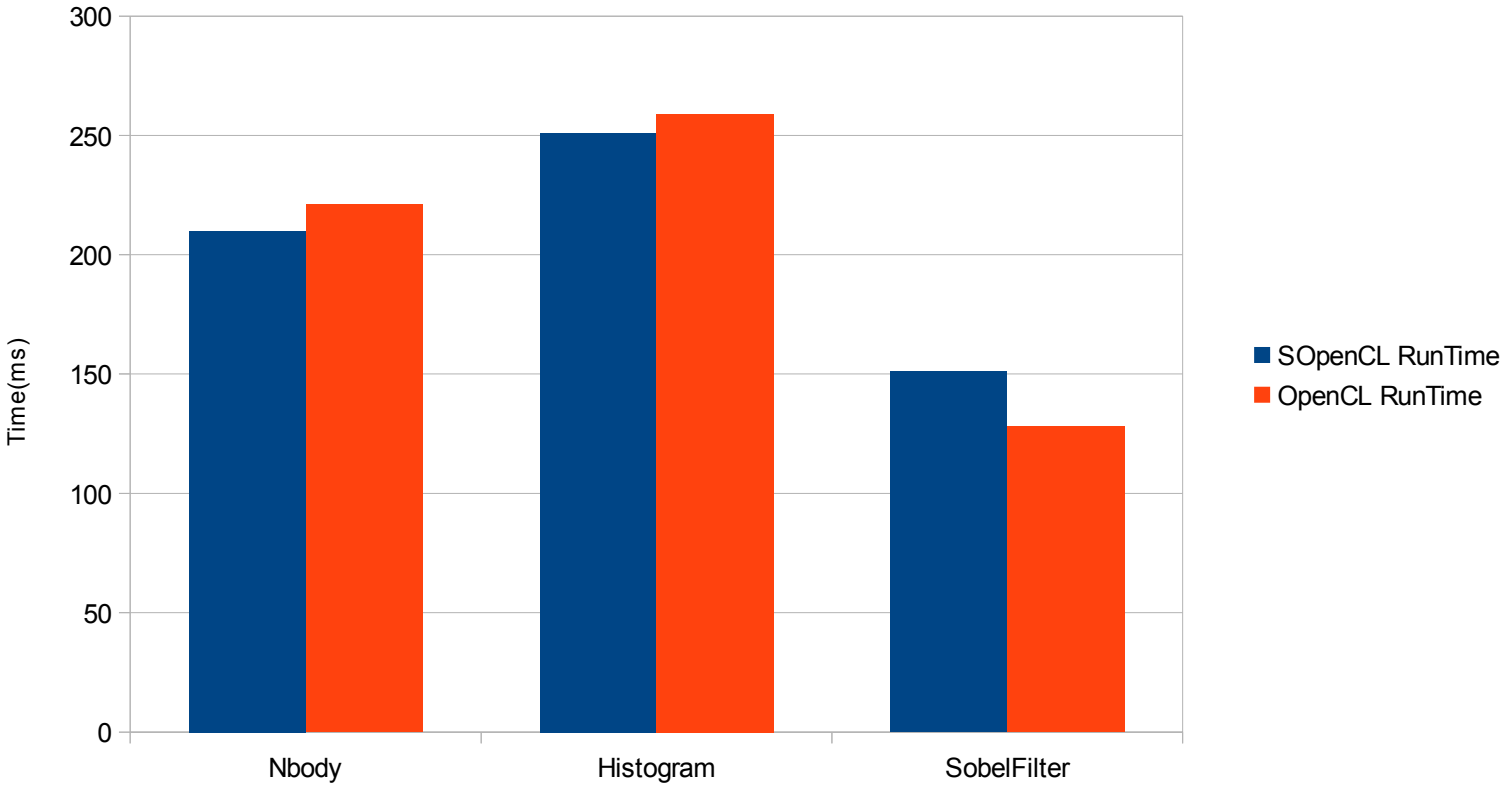
Απόδοση του Συστήματος

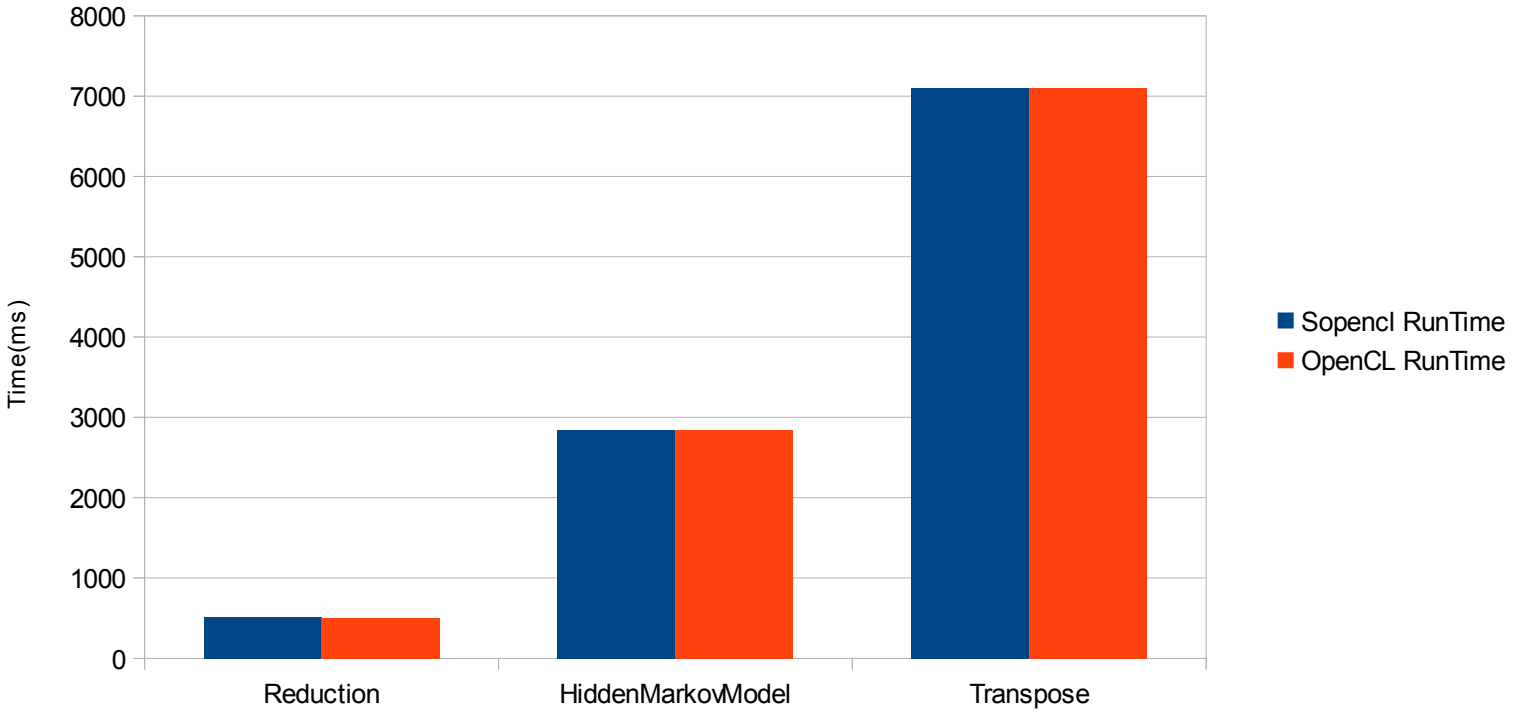
4.1 BenchMarks Applications

Στο κεφάλαιο αυτό παραθέτονται ορισμένα διαγράμματα χρόνων εκτέλεσης από εφαρμογές από το SDK της OpenCL. Με το πέρας των πειραμάτων είδαμε ότι οι χρόνοι εκτέλεσης των εφαρμογών με τη run-time library που αναπτύξαμε δε διαφέρουν από τους χρόνους εκτέλεσης με τη run-time library της OpenCL.



