Πανεπιστήμιο Πειραιώς – Τμήμα Πληροφορικής

Πρόγραμμα Μεταπτυχιακών Σπουδών

«Πληροφορική»

Μεταπτυχιακή Διατριβή

Τίτλος Διατριβής	**Επιτάχυνση του αλγορίθμου Smith-Waterman για ανίχνευση ηχητικών εφέ με χρήση GPU** **GPU-based acceleration of Smith-Waterman algorithm for audio effects detection**
Ονοματεπώνυμο Φοιτητή	**Θεόδωρος Πασχάλης**
Πατρώνυμο	**Σωτήριος**
Αριθμός Μητρώου	**ΜΠΣΠ/13087**
Επιβλέπων	**Μιχάλης Ψαράκης, Επίκουρος Καθηγητής**

Ημερομηνία Παράδοσης **Ιούλιος 2017**

Τριμελής Εξεταστική Επιτροπή

(υπογραφή)	(υπογραφή)	(υπογραφή)
Μιχάλης Ψαράκης Επίκουρος Καθηγητής	Παναγιώτης Κοτζανικολάου Επίκουρος Καθηγητής	Άγγελος Πικράκης Επίκουρος Καθηγητής

Περίληψη

Η αναγνώριση προτύπων σε ένα μεγάλο πλήθος δεδομένων αποτελεί έναν από τους σημαντικότερους τομείς στην σύγχρονη πληροφορική. Ένα τέτοιο πρόβλημα στο χώρο της βιοπληροφορικής είναι η ευθυγράμμιση μοριακών ακολουθιών, δηλαδή η εύρεση παρόμοιων υποακολουθιών ανάμεσα σε δύο ακολουθίες πρωτεϊνών. Ένας αλγόριθμος που χρησιμοποιείται για την ευθυγράμμιση μοριακών ακολουθιών είναι ο δυναμικός αλγόριθμος Smith-Waterman. Η ακρίβεια του αλγόριθμου αυτού είναι πολύ μεγάλη, αλλά αντισταθμίζεται από τις μεγάλες υπολογιστικές απαιτήσεις του. Η εξυπηρέτηση των μεγάλων υπολογιστικών απαιτήσεων του αλγόριθμου Smith-Waterman, μπορεί να πραγματοποιηθεί με την χρήση υλικού παράλληλης αρχιτεκτονικής, όπως οι κάρτες γραφικών του υπολογιστικού μοντέλου της CUDA. Οι κάρτες γραφικών μας δίνουν την δυνατότητα να εκτελέσουμε χιλιάδες νήματα, τα οποία μπορούν να πραγματοποιήσουν χιλιάδες υπολογισμούς παράλληλα.

Σκοπός της παρούσας διπλωματικής είναι η επιτάχυνση του αλγόριθμου Smith - Waterman για την ανίχνευση ηχητικών εφέ σε μία ροή ηχητικών δεδομένων. Για να ικανοποιηθούν οι μεγάλες υπολογιστικές απαιτήσεις, ο αλγόριθμος υλοποιείται πάνω στο προγραμματιστικό μοντέλο της CUDA, για να εκμεταλλευτούμε τις μεγάλες δυνατότητες παραλληλίας που μας παρέχουν οι κάρτες γραφικών. Το προγραμματιστικό μοντέλο της CUDA αποτελείται από δύο κομμάτια: το κομμάτι που εκτελείται από την CPU και το κομμάτι που εκτελείται από την GPU. Στην συγκεκριμένη εφαρμογή το κομμάτι της CPU αναλαμβάνει κυρίως το φόρτωμα των αρχείων στο σύστημα και την μεταφορά των απαραίτητων δεδομένων στην GPU. Το κομμάτι της GPU αναλαμβάνει τον υπολογισμό του πίνακας κόστους αντικατάστασης και την εκτέλεση του αλγόριθμου Smith – Waterman. Τα ηχητικά αρχεία μορφοποιούνται πάνω στην μορφή ASE, ώστε να έχουμε μια ακολουθία διανυσμάτων 62 διαστάσεων. Ο πίνακας κόστους αντικατάστασης που χρησιμοποιούμε σε αναλογία με τον αντίστοιχο της βιοπληροφορικής, είναι ένας πίνακας που μας δίνει το συνημίτονο μεταξύ δύο συγκρινόμενων διανυσμάτων.

Το μεγάλο μειονέκτημα της κάρτας γραφικών είναι ο περιορισμός των επιδόσεων σε εφαρμογές που υπάρχει συχνά επικοινωνία μεταξύ επεξεργαστή και μνήμη. Ο αλγόριθμος Smith – Waterman απαιτεί πολλές προσπελάσεις στην μνήμη και αυτό περιορίζει τις επιδόσεις της εφαρμογής. Παρ' όλους τους περιορισμούς που έχουμε λόγω της συχνής επικοινωνίας με την μνήμη, η χρήση της κάρτας γραφικών μας δίνει σημαντικά βελτιωμένους χρόνους σε σχέση με τις επιδώσεις της CPU.

Abstract

The pattern recognition in a large number of data is one of the most important sectors in the modern computing. One such problem in bioinformatics field is to align the molecular sequences, i.e. finding similar subsequences between two protein sequences. An algorithm used to align molecular sequences is the dynamic algorithm Smith-Waterman. The accuracy of this algorithm is very large, but is offset by the large computational requirements. To satisfy the large computational requirements of the Smith-Waterman algorithm we can utilize a parallel hardware architecture, such as video cards based on the computational model of CUDA. The graphics cards enable you to execute thousands of threads, which can perform thousands of calculations in parallel.

The purpose of this thesis is the acceleration of the algorithm Smith - Waterman for detecting sound effects in a stream of audio data. To meet the large computational requirements, the algorithm is implemented on the CUDA programming model, to take advantage of the great potential parallelism provided by graphics cards. The CUDA programming model consists of two parts: the part that is executed by the CPU and the one that is executed by the GPU. In this application, the piece of CPU mainly undertakes the uploading of files to the system and transfer the necessary data to the GPU. The part that is executed by the GPU, is to calculate the replacement cost matrix and the execution of the Smith – Waterman algorithm. The audio files are formatted on the ASE format, so that we have a sequence of 62-dimensional vectors. The replacement cost matrix used in analogy with the corresponding bioinformatics, is a table that gives the cosine between two compared vectors.

The big disadvantage of the graphics card is the limiting performance in communication between processor and memory. The algorithm Smith - Waterman requires many accesses to memory, and this limits the application's performance. Despite all the limitations we have due to the frequent communication with the memory, the use of the graphics card provides significantly improved times in comparison with the CPU implementation.

Ευχαριστίες

Η παρούσα διπλωματική πραγματοποιήθηκε στα πλαίσια του μεταπτυχιακού προγράμματος «Προηγμένα Συστήματα Πληροφορικής» του τμήματος Πληροφορικής.

Πριν την παρουσίαση θα ήθελα να ευχαριστήσω θερμά τον επιβλέποντα καθηγητή της διπλωματικής εργασίας, Επίκουρο Καθηγητή Μιχάλη Ψαράκη, για την διαρκεί καθοδήγηση του, τις πολύτιμες υποδείξεις του και την εμπιστοσύνη που μου έδειξε, καθ' όλη την διάρκεια της συνεργασίας μας.

Επίσης, θα ήθελα να ευχαριστήσω τους γονείς μου και τον αδερφό μου, για την υποστήριξη τους, είτε ηθική είτε οικονομική. Η βοήθεια και το κουράγιο που μου πρόσφεραν κρίθηκαν απαραίτητα για την ολοκλήρωση της μεταπτυχιακής μου διατριβής.

Περιοχόμενα

1. Εισαγωγή

Οι κάρτες γραφικών στην σύγχρονη πληροφορική δεν αποτελούν απλώς μια μονάδα προβολής εικόνας. Λόγων των μεγάλων δυνατοτήτων στην εκτέλεση πολλών παράλληλων διεργασιών αποτελούν βασικές μονάδες για την επίλυση προβλημάτων που απαιτούν μεγάλους υπολογιστικούς πόρους.

Με την εξέλιξη των CPU, οδηγηθήκαμε σε ένα τέλμα λόγω των υψηλών θερμοκρασιών που εμφανίζουν οι επεξεργαστές με πολύ υψηλές συχνότητες. Οι κατασκευαστές υπολογιστικών συστημάτων στράφηκαν τότε σε αρχιτεκτονικές που υποστηρίζουν παράλληλες διεργασίες. Έτσι, μείωσαν την συχνότητα λειτουργίας των επεξεργαστών, αλλά αυξήσαν τους πυρήνες τους για να γίνεται δυνατή η εκτέλεση πολλών διεργασιών παράλληλα και να αυξήσουν τις επιδόσεις. Πάνω σε αυτή την ιδέα βασίστηκε και η παραγωγή των καρτών γραφικών. Σκοπός των κατασκευαστών ήταν η δημιουργία του κατάλληλου υλικού που θα μπορεί να εκτελεί παράλληλες διεργασίες, οι οποίες θα πραγματοποιούν υπολογισμούς για κάθε pixel της οθόνης. Έτσι η κάρτα γραφικών μπορεί να εκτελέσει παράλληλα χιλιάδες νήματα και τις αντίστοιχες διεργασίες. Παράλληλα όμως, η κάρτα γραφικών έγινε και μια μονάδα που μας δίνει την δυνατότητα εκτέλεσης γενικού τύπου υπολογισμών με δυνατότητες μεγάλης παραλληλίας.

Οι δυνατότητες αυτές της κάρτας γραφικών βρίσκουν εφαρμογή σε προβλήματα τα οποία έχουν μεγάλες απαιτήσεις σε πόρους. Προβλήματα τα οποία απαιτούν πολλούς υπολογισμούς σε πολλά δεδομένα επωφελούνται από τις δυνατότητες παραλληλίας της κάρτας γραφικών. Ειδικότερα προβλήματα με υπολογισμούς σε δεδομένα οι οποίοι είναι ανεξάρτητοι μεταξύ τους, μπορούν να εκτελεστούν παράλληλα και να επιταχυνθούν τα αποτελέσματα σε μεγάλο βαθμό. Ένα πρόβλημα, το οποίο προέρχεται από τον τομέα της βιοπληροφορικής και έχει μεγάλες απαιτήσεις σε πόρους είναι η ευθυγράμμιση μοριακών ακολουθιών. Ένας αλγόριθμος που πραγματοποιεί την ευθυγράμμιση είναι ο αλγόριθμος Smith – Waterman. Αποτελεί ένα δυναμικό αλγόριθμο ο οποίος έχει μεγάλη ακρίβεια, αλλά και μεγάλες απαιτήσεις σε πόρους.

Ο εντοπισμός ομοιότητας σε δυο υποακολουθίες δυο ακολουθιών μας χρησιμοποιήθηκε στο τομέα της βιολογίας για να εντοπίζονται ακολουθίες πρωτεϊνών ή DNA, οι οποίες έχουν παρόμοια χαρακτηριστικά. Οι πρωτεΐνες λόγω της ομοιότητας που μπορεί να παρουσιάζουνε σε κάποιο κομμάτι τους, έχουν παρόμοιο συμπεριφορά και έτσι μπορούν να εντοπιστούν διάφορα βιολογικά φαινόμενα. Ο αλγόριθμος Smith-Waterman είναι βασισμένος στο σκεπτικό της σύγκρισης οποιουδήποτε κομματιού μιας ακολουθίας με αυτά μιας άλλης, έτσι ώστε να βρεθούν τα κομμάτια με την μεγαλύτερη δυνατή ομοιότητα. Συμπεραίνουμε λοιπόν, ότι αυτός ο αριθμός των υποακολουθιών και των συγκρίσεων που προκύπτουν είναι πολύ μεγάλος. Ο αλγόριθμος Smith-Waterman έχει την ευαισθησία να εντοπίσει τις υποακολουθίες με την μεγαλύτερη δυνατή ομοιότητα, αλλά είναι ένας πολύ απαιτητικός αλγόριθμος σε πόρους και σε χρόνο.

Ο σκοπός αυτής της διπλωματικής είναι η χρήση του αλγόριθμου Smith – Waterman για την ανίχνευση ηχητικών εφέ σε ροή ηχητικών δεδομένων, με την χρήση των καρτών γραφικών CUDA, για την επιτάχυνση των αποτελεσμάτων. Σε αναλογία με την βιολογία, αντί για μοριακές ακολουθίες, έχουμε ακολουθίες ηχητικών δεδομένων. Με τη χρήση της κάρτας γραφικών εκμεταλλευόμαστε τις δυνατότητες παραλληλίας που μας παρέχει για να επιταχύνουμε τα αποτελέσματα μας.

Στο κεφάλαιο 2 αρχικά πραγματοποιούμε μια περιγραφή της εξέλιξης των καρτών γραφικών. Περιγράφεται η αρχική χρήση της κάρτας γραφικών και πως αυτή εξελίχθηκε σε μια ισχυρή μονάδα γενικών υπολογισμών. Παρατίθεται οι λόγοι για τους οποίους μπόρεσε να ξεπεράσει σε επιδόσεις την CPU. Στην συνέχεια περιγράφεται η δομή των παράλληλων υπολογιστικών συστημάτων, ένα εκ των οποίων είναι και η κάρτα γραφικών. Παρουσιάζονται οι βασικές διαφορές της κάρτας γραφικών σε σχέση με την CPU και παρατίθενται διάφορες εναλλακτικά παράλληλα συστήματα και γλώσσες παράλληλου προγραμματισμού, μαζί με μια σύντομη περιγραφή των πλεονεκτημάτων και των μειονεκτημάτων. Ένα από αυτά τα μοντέλα είναι και το μοντέλο της CUDA. Η CUDA είναι μια παράλληλη υπολογιστική πλατφόρμα και ένα προγραμματιστικό μοντέλο που μας δίνει την δυνατότητα να υλοποιήσουμε παράλληλα προγράμματα πάνω σε κάρτες γραφικών NVIDIA. Παρουσιάζουμε τα βασικά στοιχεία από τα οποία αποτελείται η πλατφόρμα της CUDA και τέλος παραθέτουμε οι διεργασίες που επωφελούνται όταν εκτελούνται από την πλατφόρμα της CUDA.

Στο κεφάλαιο 3 παρουσιάζεται η αρχιτεκτονική μιας κάρτας γραφικών που είναι βασισμένη πάνω στο μοντέλο της CUDA. Οι σημαντικότερες μονάδες μέσα σε μια κάρτα γραφικών του μοντέλου της CUDA είναι οι επεξεργαστές συνεχούς ροής(Streaming Multiprocessor) και η μνήμη. Η κάρτα γραφικών

έχει διάφορους τύπους μνήμης, η καθεμία από τις οποίες χρησιμοποιείται για διαφορετικού σκοπούς. Τα είδη της μνήμης που έχει μια κάρτα γραφικών είναι: global memory, shared memory, registers, constant memory, texture memory και local memory. Υλικά οι καταχωρητές και η shared memory βρίσκονται πάνω στο SM, ενώ οι υπόλοιπες πάνω στην μνήμη της κάρτας. Από αυτές οι πιο σημαντικές είναι η global memory και η shared memory. Η global memory είναι η μεγαλύτερη σε μέγεθος μνήμη, αλλά και η πιο αργή μνήμη και χρησιμοποιείται για να αποθηκεύονται τα δεδομένα τα οποία μεταφέρονται από την CPU και θα χρησιμοποιηθούν για να πραγματοποιηθούν οι διάφορες διεργασίες. Η κοινόχρηστη μνήμη(shared memory) είναι μια από τις πιο γρήγορες μνήμης της GPU και χρησιμοποιείται για να επικοινωνούν τα νήματα ενός μπλοκ μεταξύ τους. Ο επεξεργαστείς συνεχούς ροής αποτελεί την σημαντικότερη μονάδα της κάρτας γραφικών του μοντέλου της CUDA, αφού σε αυτόν πραγματοποιούνται όλοι οι υπολογισμοί. Σε αυτό το κομμάτι παρουσιάζουμε τις διάφορες αρχιτεκτονικές του SM ανάλογα με την γενιά της κάρτας γραφικών. Αναλύουμε το κάθε τμήμα της κάθε αρχιτεκτονικές, τις καινοτομίες που εισήχθησαν σε κάθε γενιά, καθώς και το πως επηρεάζονται επιδόσεις.

Στο κεφάλαιο 4: Διατάξεις υλικού των συστημάτων CUDA, περιγράφουμε τις διατάξεις του υλικού σε κάθε παράλληλο σύστημα που βασίζεται πάνω στις κάρτες γραφικών CUDA. Μελετάμε την τοπολογία διαφόρων παράλληλων υπολογιστικών συστημάτων, από το επίπεδο του συστήματος μέχρι τις λειτουργικές μονάδες μέσα στην κάρτα γραφικών. Παρουσιάζεται η δομή του υπολογιστικού συστήματος από την κεντρική μονάδα μέχρι την κάρτα γραφικών και πως αυτά τα δύο μέρη επικοινωνούν και αλληλοεπιδρούν μεταξύ τους.

Στο κεφάλαιο 5: **Error! Reference source not found.** παρουσιάζουμε το προγραμματιστικό μοντέλο της CUDA. Αρχικά, αναλύεται η δομή εκτέλεσης ενός προγράμματος της CUDA. Το κάθε πρόγραμμα εκτελείται από ένα σύνολο νημάτων τα οποία αποτελούνε ένα πλέγμα και το κάθε πλέγμα αποτελείται από μπλοκ νημάτων. Εδώ περιγράφεται και η ιεραρχία της μνήμης σε αναλογία με την δομή των νημάτων, δηλαδή η πρόσβαση που έχουν τα νήματα στα διάφορα είδη της μνήμης σε σχέση με την δομή των νημάτων. Στην συνέχεια, παρατίθενται τα τμήματα ενός προγράμματος, αναλύοντας κυρίως τον προγραμματιστικό πυρήνα(kernel) που είναι το τμήμα του προγράμματος που θα εκτελεστεί από την κάρτα γραφικών. Τέλος, μελετάμε την αντιστοιχία μεταξύ της δομής του προγράμματος σε σχέση με την δομή της κάρτας γραφικών, δηλαδή το πως ομαδοποιούνται τα νήματα σε στημόνια και εκτελούνται από τις δομικές μονάδες της κάρτας γραφικών.

Στο κεφάλαιο 6: Ο αλγόριθμος Smith-Waterman, μελετάμε την λογική του αλγόριθμου Smith – Waterman. Ο αλγόριθμος Smith – Waterman είναι ένας αλγόριθμος που προέρχεται από τον τομέα της βιοπληροφορικής. Σκοπός αυτού του αλγόριθμου είναι η ευθυγράμμιση μοριακών ακολουθιών, δηλαδή η αναγνώριση παρόμοιων τμημάτων ανάμεσα σε ακολουθίες. Η λογική του βασίζεται πάνω στην λογική των δυναμικών αλγόριθμων, δηλαδή η λύση ενός μεγάλου προβλήματος πραγματοποιείται με την διάσπαση του σε ένα σύνολο μικρότερων προβλημάτων, που ο συνδυασμός τους μας δίνει την λύση του αρχικού προβλήματος. Εδώ λοιπόν περιγράφουμε την λειτουργία του αλγόριθμου και τα διάφορα μέρη από τα οποία αποτελείται.

Στο κεφάλαιο 7 περιγράφεται η εφαρμογή που πραγματοποιεί ανίχνευση ηχητικών εφέ μέσα σε αρχεία ταινιών, με την χρήση μίας κάρτας γραφικών CUDA και την χρήση του αλγόριθμου Smith – Waterman. Αρχικά, παρουσιάζεται δύο μέθοδοι με τους οποίους μπορεί να υλοποιηθεί ο αλγόριθμος Smith – Waterman. Η μία μέθοδος είναι η Inter-task παραλληλοποίηση και η άλλη είναι η Intra-task. Στην συγκεκριμένη εφαρμογή επιλέχθηκε η μέθοδος Inter-task λόγω των καλύτερων επιδόσεων που παρουσιάζει. Στη μέθοδο Inter-task, σε κάθε νήμα ανατίθεται μια ολοκληρωμένη εργασία, η οποία πραγματοποιείται πάνω σε συγκεκριμένα δεδομένα. Έτσι κάθε νήμα πραγματοποιεί την ίδια εργασία, αλλά πάνω σε διαφορετικά δεδομένα. Στη συνέχει παρουσιάζονται τα μέρη της εφαρμογής. Το μέρος της εφαρμογής που εκτελείται από την CPU αναλαμβάνει την φόρτωση των αρχείων και την μεταφορά των δεδομένων στην GPU. Τα αρχεία είναι μορφοποιημένα στην μορφή Audio Spectrum Envelop. Παρουσιάζουμε την πρώτη υλοποίηση με την οποία εκτελείται ο αλγόριθμος και πραγματοποιείται η ανίχνευση των ηχητικών εφέ. Στην συνέχεια, παρουσιάζουμε μια δεύτερη υλοποίηση η οποία αλλάζει την δομή της παραλληλοποίησης έτσι ώστε να κάνουμε χρήση περισσοτέρων νημάτων και παραθέτουμε τα αποτελέσματα αυτής της αλλαγής. Στην 3η Υλοποίηση της εφαρμογής πραγματοποιούμε κάποιες αλλαγές στην δέσμευση της μνήμης έτσι ώστε να έχουμε δυναμική δέσμευση. Αυτό, όμως έχει αρνητική επίπτωση στις επιδόσεις της εφαρμογής. Στην 4η Υλοποίηση περιορίζουμε την χρήση της μνήμης όσο το δυνατόν περισσότερο για να έχουμε βελτίωση των επιδόσεων. Στην 5η Υλοποίηση πραγματοποιήσαμε αλλαγές στην δομή του προγράμματος έτσι ώστε να κάνουμε χρήση της γρήγορης κοινόχρηστης μνήμης και παρουσιάζουμε τα αποτελέσματα.

2. Επεξεργαστές γραφικών και προγραμματιστικά μοντέλα

Τα τελευταία χρόνια η βιομηχανία υπολογιστών έστρεψε την παραγωγή της στις σε αρχιτεκτονικές που υποστηρίζουν παράλληλες διεργασίες. Όλοι οι σύγχρονοι προσωπικοί υπολογιστές χρησιμοποιούν πολυπύρηνους επεξεργαστές. Ηλεκτρονικές συσκευές όπως κινητά τηλέφωνα και tablet είναι εξοπλισμένα με πολυπύρηνους επεξεργαστές που μπορούν να υποστηρίξουν την εκτέλεση πολλών διεργασιών παράλληλα. Οι προγραμματιστές ακολουθώντας την εξέλιξη αυτή των επεξεργαστών είναι αναγκασμένοι να φτιάχνουν προγράμματα που εκτελούν πολλές διεργασίες παράλληλα και όχι σειριακά όπως συνέβαινε μέχρι σήμερα.

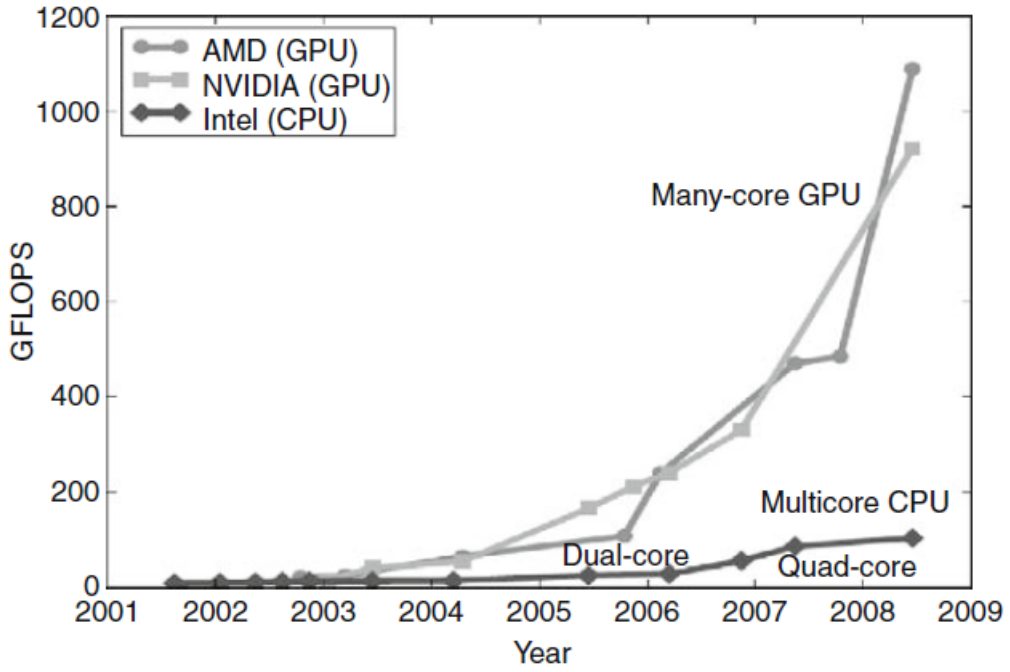
Οι μικροεπεξεργαστές που είναι βασισμένοι πάνω στην αρχιτεκτονική της κεντρικής μονάδας επεξεργασίας(CPU), οδήγησαν στην αύξηση των επιδόσεων και την μείωση του κόστους των εφαρμογών για πάνω από δυο δεκαετίες. Οι πρώτοι προσωπικοί υπολογιστές της δεκαετίας του 1980 είχαν ταχύτητες με μέγεθος 1MHz. Τριάντα χρόνια μετά οι επεξεργαστές αυτοί είχαν ταχύτητες μεταξύ 1GHz και 4GHz, περίπου 1000 φορές πιο γρήγοροι από τους πρώτους προγόνους τους. Οι επιδόσεις αυτών των επεξεργαστών έφτασε στο επίπεδο των GFLOPS στους προσωπικούς υπολογιστές(PC) και στα εκατοντάδες GFLOPS στα cluster. Η βελτίωση των επεξεργαστών αυτής της αρχιτεκτονικής βασιζότανε στην αύξηση της συχνότητας του ρολογιού του επεξεργαστή. Η ανάπτυξη αυτών των επεξεργαστών έδωσε την δυνατότητα στις εφαρμογές να έχουν περισσότερες δυνατότητες, ελκυστικό γραφικό περιβάλλον και καλύτερα αποτελέσματα. Οι χρήστες από την πλευρά τους είχαν την απαίτηση για ακόμα μεγαλύτερες επιδόσεις. Οι προγραμματιστές επίσης στήριξαν την λειτουργία των εφαρμογών τους πάνω στην ανάπτυξη των επεξεργαστών. Έτσι, οι εφαρμογές λειτουργούσαν όλο και καλύτερα με τις προηγμένες γενιές επεξεργαστών. Αυτή η ανάπτυξη όμως άρχισε να επιβραδύνεται με την αύξηση των απαιτήσεων σε κατανάλωση ενέργειας και την αύξηση την θερμοκρασίας, που εμφάνιζαν αυτοί οι επεξεργαστές, με την αύξηση την συχνότητας λειτουργίας. Αυτό είχε σαν αποτέλεσμα όλοι οι κατασκευαστές επεξεργαστών να στραφούν στην δημιουργίας πολυπύρηνων επεξεργαστών για να αυξήσουν τις επιδόσεις των συστημάτων τους. Αυτή η αλλαγή στην αρχιτεκτονική των επεξεργαστών είχε μεγάλες επιπτώσεις στο χώρο των προγραμματιστών.