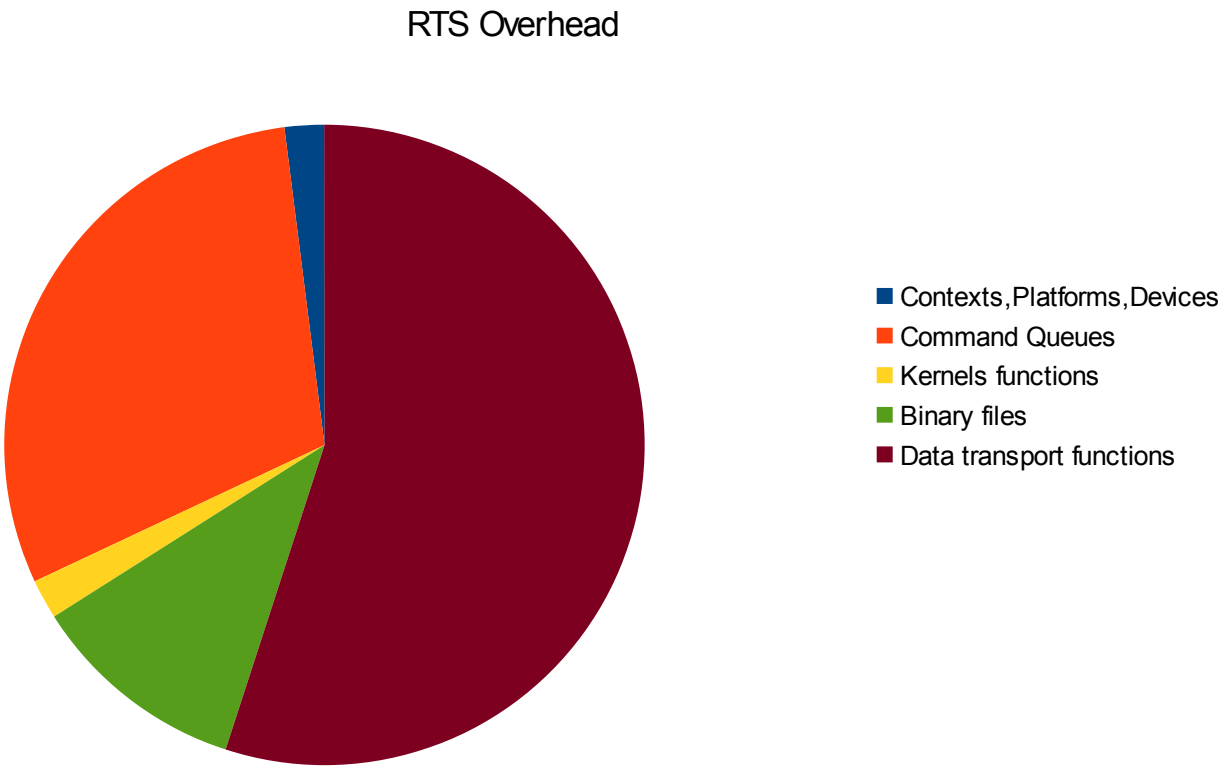
Για το VectorAdd χρησιμοποιήσαμε float numbers όπου ο κάθε πίνακας είχε `11444777` στοιχεία, Global Work Size `11444992`, local Work Size 256 και number of work Groups 44707.





4.2 RTS Overhead

Τέλος για την απόδοση του συστήματος μετρήσαμε το χρόνο που χρειάζονται για να εκτελεστούν οι συναρτήσεις της OpenCL όταν καλούνται από το πρόγραμμα υπηρεσίας. Αξίζει να αναφερθεί ότι οι συναρτήσεις που αφορούν τη δημιουργία Context, Platform, SetKerelArgs, GetDeviceIDs είχανε σχεδόν μηδαμινό χρόνο εκτέλεσης και δεν επιβάρυναν καθόλου την εκτέλεση των εφαρμογών. Η συνάρτηση clCreateCommandQueue που μετρήθηκε κατά την εκτέλεση των παραπάνω εφαρμογών χρειάζεται κατά μέσο όρο 18 ms, όσο ακριβώς χρειάζεται και από τη run-time library της OpenCL. Επίσης η διαδικασία της ανάγνωσης του FatBinary και οι απαραίτητες πράξεις με αυτό, χρειάζονται 5 ms κατά μέσο όρο με βάση τις μετρήσεις που έγιναν. Ένα μεγάλο ποσοστό του χρόνου εκτέλεσης της εφαρμογής, το οποίο είναι και λογικό, αναλώνεται στη μεταφορά των δεδομένων από και προς τη GPU. Ο χρόνος αυτός είναι ανάλογος του μεγέθους των δεδομένων και διαφέρει από εφαρμογή σε εφαρμογή.


RTS Overhead
Contexts,Platforms,Devices
Command Queues
Kernels functions
Binary files
Data transport functions

Κεφάλαιο 5

Επίλογος

5.1 Γενικά Συμπεράσματα

Η κατανόηση της αρχιτεκτονικής μιας κάρτας γραφικών είναι ύψιστης σημασίας αν θέλουμε να γράφουμε κώδικα υψηλών επιδόσεων. Σήμερα ένα προγραμματιστής ο οποίος θέλει να γράψει κώδικα για να εκτελείται σε κάρτες γραφικών ή για να γράψει παράλληλο κώδικα ο οποίος θα έχει μεγαλύτερη απόδοση από τον αντίστοιχο σειριακό ή ακόμα περισσότερο να γράψει κώδικα για ένα ετερογενές παράλληλο σύστημα πρέπει να είναι σε θέση να κατανοήσει αρκετά θέματα που αφορούν την αρχιτεκτονική του υπολογιστικού συστήματος στο οποίο θέλει να εκτελέσει τον κώδικα του.

Τα παραπάνω είναι ακόμα πιο απαιτητικά για έναν προγραμματιστή ο οποίος θέλει να φτιάξει ένα σύστημα χρόνου εκτέλεσης το οποίο θα υποστηρίζει ετερογενής συσκευές με εντελώς διαφορετικά αρχιτεκτονικά χαρακτηριστικά. Αυτό έγινε φανερό στον τρόπο με τον οποίο δομήθηκε αρχιτεκτονικά το σύστημα μας. Είδαμε ορισμένες αρχιτεκτονικές διαφορές όσον αφορά την υποστήριξη της CPU και της GPU στο ίδιο σύστημα. Για την πρώτη συσκευή χρειάστηκε ένας S/W μηχανισμός για την υποστήριξη των ασύγχρονων εντολών, ενώ στο δεύτερο τύπο συσκευής υποστηρίζεται από το H/W αυτή η λειτουργία. Επίσης με διαφορετικό τρόπο εκτελούνται οι συναρτήσεις πυρήνα στις συσκευές.