Τα παραδοσιακά προγράμματα γράφονται σε σειριακή μορφή. Έτσι, οι χρήστες με την εξέλιξη των επεξεργαστών θεωρούσαν ότι τα προγράμματα θα επιταχυνθούν. Αυτό όμως δεν είναι η πραγματικότητα στους σύγχρονους πολυπύρηνους επεξεργαστές. Το σειριακό πρόγραμμα θα εκτελεστεί σε έναν πυρήνα του επεξεργαστή με αποτέλεσμα να μην παρουσιάζει καμία επιτάχυνση αν η συχνότητα του δεν είναι μεγαλύτερη. Τα προγράμματα τα οποία επωφελούνται από την εξέλιξη των επεξεργαστών είναι αυτά που έχουν σχεδιαστεί με παράλληλο τρόπο, δηλαδή εκμεταλλεύονται πολλά νήματα τα οποία δουλεύουν μαζί για την ολοκλήρωση μιας διεργασίας. Η ανάπτυξη, όμως παράλληλων προγραμμάτων δεν είναι κάτι καινούργιο. Η κοινότητα προγραμματισμού σε συστήματα μεγάλων επιδόσεων χρησιμοποιεί παράλληλες μεθόδους εδώ και δεκαετίες. Βέβαια ο προγραμματισμός σε αυτά τα πολύ μεγάλα συστήματα περιοριζόταν σε ένα μικρό αριθμό προγραμματιστών. Σήμερα όμως, που όλοι οι καινούργιοι επεξεργαστές είναι παράλληλοι υπολογιστές ο αριθμός των εφαρμογών που δημιουργούνται για παράλληλα συστήματα έχει αυξηθεί σημαντικά.

2.1 Η Ανάπτυξη της κάρτας γραφικών

Στην δεκαετία του 2000 η βιομηχανία παραγωγής μικροεπεξεργαστών είχε πια καθιερώσει την παραγωγή δύο ειδών επεξεργαστών. Το ένα είδος είναι οι επεξεργαστές με μικρό αριθμό πυρήνων (πχ. 2-8) που εξοπλίζουν τους προσωπικούς υπολογιστές και έχουν στόχο να εξυπηρετήσουν τις ανάγκες των σειριακών εφαρμογών και το άλλο είναι οι πολυπύρηνοι επεξεργαστές που προορίζονται για τις κάρτες γραφικών. Οι επεξεργαστές αυτοί έχουν ένα μεγάλο αριθμό πυρήνων οι οποίοι διπλασιάζονται με το πέρασμα από γενιά σε γενιά. Ένα παράδειγμα ενός πολυπύρηνου επεξεργαστή είναι ο NVIDIA GeForce GTX 280 ο οποίος περιέχει 240 πυρήνες, που ο κάθε ένας μπορεί να τρέξει ένα μεγάλο αριθμό νημάτων. Οι επεξεργαστές των GPU προηγούνται τα τελευταία χρόνια σε επιδόσεις σε σχέση με τους επεξεργαστές των CPU, στους οποίους τα τελευταία χρόνια παρουσιάζουν μια στασιμότητα στις επιδόσεις τους όπως

φαίνεται και στο παρακάτω διάγραμμα(Εικόνα 2-1: Ανάπτυξη των GPU σε σχέση με τις CPU).



Εικόνα 2-1: Ανάπτυξη των GPU σε σχέση με τις CPU

Η μεγάλη διαφορά στις επιδόσεις των επεξεργαστών με τους πολλούς πυρήνες που χρησιμοποιούνται στις κάρτες γραφικών σε σχέση με τους επεξεργαστές που χρησιμοποιούνται στις CPU είναι στην διαφορετική αρχιτεκτονική φιλοσοφία. Ο σχεδιασμός των CPU πραγματοποιείται με γνώμονα την βελτιστοποίηση των σειριακών προγραμμάτων. Ο σχεδιασμός χρησιμοποιεί μια λογική ελέγχου για να επιτρέπει εντολές από ένα νήμα να εκτελούνται παράλληλα ή ακόμα και μη σειριακά σε σχέση με το πρόγραμμα, αλλά πάντα με σκοπό η εκτέλεση το προγράμματος να φαίνεται σειριακή. Επίσης, χρησιμοποιούνται μεγάλες μνήμες cache για να μειωθούν οι χρόνοι προσπέλασης την μνήμης. Αυτές οι δύο παράμετροι, η λογική ελέγχου και η μνήμη cache, δεν συνεισφέρουν στη ταχύτητα των υπολογισμών.

Σε αντίθεση η σχεδιαστική φιλοσοφία των καρτών γραφικών βασίστηκε στις απαιτήσεις της βιομηχανίας παιχνιδιών, η οποία απαιτεί την εκτέλεση μεγάλο αριθμό πράξεων κινητής υποδιαστολής πάνω σε κάθε frame. Αυτή η απαίτηση οδήγησε στον σχεδιασμό των καρτών γραφικών έτσι ώστε να μεγιστοποιηθεί ο αριθμός των νημάτων που τρέχουν παράλληλα. Πολλά νήματα τρέχουν παράλληλα όταν ένα μέρος των νημάτων είναι σε αναμονή για να προσπελάσουν την μνήμη. Χρησιμοποιούνται πολλές μνήμες cache έτσι ώστε να μην υπάρχει η ανάγκη από όλα τα νήματα να προσπελάσουν την μνήμη επαναλαμβανόμενα. Έτσι το μεγαλύτερο μέρος του επεξεργαστή της κάρτας γραφικών αφιερώνεται στην εκτέλεση πράξεων.

Οι κάρτες γραφικών είναι κατασκευασμένες σαν μηχανές που πραγματοποιούν παράλληλες υπολογιστικές πράξεις, επομένως η απόδοσή τους είναι μειωμένη όταν εκτελούν διεργασίες οι οποίες είναι σχεδιασμένες για την CPU, δηλαδή ακολουθούν μια σειριακή φιλοσοφία. Η λειτουργία μιας εφαρμογής, η οποία θα εκτελεστεί από ένα ετερογενές σύστημα, πρέπει να είναι έτσι σχεδιασμένη έτσι ώστε το σειριακό κομμάτι να πραγματοποιείται στην CPU και το παράλληλο κομμάτι που αποτελείται από σύνολο μεγάλων υπολογισμών να πραγματοποιείται από την κάρτα γραφικών. Για αυτό το λόγο το μοντέλο προγραμματισμού CUDA(Compute Unified Device Architecture) είναι σχεδιασμένο για να υποστηρίζει την παράλληλη εκτέλεση σε CPU/GPU.

2.2 Παράλληλα υπολογιστικά συστήματα

Τα παράλληλα υπολογιστικά συστήματα δημιουργήθηκαν για την βελτίωση της ταχύτητας των υπολογισμών. Από την άποψη των υπολογισμών τέτοια συστήματα μπορεί να θεωρηθούν αυτά τα οποία

μπορούν να εκτελούν πολλούς υπολογισμούς ταυτόχρονα. Έτσι, μεγάλα προβλήματα μπορούν να διαιρεθούν σε πολλά μικρότερα, τα οποία μπορούν να επιλυθούν παράλληλα. Από την πλευρά του προγραμματιστή, η δυσκολία στους ταυτόχρονους υπολογισμούς έγκειται στο πως θα αποδοθούν αυτοί οι υπολογισμοί στους πόρους του συστήματος. Έτσι, η δουλειά του προγραμματιστή είναι να διαιρέσει ένα μεγάλο πρόβλημα σε πολλά μικρότερα και να αποδώσει την επίλυση των μικρότερων προβλημάτων στους αντίστοιχους πόρους του συστήματος.

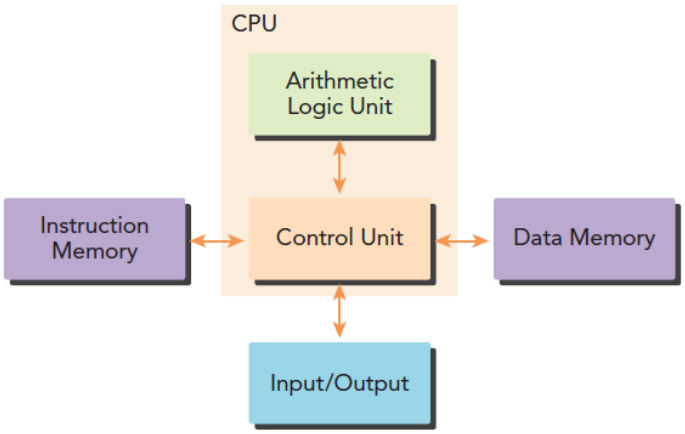
Τα μέρη των παράλληλων υπολογιστικών συστημάτων από την πλευρά του υλικού και του λογισμικού είναι στενά συνδεδεμένες μεταξύ τους. Έτσι, όταν μιλάμε για παράλληλα υπολογιστικά συστήματα έχουμε να κάνουμε με τις δύο παρακάτω μέρη:

- Αρχιτεκτονική υπολογιστών.
- Παράλληλος προγραμματισμός.

Η αρχιτεκτονική υπολογιστών εστιάζει στο να παρέχει τον παραλληλισμό στο επίπεδο του υλικού, ενώ ο παράλληλος προγραμματισμός βασίζεται στο να λύσει ένα πρόβλημα με παραλληλισμό, εκμεταλλευόμενος την παράλληλη αρχιτεκτονική του υλικού. Για να μπορέσει το λογισμικό να χρησιμοποιήσει παράλληλες μεθόδους θα πρέπει το υλικό να παρέχει στον προγραμματιστή μια πλατφόρμα που θα υποστηρίζει την παράλληλη εκτέλεση πολλών διεργασιών.

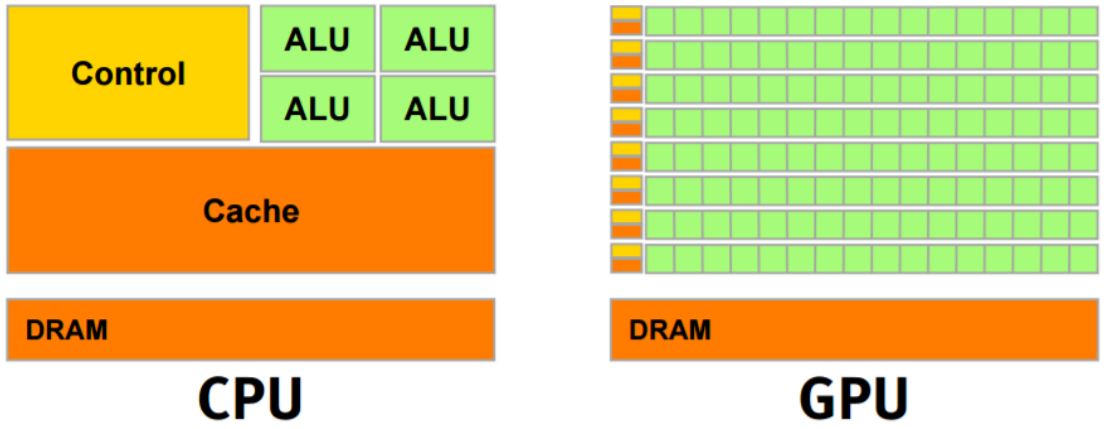
Οι περισσότεροι σύγχρονοι υπολογιστές υποστηρίζουν την αρχιτεκτονική που παρουσιάζεται στην Εικόνα 2-2: Αρχιτεκτονική Υπολογιστών. Αυτή η αρχιτεκτονική έχει τρία βασικά μέρη:

- Μνήμη(Instruction Memory – Data Memory)
- Κεντρική Μονάδα Επεξεργασίας(Control Unit – Logic Unit)
- Διεπαφές Εισόδου/Εξόσου



Εικόνα 2-2: Αρχιτεκτονική Υπολογιστών

Η σημαντικότερη μονάδα σε αυτή την αρχιτεκτονική είναι η κεντρική μονάδα επεξεργασίας(CPU), η οποία καλείται συνήθως πυρήνας. Οι παραδοσιακοί επεξεργαστές αποτελούνται από ένας έως οκτώ πυρήνες και ο προγραμματιστής όταν αναπτύσσει ένα σειριακό πρόγραμμα δεν χρειάζεται να γνωρίζει λεπτομέρειες της αρχιτεκτονικής για την δημιουργία του προγράμματος. Αντίθετα, στην περίπτωσης της κάρτας γραφικών που έχουμε πολλαπλούς τέτοιες πυρήνες, ο προγραμματιστής οφείλει να γνωρίζει βασικές λειτουργίες της αρχιτεκτονικής του υλικού για να παράγει αποδοτικά προγράμματα.



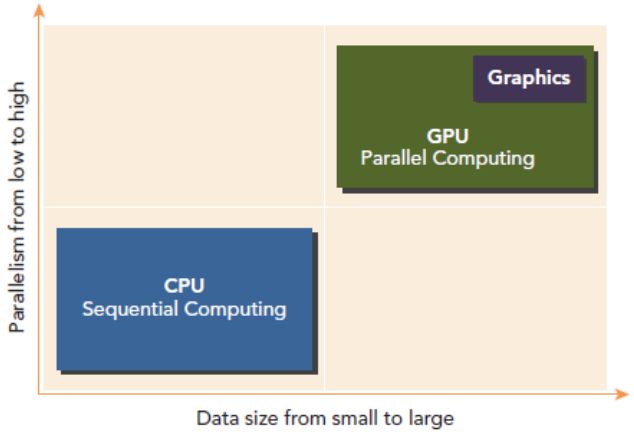
Εικόνα 2-3: Σχεδιαστικές διαφορές μεταξύ CPU και GPU

2.3 Διαφορές μεταξύ της CPU και την GPU

Ο σκοπός της χρήσεως των καρτών γραφικών δεν είναι η αντικατάσταση της CPU. Υπάρχουν διεργασίες που είναι προτιμότερο να εκτελούνται σε GPU και διεργασίες που είναι προτιμότερο να εκτελούνται σε CPU. Οι CPU είναι καλύτερες στην εκτέλεση προγραμμάτων που έχουνε έντονες λειτουργίες ελέγχου και οι κάρτες είναι καλύτερες στην εκτέλεση προγραμμάτων με διεργασίες που μπορούν να παραλληλοποιηθούν πάνω σε πολλά δεδομένα. Πολλές φορές ο συνδυασμός της CPU με την GPU μας δίνει τα καλύτερα δυνατά αποτελέσματα. Όπως φαίνεται στην Εικόνα 2-4: Χρήση CPU - GPU, έχουμε δύο μεταβλητές οι οποίες καθορίζουν την χρήση της CPU ή της GPU:

- Επίπεδο παραλληλισμού
- Όγκος δεδομένων

Αν ένα πρόβλημα έχει λίγα δεδομένα, μια εξεζητημένη ροή ελέγχου και μικρό επίπεδο παραλληλισμού, τότε η CPU είναι η καλύτερη επιλογή λόγω της δυνατότητας της να διαχειριστεί πολύπλοκες μορφές ελέγχου. Αν ένα πρόβλημα έχει μεγάλο όγκο δεδομένων και μπορεί να παραλληλοποιηθεί σε μεγάλο βαθμό τότε η GPU είναι η καλύτερη επιλογή, λόγω του ότι έχει ένα μεγάλο αριθμό πυρήνων και μπορεί να υποστηρίξει την παράλληλη εκτέλεση χιλιάδων νημάτων.



Εικόνα 2-4: Χρήση CPU - GPU

Οι βασικές διαφορές βρίσκονται κυρίως στον μοντέλο των νημάτων και στην διαφορετικές μνήμες.

- **Πόροι των νημάτων**: Η εκτέλεση των νημάτων στην CPU μπορεί να υποστηρίξει την παράλληλη εκτέλεση ενός συγκεκριμένου αριθμού νημάτων. Servers οι οποίοι έχουν τέσσερις hex-core επεξεργαστές μπορούν να εκτελέσουν το πολύ 24 νήματα παράλληλα (ή 48 αν ο επεξεργαστής υποστηρίζει Hyper-Threading). Η μικρότερη μονάδα εκτέλεσης σε μια συσκευή CUDA μπορεί να υποστηρίξει 32 παράλληλα νήματα(warp of threads). Οι σύγχρονες κάρτες

γραφικών μπορούν να υποστηρίξουν μέχρι και 1536 ενεργά παράλληλα νήματα ανά πολυεπεξεργαστή. Επομένως, μια κάρτα γραφικών με 16 πολυεπεξεργαστές μπορεί να υποστηρίξει πάνω από 24000 παράλληλα νήματα.

- **Νήματα(Threads)**: Τα νήματα της cpu είναι βαριές οντότητες που η CPU εκτελεί εναλλάξ για να υποστηρίξει την παράλληλη εκτέλεση τους. Η εναλλαγή τον νημάτων είναι μια αργή και βαριά διαδικασία. Αντίθετα τα νήματα της GPU είναι πολύ ελαφριές διεργασίες. Σε ένα απλό σύστημα, χιλιάδες threads μπαίνουν σε αναμονή για την εκτέλεση τους, τα οποία είναι ομαδοποιημένα σε στημόνια(warps) των 32 νημάτων. Αν η GPU μπει σε αναμονή κατά την εκτέλεση ενός στημονιού, τότε ξεκινάει την εκτέλεση ενός άλλου στημονιού. Επειδή, διαφορετικοί καταχωρητές κατανέμονται στα διάφορα νήματα, δεν είναι αναγκαία η αλλαγή καταχωρητών ή οποιαδήποτε αλλαγή, κατά την εναλλαγή εκτέλεσης των νημάτων στην GPU. Έτσι η εναλλαγή των νημάτων πάνω στο πολυεπεξεργαστή μπορεί να θεωρηθεί ότι γίνεται δωρεάν. Οι πόροι είναι κατανεμημένοι στο κάθε νήμα μέχρι να τελειώσει η εκτέλεση του.
- **Μνήμη(RAM)**: Το σύστημα που φιλοξενεί την κάρτα γραφικών και η κάρτα γραφικών έχουν το καθένα την δικιά του ξεχωριστή μνήμη. Από την στιγμή το σύστημα και η κάρτα γραφικών επικοινωνούν μεταξύ τους με το δίαυλο PCI Express, τα δεδομένα μεταφέρονται μεταξύ τους μέσο αυτού του διαύλου.

2.4 Γλώσσες παράλληλου προγραμματισμού και μοντέλα

Διάφορες γλώσσες παράλληλου προγραμματισμού και διάφορα μοντέλα έχουν προταθεί τις τελευταίες δεκαετίες. Οι πιο ευρέως διαδομένες είναι η MPI(Message Passing Interface) για σύμπλεγμα υπολογιστών και η OpenMP για συστήματα πολυεπεξεργαστών με διαμοιραζόμενη μνήμη. Το κάθε μοντέλο έχει τα πλεονεκτήματα και τα μειονεκτήματα του. Ανάλογα με το υπολογιστικό σύστημα που έχουμε μπορούμε να επιλέξουμε το μοντέλο που είναι πιο αποδοτικό για το συγκεκριμένο σύστημα. Η CUDA σαν απόγονος αυτών των μοντέλων προσπαθεί να εκμεταλλευτεί έχει δημιουργηθεί με τέτοιο τρόπο ώστε να εκμεταλλευτεί τα πλεονεκτήματα των προκατόχων της, αλλά και να περιορίσει όσο το δυνατό τα μειονεκτήματα τους.

Το μοντέλο MPI είναι ένα τυποποιημένο πρότυπο μετάδοσης μηνυμάτων σε παράλληλα υπολογιστικά συστήματα. Προορίζεται για σύμπλεγμα υπολογιστών τα οποία δεν διαμοιράζονται την μνήμη. Ο διαμοιρασμός των δεδομένων και η αλληλεπίδραση μεταξύ των επεξεργαστών γίνεται με την μετάδοση μηνυμάτων μεταξύ τους. Η γλώσσα MPI βρίσκει μεγάλη εφαρμογή σε υπολογιστικά συστήματα υψηλών επιδόσεων που προορίζονται για απαιτητικούς επιστημονικούς υπολογισμούς. Εφαρμογές που είναι γραμμένες στην γλώσσα MPI τρέχουν πάνω σε σύμπλεγμα υπολογιστών με πάνω από 100000 κόμβους. Η προσπάθεια όμως που χρειάζεται για την δημιουργία των εφαρμογών αυτών είναι αρκετά μεγάλη, λόγω του ότι τα συστήματα δεν διαμοιράζονται μνήμη και έτσι είναι αναγκαία πολλά μηνύματα για να επικοινωνήσουν τα διάφορα συστήματα μεταξύ τους. Η CUDA από την άλλη, μας δίνει την δυνατότητα διαμοιραζόμενης μνήμης για να παρακάμψουμε αυτές τις δυσκολίες. Όσων αφορά την επικοινωνία μεταξύ της CPU και της GPU η CUDA παρέχει περιορισμένες δυνατότητες. Η μεταφορά δεδομένων μεταξύ CPU και GPU γίνεται με μονόπλευρη μεταφορά μηνυμάτων, μια δυνατότητα που η απουσία στο μοντέλο MPI θεωρείται μια σημαντική αδυναμία.

Μερικά από τα πλεονεκτήματα του μοντέλου MPI αναφέρονται παρακάτω:

- Εφαρμόζεται σε κατανεμημένα συστήματα μνήμης.
- Μπορεί να χρησιμοποιηθεί σε μεγαλύτερο πλήθος εφαρμογών από ότι η OpenMP.
- Κάθε διεργασία έχει τις δικές της τοπικές μεταβλητές.
- Τα κατανεμημένα συστήματα έχουν μικρότερο κόστος από τα μεγάλα υπολογιστική συστήματα με διαμοιραζόμενη μνήμη.

Μερικά από τα μειονεκτήματα του μοντέλου MPI αναφέρονται παρακάτω:

- Χρειάζονται περισσότερες αλλαγές για να μεταβούμε από ένα σειριακό πρόγραμμα σε ένα παράλληλο πρόγραμμα.
- Είναι δύσκολο στην επίλυση λαθών στον κώδικα.
- Η επίδοση περιορίζεται από την επικοινωνία μεταξύ των κόμβων του συστήματος.

Το μοντέλο OpenMP(Open Multi-Processing) είναι μια γλώσσα προγραμματισμού εφαρμογών που υποστηρίζει συστήματα πολυεπεξεργαστών με διαμοιραζόμενη μνήμη. Επειδή υποστηρίζει διαμοιραζόμενη μνήμη, παρουσιάζει τα ίδια πλεονεκτήματα που έχει η CUDA, σε ότι έχει να κάνει με την ευκολία από προγραμματιστική πλευρά. Το μοντέλο αυτό είναι μια εφαρμογή του multithreading,

όπου ένα κύριο νήμα δημιουργεί έναν αριθμό δευτερευόντων νημάτων και το σύστημα μοιράζει τις διάφορες διεργασίες ανάμεσα σε αυτά τα νήματα Τα νήματα αυτά τρέχουν παράλληλα, με το σύστημα να διανέμει τα νήματα στους διαφορετικούς επεξεργαστές. Παρόλα αυτά, αυτό το μοντέλο δεν μπορεί να εφαρμοστεί σε συστήματα με κόμβους πάνω από μερικές εκατοντάδες εξαιτίας της διαχείρισης των νημάτων και προβλημάτων διαχείρισης της μνήμης cache. Η CUDA μπορεί να υποστηρίξει μεγαλύτερη επεκτασιμότητα απλή διαχείριση νημάτων μικρού κόστους και χωρίς απαίτηση για ομοιομορφία της cache. Παρόλα η CUDA δεν μπορεί να υποστηρίξει τόσες εφαρμογές όσες η OpenMP λόγω αυτών των παραχωρήσεων για να υποστηρίξει την επεκτασιμότητα.

Μερικά από τα πλεονεκτήματα της γλώσσας OpenMP είναι τα παρακάτω:

- Υποστηρίξει συστημάτων με διαμοιραζόμενη μνήμη για κοινή χρήση των δεδομένων σε διεργασίες
- Απλή χρήση μηνυμάτων για την επικοινωνία των νημάτων.
- Ενοποιημένος κώδικας για σειριακές και παράλληλες εφαρμογές.
- Φορητός πολυνηματικός κώδικας

Μερικά από τα μειονεκτήματα της OpenMP είναι τα παρακάτω:

- Υποστηρίζει μόνο συστήματα διαμοιραζόμενης μνήμης.
- Περιορισμοί στην επεκτασιμότητας λόγω της αρχιτεκτονικής της μνήμης
- Δυσκολία στην επίλυση λαθών συγχρονισμού και ανταγωνισμού των νημάτων.

Η CUDA έχει πολλές ομοιότητες με τις δυο αυτές γλώσσες, αφού και στα τρία μοντέλα ο προγραμματιστής καθορίζει την δομή της παραλληλίας, παρόλο που ο compiler της OpenMP αυτοματοποιεί περισσότερες διαδικασίες από τα άλλα δυο μοντέλα. Σήμερα, στην CUDA γίνεται προσπάθεια να αυτοματοποιηθούν η διαχείριση της παραλληλία και η βελτιστοποίηση της απόδοσης, για την διευκόλυνση του έργου των προγραμματιστών. Οι προγραμματιστές που έχουν κάποια γνώση με τις γλώσσες MPI και OpenMP, μπορούν εύκολα να εξοικειωθούν με την CUDA, αφού πολλές από τις τεχνικές βελτιστοποίησης σε παράλληλα συστήματα είναι κοινές και στα τρία μοντέλα.

Τα τελευταία χρόνια γίνεται προσπάθεια από μεγάλες κατασκευαστικές εταιρίες, όπως η Apple, Intel, AMD/ATI και NVIDIA, να συνεργαστούν ώστε να αναπτύξουν ένα κοινό μοντέλο παράλληλου προγραμματισμού, το OpenCL. Όπως και η CUDA, περιέχει κάποιες επεκτάσεις για γλώσσες προγραμματισμού και κάποιες βιβλιοθήκες που επιτρέπουν στους προγραμματιστές να διαχειριστούν την δομή της παραλληλία και την διανομή των δεδομένων στους διάφορους παράλληλους επεξεργαστές. Ουσιαστικά, είναι ένα τυποποιημένο μοντέλο προγραμματισμού στο οποίο, οι εφαρμογές που αναπτύσσονται με βάση την γλώσσα OpenCL, μπορούν να τρέξουν σε οποιοδήποτε σύστημα που υποστηρίζει το μοντέλο OpenCL. Το μειονέκτημα της γλώσσας αυτής, είναι ότι βρίσκεται ακόμα σε εμβρυικό στάδιο και ο προγραμματισμός σε αυτό γίνεται σε πολύ χαμηλό επίπεδο.

2.5 CUDA

Η πρώτη κάρτα γραφικών που κατασκευάστηκε από την NVIDIA με τη χρήση της αρχιτεκτονικής CUDA ήταν ή GeForce 8800 GTX. Αυτή η αρχιτεκτονική συμπεριλάμβανε πολλά νέα χαρακτηριστική σχεδιασμένα αποκλειστικά για κάρτες γραφικών, με σκοπό να ξεπεραστούν πολλοί περιορισμοί που υπήρχαν σε προηγούμενες κάρτες γραφικών και δεν μπορούσαν να χρησιμοποιηθούν για γενικού σκοπού υπολογισμούς. Πριν την χρήση την αρχιτεκτονικής της CUDA, οι κάρτες γραφικών χρησιμοποιούσαν διαφορετικούς πόρους για τον χρωματισμό των pixel(shader) και την δημιουργία σχημάτων(vertex). Η αρχιτεκτονική CUDA επέτρεψε την χρήση μιας αριθμητικής λογικής μονάδας για τον υπολογισμό γενικών πράξεων. Επιπλέον, η μονάδες εκτέλεσης την κάρτας γραφικών μπορούν να προσπελάσουν την μνήμη, όπως επίσης και να κάνουν χρήση μέσω του λογισμικού ενός ειδικού τύπου μνήμης cache(Shared Memory). Αυτές οι επιπλέον λειτουργίες της αρχιτεκτονικής CUDA προστέθηκαν στην κάρτα γραφικών έτσι ώστε να πραγματοποιεί, εκτός από διεργασίες γραφικών, και γενικούς υπολογισμούς. Για την χρήση αυτής της αρχιτεκτονικής και για να διευρυνθεί ο αριθμός των προγραμματιστών η NVIDIA χρησιμοποίησε την γλώσσα C με την προσθήκη ορισμένων λέξεων-κλειδιά που ενεργοποιούν τις ειδικές λειτουργίες την κάρτας γραφικών.