5.2 Μελλοντική ανάπτυξη του συστήματος

Το σύστημα χρόνου εκτέλεσης που αναπτύχθηκε για τους σκοπούς της διπλωματικής εργασίας έχει τη δυνατότητα περαιτέρω ανάπτυξης για την υποστήριξη περαιτέρω αρχιτεκτονικών συσκευών. Αρχικά ένα πρώτο βήμα για την επιπλέον ανάπτυξη του είναι η ενσωμάτωση στο σύστημα αυτό μιας διαφορετικής αρχιτεκτονικής GPU, όπως είναι αυτές της Nvidia. Η ενσωμάτωση δύο διαφορετικών αρχιτεκτονικών GPU στο σύστημα μας, πρώτον κάνει το σύστημα μας ακόμη πιο ετερογενές και δίνει τη δυνατότητα για επιπλέον αύξηση της απόδοσης, γεννάει ωστόσο και νέα ερωτήματα και προκλήσεις όπως σε πια αρχιτεκτονική συσκευής να τρέξει μια εφαρμογή, με πια κριτήρια. Με την ενσωμάτωση αυτή γεννάτε επίσης η πρόκληση της εκτέλεσης μιας συνάρτησης πυρήνα σε δυο διαφορετικές συσκευές, να τρέχει δηλαδή ένα τμήμα της συνάρτησης πυρήνα σε μια GPU και ένα άλλο τμήμα της στην άλλη GPU. Αυτό φυσικά προϋποθέτει το σύστημα μας να δέχεται κατάλληλη πληροφορία από τον compiler για τις εξαρτήσεις μνήμης, πια κομμάτια του κώδικα μπορούν να τρέξουν σε ξεχωριστές GPUs, με ποιον τρόπο θα πρέπει να σπάσει η γεωμετρία της εφαρμογής μας ώστε να πετύχουμε μεγαλύτερη απόδοση.

Η ενσωμάτωση στην αρχιτεκτονική του συστήματος των FPGA's είναι μια ακόμη μεγάλη πρόκληση για την επιπλέον ανάπτυξη του συστήματος μας. Ζητήματα που πρέπει να λυθούν είναι ο τρόπος επικοινωνίας των GPU με την FPGA(π.χ μέσω PCI-e), με ποιον τρόπο το σύστημα μας θα αναγνωρίζει αν η FPGA είναι configure με το σωστό H/W που χρειάζεται η εφαρμογή μας. Επίσης η ανάπτυξη κατάλληλων drivers προκειμένου να γίνεται η επικοινωνία του S/W με το H/W. Τέλος ένα θέμα που πρέπει να λυθεί για να μπορέσουμε να έχουμε την επικοινωνία και τη μεταφορά δεδομένων μεταξύ της FPGA, GPU και CPU, είναι η ανάπτυξη του H/W infrastructure στην FPGA ώστε να μπορέσει να υποστηριχθεί αυτή η διαδικασία.

Παράρτημα

Τεχνικά ζητήματα

Σκοπός της ενότητας αυτής είναι να τα βήματα τα οποία είναι απαραίτητα προκειμένου να μπορέσουμε να τρέξουμε μια εφαρμογή της OpenCL με το σύστημα χρόνου εκτέλεσης που υλοποιήθηκε για το σκοπό της διπλωματικής αυτής εργασίας. Θα περιγράψουμε τη διαδικασία σε βήματα για να γίνει όσο το δυνατό πιο καθαρή η διαδικασία.

1. Πρώτο βήμα της διαδικασίας είναι να παραχθεί η βιβλιοθήκη του RTS. Αρχικά πάμε στο φάκελο όπου έχουμε το κώδικα του συστήματος. Στη συνέχεια στο directory Release/src υπάρχει ένα αρχείο με το όνομα subdir.mk. Ανοίγουμε το αρχείο αυτό και στη σειρά 62 αντικαθιστούμε το directory path με το δικό μας path directory στο οποίο βρίσκεται ο φάκελος include του συστήματος μας. Αυτό γίνεται ώστε να ξέρει ο compiler το directory στο οποίο είναι τα header files τα οποία χρειάζεται το σύστημα μας. Στην συνέχεια πίσω στο directory Release. Εκτελούμε την εντολή make clean και στη συνέχεια την εντολή make. Η run-time library είναι έτοιμη με το όνομα libglOpenCL_runtime_lib.a