2.5.1 Βασικά στοιχεία της CUDA

Σε αυτή την ενότητα θα περιγράψουμε κάποια βασικά στοιχεία της CUDA.

- Η κάρτες γραφικών CUDA είναι συνήθως ξεχωριστές συσκευές οι οποίες εγκαθίστανται πάνω σε ένα κεντρικό υπολογιστικό σύστημα που τις φιλοξενεί. Συνήθως, η επικοινωνία με τον κεντρικό σύστημα και την CPU γίνεται μέσο ενός διαύλου PCIe(Peripheral Component Interconnect Express). Στα κοινά υπολογιστική συστήματα μπορούν να φιλοξενηθούν από μια μέχρι 4 κάρτες γραφικών. Το πόσες μπορούν να φιλοξενηθούν εξαρτάται από το πόσες υποδοχές PCIe υπάρχουν στο σύστημα, από την ψύξη και από την ισχύ που είναι διαθέσιμη. Η κάθε συσκευή λειτουργεί ασύγχρονα με την CPU, δηλαδή μπορούν να εκτελούνται διεργασίες παράλληλα στην CPU και στην GPU. Ο δίαυλος PCI χρησιμοποιείται τόσο για την μεταφορά δεδομένων όσο και για την μεταφορά εντολών μεταξύ των συσκευών.
- Η GPU τρέχει στην δικιά της ξεχωριστή μνήμη ανεξάρτητα από την CPU. Εκτός από κάποιες ενσωματωμένες κάρτες γραφικών, όλες οι κάρτες γραφικών έχουν την δικιά τους μνήμη, οι οποίες είναι σχεδιασμένες, ώστε να μπορούν να εξυπηρετήσουν μεγαλύτερο εύρος ζώνης σε σχέση με τις παραδοσιακές μνήμες. Πολλές σύγχρονες κάρτες γραφικών μπορούν να εξυπηρετήσουν μέχρι και 160 – 200 GB/s. Μια παραδοσιακή μνήμη μπορεί να εξυπηρετήσει μέχρι και 8-20 GB/s.
- Τα προγράμματα τις CUDA χρησιμοποιούν κάποιες υπορουτίνες, που ονομάζονται kernel και καλούνται από την κεντρική μονάδα, αλλά εκτελούνται στην κάρτα γραφικών. Να αναφέρουμε εδώ ότι οι kernels δεν είναι οι κλασσικές συναρτήσεις της C, γιατί δεν μπορούν να επιστρέψουν κάποια τιμή. Τα περισσότερα προγράμματα δαπανούν τον περισσότερο χρόνο τους σε μερικές υπολογιστικές διεργασίες. Για να επιταχύνουμε αυτά τα προγράμματα τις περισσότερες φορές ο σκοπός μας είναι να μετατρέψουμε τις διεργασίες αυτές σε CUDA kernels, για να μπορέσουμε να εκμεταλλευτούμε την υπολογιστική ισχύ της κάρτας γραφικών. Βέβαια, αυτές οι διεργασίες θα πρέπει να υποστηρίζουν την παραλληλοποίηση των υπολογισμών τους, για να μπορέσουμε να έχουμε επιτάχυνση από την κάρτα γραφικών.
- Η εκτέλεση των kernel είναι ασύγχρονη, δηλαδή η κεντρική μονάδα που φιλοξενεί την κάρτα γραφικών συνεχίζει την λειτουργίας χωρίς να περιμένει την ολοκλήρωση του kernel. Λόγω αυτής της ασύγχρονης λειτουργίας της κάρτας γραφικών δεν μπορεί ο kernel να επιστέψει κάποια τιμή στην κεντρική μονάδα. Για να μπορέσουμε να έχουμε σε λειτουργία την κάρτα γραφικών όσο πιο αποτελεσματικά έχει δημιουργία μια τεχνολογία ουράς των kernel που καλούνται από την κεντρική μονάδα. Έτσι, απαιτείται από την κεντρική μονάδα κάποια λειτουργία συγχρονισμού για να μπορεί να έχει γνώση πότε τελείωσε ένας kernel ή μια ροή από kernels. Δυο μηχανισμοί που χρησιμοποιούνται συγχρονισμού είναι οι ακόλουθοι:
    1. Η κλήση της εντολής cudaThreadSynchronize(), η οποία λειτουργεί σαν φράγμα και αναγκάζει τις λειτουργίες της CPU να σταματήσουν και να περιμένουν την ολοκλήρωση των kernel που έχουν ήδη καλεστεί.
    2. Η πραγματοποίηση μιας μεταφοράς με την εντολή cudaMemcpy(), έχει σαν αποτέλεσμα τον συγχρονισμό της CPU με την GPU, αφού στο εσωτερικό της εντολής αυτής εκτελείται και η εντολή cudaThreadSynchronize().
- Ένας βασικός μηχανισμός της κάρτας γραφικών είναι το νήμα(thread). Το κάθε νήμα έχει την λειτουργεί ανεξάρτητα από κάθε άλλο νήμα. Λειτουργεί σαν να έχει τον δικό του επεξεργαστή με τους δικούς του καταχωρητές και την δικιά του ταυτότητα μέσα σε ένα περιβάλλον κοινόχρηστης μνήμης. Το υλικό καθορίζει τον αριθμό των νημάτων που μπορούν να τρέξουν παράλληλα. Ο χρονοπρογραμματιστής των νημάτων της κάρτας γραφικών αποφασίζει ποια ομάδα νημάτων θα τρέξει και έχει την δυνατότητα να εναλλάσσει της ομάδες των νημάτων. Αυτή η εναλλαγή από πλευράς χρονικού κόστους μπορεί να θεωρηθεί δωρεάν.

2.5.2 Τι τρέχει πάνω σε μια συσκευή CUDA

Οι παρακάτω παράμετροι θα πρέπει να λαμβάνονται υπόψιν όταν διερευνάται τα μέρη της εφαρμογής που θα τρέξουν πάνω στην κάρτα γραφικών:

- Η συσκευή είναι ιδανική για υπολογισμούς που μπορούν να πραγματοποιηθούν σε πολλά δεδομένα ταυτόχρονα. Μια τέτοια εργασία πρέπει να περιλαμβάνει υπολογισμούς πάνω σε μεγάλο όγκο δεδομένων, όπου η ίδια εργασία πρέπει να εκτελεστεί πάνω σε χιλιάδες, αν όχι εκατομμύρια, στοιχεία ταυτόχρονα. Αυτό είναι ένα χαρακτηριστικό για να έχουμε καλή επίδοση

σε CUDA, γιατί το λογισμικό πρέπει να χρησιμοποιήσει ένα μεγάλο αριθμό ταυτόχρονων νημάτων, έτσι ώστε να εκμεταλλευτεί τις δυνατότητες παραλληλίας της κάρτας γραφικών.

- Για καλύτερες επιδόσεις, θα πρέπει να υπάρχει μια αλληλουχία στις προσβάσεις στην μνήμη από γειτονικά νήματα. Συγκεκριμένα πρότυπα πρόσβασης της μνήμης δίνουν την δυνατότητα στην κάρτα γραφικών να πραγματοποιεί προσβάσεις ομαδοποιημένων δεδομένων στην μνήμη με μία μόνο ενέργεια. Δεδομένα που δεν ευνοούν αυτή την ομαδοποιημένη χρήση δεδομένων της μνήμης έχουνε χαμηλότερη επιτάχυνση.
- Για να χρησιμοποιήσουμε την CUDA, τα δεδομένα πρέπει να μεταφέρονται από την κεντρική μονάδα στην κάρτα μέσο του διαύλου PCI Express. Αυτές οι μεταφορές είναι αρκετά χρονοβόρες και θα πρέπει να περιορίζονται το δυνατόν περισσότερο έτσι ώστε να μην επηρεάζουν την επίδοση του προγράμματος. Έτσι, για να γίνει η μεταφορά των δεδομένων στην κάρτα γραφικών θα πρέπει να δικαιολογείτε το φόρτο εργασίας που πρέπει να πραγματοποιηθεί πάνω στα δεδομένα. Επίσης τα δεδομένα θα πρέπει να αποθηκεύονται στην μνήμα της κάρτας όσο το δυνατόν περισσότερο. Έτσι πολλαπλοί kernels που τρέχουν πάνω στα ίδια δεδομένα πρέπει να ευνοούν την παραμονή αυτών των δεδομένων στην μνήμη της κάρτας, από το να γίνονται πολλές μεταφορές ενδιάμεσων αποτελεσμάτων.

3. GPU Αρχιτεκτονική

Υπάρχουν τρεις διαφορετικές GPU αρχιτεκτονικές που μπορούν να τρέξουν την CUDA.

1. Η αρχιτεκτονική Tesla που εμφανίστηκε το 2006, με την GeForce 8800 GTX.
2. Η αρχιτεκτονική Fermi που εμφανίστηκε το 2010, με την GeForce 480.
3. Η αρχιτεκτονική Kepler που εμφανίστηκε το 2012, με την GeForce 680.

Οι αρχιτεκτονικές Tesla και Fermi ακολουθούν την παραδοσιακή αρχιτεκτονική όπου έχουν ένα μεγάλο επεξεργαστή που μπορεί να πετύχει μεγάλες επιδόσεις. Αντίθετα, η αρχιτεκτονική Kepler περιλαμβάνει chip τα όποια είναι μικρότερης ισχύος.
Μια απλοποιημένη περιγραφή της αρχιτεκτονικής της GPU περιλαμβάνει τα ακόλουθα:

- Την διεπαφή με την κεντρική μονάδα που συνδέει την GPU με τον δίαυλο PCI Express.
- 0 – 2 μηχανές αντιγραφής(copy engines).
- Την DRAM διεπαφή που συνδέει την GPU με την μνήμη της κάρτας γραφικών.
- Έναν αριθμό TPCs(texture processing clusters) ή GPCs(graphic processing clusters), που το καθένα περιλαμβάνει caches και ένα αριθμό streaming multprocessors(SMs).

Διεπαφή Κεντρική Μονάδας
Η διεπαφή κεντρικής μονάδας(host interface) διαβάζει εντολές της GPU(πχ. memcpy), τις διανέμει στις κατάλληλες μονάδες του υλικού και συγχρονίζει την GPU με την CPU, τα διαφορετικά μέρη της GPU και διαφορετικές GPU μεταξύ τους.

Μηχανές Αντιγραφής
Οι μηχανές αντιγραφής πραγματοποιούν μεταφορές στις μνήμες μεταξύ της κεντρικής μονάδας και της κάρτας γραφικών ενώ τα SMs πραγματοποιούν υπολογισμούς. Οι πρώτες κάρτες CUDA δεν είχαν μηχανές αντιγραφής. Οι επόμενες γενιές περιλάμβαναν στην αρχιτεκτονική τους μια μηχανή αντιγραφή οι οποίες μπορούν να μεταφέρουν σειριακή μνήμη της κάρτας αλλά όχι πίνακες CUDA. Οι τελευταίες γενιές καρτών περιλαμβάνου δυο μηχανές αντιγραφής που μπορούν να μεταφέρουν και σειριακή μνήμη, αλλά και πίνακες CUDA.

Διεπαφή DRAM
Η διεπαφή DRAM υποστηρίζει εύρος ζώνης μέχρι 100GB/s και μεταφορές ομαδοποιημένων γειτονικών δεδομένων. Οι πιο πρόσφατες κάρτες γραφικών CUDA το υλικό έχει πιο εξελιγμένες διεπαφές DRAM. Οι παλαιότερες γενιές (1.x SM) είχαν απαιτήσεις ομαδοποιήσεων, που απαιτούσαν οι διευθύνσεις να είναι συνεχόμενες και 64-, 128- ή 256-byte ομαδοποιημένες. Από τις γενιές SM 1.2 (GT200 ή GeForce GTX 280), οι διευθύνσεις μπορούν να ομαδοποιηθούν με βάση την τοποθεσία, ανεξάρτητα από την διεύθυνση ευθυγράμμισης.

TPCs και GPCs
Οι μονάδες αυτές βρίσκονται ανάμεσα στην κάρτα γραφικών και τους πολυεπεξεργαστές που πραγματοποιούν τους υπολογισμούς. Οι κάρτες αρχιτεκτονικής Tesla ομαδοποιούν τα SMs σε TPCs που περιέχουν υλικός υφής(texture cache) και δυο ή τρεις πολυεπεξεργαστές. Οι κάρτες αρχιτεκτονικής Fermi έχουν υλικό που ομαδοποιεί τα SMs σε GPCs(graphic processing clusters) από τα οποίας το καθένα περιλαμβάνει μια μονάδα raster και τέσσερα SMs.

3.1 Μνήμη

Στην CUDA, η κεντρική μονάδα και η κάρτα γραφικών έχουν ξεχωριστές μνήμες. Η κάρτα γραφικών έχει την δικιά της δυναμική μνήμη τυχαίας προσπέλασης(DRAM). Οι κάρτες γραφικών CUDA έχουν μνήμες πάνω στο μικροτσίπ(on-chip) και πάνω στην πλακέτα(on-board). Η γρήγορη μνήμη που βρίσκεται πάνω στο μικροτσίπ είναι περιορισμένη και της τάξης των kB. Η μνήμη “Global Memory” που βρίσκεται πάνω στην πλακέτα είναι κοινή για όλα τα SM, είναι της τάξης των GB και είναι η πιο αργή μνήμη της κάρτας γραφικών. Για να τρέξει ένας πυρήνας θα πρέπει ο προγραμματιστής να δεσμεύσει μνήμη στην κάρτα και να μεταφέρει τα αναγκαία δεδομένα από την κεντρική μονάδα στην κάρτα γραφικών. Παρόμοια με το τέλος των διεργασιών στην κάρτα γραφικών πρέπει να μεταφερθούν τα

αποτελέσματα από την κάρτα στην κεντρική μονάδα. Η CUDA διαθέτει τις κατάλληλες βιβλιοθήκες για να γίνουν αυτές οι μεταφορές δεδομένων. Για να μεγιστοποιήσει την απόδοση η CUDA χρησιμοποιεί διάφορα είδη μνήμης που εξαρτάται από την χρήση που θέλουμε να κάνουμε. Τα διαφορετικά χαρακτηριστικά αυτών των μνημών παρουσιάζονται στο παρακάτω πίνακα.

Memory	Location on/off chip	Cached	Access	Scope	Lifetime
Register	On	n/a	R/W	1 thread	Thread
Local	Off	Yes††	R/W	1 thread	Thread
Shared	On	n/a	R/W	All threads in block	Block
Global	Off	†	R/W	All threads + host	Host allocation
Constant	Off	Yes	R	All threads + host	Host allocation
Texture	Off	Yes	R	All threads + host	Host allocation
† Cached in L1 and L2 by default on devices of compute capability 2.x; cached only in L2 by default on devices of higher compute capabilities, though some allow opt-in to caching in L1 as well via compilation flags.					
†† Cached in L1 and L2 by default on devices of compute capability 2.x and 3.x; devices of compute capability 5.x cache locals only in L2.					

Εικόνα 3-1: Είδη μνήμης κάρτας γραφικών και χαρακτηριστικά

Η μνήμη της κάρτας γραφικών η οποία είναι προσαρτημένη πάνω της, την διαχειρίζεται ένας ελεγκτής μνήμης. Έτσι έχουμε τα παρακάτω είδη μνήμης της κάρτας γραφικών.

3.1.1 Global memory

Είναι η μνήμη με το μεγαλύτερο μέγεθος πάνω στην κάρτα γραφικών, αλλά είναι και η πιο αργή μνήμη της. Λόγου του μεγάλου μεγέθους της είναι και αυτή που χρησιμοποιηθεί πιο συχνά, αφού τα δεδομένα από την CPU περνάνε πρώτα από αυτή. Η global memory μπορεί να εγγραφεί και από την CPU και από την GPU. Μπορεί να εγγραφεί από οποιαδήποτε άλλη κάρτα η οποία είναι πάνω στον δίαυλο PCI-E. Σε αρχιτεκτονικές με πολλές κάρτες, οι κάρτες γραφικών μπορούν να μεταφέρουν δεδομένα η μία στην άλλη χωρίς την παρεμβολή της CPU.

Η συνηθισμένη τακτική είναι η CPU να μεταφέρει τα δεδομένα στην GPU, στην συνέχεια να γίνεται οι επιθυμητές διεργασίες πάνω στα δεδομένα και τέλος να μεταφέρονται τα αποτελέσματα από την GPU στην CPU. Τα δεδομένα της global memory είναι ορατά από όλα τα μπλοκ που τρέχουν στην GPU και μένουν σε αυτή καθ' όλη την διάρκεια της εκτέλεσης της εφαρμογής. Η μνήμη αυτή μπορεί να δηλωθεί στατικά ή δυναμικά. Η global memory δεσμεύεται από την CPU μέσω της εντολής *cudaMalloc* και ελευθερώνεται από την εντολή *cudaFree*.

Η σωστή χρήση της global memory είναι από τους βασικότερους παράγοντες ώστε να έχουμε αποδοτικά προγράμματα στην CUDA. Για να πραγματοποιηθεί πρέπει να τηρούνται οι παρακάτω 3 κανόνες:

1. Μεταφορά των δεδομένων στη κάρτα γραφικών και διατήρηση τους.
2. Να δώσουμε στην κάρτα γραφικών μεγάλο φόρτο εργασίας για να εκμεταλλευτούμε τις δυνατότητες της.
3. Να γίνεται επαναχρησιμοποίηση των δεδομένων έτσι ώστε να αποφεύγονται οι καθυστερήσεις λόγω των προσπελάσεων στην μνήμη και του χαμηλού εύρους ζώνης της μνήμης.

Η πρόσβαση σε αυτή γίνεται μέσω δεικτών μέσα στους υπολογιστικούς πυρήνες(kernels) της CUDA. Ο προγραμματιστής πρέπει να δεσμεύσει την μνήμη που θα διαβάσει ο πυρήνας και να μεταφέρει τα δεδομένα από την κεντρική μονάδα στην μνήμη αυτή πριν τρέξει ο πυρήνας. Επειδή η μνήμη είναι

προσαρτημένη πάνω στην κάρτα γραφικών και χρησιμοποιείται μέσω ενός controller που είναι και αυτός ενσωματωμένος πάνω στην κάρτα γραφικών, το εύρος ζώνης μπορεί να φτάσει μέχρι τα 100G/s.

3.1.2 Καταχωρητές(Registers)

Οι καταχωρητές είναι η πιο γρήγορη μνήμη στην κάρτα γραφικών. Σε αντίθεση με την CPU, η GPU έχει χιλιάδες καταχωρητές ανά SM. Είναι από τα σημαντικότερα στοιχεία, γιατί έχουν μεγάλο εύρος ζώνης, πολύ μικρή καθυστέρηση και με την χρήση τους επιτυγχάνονται οι μέγιστες επιδόσεις.

Μια βασική διαφορά μεταξύ της CPU και της GPU είναι ο τρόπος που κατανέμουν τους καταχωρητές. Για να τρέξει πολλά νήματα η CPU χρησιμοποιεί την εναλλαγή των καταχωρητών και την μνήμη stack. Για να τρέξει μια καινούργια διεργασία η CPU χρειάζεται να αλλάξει το πλαίσιο εκτέλεσης, στο οποίο περιλαμβάνεται και η αποθήκευση όλων των καταχωρητών στην μνήμη stack. Στην συνέχεια γίνεται επαναφορά του πλαισίου στην κατάστασης που ήταν την τελευταία φορά που έτρεξε το καινούργιο νήμα. Αυτή η εναλλαγή της κατάστασης στην οποία τρέχει το κάθε νήμα απαιτεί αρκετούς κύκλους ρολογιού. Αν τρέξουμε πολλά νήματα πάνω σε μια CPU ο περισσότερος χρόνος δαπανείται σε αυτή την εναλλαγή των καταχωρητών και γενικότερα του πλαισίου για κάθε νήμα. Σε αντίθεση η κάρτα γραφικών λειτουργεί με τον αντίθετο τρόπο. Χρησιμοποιεί τα νήματα για να κρύψει τις καθυστερήσεις από τις προσπελάσεις της μνήμης και από την εκτέλεση των εντολών. Έτσι, αν τρέχουν πολύ λίγα νήματα σημαίνει ότι η κάρτα θα μένει ανεργή περιμένοντας κυρίως δοσοληψίες με την μνήμη. Επίσης, η κάρτα γραφικών δεν πραγματοποιεί εναλλαγές στους καταχωρητές για να εκτελεστεί το κάθε νήμα. Το κάθε νήμα αφιερώνει σε κάθε νήμα πραγματικούς καταχωρητές. Έτσι, κατά την εναλλαγή του πλαισίου για την εκτέλεση μια άλλης ομάδας νημάτων υπάρχει μηδενικό κόστος.

Κάθε SM μπορεί να εκτελέσει έναν αριθμό μπλοκ. Τα μπλοκ σε επίπεδο SM είναι απλά ομάδες ανεξάρτητων στημονιών(warps). Ο αριθμός των καταχωρητών ανά νήμα υπολογίζεται κατά τη μεταγλώττιση. Όλα τα μπλοκ έχουν το ίδιο μέγεθος και έχουν ένα γνωστό αριθμό νημάτων, έτσι η απαίτηση των καταχωρητών ανά μπλοκ είναι γνωστή και σταθερή. Κατά συνέπεια, η GPU μπορεί να διαθέσει ένα σταθερό αριθμό καταχωρητών για κάθε μπλοκ. Ωστόσο, ένας πυρήνας που απαιτεί πολλούς καταχωρητές ανά νήμα μπορεί να περιορίσει τον αριθμό των μπλοκ που μπορούν να τρέξουν πάνω σε ένα SM, και επομένως τον συνολικό αριθμό των νημάτων.

Ανάλογα με την γενιά της κάρτας γραφικών μπορούν να υπάρχουν 8 K, 16 K, 32 K ή 64 K καταχωρητών ανά SM, για όλα τα νήματα του SM. Έτσι, μια απλή τοπική μεταβλητή float στη C θα χρειαστεί N καταχωρητές, όπου N είναι ο αριθμός των νημάτων που είναι προγραμματισμένα να τρέξουν. Με την αρχιτεκτονική Fermi, έχουμε 32K καταχωρητών ανά SM. Αν έχουμε 256 νήματα ανά μπλοκ, έχουμε ((32.768 / 4 bytes ανά καταχωρητή) / 256 νήματα) = 32 καταχωρητές ανά νήμα διαθέσιμα. Για να επιτευχθεί ο μέγιστος αριθμός των καταχωρητών ανά νήμα στην αρχιτεκτονική Fermi, 64 (128 για G80 / GT200), τότε θα πρέπει να μειώσουμε κατά το ήμισυ τον αριθμό των νημάτων, δηλαδή σε 128. Έτσι, εναλλάσσοντας τον αριθμό των νημάτων ανά μπλοκ και των αριθμό των μπλοκ ανά SM μπορούμε να επιτύχουμε τον μέγιστο αριθμό καταχωρητών ανά νήμα.

Register Availability by Thread Usage on Fermi

No. of Threads	Maximum Register Usage					
192	16	20	24	28	32	64
Blocks Scheduled	8	8	7	6	5	2
No. of Threads	**Maximum Register Usage**					
256	16	20	24	28	32	64
Blocks Scheduled	6	6	5	4	4	2

Εικόνα 3-2: Μέγιστος αριθμός καταχωρητών ανά νήμα

3.1.3 Constant memory

Η constant memory είναι μια μορφή εικονικής διευθυνσιοδότησης της global memory, χωρίς να υπάρχει κάποιο συγκεκριμένο υλικό κομμάτι μνήμης. Η μνήμη αυτή είναι read-only και είναι προσβάσιμη μέσω συγκεκριμένων εντολών που σκοπό έχουν να κατανέμουν τα δεδομένα στα διάφορα νήματα. Το μέγεθος

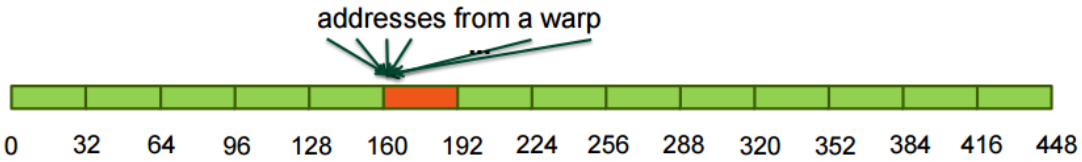
της είναι 64K. Όπως και στην global memory, έτσι και εδώ μπορούν να μεταφερθούν δεδομένα από την κεντρική μονάδα. Στις συσκευές με υπολογιστική ικανότητα 1.0 η constant memory είναι ένας καλός τρόπος ώστε να αποθηκεύσουμε read-only δεδομένα και να τα αναμεταδοθούν σε όλα τα νήματα της GPU. Η cache της constant memory έχει μέγεθος 64KB. Μπορεί να αναμεταδώσει 32-bits ανά στημόνι ανά 2 κύκλους ρολογιού και πρέπει να χρησιμοποιείται όταν όλα τα νήματα σε ένα στημόνι διαβάζουν από την ίδια διεύθυνση.

Στην αρχιτεκτονική Fermi (υπολογιστική ικανότητα 2.x) και μετά, υπάρχει μια μνήμη (L2) cache. Οι αρχιτεκτονικές αυτές διαμοιράζουν την μνήμη L2 cache ανάμεσα στα SM. Όλες οι προσπελάσεις μνήμης αποθηκεύονται προσωρινά από την μνήμη L2 cache. Επιπροσθέτως, το μέγεθος της μνήμης cache L1 μπορεί να αυξηθεί από 16 K σε 48 K, θυσιάζοντας 32 K από την κοινόχρηστη μνήμη ανά SM.

Η αρχιτεκτονική Fermi, σε αντίθεση με τις συσκευές υπολογιστικής ικανότητας 1.x, επιτρέπει σε όλα τα σταθερά δεδομένα να αντιμετωπίζονται ως constant μνήμη, ακόμη και αν δεν έχουν δηλωθεί ως τέτοια. Η constant memory σε συσκευές 1.x, πρέπει να αντιμετωπίζεται από ειδικού τύπου συναρτήσεις, όπως cudaMemcpyToSymbol ή να δηλώνεται κατά τη μεταγλώττιση. Με την αρχιτεκτονική Fermi, οποιαδήποτε nonthread-based πρόσβαση σε μια περιοχή της μνήμης η οποία έχει δηλωθεί ως constant (απλά με την λέξη-κλειδί const) περνά μέσα από τη μνήμη cache. Με τον όρο nonthread-based πρόσβαση, εννοούμε ότι η πρόσβαση δεν περιλαμβάνει το threadIdx.x στην εύρεση του στοιχείου. Σε περίπτωση που είναι αναγκαία η πρόσβαση σε σταθερά δεδομένα σε per-thread-based πρόσβαση, τότε θα πρέπει να χρησιμοποιήσουμε το λεκτικό *__constant__* ή την συνάρτηση *cudaMemcpyToSymbol* όπως συμβαίνει με τις συσκευές υπολογιστικής ικανότητας 1.x.

Οι συσκευές με υπολογιστική ικανότητα 2.0 και πάνω επιτρέπουν στον προγραμματιστή να έχει πρόσβαση στην global memory μέσω της constant memory, που είναι πιο αποδοτική, όταν ο compiler αναγνωρίσει μια εντολή ομοιόμορφης φόρτωσης(LoaD Uniform instruction). Πιο συγκεκριμένα τα δεδομένα πρέπει:

1. Να βρίσκονται στην global memory.
2. Να είναι προσβάσιμα στον kernel μόνο ως read-only.
3. Να μην εξαρτώνται από την ταυτότητα του νήματος(threadId).



Εικόνα 3-3: Ομοιόμορφη πρόσβαση μνήμης

3.1.4 Local memory

Η τοπική μνήμη ονομάζεται τοπική επειδή σε αυτή έχει μόνο πρόσβαση τοπικά το νήμα και όχι από την υλική τοποθεσία της πάνω στην κάρτα. Υλικά η τοπική μνήμη δεν βρίσκεται μέσα στο μικροτσίπ της κάρτας, αλλά πάνω στην global memory και για αυτό η πρόσβαση σε αυτή είναι όσο χρονοβόρα είναι και η πρόσβαση στην Global memory. Η τοπική μνήμη χρησιμοποιείται από τον nvcc compiler όταν υπολογίζει ότι δεν επαρκούν οι καταχωρητές για την αποθήκευση κάποιας μεταβλητής. Ένα παράδειγμα μιας τέτοιας μεταβλητής είναι ένας πίνακας που χρησιμοποιείται τοπικά μόνο από ένα νήμα και ο χώρος των καταχωρητών δεν επαρκεί για την αποθήκευσή του. Μεταβλητές που είναι πιθανό ο μεταγλωττιστής να τοποθετήσει στην local memory είναι οι ακόλουθες:

- Τοπικούς πίνακες που το μέγεθος τους δεν μπορεί να καθοριστεί κατά την διάρκεια της μεταγλώττισης του προγράμματος.
- Μεγάλους τοπικούς πίνακες ή δομές δεδομένων που θα καταναλώσουν μεγάλος μέρος των καταχωρητών.
- Οποιαδήποτε μεταβλητή η οποία ξεπερνάει το όριο των καταχωρητών.

Στις κάρτες γραφικών με υπολογιστική ικανότητα 2.0 και πάνω, η local memory περνάει από cache ανά SM L1 και ανά συσκευή L2.

3.1.5 Texture memory

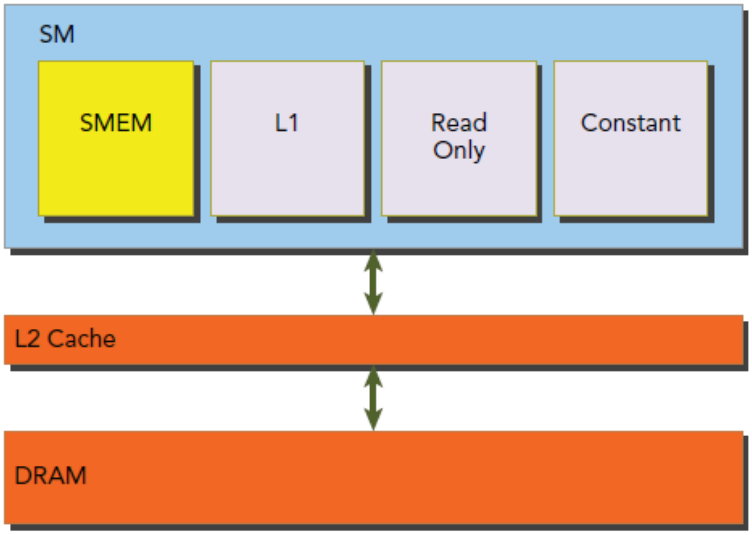
Στις συσκευές με υπολογιστική ικανότητα 1.x δεν υπάρχει cache, επομένως η χρήση της texture memory έδινε την δυνατότητα ενός είδους cache. Με την ανάπτυξη όμως της αρχιτεκτονικής Fermi και τα 48K L1 cache και τα 768K L2 cache, η χρήση θεωρείται ξεπερασμένη. Η ύπαρξη της ακόμα και στις Fermi GPU έγινε για να υπάρχει συμβατότητα με κώδικα που αναφερόταν σε παλιές γενιές καρτών γραφικών. Η χρήση της προορίζεται για τοπικές μεταβλητές που συνδέονται με τα νήματα που τρέχουν στο συγκεκριμένο SM στο οποίο βρίσκεται και η texture memory.

Ενεργοποιείται μόνο με την χρήση συγκεκριμένων εντολών και είναι read-only μνήμη που εξυπηρετείται από μια cache μνήμη για την βελτιστοποίησή των αναγνώσεων. Χρησιμοποιείται σαν μια μορφή cache μνήμης για να αποφύγουμε την αργή global memory. Το μέγεθος όμως αυτής της μνήμης είναι αρκετά μικρό, της τάξης των 8KB.

3.1.6 Shared Memory

Η κοινόχρηστη μνήμη (shared memory) είναι μια από τις βασικότερες μονάδες των καρτών γραφικών της αρχιτεκτονικής CUDA. Υλικά η μνήμη αυτή περιέχεται μέσα στο SM, είναι πολύ γρήγορη και την μοιράζονται όλα τα νήματα τα οποία τρέχουν από το SM. Ο σκοπός της είναι, χρησιμοποιώντας την, να αποφευχθεί η χρήση της αργής global memory. Επειδή το περιεχόμενο της κοινόχρηστης μνήμης, το διαχειρίζεται η εφαρμογή, συχνά περιγράφεται ως διαχειρίσιμη cache. Τα δεδομένα της, δηλαδή φορτώνονται από τον προγραμματιστή και όχι από την λογική της αρχιτεκτονικής, όπως θα ήταν αναμενόμενο από μια κοινή cache.

Όπως φαίνεται στην Εικόνα 3-4: Τοπολογία μνημών, κάθε φόρτωση και αποθήκευση στην global memory περνάει από την L2 Cache, που είναι το σημείο που τα δεδομένα των SM ενοποιούνται. Η κοινόχρηστη μνήμη και η L1 Cache είναι πιο κοντά στο SM από την L2 cache και από την global memory. Σαν αποτέλεσμα η κοινόχρηστη μνήμη έχει 20 με 30 φορές λιγότερη καθυστέρηση σε σχέση με την global memory.



Εικόνα 3-4: Τοπολογία μνημών

Η κοινόχρηστη μνήμη έχει μέγεθος 16KB ή 48KB ανά SM και είναι οργανωμένη σε 32 τράπεζες μνήμης (memory banks), μήκους 32 bit. Υλικά δεν βρίσκεται πάνω στην κάρτα, αλλά πάνω στο μικροτσίπ της κάρτας γραφικών. Ένα συγκεκριμένο μέγεθος κοινόχρηστης μνήμης αποδίδεται σε κάθε μπλοκ νημάτων και όλα τα νήματα του έχουν πρόσβαση στην μνήμη. Τα δεδομένα της κοινόχρηστης μνήμης έχουν ζωή όσο διαρκεί και η ζωή του μπλοκ των νημάτων.

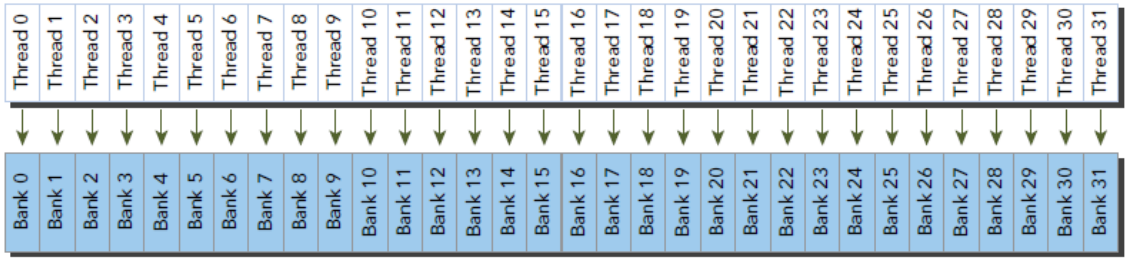
Η κοινόχρηστη μνήμη αποτελείται από 32 τράπεζες μνήμης πάνω στο SM. Σε ιδανικές συνθήκες 32 νήματα μπορούν να προσπελάσουν την κοινόχρηστη μνήμη χωρίς καμία καθυστέρηση. Παρόλα αυτά έχουμε πολλές φορές συγκρούσεις στην κοινόχρηστη μνήμη, όταν περισσότερα από ένα νήματα θέλουν να προσπελάσουν τα δεδομένα στην ίδια τράπεζα μνήμης. Αυτές οι συγκρούσεις μπορεί να είναι για την

ίδια διεύθυνση ή για διαφορετικές διευθύνσεις μέσα στην ίδια τράπεζα. Σε αυτές τις περιπτώσεις το υλικό μετατρέπει τα αιτήματα σε σειριακά αιτήματα. Έτσι, αν σε ένα στημόνι έχουμε ν νήματα που θέλουν να έχουν πρόσβαση στην ίδια τράπεζα μνήμης, θα έχουμε ν φορές καθυστέρηση σε αυτό το SM.

Όταν πολλαπλές διευθύνσεις, σε ένα κοινό αίτημα μνήμης προορίζονται στην ίδια τράπεζα μνήμης, τότε έχουμε σύγκρουση για την ίδια τράπεζα μνήμης, προκαλώντας επανάληψη του αιτήματος. Το υλικό χωρίζει ένα αίτημα σε μια τράπεζα μνήμη, όπου έχουμε σύγκρουση, σε πολλές χωριστές αιτήσεις χωρίς συγκρούσεις. Το εύρος ζώνης μειώνεται, κατά ένα συντελεστή ίσο με τον αριθμό των ξεχωριστών απαιτούμενων συναλλαγών. Υπάρχουν τρείς περιπτώσεις, όταν έχουμε ένα αίτημα στην κοινόχρηστη μνήμη:

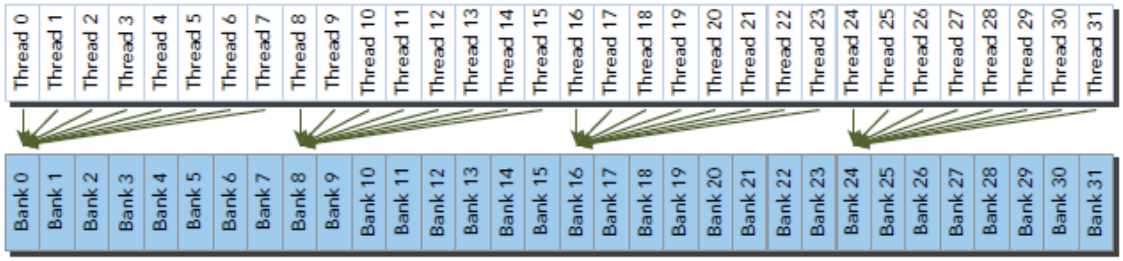
- Παράλληλη προσπέλαση: έχουμε προσπέλαση σε πολλές διευθύνσεις σε διαφορετικές τράπεζες μνήμη.
- Σειριακή προσπέλαση: έχουμε προσπέλαση πολλών διευθύνσεων στην ίδια τράπεζα μνήμης.
- Μετάδοση προσπέλασης(Broadcast access): έχουμε προσπέλαση μιας διεύθυνσης σε μια τράπεζα μνήμης.

Η παράλληλη προσπέλαση είναι το πιο κοινό μοτίβο, όπου έχουμε προσπέλαση πολλών διευθύνσεων από ένα στημόνι, οι οποίες βρίσκονται σε διαφορετικές τράπεζες μνήμης. Σε αυτό το μοτίβο, όλες οι διευθύνσεις μπορούν να προσπελαστούν με μια μόνο δοσοληψία.



Εικόνα 3-5: Παράλληλη προσπέλαση κοινόχρηστης μνήμης

Η σειριακή προσπέλαση είναι το χειρότερο μοτίβο. Όταν όλες οι διευθύνσεις βρίσκονται στην ίδια τράπεζα, τότε το αίτημα πρέπει να γίνει σειριακό. Αν και τα 32 νήματα ενός στημονιού θέλουν να προσπελάσουν διαφορετικές διευθύνσεις μέσα στην ίδια τράπεζα μνήμης, τότε πρέπει να πραγματοποιηθούν 32 δοσοληψίες για να ικανοποιηθεί αυτό το αίτημα. Αποτέλεσμα αυτό το αίτημα είναι 32 φορές πιο αργό από ένα αίτημα παράλληλης προσπέλασης. Στην Εικόνα 3-6: Σειριακή προσπέλαση κοινόχρηστης μνήμης βλέπουμε ένα παράδειγμα που έχουμε συγκρούσεις σειριακής μνήμης, όπου 7 διαφορετικά μνήματα θέλουν να προσπελάσουν την ίδια τράπεζα μνήμης.



Εικόνα 3-6: Σειριακή προσπέλαση κοινόχρηστης μνήμης

Στην περίπτωση της μετάδοσης προσπέλασης, όλα τα νήματα σε ένα στημόνι διαβάζουν της ίδια διεύθυνση σε μια τράπεζα μνήμης. Μια δοσοληψία εκτελείται και η διεύθυνση η οποία διαβάζεται αναμεταδίδεται σε όλα τα νήματα. Παρόλο που μόνο μία δοσοληψία πραγματοποιείται δεν έχουμε καλή χρήση του εύρους ζώνης, αφού μεταδίδεται μόνο ένα μικρό μέρος μνήμης.

Υπάρχουν τρεις τρόποι για να δηλώσουμε και να χρησιμοποιήσουμε την κοινόχρηστη μνήμη:

1. Στατικά μέσα στο kernel χρησιμοποιώντας το λεκτικό __shared__.
2. Δυναμικά μέσα στο kernel καλώντας την συνάρτηση cuFuncSetSharedSize της CUDA.

3. Δυναμικά μέσω της παραμετροποίησης της εκτέλεση του kernel, κατά την κάλεσμα του από την CPU.

Όπως είδαμε παραπάνω κάθε SM έχει πάνω στο τσιπ μια μνήμη μεγέθους 64KB. Αυτή την μνήμη την διαμοιράζονται η κοινόχρηστη μνήμη και η L1 cache. Η CUDA μας δίνει την δυνατότητα να καθορίσουμε την L1 cache και την κοινόχρηστη μνήμη με δύο τρόπους:

- Ανά συσκευή.
- Ανά υπολογιστικό πυρήνα.

Μπορούμε να ρυθμίσουμε το μέγεθος της L1 cache και της κοινόχρηστης μνήμης που θα χρησιμοποιηθεί από τους υπολογιστικούς πυρήνες μέσω της παρακάτω συνάρτησης:

```
cudaError_t cudaDeviceSetCacheConfig(cudaFuncCache cacheConfig)
```

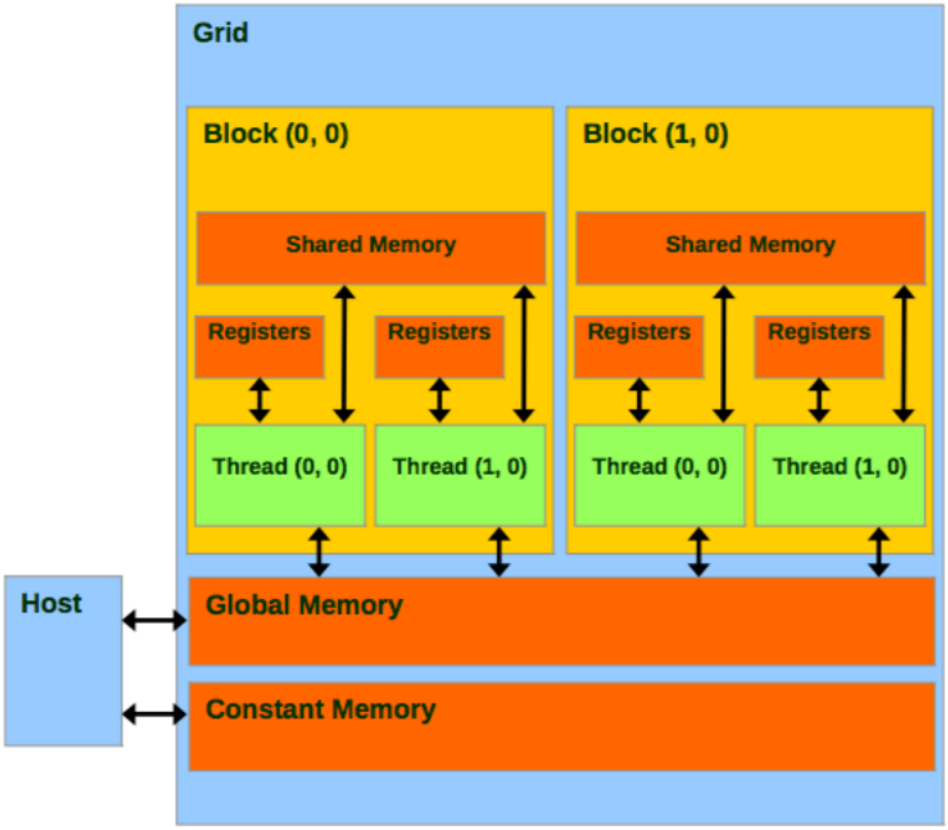
Η παράμετρος *cacheConfig* το ποσό από την μνήμη που βρίσκεται πάνω στο SM, που θα μοιραστεί στην κοινόχρηστη μνήμη και στην L1 cache. Η υποστηριζόμενες ρυθμίσεις είναι οι παρακάτω:

```
cudaFuncCachePreferNone:   προκαθορισμένες ρυθμίσεις
cudaFuncCachePreferShared: 48KB shared memory και 16 KB L1 cache
cudaFuncCachePreferL1:     48KB L1 cache και 16 KB shared memory
cudaFuncCachePreferEqual:  32KB L1 cache και 32 KB shared memory
```

Το ποια ρύθμιση θα προτιμήσουμε εξαρτάται από την λειτουργία της εφαρμογής μας. Στην περίπτωση που έχουμε μια εφαρμογή, όπου οι υπολογιστικοί πυρήνες χρησιμοποιούν πολύ την κοινόχρηστη μνήμη τότε θα πρέπει να επιλέξουμε η κοινόχρηστη μνήμη να έχει μέγεθος 48K. Στην περίπτωση που η εφαρμογή μας δεν χρησιμοποιεί πολύ κοινόχρηστη μνήμη, αλλά αντιθέτως πολλούς καταχωρητές τότε είναι καλύτερο να επιλέξουμε η L1 cache να είναι 48KB.

Μια άλλη σημαντική λειτουργία της CUDA που παίζει σημαντικό ρόλο στην χρήση της κοινόχρηστης μνήμης είναι ο συγχρονισμός των νημάτων

Συνήθως οι πυρήνες που χρησιμοποιούν την shared memory είναι γραμμένοι σε τρεις φάσεις. Πρώτα φορτώνουν τα δεδομένα στην shared memory και συγχρονίζουν τα νήματα. Στη συνέχεια πραγματοποιούν όλες τις διεργασίες που απαιτούνται πάνω στα δεδομένα αυτά και συγχρονίζουν πάλι τα νήματα. Τέλος, μεταφέρουμε τα αποτελέσματα στην Global memory.



Εικόνα 3-7: Αρχιτεκτονική Μνήμης CUDA

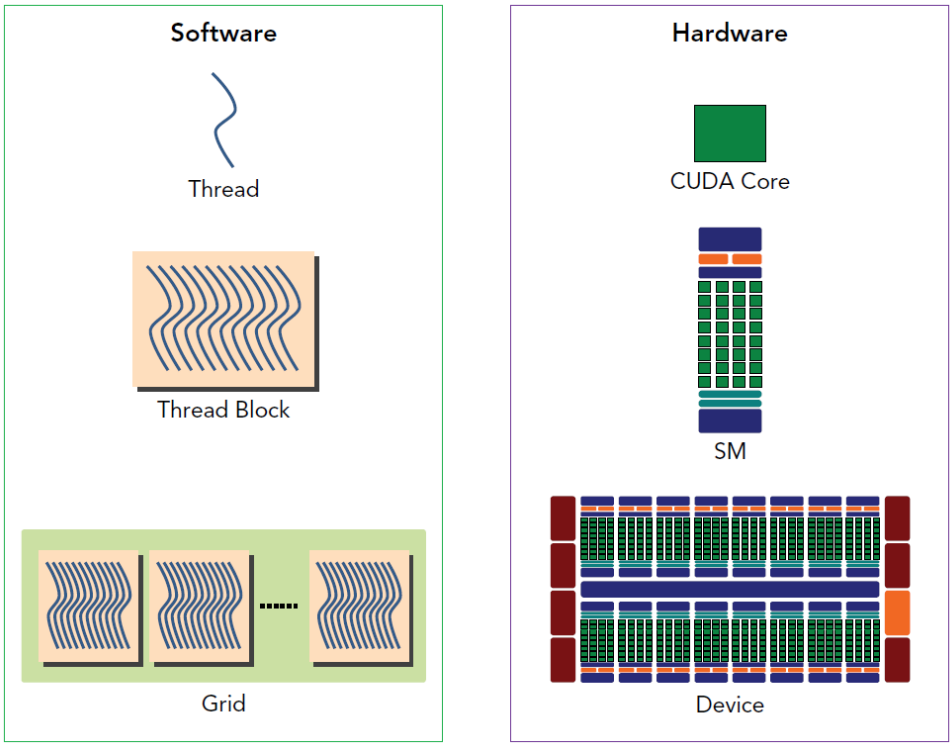
3.2 Επεξεργαστής Συνεχούς Ροής(SMs)

Η δύναμη της κάρτας γραφικών βρίσκεται στους επεξεργαστές συνεχούς ροής(SMs). Οι γενιές 1.x των καρτών γραφικών της NVIDIA περιλαμβάνουν 2 ή 3 SMs, ενώ οι γενιές 2.x περιλαμβάνουν 4 SMs.

Ένα SM περιέχει τα παρακάτω μέρη:

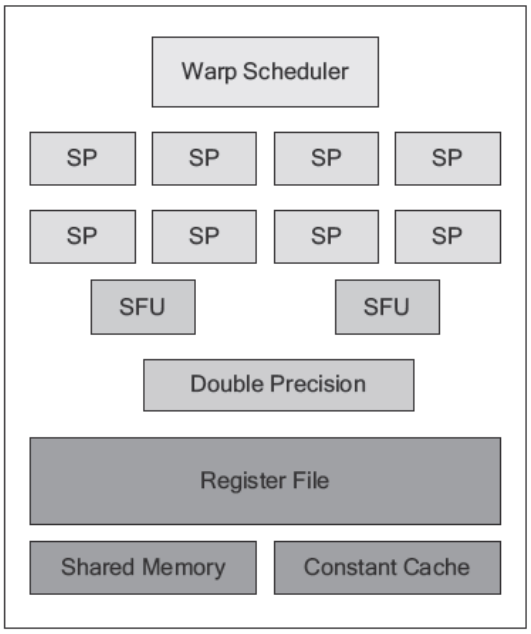
- Μονάδα επεξεργασίας που μπορεί να πραγματοποιήσει 32-bit πράξεις ακεραίων και διπλής ακρίβειας πράξεις κινητής υποδιαστολής.
- Ειδικές μονάδες(SFU-Special Function Units) για τον υπολογισμό πράξεων log/exp, sin/cos και rcp/rsqrt.
- Ένα χρονοπρογραμματιστή στημονιών(warp scheduler) για την κατανομή των διεργασιών στις μονάδες επεξεργασίας.
- Μια μνήμη cache για την μεταφορά δεδομένων στα SMs.
- Την διαμοιραζόμενη μνήμη(share memory) για ανταλλαγή δεδομένων μεταξύ των νημάτων.

Κάθε SM μέσα στην GPU είναι σχεδιασμένο ώστε να υποστηρίζει την εκτέλεση εκατοντάδων παράλληλων νημάτων. Σε μια κάρτα γραφικών υπάρχουν πολλά SM, έτσι ώστε να υποστηρίζεται η εκτέλεση χιλιάδων νημάτων παράλληλα. Όταν εκτελείται ένας υπολογιστικός πυρήνας, τα νήματα των μπλοκ του πλέγματος του πυρήνα διανέμονται στα SM της κάρτας για την εκτέλεση τους. Αφού διανεμηθούν στα SM τα νήματα του κάθε SM, θα εκτελεστούν παράλληλα. Πολλά μπλοκ μπορούν να ανατεθούν σε ένα SM και θα εκτελεστούν την στιγμή, που θα υπάρχουν οι διαθέσιμοι πόροι από το SM. Ένα μπλοκ νημάτων ανατίθεται μόνο σε ένα SM και μένει σε αυτό μέχρι να ολοκληρωθεί η εκτέλεση του. Στην Εικόνα 3-8: Αντιστοίχιση λογικής CUDA στο υλικό, παρουσιάζεται η αντιστοίχιση της λογικής της CUDA, πάνω στο υλικό της κάρτας.



Εικόνα 3-8: Αντιστοίχιση λογικής CUDA στο υλικό

Η Εικόνα 3-9: Επεξεργαστής Συνεχούς Ροής 1.x(Streaming Multiprocessor) μας δείχνει έναν επεξεργαστή ροής αρχιτεκτονικής Tesla. Η κάρτα γραφικών Tesla περιλαμβάνει 8 SMs οι οποίοι υποστηρίζουν 32-bit πράξεις ακεραίων και μονής ακρίβειας πράξεις κινητής υποδιαστολής. Οι πρώτες κάρτες γραφικών CUDA δεν υποστήριζαν διπλής ακρίβειας κινητής υποδιαστολής μονάδες.

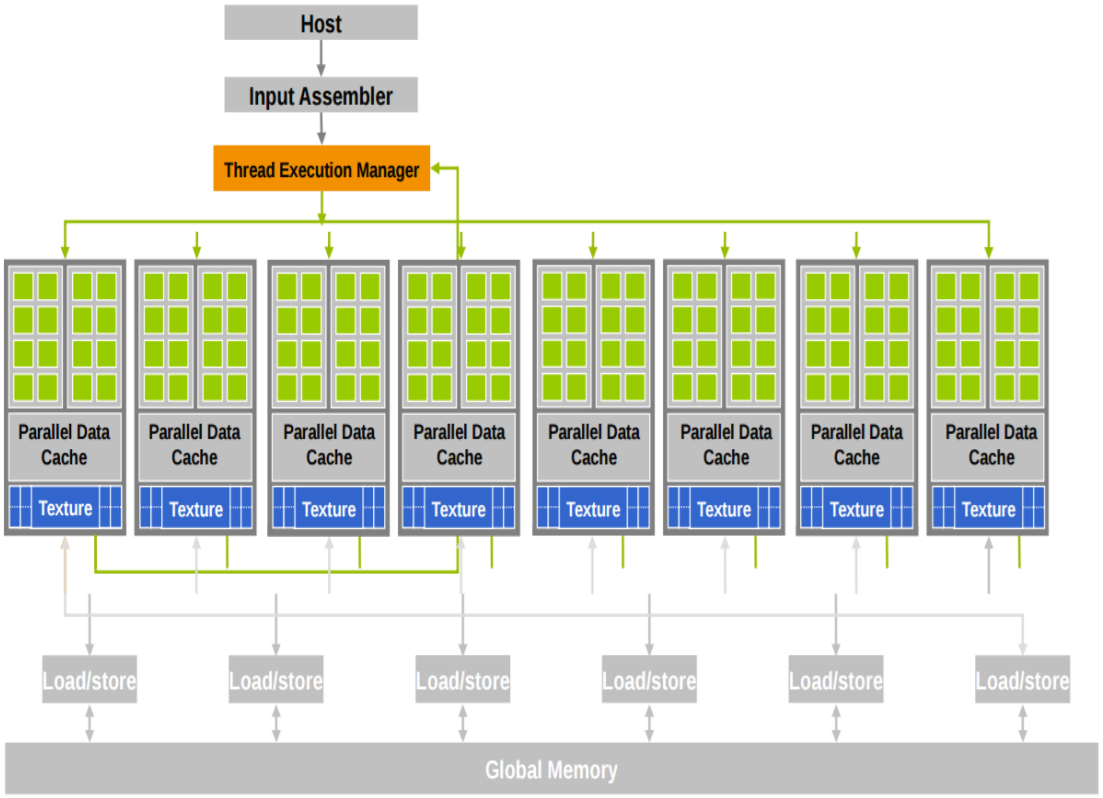


Εικόνα 3-9: Επεξεργαστής Συνεχούς Ροής 1.x(Streaming Multiprocessor)

3.2.1 Αρχιτεκοντική G80

Η πρώτη κάρτα γραφικών που υποστήριζε την αρχιτεκτονική CUDA ήταν σχεδιασμένη πάνω στην αρχιτεκτονική G80, και εμφανίστηκε το 2006 με την GeForce 8800. Αυτή η αρχιτεκτονική έφερε διάφορες καινοτομίες:

- Ήταν η πρώτη κάρτα γραφικών η οποία υποστήριζε την γλώσσα προγραμματισμού C, επιτρέποντας στους προγραμματιστές να χρησιμοποιήσουν την κάρτα γραφικών χωρίς να χρειάζεται μια νέα γλώσσα.
- Ήταν η πρώτη κάρτα γραφικών που αντικατέστησε τους ξεχωριστούς επεξεργαστές κορυφής(vertex) και pixel, αντικαθιστώντας τους με ένα νέο επεξεργαστή που εκτελούσε διεργασίες κορυφής, γεωμετρίας, pixel και υπολογιστικές.
- Εισήγαγε το μοντέλο SIMT(Single-Instruction Multiple-Threads), όπου πολλά νήματα εκτελούν παράλληλα την ίδια οδηγία.
- Εισήγαγε την κοινόχρηστη μνήμη(Shared Memory) και τον συγχρονισμό των νημάτων για ενδοεπικοινωνία των νημάτων.