2. Στη συνέχεια δημιουργούμε ένα directory και κάνουμε copy paste το αρχείο clbingenerate.c και τον αρχείο που περιέχει τον kernel που εκτελεί η εφαρμογή που θέλουμε να εκτελέσουμε. Το αρχείο clbingenerate.c περιέχει τον κώδικα ο οποίος εξάγει το IL_AMD κώδικα από τον kernel της εφαρμογής μας. Για να γίνει αυτό ανοίγουμε το αρχείο και στη σειρά 397 στο τρίτο όρισμα της συνάρτησης generateAmdBin βάζουμε το όνομα του αρχείου που περιέχει τον kernel(με την κατάληξη .cl) και αν θέλουμε αλλάζουμε και το όνομα του εκτελέσιμου αρχείου που θα παραχθεί(δίνεται στο τέταρτο όρισμα με default name “kernel.bin.0”). Στη συνέχεια κάνουμε compile το αρχείο αυτό με την εντολή gcc -o clbingenerate.c -lelf /usr/libOpenCL.so -I$AMDAPPSDKROOT/include. Με το όρισμα -lelf κάνουμε link με την βιβλιοθήκη που χειρίζεται τα elf files(μπορεί να μην είναι εγκατεστημένη στο σύστημα μας και να χρειαστεί να την εγκαταστήσει ο χρήστης). Το όρισμα /usr/libOpenCL.so το βάζουμε για να μπορέσουμε να διαχειριστούμε συναρτήσεις της OpenCL που ο κώδικας χρησιμοποιεί. Τέλος το όρισμα -I$AMDAPPSDKROOT/include χρειάζεται για να μπορεί να ξέρει ο compiler που θα βρει τα header files από το SDK της OpenCL AMD, τα οποία χρησιμοποιεί ο κώδικας του αρχείου. Αφού γίνει επιτυχώς η διαδικασία του compilation στη συνέχεια τρέχουμε το εκτελέσιμο αρχείο ./clbingenerate. Δημιουργήται έτσι ένα άλλο

εκτελέσιμο αρχείο με το όνομα kernel.bin.0. Το αρχείο αυτό είναι που θα βάλουμε στο fat binary file.

3. Με την επιτυχή δημιουργία του kernel.bin.0 είμαστε τώρα έτοιμη να ενσωματώσουμε το αρχείο αυτό μαζί με όσο άλλα εκτελέσιμα αρχεία επιθυμούμε στο fat binary file. Στο directory fatbinaries κάνουμε compile τα αρχεία compine.c και main.c με την εντολή gcc -Wall compine.c main.c -o compine_files. Παράγεται λοιπόν ένα εκτελέσιμο αρχείο με το όνομα compine_files το οποίο είναι υπεύθυνο για τη δημιουργία του fat binary αρχείου που θέλουμε. Για να γίνει αυτό εκτελέσουμε το αρχείο αυτό με πρώτο όρισμα το όνομα του fat binary file που θέλουμε να δημιουργηθεί, με κατάληξη .fatbin. Τα επόμενα ορίσματα είναι τα ονοματα των kernels που θέλουμε να περιέχει το fatbin. Ένα παράδειγμα: ./compine_files kernelsfunction.fatbin kernel0.bin.0 kernel1.bin.0 kernel2.bin.0.