Εικόνα 3-10: Αρχιτεκτονική G80

3.2.2 Αρχιτεκτονική Fermi

Η αρχιτεκτονική Fermi ήταν το πιο σημαντικό βήμα στην ανάπτυξη των καρτών γραφικών έπειτα από την πρώτη γενιά καρτών γραφικών G80. Η αρχιτεκτονική GT200 πραγματοποίησε κάποιες βελτιώσεις στις επιδόσεις του προκατόχου του G80. Στη συνέχεια η αρχιτεκτονική Fermi επέκτεινε τις λειτουργίες και τις επιδόσεις των δύο προκατόχων της. Συλλέγοντας την εμπειρία από την προηγούμενη γενιά καρτών γραφικών η αρχιτεκτονική Fermi σχεδιάστηκε δίνοντας έμφαση στους παρακάτω τομείς:

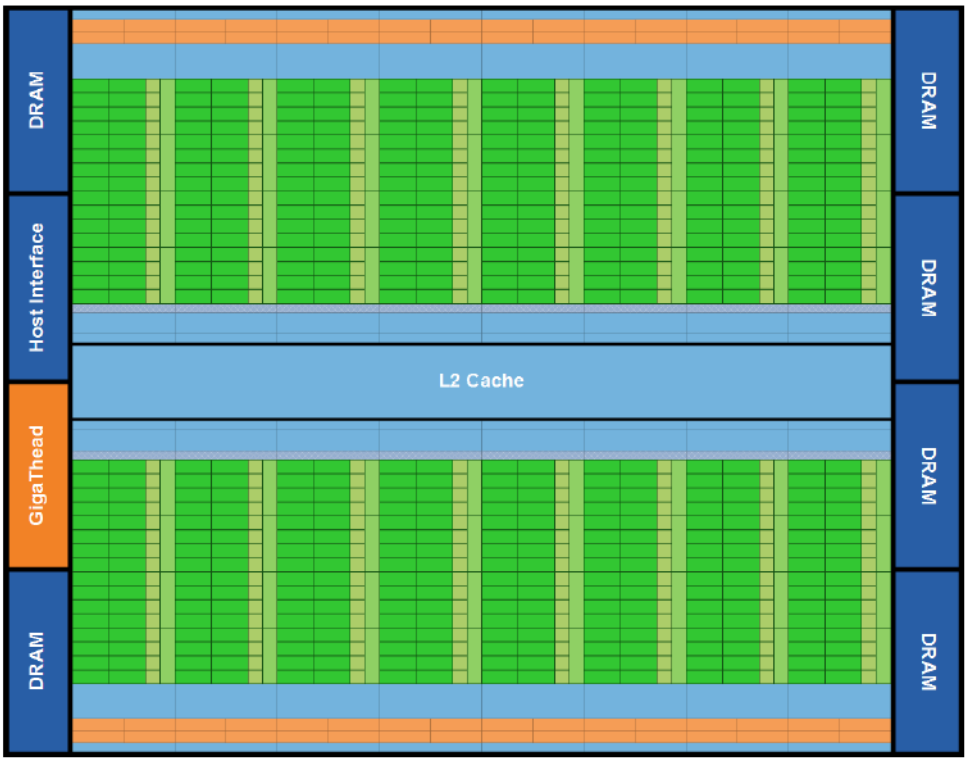
- **Βελτιστοποίηση των πράξεων διπλής ακρίβειας:** ενώ οι επιδόσεις των πράξεων μονής ακρίβειας ήταν της τάξης 10x σε σχέση με τις CPU, μερικές εφαρμογές GPU απαιτούσαν βέλτιστες επιδόσεις στις πράξεις διπλής ακρίβειας.

- **ECC υποστήριξη:** Το ECC επιτρέπει στους χρήστες καρτών γραφικών να εγκαθιστούν μεγάλο αριθμό καρτών σε μεγάλα υπολογιστική συστήματα και να αποφεύγονται τα λάθη μνήμης σε εφαρμογές με ευαίσθητα δεδομένα, όπως εφαρμογές ιατρικού περιεχομένου, κλπ.
- **Ιεραρχία Cache:** Κάποιες εφαρμογές λόγω της λειτουργίας τους δεν είχαν την δυνατότητα να κάνουν χρήση της shared memory, και έτσι οι χρήστες είχαν την απαίτηση μιας αρχιτεκτονικής cache για την βελτίωση των επιδόσεων.
- **Μεγαλύτερη Shared Memory:** Κάποιοι χρήστες απαίτησαν μεγαλύτερη από 16K shared memory για να επιταχύνουν της εφαρμογές τους.
- **Ταχύτερες Ατομικές Διεργασίες:** Οι χρήστες απαιτούσαν ταχύτερες εγγραφές και αναγνώσεις σε ατομικές μεταβλητές, που μπορούσαν να μεταβληθούν από πολλά παράλληλα νήματα.

Με αυτές τις προδιαγραφές η αρχιτεκτονική Fermi αναπτύχθηκε έτσι ώστε να προσφέρει μεγαλύτερες επιδόσεις και μέσο νέων καινοτομιών να διευκολύνει την δημιουργία εφαρμογών που υποστηρίζουν την χρήση καρτών γραφικών. Οι κύριες αρχιτεκτονικές καινοτομίες της αρχιτεκτονικής Fermi είναι η εξής:

- **Τρίτης γενιάς επεξεργαστές ροής(SM)**
  - 32 CUDA πυρήνες ανά SM
  - 8x αύξηση της επίδοσης των πράξεων διπλής ακρίβειας
  - Διπλούς χρονοπρογραμματιστές στημονιών που μπορούν να προγραμματίσουν και να εκπέμψουν οδηγίες σε δύο ανεξάρτητα στημόνια
  - 64KB μνήμης, που ένα μέρος της μπορεί να χρησιμοποιηθεί σαν shared και σαν L1 cache
- **Δεύτερης Γενιάς Παράλληλη Εκτέλεση Νημάτων ISA**
- **Βελτιωμένο Σύστημα Υπομνήμης**
  - Βελτιωμένες επιδόσεις σε ατομικές περιοχές μνήμης
  - Υποστήριξη ECC
- **NVIDIA GigaThread μηχανή**
  - Παράλληλη εκτέλεση υπολογιστικών πυρήνων(kernel)
  - 10x ταχύτερη εναλλαγή πλαισίου εφαρμογής
  - Ασύγχρονη εκτέλεση των μπλοκ

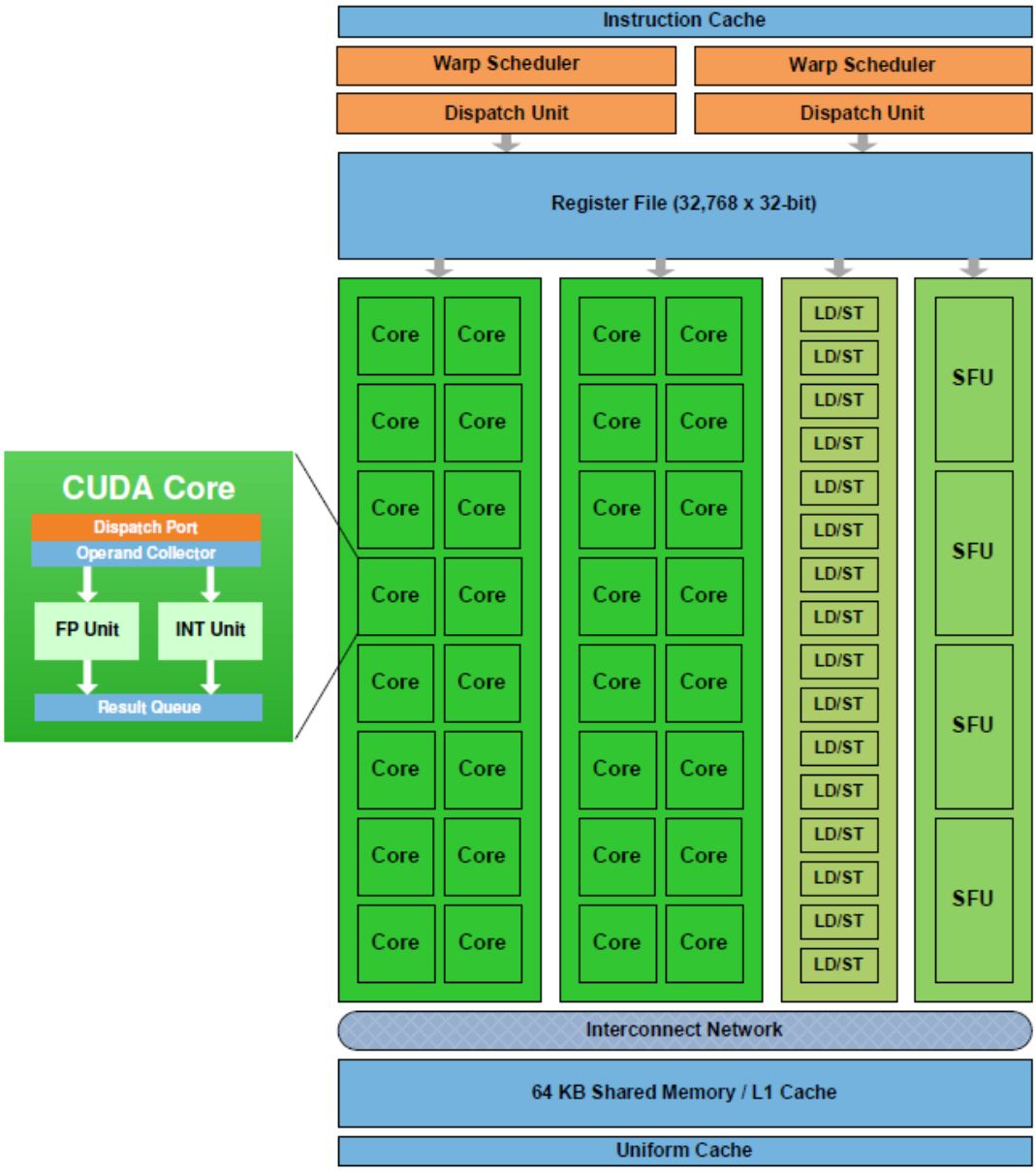
Η πρώτη κάρτα γραφικών που βασίστηκε στην αρχιτεκτονική Fermi περιείχε 3 δισεκατομμύρια τρανζίστορ, τα οποία παρείχαν 512 CUDA πυρήνες. Ένας πυρήνας CUDA εκτελεί μια εντολή κινητής υποδιαστολής ή ακεραίου σε ένα κύκλο ρολογιού για κάθε νήμα. Οι 512 CUDA πυρήνες είναι οργανωμένοι σε 16 επεξεργαστές ροής(Streaming Multiprocessor) 32 πυρήνων ο καθένας. Η GPU έχει έξι 64-bit μέρη μνήμης, για μία 384-bit διεπαφή μνήμης, που υποστηρίζει μέχρι 6 GB GDDR5 DRAM μνήμης. Μια διεπαφή στην κεντρική μονάδα ενώνει την GPU με την CPU μέσω PCI-Express. Το υπερνήμα(GigaThread) χρονοπρογραμματισμού διανέμει τα μπλοκ νημάτων στους χρονοπρογραμματιστές των SM.



Εικόνα 3-11: Κάτοψη Αρχιτεκτονικής Fermi

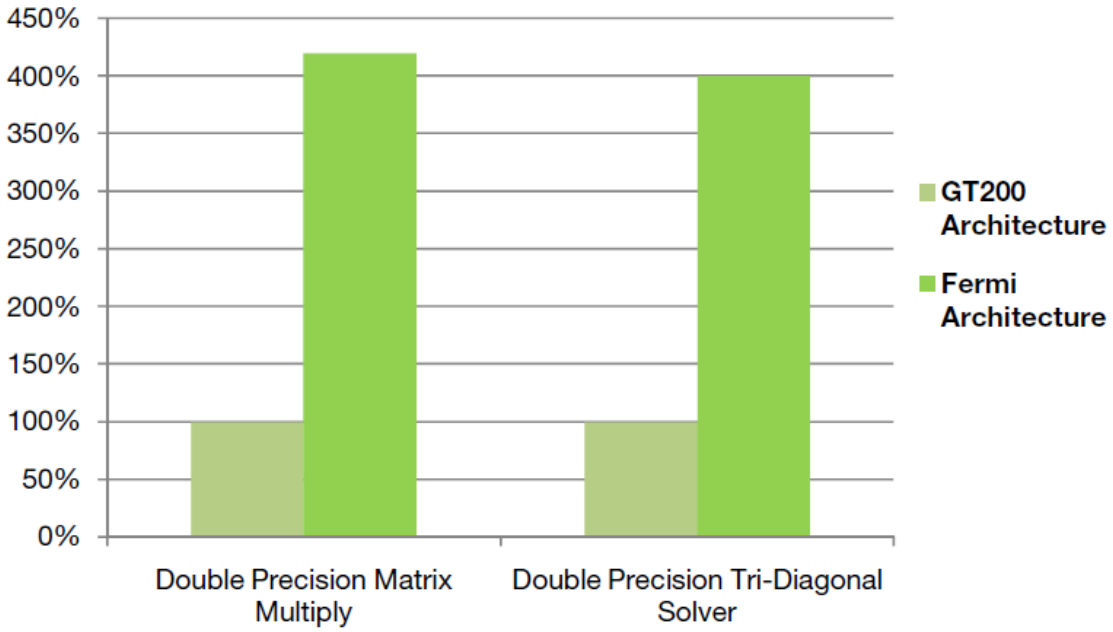
Η *Εικόνα 3-12: Fermi Streaming Multiprocessor*, μας δείχνει μια κάρτα γραφικών αρχιτεκτονικής Fermi με ένα επεξεργαστή ροής 3^{ης} γενιάς. Κάθε επεξεργαστής ροής έχει 32 CUDA επεξεργαστές. Κάθε επεξεργαστής έχει μια αριθμητική λογική μονάδα ακεραίων(ALU) και μια μονάδα κινητής υποδιαστολής(FPU). Στης κάρτες γραφικών GT200, η αριθμητική λογική μονάδα ALU είχε όριο τα 24-bit ακρίβειας για πράξεις πολλαπλασιασμού. Στην αρχιτεκτονική Fermi, οι καινούργιες αριθμητικές λογικές μονάδες υποστηρίζουν 32-bit ακρίβειας εντολές, που είναι συμβατές με τις περισσότερες γλώσσες προγραμματισμού. Η αριθμητική λογική μονάδα ακεραίων είναι επίσης βελτιστοποιημένη για να υποστηρίζει 64-bit πράξεις.

Κάθε SM έχει 16 μονάδες φόρτωσης και αποθήκευσης, επιτρέποντας να υπολογίζονται πηγαίες διευθύνσεις και διευθύνσεις προορισμού για 16 νήματα ανά κύκλο ρολογιού. Επίσης, υποστηρικτικές μονάδες αποθηκεύουν ή φορτώνουν τα δεδομένα που προορίζονται για την κάθε διεύθυνση στην cache ή στην DRAM. Σε κάθε SM υπάρχουν και 4 μονάδες ειδικών συναρτήσεων(Special Function Units – SFU) που εκτελούν πράξει όπως το sin, cos και τετραγωνικής ρίζας. Κάθε εκτελεί μια εντολή ανά νήμα και ανά κύκλο ρολογιού. Ένα στημόνι εκτελείται πάνω σε 8 κύκλους ρολογιού.



Εικόνα 3-12: Fermi Streaming Multiprocessor

Οι επεξεργαστές ροής τρίτης γενιάς είναι ειδικά σχεδιασμένη για να βελτιώσουν τις επιδόσεις σε πράξεις διπλής ακρίβειας, τις οποίες απαιτούν πολλές εφαρμογές. Μπορούν να υποστηριχτούν μέχρι και 16 πράξεις πρόσθεσης αφαίρεσης διπλής ακρίβειας ανά SM και ανά κύκλο ρολογιού. Σε σχέση με τον προκάτοχο τους GT200 έχουμε μια αρκετά μεγάλη αύξηση στις επιδόσεις.



Εικόνα 3-13: Σύγκριση πράξεων διπλής ακρίβειας

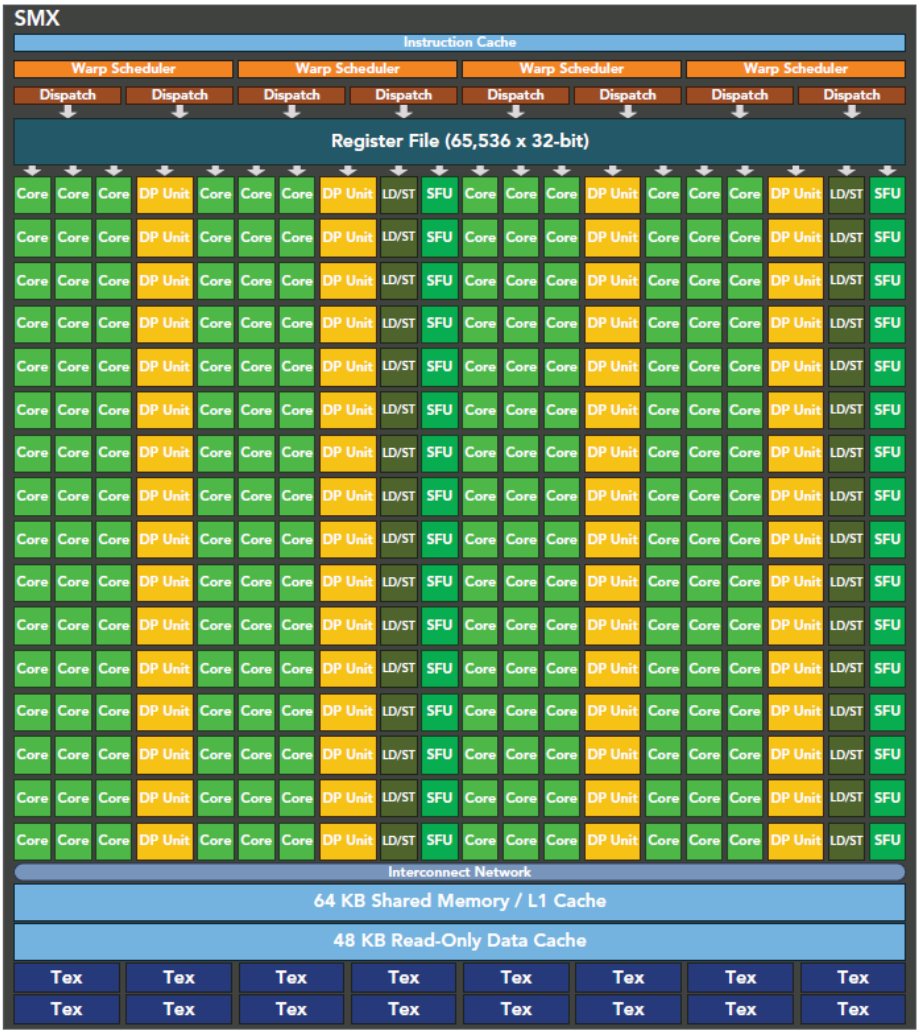
3.2.3 Αρχιτεκτονική Kepler

Η αρχιτεκτονική Kepler GPU, που κυκλοφόρησε το φθινόπωρο του 2012, είναι μια γρήγορη, αποδοτική, υψηλής απόδοσης αρχιτεκτονική υπολογιστών. Η Εικόνα 3-14: Αρχιτεκτονική Kepler απεικονίζει το μπλοκ διάγραμμα του μικροτσίπ Kepler K20X, που περιέχει 15 πολυεπεξεργαστές ροής(SMs) και έξι ελεγκτές μνήμης 64-bit. Οι τρεις σημαντικότερες καινοτομίες στην αρχιτεκτονική Kepler είναι:

- Ενισχυμένα SMs.
- Δυναμικός παραλληλισμός.
- Hyper-Q

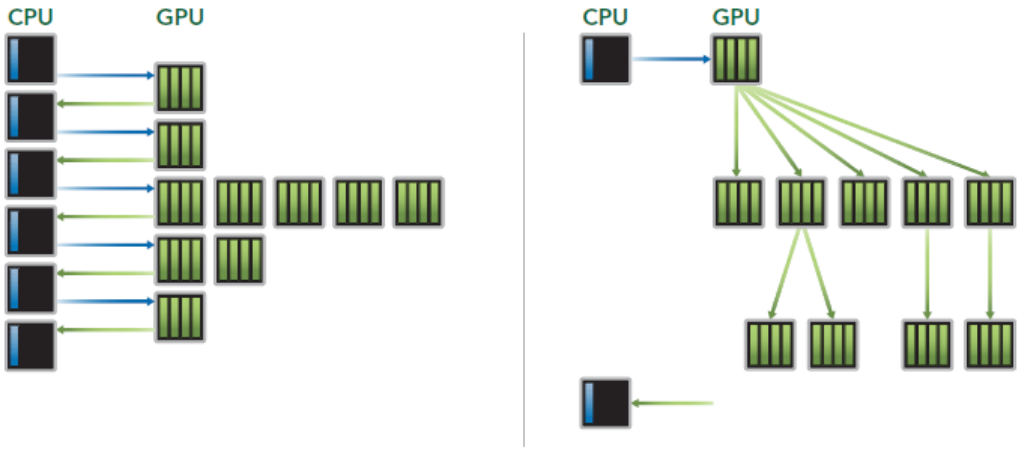
Εικόνα 3-14: Αρχιτεκτονική Kepler

Στην καρδιά της Kepler βρίσκεται ένας καινούργιος επεξεργαστής ροής, οποίος συνδυάζει διάφορες καινοτομίες για την βελτίωση του προγραμματισμού και της εξοικονόμησης ενέργειας. Κάθε επεξεργαστής συνεχούς ροής στην αρχιτεκτονική Kepler αποτελείται από 192 μονής ακρίβειας CUDA πυρήνες, 64 διπλής ακρίβειας μονάδες, 32 μονάδες ειδικών συναρτήσεων. Κάθε Kepler SM έχει τέσσερις χρονοπρογραμματιστές στημονιών και 8 αποστολείς εντολών, δίνοντας την δυνατότητα σε 4 στημόνια να εκτελεστούν παράλληλα από ένα SM.



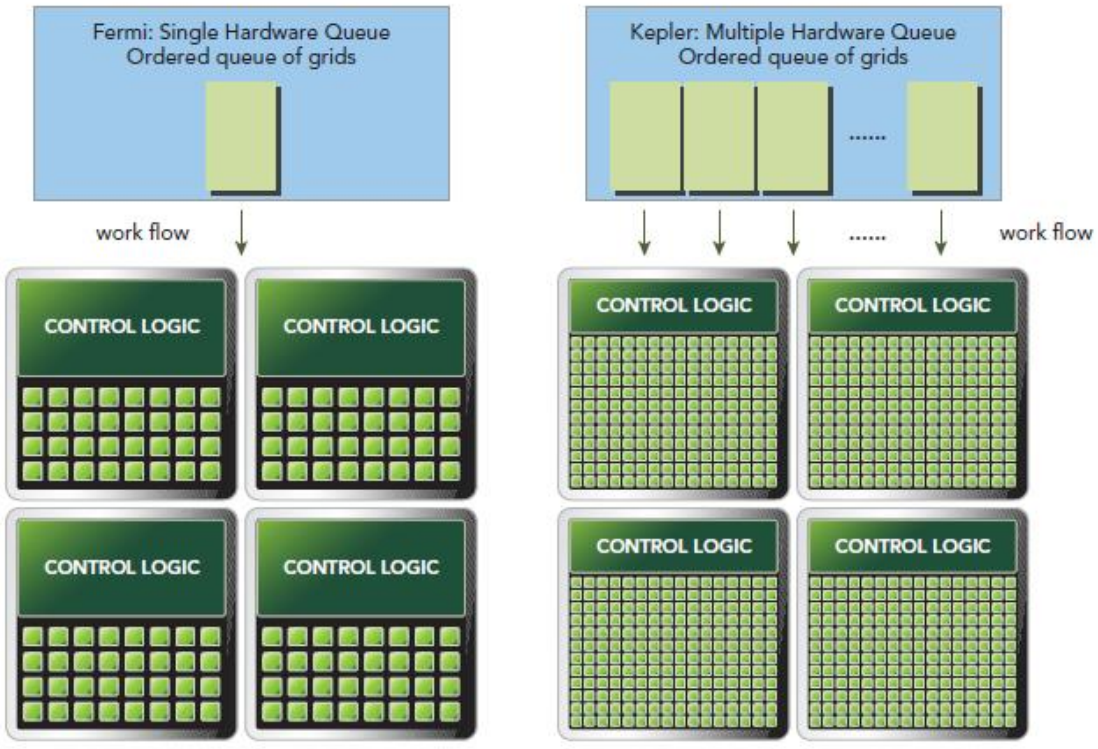
Εικόνα 3-15: Kepler επεξεργαστής συνεχούς ροής

Ο δυναμικός παραλληλισμός είναι μία καινούργια λειτουργία που εισήχθη στην αρχιτεκτονική Kepler και δίνει την δυνατότητα για την δημιουργία νέων πλεγμάτων δυναμικά. Με αυτή την λειτουργία κάθε πυρήνας μπορεί να δημιουργήσει ένα καινούργιο πλέγμα. Όπως φαίνεται στο παρακάτω σχήμα χωρίς τον δυναμικό παραλληλισμό, ο κάθε πυρήνας πρέπει να καλεστεί από την κεντρική μονάδα. Με τον δυναμικό παραλληλισμό μπορεί ένα πυρήνας να καλέσει ένα άλλο εμφωλευμένο πυρήνα, περιορίζοντας έτσι την επικοινωνία με την κεντρική μονάδα.



Εικόνα 3-16: Δυναμικός παραλληλισμός

Η λειτουργία Hyper-Q προσθέτει παραπάνω παράλληλες συνδέσεις μεταξύ της CPU και της GPU, έτσι οι πυρήνες της CPU μπορούν να τρέχουν περισσότερες διεργασίες στην GPU. Η GPU αξιοποιείται καλύτερα και έχουμε μειωμένους χρόνου που η CPU είναι ανενεργή. Η αρχιτεκτονική Fermi βασίζεται σε μία μοναδική ουρά για να στέλνονται οι διεργασίες από την CPU στην GPU, το οποίο έχει σαν αποτέλεσμα μια βαριά διεργασία να καθυστερεί όλες τις υπόλοιπες διεργασίες που βρίσκονται στην ουρά πίσω από αυτήν. Η αρχιτεκτονική Kepler αφαιρεί αυτό το πρόβλημα και παρέχει 32 ουρές υλικού μεταξύ κεντρική μονάδας και κάρτας γραφικών.



Εικόνα 3-17: Hyper-Q

3.3 Διατάξεις υλικού των συστημάτων CUDA

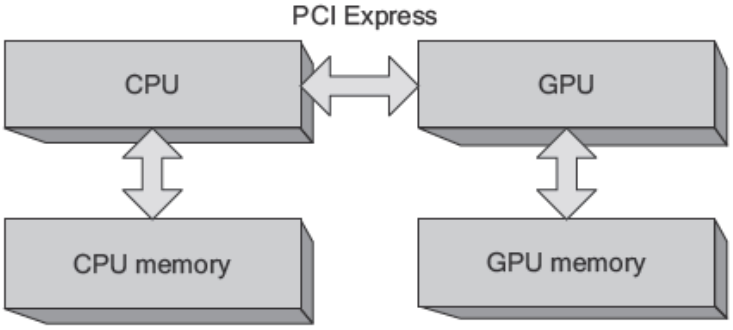
Σε αυτό το κεφάλαιο περιγράφουμε την τοπολογία διαφόρων παράλληλων υπολογιστικών συστημάτων, από το επίπεδο του συστήματος μέχρι τις λειτουργικές μονάδες μέσα στην κάρτα γραφικών. Παρουσιάζεται η δομή του υπολογιστικού συστήματος από την κεντρική μονάδα μέχρι την κάρτα γραφικών και πως αυτά τα δύο μέρη επικοινωνούν και αλληλοεπιδρούν μεταξύ τους.

3.3.1 CPU Διαμόρφωση

Σε αυτή την ενότητα θα περιγράψουμε διάφορες αρχιτεκτονικές CPU/GPU και πως αυτές επηρεάζουν τον τρόπο που προγραμματίζουμε τα εκάστοτε προγράμματα.

Η πρώτη αρχιτεκτονική που θα εξετάσουμε είναι αυτή που παρουσιάζεται στην Εικόνα 3-18: Απλοποιημένη Αρχιτεκτονική CPU/GPU. Μια σημαντική παράλειψη του σχήματος αυτού είναι, το μικροτσίπ, ή η λογική με την η CPU επικοινωνεί με τα διάφορα εξωτερικά μέρη. Οποιοδήποτε στοιχείο εξόδου ή εισόδου, όπως δίσκοι, ελεγκτές δικτιού, συσκευές, μέχρι και η κάρτα γραφικών περνάνε από αυτό το τσιπ. Μέχρι πρόσφατα, αυτά τα τσιπ ήταν χωρισμένα στο κομμάτι “Southbridge”, που συνδέει τις διάφορες περιφερειακές συσκευές, και το κομμάτι “Northbridge”, που συνδέει τον δίαυλο γραφικών(PCI Express) και τον ελεγκτή μνήμης που είναι συνδεδεμένος με την μνήμη της CPU.

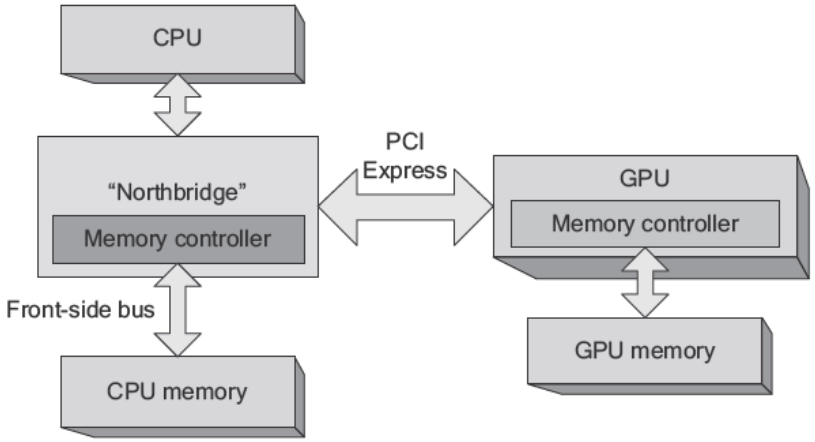
Κάθε γραμμή του PCI Express μπορεί να φτάσει ένα εύρος ζώνης της τάξης των 500MB/s, και ο αριθμός των γραμμών για μια περιφερειακή συσκευή μπορεί να είναι 1,4,8 ή 16. Οι κάρτες γραφικών απαιτούν το μεγαλύτερο εύρος ζώνης από οποιαδήποτε άλλη συσκευή και για αυτό είναι σχεδιασμένες για να τοποθετούνται σε συνδέσεις PCI με 16 γραμμές. Αυτές οι συνδέσεις μπορούν πρακτικά να φτάσουν μέχρι και τα 6G/s.



Εικόνα 3-18: Απλοποιημένη Αρχιτεκτονική CPU/GPU

3.3.2 Μπροστινός Δίαυλος

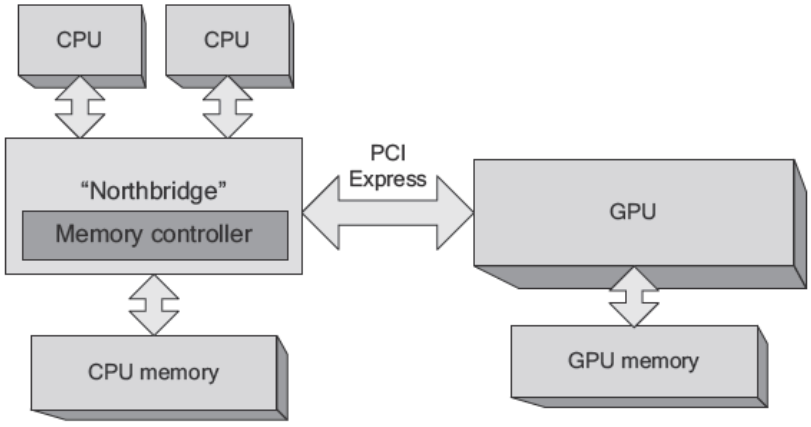
Στο επόμενο σχήμα έχει προστεθεί το κομμάτι “Northbridge” που αναφέρθηκε προηγουμένως και ο ελεγκτής μνήμης. Σε αυτό το σχήμα επίσης έχει προστεθεί και ο ελεγκτής μνήμης της κάρτας γραφικών. Ο ελεγκτής αυτός είναι σχεδιασμένος να λειτουργεί με διαφορετικό τρόπο από ότι ο ελεγκτής μνήμης της CPU. Η GPU θα πρέπει να εξυπηρετεί ισόχρονές κλήσεις, όπως για παράδειγμα την προβολή βίντεο όπου το εύρος ζώνης είναι προκαθορισμένο και αδιαπραγμάτευτο. Ο ελεγκτής μνήμης την κάρτας γραφικών είναι επίσης σχεδιασμένος έτσι ώστε να ανέχονται ένα συγκεκριμένο χρονικό όριο σε καθυστέρηση και να έχουν ένα σχετικά μεγάλο εύρος ζώνης. Οι σημερινές κάρτες γραφικών μπορούν να εξυπηρετήσουν τοπικό εύρος ζώνης στην μνήμη τους ύψους 100G/s.



Εικόνα 3-19: CPU/GPU Αρχιτεκτονική - Northbridge

3.3.3 Συμμετρικοί Πολυεπεξεργαστές

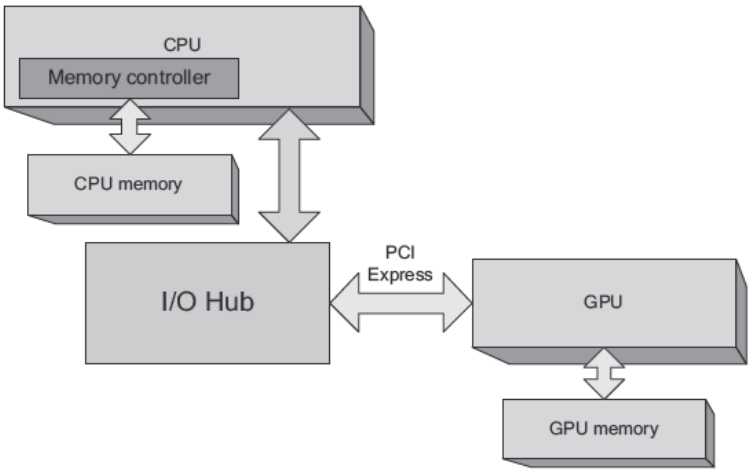
Στην Εικόνα 3-20: Αρχιτεκτονική με πολλαπλές CPU, βλέπουμε ένα σύστημα με πολλαπλές CPU σε μια "Northbridge" αρχιτεκτονική. Πριν τους πολυπύρηνους επεξεργαστές, οι εφαρμογές έπρεπε να χρησιμοποιήσουν πολλαπλά νήματα για να εκμεταλλευτούν την ισχύ των πολλαπλών επεξεργαστών. Η αρχιτεκτονική "Northbridge" θα πρέπει να εξασφαλίσει ότι όλοι οι επεξεργαστές έχουν πρόσβαση σε όλη την μνήμη. Επειδή σε αυτή την αρχιτεκτονική οι επεξεργαστές έχουν πρόσβαση στην μνήμη με τον ίδιο τρόπο έχουμε ομοιόμορφη επίδοση.



Εικόνα 3-20: Αρχιτεκτονική με πολλαπλές CPU

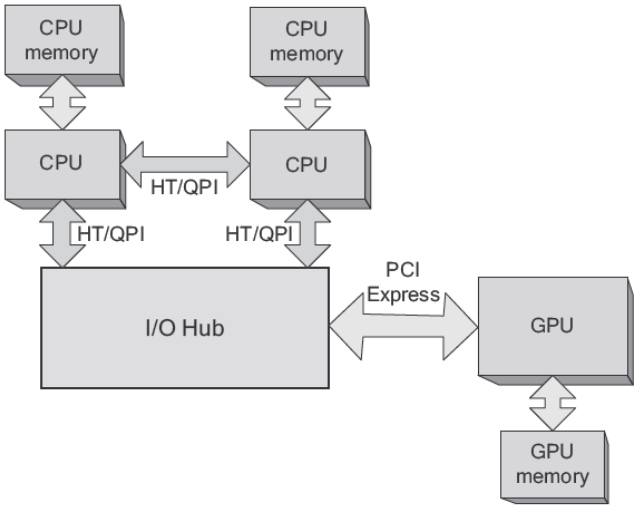
3.3.4 Μη ομοιόμορφη προσπέλαση μνήμης

Η AMD και η Intel με τους επεξεργαστές Opteron και Nehalem αντίστοιχα ενσωματώσανε τον ελεγκτή μνήμης μέσα στον επεξεργαστή όπως φαίνεται στο σχήμα. Η αρχιτεκτονική αυτή βελτιώνει τις επιδόσεις της CPU.



Εικόνα 3-21: CPU με ενσωματωμένο ελεγκτή μνήμης

Η αρχιτεκτονική που φαίνεται στο σχήμα βλέπουμε ότι κάθε CPU έχει την δικιά της μνήμη. Λόγω των πολλών νημάτων που αναπτύσσονται σε αυτά τα συστήματα και της ανάγκης να υπάρχει συνάφεια μεταξύ των μνημών cache και της μνήμης, αναπτύχθηκαν οι τεχνολογίες HyperTransport(HT) και QuickPath Interconnect(QPI). Αυτές οι τεχνολογίες συνδέουν μεταξύ τους τις CPU. Με αυτές τις τεχνολογίες η κάθε CPU έχει πρόσβαση σε οποιοδήποτε κομμάτι της μνήμης, αλλά έχουν πολύ μεγαλύτερη ταχύτητα στις μνήμη που είναι κατευθείαν συνδεδεμένες σε αυτές.

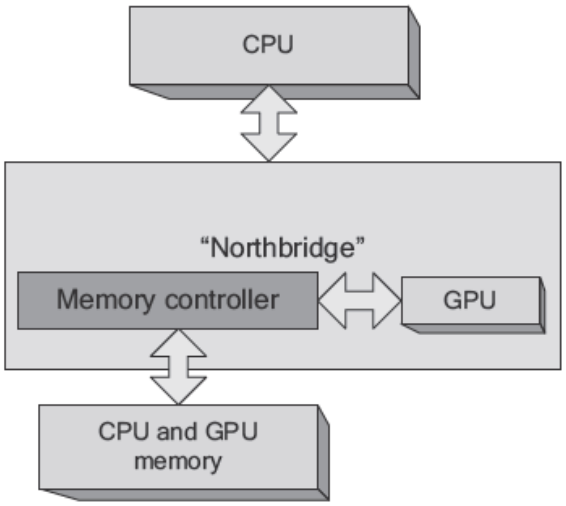


Εικόνα 3-22: Πολλαπλές CPU

3.3.5 Ενσωματωμένες GPUs

Η ενσωματωμένες κάρτα γραφικών βρίσκεται μέσα στο τσιπ. Όπως φαίνεται στο παρακάτω σχήμα η μνήμη που πριν ανήκε μόνο στην CPU, τώρα διαμοιράζεται μεταξύ CPU και GPU. Οι βιβλιοθήκες για την διαχείριση της μνήμης έχουν ειδικό νόημα για τις ενσωματωμένες κάρτες γραφικών. Οι βιβλιοθήκες αυτές, οι οποίες δεσμεύουν μνήμη για να την χρησιμοποιήσουν οι kernel, ονομάζονται και zero-copy γιατί δεν χρειάζεται να αντιγραφούν μέσο του διαύλου αφού αυτή η μνήμη είναι διαμοιραζόμενη. Μια ενσωματωμένη κάρτα γραφικών έχει πολύ καλύτερες επιδόσεις από μια ξεχωριστή κάρτα γραφικών σε ότι να κάνει με διεργασίες μεταφοράς δεδομένων. Οι εγγραφές στην μνήμη έχουν επίσης καλές επιδόσεις στις ενσωματωμένες κάρτες, επειδή οι εγγραφές αυτές χρησιμοποιούν την cache της CPU, αρκεί βέβαια να μην διαβάσει η CPU την μνήμη την ίδια στιγμή.

Η ενσωματωμένες κάρτες γραφικών μπορούν να συνυπάρξουν με αυτόνομες κάρτες γραφικών. Σε αυτά τα συστήματα η CUDA προτιμάει να τρέχει πάνω στην ξεχωριστή κάρτα γραφικών, έτσι ένα πρόγραμμα που είναι γραμμένο για μία κάρτα γραφικών θα τρέξει πάνω στην ξεχωριστή κάρτα γραφικών.

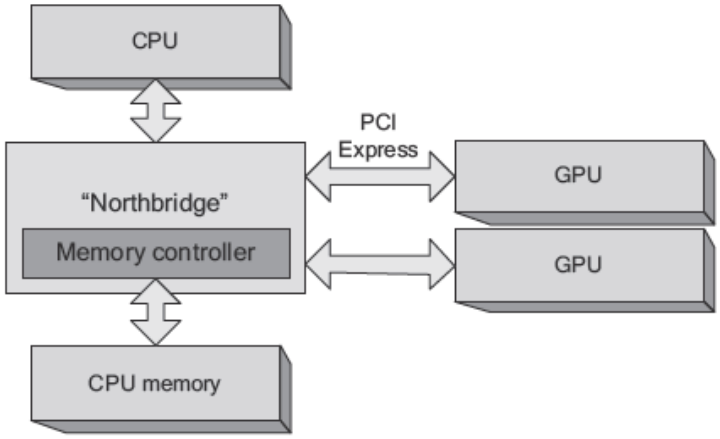


Εικόνα 3-23: Ενσωματωμένη Κάρτα Γραφικών

3.3.6 Πολλαπλές Κάρτες Γραφικών

Σε αυτήν την ενότητα τα περιγράψουμε τους διαφορετικούς τρόπους με τους οποίους μπορούν πολλές κάρτες γραφικών να εγκατασταθούν πάνω σε ένα σύστημα. Στα παρακάτω σχήματα θα παραλείψουμε την μνήμη της κάρτας γραφικών και εννοείται ότι η κάθε κάρτα έχει την δικιά της μνήμη.

Η NVIDIA το 2004 εισήγαγε μια νέα τεχνολογία (SLI-Scalable Link Interface) που επιτρέπει σε πολλές κάρτες γραφικών να λειτουργήσουν παράλληλα και να παρέχουν μεγάλες επιδόσεις. Σε μητρικές για παράδειγμα που επιτρέπουν την εγκατάσταση δύο καρτών, μπορεί ο χρήστης να εγκαταστήσει δύο κάρτες και να διπλασιάσει τις επιδόσεις του συστήματος. Οι κάρτες αυτές, μέσο των τεχνολογιών τις NVIDIA, λειτουργούν σαν να είχαμε μια κάρτα γραφικών με τις διπλάσιες επιδόσεις.

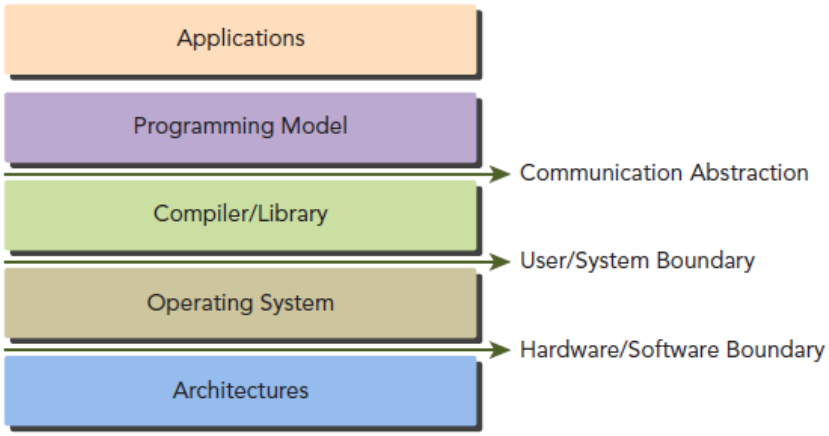


Εικόνα 3-24: Πολλαπλές Κάρτες Γραφικών

4. Προγραμματιστικό μοντέλο CUDA

Η CUDA είναι μια παράλληλη υπολογιστική πλατφόρμα και ένα προγραμματιστικό μοντέλο με ένα μικρό σύνολο επεκτάσεων πάνω στην γλώσσα C. Με την CUDA μπορούμε να υλοποιήσουμε παράλληλους αλγορίθμους χρησιμοποιώντας την γλώσσα C. Σε αυτό το κεφάλαιο θα αναλύσουμε τα κύρια χαρακτηριστική του προγραμματισμού σε CUDA. Θα παρουσιάσουμε τα βασικά μέρη ενός προγράμματος που είναι γραμμένο σε CUDA και θα δώσουμε μια συνοπτική περιγραφή τους.

Ένα προγραμματιστικό μοντέλο είναι μια αφαιρετική έννοια που σκοπός είναι να αποτελέσει την γέφυρα μιας εφαρμογής και του υλικού που θα υλοποιήσει αυτή την εφαρμογή(Εικόνα 4-1: Προγραμματιστικό Μοντέλο). Το σημείο επικοινωνίας είναι το σημείο στο οποίο έχει υλοποιηθεί το πρόγραμμα μέσω του προγραμματιστικού μοντέλου και αναλαμβάνει ο μεταγλωττιστής μαζί με τις βιβλιοθήκες. Το προγραμματιστικό μοντέλο, καθορίζει το πως τα διάφορα μέρη του προγράμματος επικοινωνούν μεταξύ τους και συντονίζουν τις διεργασίες τους. Ουσιαστικά μας παρέχει μια λογική άποψη της κάθε υπολογιστικής αρχιτεκτονικής.



Εικόνα 4-1: Προγραμματιστικό Μοντέλο

Η CUDA μας παρέχει πολλές λειτουργίες που είναι κοινές σε πολλά παράλληλα προγραμματιστικά μοντέλα. Εκτός όμως από αυτά μας παρέχει και συγκεκριμένα χαρακτηριστικά για την εκμετάλλευση της υπολογιστικής ικανότητας των καρτών γραφικών. Το προγραμματιστικό μοντέλο της CUDA μας παρέχει:

- Μια μέθοδο για να οργανώσουμε τα νήματα της GPU μέσα από μια ιεραρχική δομή.
- Μια μέθοδο πρόσβασης της μνήμης της GPU μέσα από μια ιεραρχική δομή.

4.1 Δομή προγραμματιστικού μοντέλου CUDA

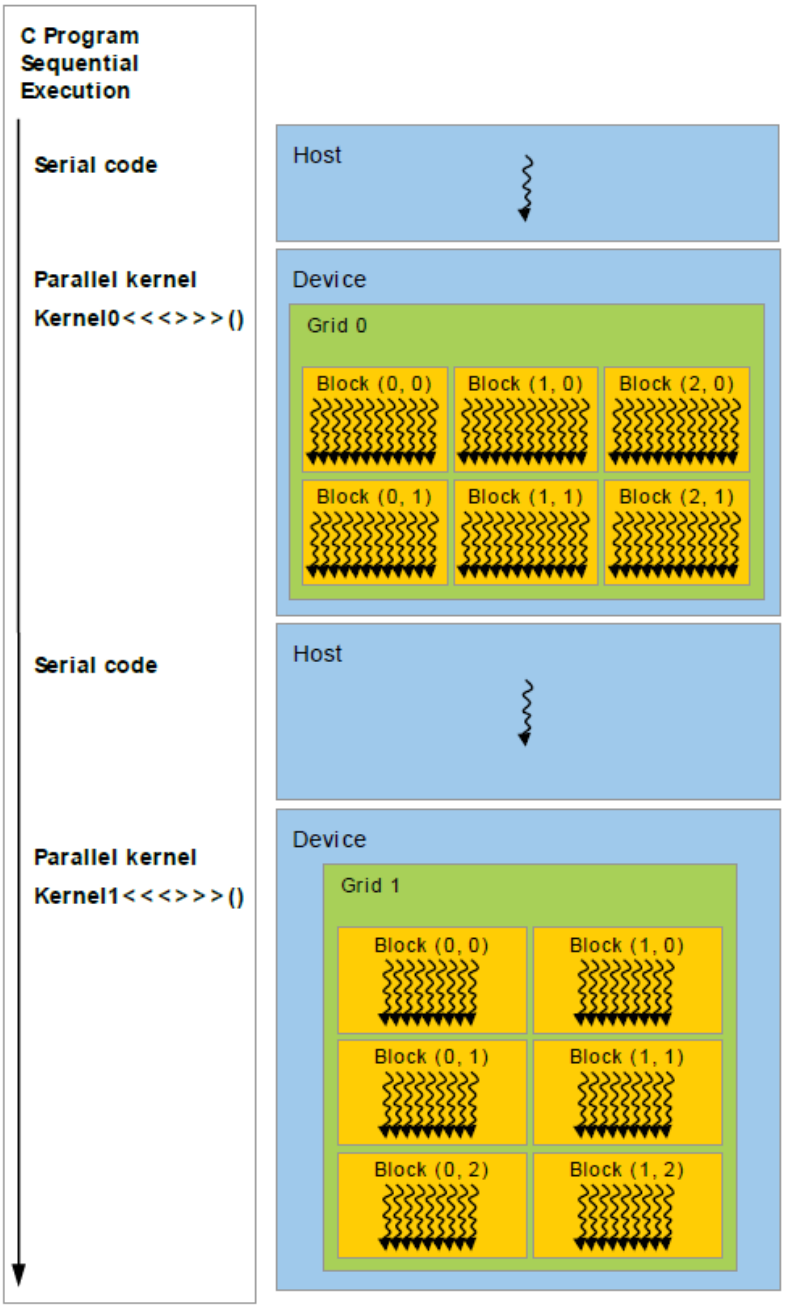
Το προγραμματιστικό μοντέλο της CUDA μας δίνει την δυνατότητα να εκτελέσουμε εφαρμογές πάνω σε ετερογενή συστήματα, απλώς επισημαίνοντας κάποια κομμάτια του κώδικα με κάποιες λεξιλογικές επεκτάσεις της γλώσσας C. Ένα ετερογενές υπολογιστικό περιβάλλον αποτελείται από κάποιες CPU μαζί με ένα αριθμό GPU, όπου το κάθε στοιχείο έχει την δικιά του μνήμη και συνδέονται μεταξύ τους μέσω ενός διαύλου PCI-Express. Έτσι, σε ένα ετερογενές σύστημα μπορούμε να ξεχωρίσουμε τα εξής μέρη:

- Κεντρική Μονάδα: η CPU μαζί με την μνήμη της.
- Συσκευή: η κάρτα γραφικών μαζί με την δικιά της μνήμη.

Η κεντρική μονάδα μπορεί να λειτουργεί ανεξάρτητα από την συσκευή. Όταν καλείται ένας πυρήνας, που είναι το κομμάτι κώδικα που εκτελείται από την συσκευή, ο έλεγχος επιστρέφει στην CPU και μπορεί να εκτελέσει περαιτέρω διεργασίες, ενώ παράλληλα ο εκτελείται το κομμάτι του προγράμματος που αναλογεί στην κάρτα γραφικών. Το προγραμματιστικό μοντέλο της CUDA είναι κατά κύριο λόγω ασύγχρονο. Ένα τυπικό πρόγραμμα της CUDA αποτελείται από το σειριακό κομμάτι που θα εκτελεστεί στην κεντρική μονάδα και από το παράλληλο κομμάτι που θα εκτελεστεί στην κάρτα γραφικών. Το σειριακό κομμάτι γράφεται σε γλώσσα ANSI C και το παράλληλο κομμάτι σε CUDA C.

Ένα τυπικό πρόγραμμα της CUDA ακολουθεί συνήθως την παρακάτω ροή:

1. Αντιγραφεί των δεδομένων από την μνήμη της κεντρικής μονάδας στην μνήμη της κάρτας γραφικών.
2. Εκτέλεση των πυρήνων που θα εκτελέσουν διεργασίες πάνω στα δεδομένα που είναι αποθηκευμένα στην μνήμη της κάρτας γραφικών.
3. Αντιγραφή δεδομένων-αποτελεσμάτων από την κάρτα γραφικών στην κεντρική μονάδα.



Εικόνα 4-2: Τυπική ροή ενός προγράμματος CUDA

4.1.1 Υπολογιστικός πυρήνας(Kernel)

Το βασικότερο μέρος ενός προγράμματος CUDA είναι ο πυρήνας(Kernel). Στην συνάρτηση του πυρήνα καθορίζουμε τους υπολογισμούς που θα εκτελέσει ένα νήμα. Όταν εκτελεστεί ο πυρήνας πολλά

νήματα θα εκτελέσουν τους ίδιους υπολογισμούς που έχουμε ορίσει μέσα στον πυρήνα. Ο πυρήνας είναι μια συνάρτηση που δηλώνεται μέσο του λεκτικού "__global__" και είναι το κομμάτι κώδικα που θα εκτελεστεί από την κάρτα γραφικών:

```
__global__ void kernel_name (argument list);
```

Το λεκτικό *global* ειδοποιεί τον compiler, στην περίπτωση της CUDA τον nvcc, ότι αυτό το κομμάτι θα τρέξει πάνω στην κάρτα γραφικών. Έτσι, αυτό το κομμάτι θα εκτελεστεί από τα x νήματα που θα τρέξουν στην κάρτα. Η συνάρτηση αυτή καλείται από την CPU και μέσα σε αυτή την κλήση περιλαμβάνονται παράμετροι, όπως ο αριθμός των νημάτων που θα τρέξουν και ο αριθμός των μπλοκ.

```
kernel_name<<<dimGrid, dimBlock>>> (argument list);
```

Για να μπορούμε να ξεχωρίσουμε τα νήματα που τρέχουν στον πυρήνα, το κάθε νήμα έχει τη δικιά του ταυτότητα(threadIdx) και ανήκει σε ένα συγκεκριμένο μπλοκ νημάτων που έχει επίσης την δικιά του μοναδική ταυτότητα(blockIdx). Τα νήματα μπορούν να κατανεμηθούν ανάλογα με την διαδικασία σε 1,2 ή 3 διαστάσεις. Η επιλογή που κάνουμε εξαρτάται από την διεργασία που θέλουμε να εκτελέσουμε και πως θέλουμε να ταξινομήσουμε στο χώρο. Για παράδειγμα αν θέλουμε να επεξεργαστούμε μια εικόνα θα μπορούσαμε να το χωρίσουμε σε δυο διαστάσεις έτσι ώστε το κάθε νήμα να επεξεργάζεται ένα pixel της εικόνας. Με τον ίδιο τρόπο χωρίζονται και τα block.