4. Στη συνέχεια πρέπει να δημιουργήσουμε το αρχείο info.stub.txt. Στο αρχείο αυτό όπως αναλύσαμε σε παραπάνω ενότητα πρέπει ο χρήστης να δώσει τον αριθμό των ορισμάτων και τον τύπο αυτών. Σαν πρώτο όρισμα δίνεται ο αριθμός των kernel functions που έχει η εφαρμογή μας. Σαν δεύτερο όρισμα δίνεται το όνομα της συνάρτησης του kernel. Τρίτο όρισμα είναι ο αριθμός των παραμέτρων της συνάρτησης. Στη συνέχεια δίνονται τα ορίσματα με τον εξής τρόπο. Με 0 αναφέρεται ότι το όρισμα είναι global ενώ με 3 ότι είναι local. Για παράδειγμα για τη συνάρτηση Vector Add με ορίσματα __kernel void VectorAdd(__global const float* a, __global const float* b, __global float* c, int iNumElements) το αρχείο θα είναι 01 VectorAdd 04 0 0 0 0 0 0 3 4. Το 01 στην αρχή μας λέει ότι έχουμε μια συνάρτηση πυρήνα. Το VectorAdd είναι το όνομα της συνάρτησης πυρήνα. Το 04 στη συνέχεια μας δείχνει ότι η συνάρτηση έχει 4 ορίσματα. Το 0 που ακολουθεί δείχνει ότι το πρώτο όρισμα είναι global ενώ το 0 που ακολουθεί μας λέει ότι έχει μέγεθος 0 bytes(είναι pointer). Το 3 μας λέει ότι το όρισμα είναι local ενώ το 4 μας λέει ότι το μέγεθος του ορίσματος είναι 4 bytes(1*int(4bytes)=4 bytes).

5. Τελευταίο βήμα είναι να εκτελέσουμε την εφαρμογή της OpenCL που επιθυμούμε. Στο directory όπου βρίσκεται η εφαρμογή μας αντιγράφουμε το info.stub.txt το οποίο δημιουργήσαμε προηγουμένως, το fat binary αρχείο και τη run-time library που δημιουργήσαμε στο βήμα 1. Για να κάνουμε το compile: gcc -o “όνομα εφαρμογής” “όνομα εφαρμογής.c” -lelf -L. -l glOpenCL_runtime_lib -ldl -I “directory path του include φακέλου του κώδικα του RTS” -lpthread.

Βιβλιογραφία

[1] Muhsen Owaida, Nikolaos Bellas, Christos Antonopoulos, Konstantis Daloukas, Charalambos Antoniadis. "Massively parallel Programming Models Used as Hardware Description Languages : The OpenCL case".

[2]. Konstantis Daloukas, Christos D.Antonopoulos, Nikolaos Bellas "GLOpenCL:OpenCL support on Hardware-and Software-Managed Cache Multicores"

[3]. Muhsen Owaida, Nikoaos Bellas, Christos Antonopoulos, Konstantis Daloukas, Charalambos Antoniadis. "Synthesis of Platform Architectures from OpenCL Programs"

[4].B. Jang, P. Mistry, D. Schaa, , and D. Kaeli. "Exploiting memory access patterns to improve memory performance in data parallel architectures". Parallel and Distributed Systems, IEEE Transactions on (submitted), 2010.

[5].NVIDIA. CUDA Programming Guide 3.0, Feb 2010.

[6].Static Memory Access Pattern Analysis on a Massively Parallel GPU Byunghyun Jang, Dana Schaa, Perhaad Mistry, and David Kaeli Computer Architecture Labratory Northeastern University

[7]."Achieving a Single Compute Device Image in OpenCL for Multiple GPUs" Jungwon Kim Honggyu Kim Joo Hwan Lee Jaejin Lee Center for Manycore Programming School of Computer Science and Engineering Seoul National University

[8].ATI Stream Software Development Ket (SDK) v2.1. AMD, 2010. http://developer.amd.com/gpu/atistreamsdk/pages/default.aspx.

[9].Khronos OpenCL Working Group. The OpenCL Specification Version1.0. Khronos Group, 2009. http://www.khronos.org/OpenCL

[10].NVIDIA CUDA C Best Practices Guide 3.1. NVIDIA, May 2010

[11].Programming Massively Parallel Processors: A Hands-on Approach (David B. Kirk and Wen-mei W. Hwu)

[12].H. Franke and R. Russel. Fuss, Futexes and Furlocks:Fast User-Space Locking in Linux. In Proceedings of the Otawa Linux Symposium, pages 85{97, 2002.

[13] AMD Tutorial

[14] OpenCL Tutorial Guide

[15] Nvdia Guide

[16] http://en.wikipedia.org/wiki/Executable_and_Linkable_Format