Αναγνωριστικό	Εκτελείται	Καλείται
__device__	Συσκευή	Συσκευή
__global__	Συσκευή	Κεντρική Μονάδα
__host__	Κεντρική Μονάδα	Κεντρική Μονάδα

Πίνακας 1: CUDA επεκτάσεις στις δηλώσεις συναρτήσεων C

Οι υπολογιστικοί πυρήνες της CUDA είναι σαν τις συναρτήσεις της C, αλλά έχουν τους παρακάτω περιορισμούς:

- Έχουν πρόσβαση μόνο στην μνήμη της κάρτας γραφικών.
- Η τιμή που επιστρέφουν πρέπει να είναι *void*.
- Δεν υποστηρίζουν στατικές μεταβλητές.
- Δεν υποστηρίζουν δείκτες συναρτήσεων.
- Έχουν ασύγχρονη συμπεριφορά.

4.1.2 Οργάνωση των νημάτων

Όταν καλείται η συνάρτηση ενός πυρήνα από την κεντρική μονάδα, η εκτέλεση μετακινείται στην κάρτα γραφικών, όπου ένας μεγάλος αριθμός νημάτων δημιουργείται και κάθε νήμα εκτελεί τις εντολές που περιέχονται μέσα στην συνάρτηση του πυρήνα. Η οργάνωση των νημάτων για την εκτέλεση του προγράμματος είναι πολύ σημαντικό στην προγραμματισμό της CUDA. Το μοντέλο της CUDA διαχωρίζει ιεραρχικά τα νήματα για να μας δώσει την δυνατότητα να οργανώσουμε καλύτερα τα νήματα. Υπάρχει λοιπόν μια διπλού επιπέδου ιεραρχία που αποτελείται από τα μπλοκ των νημάτων και το πλέγμα των μπλοκ.

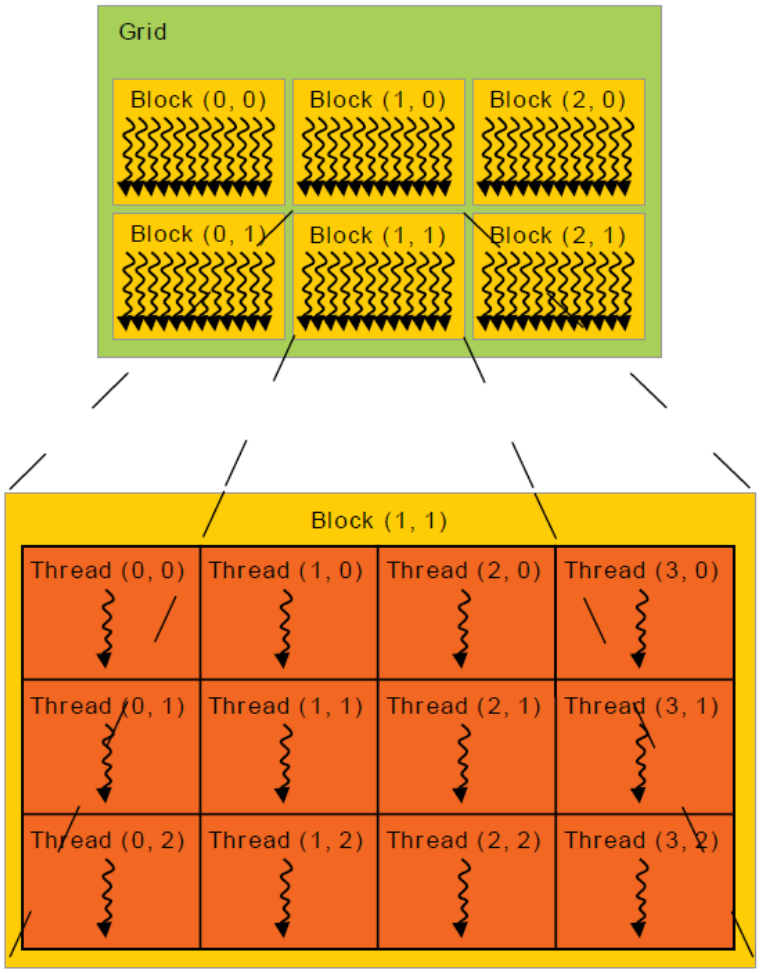
Από την πλευρά του προγραμματιστή, τα βήματα για τον καθορισμό του μεγέθους του πλέγματος και των μπλοκ είναι τα ακόλουθα:

- Να καθορίσει το μέγεθος των μπλοκ, δηλαδή τα νήματα που θα περιέχονται σε κάθε μπλοκ.
- Να υπολογίσει το μέγεθος του πλέγματος, ανάλογα με των όγκο των δεδομένων της εφαρμογής και το μέγεθος των μπλοκ.

Για τον καθορισμό των διαστάσεων του μπλοκ θα πρέπει να λάβουμε υπόψη μας τις παρακάτω παραμέτρους:

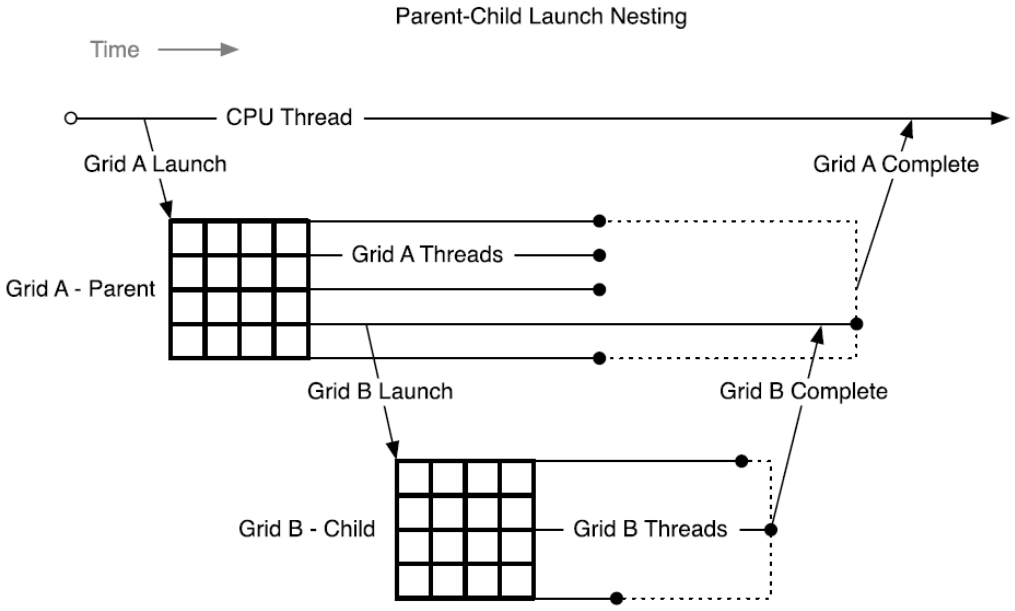
- Τα χαρακτηριστική του πυρήνα για την μεγιστοποίηση των επιδόσεων.
- Τους περιορισμούς στους πόρους της κάρτας γραφικών.
- Η δομή των δεδομένων του προβλήματος και οι υπολογισμοί που θα πραγματοποιηθούν πάνω σε αυτά.

Σε ένα μπλοκ μπορούν να τρέξουν ένας συγκεκριμένος αριθμός νημάτων. Τα νήματα ενός μπλοκ μπορούν να μοιραστούν την ίδια Shared Memory. Στις σύγχρονες κάρτες γραφικών ο μέγιστος αριθμός νημάτων που μπορούν να τρέξουν σε ένα μπλοκ είναι 1512. Ο πυρήνας όμως, μπορεί να τρέξει πολλά μπλοκ παράλληλα έτσι ο αριθμός των συνολικών νημάτων είναι ο αριθμός των νημάτων που περιέχονται σε ένα μπλοκ επί των αριθμό των μπλοκ. Στην Εικόνα 4-3: Δομή των νημάτων φαίνεται η δομή των νημάτων, των μπλοκ και του πλέγματος.



Εικόνα 4-3: Δομή των νημάτων

Ο αριθμός των νημάτων και των μπλοκ, όπως και οι διαστάσεις, που θα τρέξουν καθορίζεται από τον προγραμματιστή όταν καλέσει από την CPU τον πυρήνα που θα τρέξει πάνω στην κάρτα γραφικών. Όλα τα νήματα και τα μπλοκ αποτελούν ένα πλέγμα(Grid). Ένα πρόγραμμα μπορεί να αποτελείται από πολλά πλέγματα τα οποία ουσιαστικά δημιουργούνται από διαφορετικού πυρήνες. Ένα νήμα μπορεί επίσης να καλέσει ένα καινούργιο πλέγμα. Αυτό το πλέγμα ανήκει στο πλέγμα στο οποίο το κάλεσε(parent-grind) και το πλέγμα αυτό το οποίο δημιουργήθηκε είναι ένα πλέγμα παιδί(child-grid). Όλα τα child-grid είναι εμφωλευμένα μέσα στο parent-grid πράγμα το οποίο σημαίνει, ότι το parent-grid δεν θεωρείται ότι έχει τελειώσει όλες τις διεργασίες του, αν όλα τα child-grid δεν έχουν ολοκληρωθεί. Στο παρακάτω σχήμα παρουσιάζεται η ροή αυτή που ξεκινάει από το αρχικό νήμα την CPU.



Εικόνα 4-4: Ροή των πλεγμάτων

Ένα άλλο σημαντικό κομμάτι του προγράμματος είναι η μνήμη που θα χρησιμοποιήσει η κάρτα γραφικών για να τρέξει το οποιοδήποτε πρόγραμμα. Όπως αναφέρθηκε, το μοντέλο προγραμματισμού της CUDA προϋποθέτει ότι το σύστημα μας αποτελείται από την κεντρική μονάδα και την κάρτα γραφικών, όπου το καθένα έχει την δικιά του ξεχωριστή μνήμη. Για έχουμε πλήρη έλεγχο της μνήμης της κάρτας γραφικών η CUDA μας εξοπλίζει με ένα σύνολο συναρτήσεων που πραγματοποιούνε εντολές δέσμευσης της μνήμης. Στον Η μνήμη που δεσμεύεται για να τρέξουμε ένα πρόγραμμα στη κάρτα γραφικών γίνεται από το κομμάτι του κώδικα που τρέχει στην CPU με την εντολή cudaMalloc(). Έτσι τα δεδομένα που θα χρησιμοποιηθούν σε κάθε πρόγραμμα πρώτα φορτώνονται από την κεντρική μονάδα, στην συνέχεια γίνεται δέσμευση ενός μέρος της μνήμης στην κάρτα γραφικών και τέλος μεταφέρονται τα δεδομένα από την κεντρική μονάδα στην κάρτα γραφικών μέσο την εντολής cudaMemcpyHostToDevice(). Μετά την ολοκλήρωση των υπολογισμών μεταφέρουμε τα αποτελέσματα από την κάρτα στην κεντρική μονάδα μέσο την εντολής cudaMemcpyDeviceToHost().

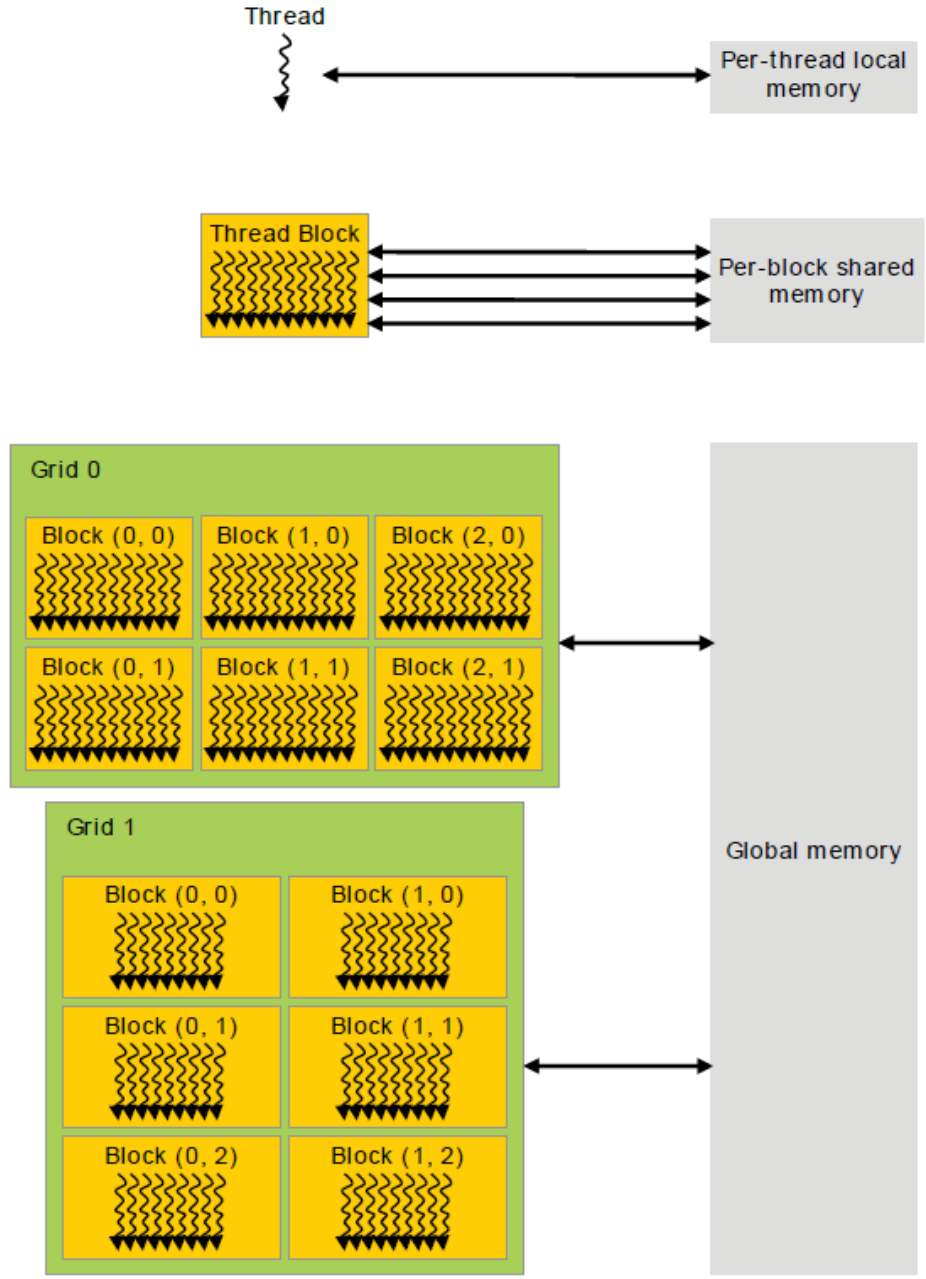
Συνάρτηση C	Συνάρτηση CUDA
malloc	cudaMalloc
memcpy	cudaMemcpy
memset	cudaMemset
free	cudaFree

Πίνακας 2: Συναρτήσει δέσμευσης μνήμης C - CUDA

4.2 Ιεραρχία Μνήμης

Τα νήματα της CUDA έχουν πρόσβαση σε διάφορα μέρη της μνήμης όπως φαίνεται στο παρακάτω σχήμα. Κάθε νήμα έχει διαθέσιμη την δικιά του τοπική μνήμη(local memory). Αυτό δεν σημαίνει αναγκαία ότι αυτή η μνήμη του νήματος βρίσκεται και υλικά μέσα στο SM. Για παράδειγμα, αν έχουμε ένα πολύ μεγάλο πίνακα τον, ο οποίος δηλώνεται από ένα νήμα και δεν χωράει μέσα στους καταχωρητές, τότε αυτό ο πίνακας θα αποθηκευτεί πάνω στην global memory, αλλά πρόσβαση σε αυτόν θα έχει μόνο το νήμα. Κάθε μπλοκ νημάτων έχει πρόσβαση στην κοινόχρηστη μνήμη(shared memory) η οποία υπάρχει όσο τρέχει το μπλοκ. Όλα τα νήματα έχουν πρόσβαση στην global memory.

Υπάρχουν επίσης και δύο ακόμα είδη μνήμης read-only: texture memory και constant memory. Αυτά τα δυο είδη μνήμης βρίσκονται πάνω στην global memory, αλλά είναι βελτιστοποιημένες για συγκεκριμένες χρήσης της μνήμης.



Εικόνα 4-5: Ιεραρχία Μνήμης

4.3 Το μοντέλο εκτέλεσης CUDA

Η δύναμη της επίδοσης και της επεκτασιμότητας της CUDA, βασίζεται στον διαχωρισμό της εκτέλεσης μίας διεργασίας σε ένα ορισμένο πλήθος νημάτων, που ομαδοποιούνται σε έναν αριθμό μπλοκ. Αυτά τα μπλοκ νημάτων αποτελούν την χαρτογράφηση μεταξύ της παραλληλίας της εφαρμογής και την παράλληλη εκτέλεση των διεργασιών από το υλικό της κάρτας γραφικών. Η αύξηση των επιδόσεων και η μεγιστοποίηση της παραλληλίας μπορεί να γίνει με την αύξηση των επεξεργαστών συνεχούς ροής(SMs).

Μεταβάλλοντας τον αριθμό των SMs, οι κατασκευαστές καρτών μπορούν να παράγουν προϊόντα με ανάλογες επιδόσεις και κόστος.

Ο ορισμός των μπλοκ νημάτων, από πλευράς λογισμικού, αντιστοιχίζεται στο υλικό, στην εκτέλεση ενός kernel από έναν αριθμό SMs. Επίσης, ένα μπλοκ λειτουργεί και σαν ένα πλαίσιο μέσα στο οποίο τα νήματα μπορούν να συνεργαστούν, αφού μόνο μέσα σε αυτό μπορούν τα νήματα να μοιραστούν δεδομένα. Έτσι, τα μπλοκ αποτελούν την έκφραση της παραλληλίας μιας εφαρμογής και του διαμερισμού της παραλληλίας, όπου το κάθε μέρος λειτουργεί ανεξάρτητα από τα άλλα.

Η αντιστοίχισή του μπλοκ πάνω στην κάρτα είναι σχετικά απλή. Κάθε επεξεργαστής συνεχούς ροής θα τρέξει ένα ή παραπάνω μπλοκ. Έτσι, η αντιστοίχιση των μπλοκ:

- Επεκτείνεται σε οποιοδήποτε αριθμό SMs.
- Δεν περιορίζεται χωρικά από την θέση του SM, δηλαδή η CUDA μας δίνει την δυνατότητα να τρέξουμε μια εφαρμογή σε πολλές συσκευές χωρίς καμία αλλαγή.
- Δίνει την δυνατότητα στο υλικό να αναμεταδώσει τον εκτελέσιμο kernel και τις παραμέτρους του χρήστη στο υλικό. Η παράλληλη αναμετάδοση είναι ο πιο επεκτάσιμος και αποδοτικότερος τρόπος επικοινωνίας, για την μετάδοση δεδομένων στα διάφορα μέρη επεξεργασίας.

Επειδή όλα τα νήματα ενός μπλοκ εκτελούνται κάτω από το ίδιο SM, η κάρτα γραφικών μας δίνει την δυνατότητα να κάνουμε χρήση της γρήγορης κοινόχρηστης μνήμης. Αυτή η δομή μας δίνει την δυνατότητα να αποφύγουμε τον περιορισμό της ομοιόμορφης cache σε πολλούς επεξεργαστές. Η ομοιόμορφη cache εγγυάται την αξιοπιστία μιας μεταβλητής, όταν αυτήν μπορούν να την μεταβάλουν πολλές μονάδες επεξεργασίας.

4.4 Χρονοπρογραμματισμός Νημάτων

Ο διαμοιρασμός των διεργασιών στους επεξεργαστές συνεχούς ροής πραγματοποιείται από ένα υπερνήμα(Giga-Thread) χρονοπρογραμματισμού. Το νήμα αυτό, βασισμένο πάνω στις παραμέτρους που παρέχονται από τον προγραμματιστή, δηλαδή των αριθμό των μπλοκ και των αριθμό των νημάτων, διαμοιράζει ένα ή παραπάνω μπλοκ σε κάθε επεξεργαστή. Το πόσα μπλοκ ανατίθενται σε κάθε SM και το πόσα νήματα μπορεί να υποστηρίξει εξαρτάται από το SM.

Η NVIDIA κατηγοριοποιεί τις κάρτες γραφικών ανάλογα με την υπολογιστική ικανότητα. Ένα χαρακτηριστικό που καθορίζει την υπολογιστική ικανότητα είναι ο μέγιστος αριθμός νημάτων που μπορούν να τρέξουν κάτω από ένα SM. Για παράδειγμα, μια κάρτα πρώτης γενιάς της CUDA με υπολογιστική ικανότητα 1.0 έχει την δυνατότητα να διαχειριστεί 768 νήματα ανά επεξεργαστή. Υπό την σκέπη ενός SM θα μπορούσαν να εξυπηρετηθούν οι εξής συνδυασμοί νημάτων:

- 1 μπλοκ των 768 νημάτων
- 3 μπλοκ των 256 νημάτων
- 6 μπλοκ των 128 νημάτων

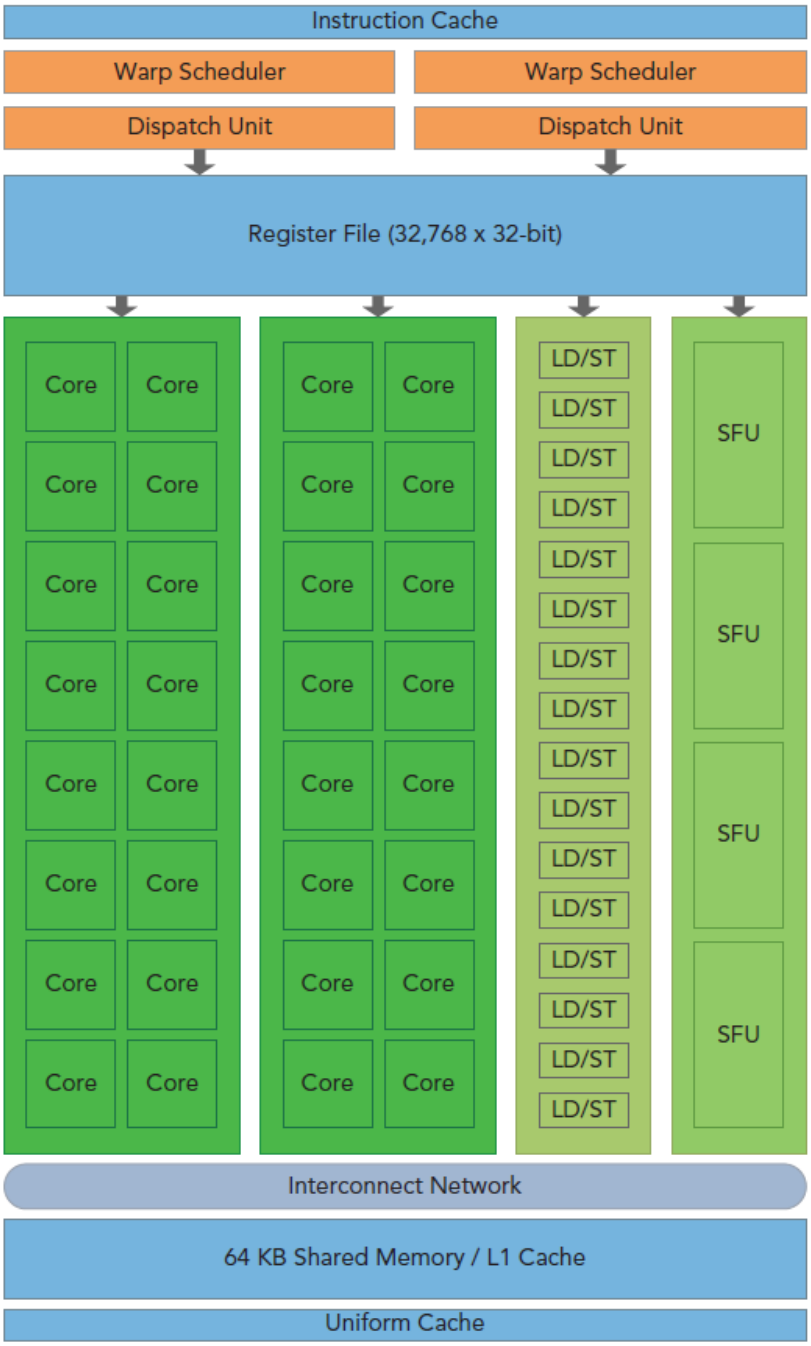
Μια κάρτα με υπολογιστική ικανότητα 2.0 μπορεί να υποστηρίξει 1536 νήματα ανά επεξεργαστή.

Το υπερνήμα χρονοπρογραμματισμού εφαρμόστηκε στις συσκευές υπολογιστικής ικανότητας 2.0, για να υποστηρίξει την εκτέλεση πολλών kernel παράλληλα. Έτσι, οι κάρτες γραφικών με υπολογιστική ικανότητα 2.0 μπορούν να εκμεταλλευτούν καλύτερα το υλικό, όταν έχουν να εκτελέσουν ένα πλήθος από μικρούς kernels. Πολλοί kernels μπορούν να εκτελούνται παράλληλα σε μια κάρτα γραφικών αρκεί να ανήκουν σε διαφορετικές ροές. Οι kernels εκτελούνται με την σειρά που καλούνται αρκεί να υπάρχουν ελεύθερα SMs, αφού πρώτα έχουν δρομολογηθεί όλοι οι προηγούμενοι kenrels.

Ο κάθε επεξεργαστής είναι υπεύθυνος ώστε να χρησιμοποιήσει όσο το δυνατόν καλύτερα τους εσωτερικούς του πόρους, τους πυρήνες και άλλες μονάδες επεξεργασίας, ώστε να φέρει εις πέρας το έργο των νημάτων των μπλοκ που του έχουν ανατεθεί. Αυτή η διαχωρισμένη αλληλεπίδραση, μας δίνει μεγάλη επεκτασιμότητα, αφού το υπερνήμα χρειάζεται μόνο να γνωρίζει πότε ένα SM είναι απασχολημένο. Σε κάθε κύκλο του ρολογιού ο χρονοπρογραμματιστής στημονιού αποφασίζει πιο στημόνι θα εκτελεστεί από αυτά που είναι σε αναμονή, είτε γιατί περιμένανε δεδομένα από την μνήμη είτε γιατί περιμένανε την εκτέλεση προηγούμενων εντολών.

Στην Εικόνα 4-6: Αρχιτεκτονική επεξεργαστή συνεχούς ροής, παρουσιάζεται η δομή ενός επεξεργαστή συνεχούς ροής GF100. Αυτός ο επεξεργαστής διαθέτει 32 πυρήνες SIMD(Single Instruction Multiple Data). Το μοντέλο SIMD σημαίνει ότι όλοι οι πυρήνες θα εκτελέσουν την ίδια εντολή, αλλά θα χρησιμοποιήσουν διαφορετικά δεδομένα. Το πλεονέκτημα των πυρήνων που είναι βασισμένοι στο

μοντέλο SIMD είναι χρειάζονται πολύ λιγότερο χώρο και καταναλώνουν πολύ λιγότερη ενέργεια σε σχέση με τα άλλα μοντέλα. Αυτά τα πλεονεκτήματα πολλαπλασιάζονται με την δυνατότητα χρήσης πολλών επεξεργαστών μέσα σε μια κάρτα γραφικών.



Εικόνα 4-6: Αρχιτεκτονική επεξεργαστή συνεχούς ροής

Τα πλεονεκτήματα του μοντέλου SIMD τα εκμεταλλεύτηκαν οι σχεδιαστές των καρτών γραφικών για να προσθέσουν στον επεξεργαστής αριθμητικές και λογικές μονάδες, μονάδες κινητής υποδιαστολής και μονάδες ειδικών συναρτήσεων. Σαν αποτέλεσμα οι κάρτες γραφικών έχουν μεγαλύτερο λόγο flop per watt σε σχέση με τους συμβατικούς επεξεργαστές.

Μια αρχική ανάλυση του τρόπου προγραμματισμού σε κάρτες γραφικών γίνεται πάνω στην λογική των νημάτων. Μια βαθύτερη όμως ανάλυση πρέπει να γίνεται με βάση των αριθμός των νημάτων SIMD μέσα σε ένα στημόνι. Ένα στημόνι είναι μια ομάδα 32 SIMD νημάτων. Ότι είναι το μπλοκ των νημάτων

για το υπερνήμα, το στημόνι είναι μια παρόμοια μονάδα για την κατανομή της εργασίας μέσα σε ένα SM. Επειδή κάθε στημόνι είναι εξορισμού μια ομάδα νημάτων SIMD, ο χρονοπρογραμμαστιστής δεν χρειάζεται να ελέγξει για εξαρτήσεις μέσα στη ροή των εντολών. Στο διπλανό σχήμα παρατηρούμε ότι ένας SM έχει δύο χρονοπρογραμματιστές στημονιών, το οποίο μας επιτρέπει δυο στημόνια να εκτελούνται παράλληλα. Με αυτό τον τρόπο οι επεξεργαστές συνεχούς ροής μπορούν να πραγματοποιήσουν δύο εντολές ανά κύκλο ρολογιού, διαλέγοντας δύο στημόνια και εκτελώντας μια εντολή από το καθένα σε μια ομάδα 16 πυρήνων, ή σε 16 μονάδες φόρτωση/αποθήκευσης, ή 4 μονάδες ειδικών συναρτήσεων.

5. Ο αλγόριθμος Smith-Waterman

Ο αλγόριθμος Smith-Waterman είναι ένας δυναμικός αλγόριθμος για την εύρεση όμοιων υποακολουθιών μεταξύ ακολουθιών. Χρησιμοποιήθηκε πρώτη φορά για την τοπική στοίχιση δυο ακολουθιών μεταξύ ακολουθιών νουκλεοτιδίων ή πρωτεϊνών. Ο αλγόριθμος αυτός προτάθηκε πρώτη φορά το 1981 από τους Smith και Waterman για την αναγνώριση όμοιων περιοχών μεταξύ ακολουθιών. Για να βρεθεί η τοπική ομοιότητα των δυο ακολουθιών χρησιμοποιείται ένα σύστημα σκορ στο οποίο καθορίζονται τα πέναλτι κενού και το κόστος αντικατάστασης.

Ο εντοπισμός ομοιότητας σε δυο υποακολουθίες δυο ακολουθιών χρησιμοποιήθηκε στο τομέα της βιολογίας για να εντοπίζονται ακολουθίες οι οποίες έχουν παρόμοια χαρακτηριστικά. Ο αλγόριθμος Smith-Waterman είναι βασισμένος στο σκεπτικό της σύγκρισης οποιουδήποτε κομματιού μιας ακολουθίας με αυτά μιας άλλης, έτσι ώστε να βρεθούν τα κομμάτια με την μεγαλύτερη δυνατή ομοιότητα. Συμπεραίνουμε λοιπόν, ότι αυτός ο αριθμός των ζευγαριών υποακολουθιών και των συγκρίσεων που προκύπτουν είναι πολύ μεγάλος. Ο αλγόριθμος Smith-Waterman έχει την ευαισθησία να εντοπίσει τις υποακολουθίες με την μεγαλύτερη δυνατή ομοιότητα, αλλά είναι ένας πολύ απαιτητικός αλγόριθμος σε πόρους και σε χρόνο.

Ο αλγόριθμος Smith-Waterman προέκυψε από την εξέλιξη του αλγόριθμου Needleman και Wunsch. Οι Needleman και Wunsch δημιούργησαν το 1970 έναν αλγόριθμο για τον εντοπισμό ομοιοτήτων σε δύο ακολουθίες. Στην συνέχεια προτάθηκαν διάφορες εναλλακτικές με σημαντικότερη αυτή του Sellers που έφτασε πιο κοντά στις απαιτήσεις της βιολογίας υπολογίζοντας μια μετρική διαφοράς μεταξύ δυο ακολουθιών. Περαιτέρω εξέλιξη οδήγησε στην δημιουργία των Smith-Waterman που βασίστηκε στην τοπική ομοιότητα δυο ακολουθιών και όχι στην καθολική ομοιότητα των ακολουθιών σε όλη τους την έκταση.

Ο αλγόριθμος Smith-Waterman είναι ο πιο ακριβής αλγόριθμος για την εύρεση όμοιων περιοχών ακολουθιών μέσα σε μεγάλες βάσεις δεδομένων, αλλά είναι και ο πιο χρονοβόρος. Για αυτό το λόγο έχουν προταθεί διάφορα μοντέλα για την μείωση των απαιτήσεων και του χρόνου, όπως για παράδειγμα το μοντέλο BLAST(Basic Alignment Search Tool).

5.1 Λειτουργία Smith-Waterman

Ο αλγόριθμος Smith-Waterman ανιχνεύσει όμοιες υποακολουθίες συγκρίνοντας ακολουθίες. Επειδή έχουμε να κάνουμε με τοπική ομοιότητα δυο ακολουθιών, ο αριθμός των όμοιων υποακολουθιών μπορεί να είναι αρκετά μεγάλος, για αυτό η εύρεση των υποακολουθιών με την μεγαλύτερη ομοιότητα είναι πολύ σημαντική. Ουσιαστικά η λειτουργία του αλγορίθμου Smith-Waterman είναι αυτή ακριβώς, δηλαδή να βρει την καλύτερη τοπική ομοιότητα μεταξύ δύο ακολουθιών.

Η βέλτιστη τοπική ομοιότητα εντοπίζεται με την σύγκριση μιας ακολουθίας και των ακολουθιών που βρίσκονται στην βάση δεδομένων, συγκρίνοντας τους χαρακτήρες έναν προς έναν. Η διαφορά του αλγόριθμου Smith-Waterman με τον αλγόριθμο Needleman-Wunsch, είναι ότι ψάχνει για τοπικές ομοιότητες στις ακολουθίες, σε αντίθεση με τον Needleman-Wunsch που ψάχνει για ομοιότητα σε ολόκληρη την ακολουθία. Ο Needleman-Wunsch συγκρίνει ολόκληρη την ακολουθία με τις υπόλοιπες ακολουθίες. Ο Smith-Waterman συγκρίνει οποιαδήποτε κομμάτια με διάφορα μεγέθη μεταξύ των ακολουθιών.

Ο Smith-Waterman είναι βασισμένος στον δυναμικό προγραμματισμό, δηλαδή χωρίζουμε το πρόβλημα σε πολλά μικρότερα και αφού λύσουμε αυτά τα μικρότερα προβλήματα, τα ενώνουμε για λύσουμε το αρχικό πρόβλημα. Εφαρμόζοντας αυτή την τεχνική ο αλγόριθμος βρίσκει την ομοιότητα μεταξύ οποιασδήποτε υποακολουθίας, με αρχή και τέλος οποιοδήποτε σημείο των δυο ακολουθιών που συγκρίνονται. Στη συνέχεια έχοντας υπολογίσει το σκορ ομοιότητας για όλες αυτές τις συγκρίσεις κρατάει τις υποακολουθίες που παρουσιάζουν το μεγαλύτερο σκορ

Το σκορ ομοιότητας που χρησιμοποιεί ο αλγόριθμος εκφράζει το κόστος για να μεταλλάξουμε την μια υποακολουθία στην άλλη, δηλαδή αυτή με την οποία συγκρίνεται. Σε αυτή την μετάλλαξη επιτρέπονται δυο διαφορετικές αλλαγές. Η πρώτη αλλαγή που μπορούμε να κάνουμε είναι η εναλλαγή ενός στοιχείου με ένα άλλο. Για παράδειγμα αν συγκρίνουμε τις δυο υποακολουθίες από στοιχείο σε στοιχείο και βρεθούμε σε δυο στοιχεία που είναι διαφορετικά, τότε πραγματοποιούμε την αλλαγή αυτού του στοιχείου με το στοιχείο της άλλης υποακολουθίας και σε αυτή μας την αλλαγή δίνουμε ένα κόστος.

Η δεύτερη αλλαγή είναι η εισαγωγή ή η διαγραφή ενός στοιχείου. Για παράδειγμα όταν παρατηρούμε ότι η διαφορά δυο υποακολουθιών έγκειται στην έλλειψη ενός στοιχείου στην μια από αυτές, τότε το εισάγουμε προσδίδοντας κάποιο επιπλέον κόστος. Παρόμοια λειτουργούμε και στην περίπτωση ενός πρόσθετου στοιχείου. Το ζευγάρι των ακολουθιών που χρειάζεται τις λιγότερες δυνατές αλλαγές ώστε να είναι ίδιες θα έχει το υψηλότερο σκορ ομοιότητας.

Ας υποθέσουμε ότι έχουμε δυο ακολουθίες X και Y με μήκος M και N, αντίστοιχα. Ο αλγόριθμος Smith-Waterman υπολογίζει το σκορ ομοιότητας H(i,j) δυο υποακολουθιών X[0..i] και Y[0..j] με την χρήση των ακόλουθων εξισώσεων.

$$H(i,j) = \max\begin{cases} 0 \\ E(i,j) \\ F(i,j) \\ H(i-1,j-1) + S(X_i, Y_j) \end{cases} \quad \text{for } 1 \le i \le M, 1 \le j \le N$$

$$E(i,j) = \max\begin{cases} H(i,j-1) - \alpha \\ E(i,j-1) - \beta \end{cases} \quad \text{for } 0 \le i \le M, 1 \le j \le N$$

$$F(i,j) = \max\begin{cases} H(i-1,j) - \alpha \\ F(i-1,j) - \beta \end{cases} \quad \text{for } 1 \le i \le M, 0 \le j \le N$$

,όπου S είναι ο πίνακας κόστους εναλλαγή δυο στοιχείων.

Στους παραπάνω εξισώσεις το α είναι το πέναλτι για το πρώτο κενό(gap open) και το β είναι το πέναλτι για το δεύτερο κενό(gap extension). Το κενό είναι ένας κενός χαρακτήρας στην ακολουθία.

Ο αλγόριθμος αντιστοιχεί ένα σκορ σε κάθε ζευγάρι στοιχείων των δύο ακολουθιών. Αντιστοιχίζοντας ένα σκορ για ίδια στοιχεία ή για αλλαγή αυτών ή για παράλειψη ενός στοιχείου ή για εισαγωγή ενός νέου στοιχείου, δημιουργείται ένας πίνακας κόστους για όλα τα πιθανά ζευγάρια των δυο ακολουθιών και για όλα τα δυνατά μονοπάτια.

Παράδειγμα

Ας υποθέσουμε ότι έχουμε τις παρακάτω δυο ακολουθίες:

ATGCATCCCATGAC

TCTATATCCGT

Θεωρούμε σαν πέναλτι κενού το -2, σαν πέναλτι μη ομοιότητας το -3 και σαν σκορ για την ομοιότητα δυο στοιχείων το 2. Χρησιμοποιώντας τον αλγόριθμο Smith-Waterman παράγεται ο παρακάτω πίνακας σκορ(*Εικόνα 5-1: Σκορ πίνακας ομοιότητας*).

		A	T	G	C	A	T	C	C	C	A	T	G	A	C
	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
T	0	0	2	0	0	0	2	0	0	0	0	2	0	0	0
C	0	0	0	0	2	0	0	4	2	2	0	0	0	0	2
T	0	0	2	0	0	0	0	2	1	0	0	2	0	0	0
A	0	2	0	0	0	2	0	0	0	0	2	0	0	2	0
T	0	0	4	2	0	0	2	0	0	0	0	4	2	0	0
A	0	2	0	0	0	2	0	0	0	0	2	0	0	2	0
T	0	0	4	2	0	0	4	2	0	0	0	4	0	0	0
C	0	0	2	0	4	0	0	6	4	2	0	0	0	0	2
C	0	0	0	0	2	0	0	4	8	6	4	2	0	0	2
G	0	0	0	2	0	0	0	2	6	5	3	1	4	2	0
T	0	0	2	0	0	0	2	0	4	3	2	5	3	1	0

Εικόνα 5-1: Σκορ πίνακας ομοιότητας

Για να βρούμε τις ακολουθίες με την μεγαλύτερη ομοιότητα, βρίσκουμε το στοιχείο του πίνακα με την μεγαλύτερη τιμή και στην συνέχεια ακολουθούμε ένα μονοπάτι προς τα πίσω ακολουθώντας πάντα το γειτονικό στοιχεί με την μεγαλύτερη τιμή. Έτσι δημιουργείται ένα μονοπάτι που μας δείχνει τις δύο υποακολουθίες με την μεγαλύτερη ομοιότητα.

6. Ανίχνευση ηχητικών προτύπων με την GPU και του S-W

Στην παρούσα μεταπτυχιακή διατριβή υλοποιήθηκε ο αλγόριθμος Smith-Waterman για την ανίχνευση ηχητικών εφέ σε αρχεία ήχου ταινιών. Θέλοντας να ανιχνεύσουμε ένα ηχητικό εφέ μέσα σε μια ταινία τροποποιήσαμε τον αλγόριθμο S-W έτσι ώστε αντί να ανιχνεύει όμοιες ακολουθίες πρωτεϊνών, να ανιχνεύει όμοιες ηχητικές ακολουθίες. Η μετατροπή που πραγματοποιήσαμε ήταν να χρησιμοποιήσουμε ένα διαφορετικό πίνακα κόστους αντικατάστασης για να ανιχνεύσουμε όμοιες ηχητικές ακολουθίες.

Στην βιολογία χρησιμοποιούνται συγκεκριμένοι πίνακες κόστους αντικατάστασης όπως ο BLOSUM62, δηλαδή στην σύγκριση μεταξύ των ακολουθιών πρωτεϊνών δίνουμε ένα κόστος, όταν το ένα στοιχείο μιας ακολουθίας δεν είναι όμοιο με το στοιχείο της δεύτερης ακολουθίας και πρέπει να το αντικαταστήσουμε έτσι ώστε να έχουμε ομοιότητα. Έτσι ο πίνακας BLOSUM62 είναι ένας 20x20 πίνακας που μας δίνει την τιμή κόστους για την αντικατάσταση μιας πρωτεΐνης με μία άλλη.

Στην περίπτωση μας που έχουμε ακολουθίες ήχου και όχι ακολουθίες πρωτεϊνών, χρησιμοποιήθηκε ένας άλλος πίνακας κόστους αντικατάστασης. Το κάθε frame ήχου, το επεξεργαζόμαστε έτσι ώστε, να εξάγουμε από αυτό το Audio Spectrum Envelop, το οποίο είναι ένα διάνυσμα 62 διαστάσεων. Έτσι τελικά έχουμε δυο ακολουθίες διανυσμάτων, που το κάθε διάνυσμα έχει 62 διαστάσεις. Η σύγκριση λοιπόν, γίνεται μεταξύ μιας ακολουθία διανυσμάτων που αναπαριστούν το εφέ(πρότυπο), που θέλουμε να ανιχνεύσουμε, και μία ακολουθία διανυσμάτων που αναπαριστούν τον ήχο της ταινίας. Για να πραγματοποιήσουμε αυτή την σύγκριση διαμελίσουμε τον ήχο της ταινίας σε μικρά κομμάτια. Η σύγκριση, λοιπόν γίνεται μεταξύ του εφέ και του κάθε κομματιού του ήχου της ταινίας. Ο πίνακας κόστους αντικατάστασης που χρησιμοποιούμε, είναι για κάθε στοιχείο του πίνακα το αντίστοιχο συνημίτονο των δύο συγκρινόμενων διανυσμάτων, το οποίο το πολλαπλασιάζουμε επί εκατό και το στρογγυλοποιούμε στον κοντινότερο ακέραιο. Έτσι, χρησιμοποιώντας αυτόν τον πίνακα κόστους αντικατάστασης ανιχνεύουμε τα ηχητικά εφέ χρησιμοποιώντας τον αλγόριθμο Smith-Waterman.

Ο αλγόριθμος Smith Waterman βρίσκει ένα ζευγάρι υποακολουθίων το οποίο παρουσιάζει την μεγαλύτερη ομοιότητα, μεταξύ δύο ακολουθιών χρησιμοποιώντας τις παρακάτω αναδρομικές θέσεις:

$$E[i, j] = \max \{H[i, j - 1] - \alpha, E[i, j - 1] - \beta\}$$

$$F[i, j] = \max \{H[i - 1, j] - \alpha, F[i - 1, j] - \beta\}$$

$$H[i, j] = \max \{0, E[i, j], F[i, j], H[i - 1, j - 1] + sbt(Q[i - 1], T[j - 1])\}$$

,όπου το sbt είναι ένας πίνακας κόστους αντικατάστασης μεταξύ δυο στοιχείων των ακολουθιών.

Υπάρχουν δύο τρόποι για να γίνει παραλληλοποίηση του αλγόριθμου Smith – Waterman:

- **Inter-task parallelization**. Σε αυτού του τύπου την παραλληλοποίηση έχουμε μια εργασία που πρέπει να εκτελεστεί πολλές φορές για διαφορετικά δεδομένα. Το κάθε νήμα αναλαμβάνει να εκτελέσει αυτή την εργασία για συγκεκριμένα δεδομένα. Όλο το πλέγμα λειτουργεί έτσι ώστε να ολοκληρωθούν όλες οι διεργασίες παράλληλα. Σαν εργασία στην περίπτωση του αλγόριθμου Smith-Waterman μπορούμε να θεωρήσουμε την εύρεση του πίνακα κόστους μεταξύ ενός πρότυπου και ενός παραθύρου της ταινίας. Σε αυτόν τον τύπο παραλληλοποίησης, αναλαμβάνει το κάθε νήμα να ολοκληρώσει τον υπολογισμού του πίνακα κόστους.
- **Intra-task parallelization.** Στην περίπτωση αυτή μια εργασία ανατίθεται σε ένα μπλοκ από νήματα, έτσι ώστε όλα τα νήματα να συνεργαστούν μεταξύ τους για να ολοκληρωθεί η εργασία. Για να μπορεί να πραγματοποιηθεί αυτό πρέπει τα δεδομένα να μην είναι αλληλεξαρτόμενα. Στην παραλληλοποίηση του Smith-Waterman, ο σκοπός μας είναι να παραλληλοποιήσουμε τον υπολογισμό του πίνακας κόστους μεταξύ δυο υποακολουθιών. Ο υπολογισμός του κάθε στοιχείου του πίνακα δεν είναι ανεξάρτητος από τα άλλα στοιχεία. Κάθε στοιχείο του πίνακα εξαρτάται από το αριστερό του γειτονικό στοιχείο, από το πάνω γειτονικό στοιχείο και από το πάνω αριστερά στοιχείο. Τα στοιχεία που είναι ανεξάρτητα μεταξύ τους είναι αυτά που βρίσκονται στην αντιδιαγώνιο του πίνακα. Επομένως, στην περίπτωση της Intra-task parallelization, τα νήματα λειτουργούν παράλληλα ώστε να υπολογίσει το κάθε ένα στοιχείο της αντιδιαγωνίου.

Η παραλληλοποίηση Inter-task έχει μεγαλύτερες απαιτήσει σε μνήμη, αλλά επιτυγχάνει καλύτερες επιδόσεις σε σχέση με την Intra-task.

Στην συγκεκριμένη υλοποίηση έγινε εφαρμογή του αλγορίθμου Smith - Waterman μέσο της μεθόδου ***Inter-task parallelization***. Κάθε thread αναλαμβάνει μια ολοκληρωμένη εργασία, δηλαδή αναλαμβάνει:

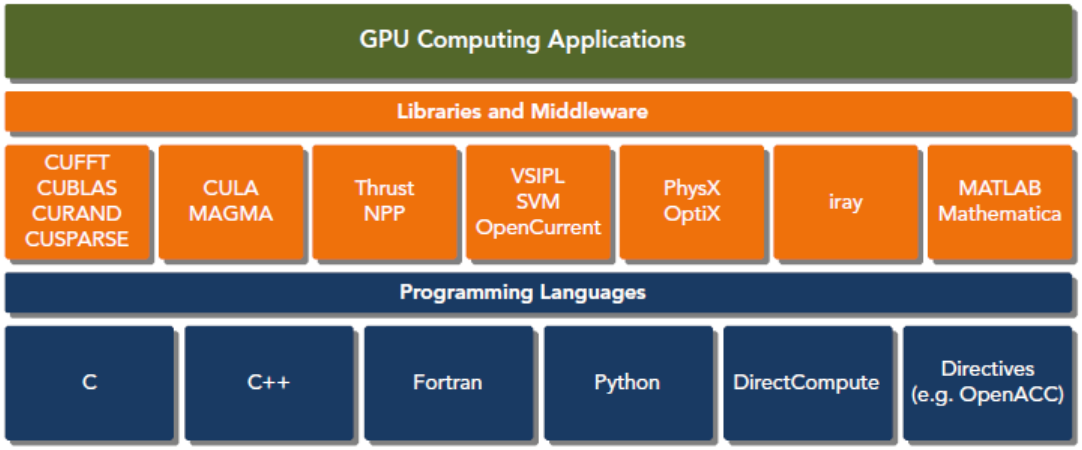
1. Τον υπολογισμό του πίνακα ομοιότητας, μεταξύ των διανυσμάτων ενός μικρού κομματιού του ήχου της ταινίας και των διανυσμάτων του pattern που θέλουμε να εντοπίσουμε.
2. Την εφαρμογή του αλγόριθμου για τις δυο ακολουθίες διανυσμάτων, την ακολουθία διανυσμάτων του αποσπάσματος ήχου της ταινίας και την ακολουθία διανυσμάτων του pattern, με βάση τον πίνακα ομοιότητας που υπολογίστηκε προηγούμενος.

Το κάθε νήμα αναλαμβάνει να ολοκληρώσει αυτές τις διεργασίες και όλο το πλέγμα θα πραγματοποιήσει αυτές τις δυο διεργασίες για το πρότυπο και για όλα τα κομμάτια ήχου της ταινίας παράλληλα. Αυτές λοιπόν οι διεργασίες, που στην περίπτωση του σειριακού προγραμματισμού θα έπρεπε να εκτελεστούν σειριακά η μια μετά την άλλη για κάθε ένα κομμάτι της ταινίας και το πρότυπου, στην παραλληλοποίηση inter-task θα πραγματοποιηθούν παράλληλα.

Η υλοποίηση της ανίχνευσης ηχητικών εφέ μέσα σε ηχητικά δεδομένα με την χρήση της CUDA αποτελείται από δυο μέρη: το μέρος του προγράμματος που εκτελείται από την CPU και το μέρος του προγράμματος που υλοποιείται από την κάρτα γραφικών μέσω των υπολογιστικών πυρήνων. Στην συγκεκριμένη υλοποίηση το πρόγραμμα που εκτελείται από την CPU αναλαμβάνει μόνο εργασίες που έχουν να κάνουν με την φόρτωση των δεδομένων από τα κατάλληλα αρχεία και την μεταφορά των δεδομένων αυτών από την CPU στην κάρτα γραφικών.

6.1 Προγραμματιστικό Περιβάλλον

Σε αυτό το κεφάλαιο θα παρουσιάσουμε το περιβάλλον στο οποίο πραγματοποιήθηκε η υλοποίηση μας. Η CUDA είναι μια υπολογιστική πλατφόρμα παράλληλου προγραμματισμού που αξιοποιεί τις κάρτες γραφικών παράλληλης αρχιτεκτονικής NVIDIA, για να επιλύσει πολύπλοκα υπολογιστική προβλήματα με αποδοτικότερο τρόπο. Η πλατφόρμα της CUDA είναι προσβάσιμη από τις βιβλιοθήκες της CUDA, τις οδηγίες του μεταγλωττιστή και από προεκτάσεις σε υπάρχουσες προγραμματιστικές γλώσσες όπως η C, C++, Fortran και Python, όπως φαίνεται στην Εικόνα 6-1: Πλατφόρμα CUDA.



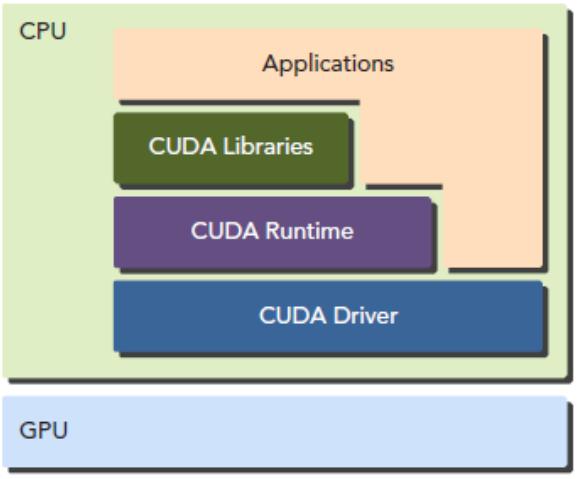
Εικόνα 6-1: Πλατφόρμα CUDA

Η CUDA C είναι μια επέκταση της κλασσική γλώσσας προγραμματισμού ANSI C, με χρήσιμες γλωσσικές επεκτάσεις για την ενεργοποίηση του ετερογενούς προγραμματισμού και με τις κατάλληλες διεπαφές για την διαχείριση των συσκευών, της μνήμης και άλλων λειτουργιών. Η CUDA είναι επίσης ένα μοντέλο προγραμματισμού που μας δίνει την δυνατότητα να δημιουργούμε εφαρμογές που μπορούν να υποστούν διεύρυνση εκτελώντας τις, σε κάρτες γραφικών με περισσότερους πυρήνες χωρίς να χρειάζεται να πραγματοποιήσουμε αλλαγές στο πρόγραμμα μας. Αναπτύσσοντας δηλαδή, μια εφαρμογή για μια συγκεκριμένη συσκευή, η CUDA μας δίνει την δυνατότητα να τρέξουμε την ίδια αυτή εφαρμογή σε μια κάρτα γραφικών με περισσότερους πόρους και να έχουμε καλύτερες επιδόσεις. Έχουμε επομένως επεκτασιμότητα χωρίς να πραγματοποιήσουμε αλλαγές στο πρόγραμμά μας.

Η CUDA παρέχει δύο επίπεδα διεπαφών για την διαχείριση της κάρτας γραφικών και την οργάνωση των νημάτων:

- CUDA Driver API
- CUDA Runtime API

Η διεπαφή οδηγών(Driver API) είναι μια διεπαφή χαμηλού επιπέδου και είναι δύσκολο να προγραμματιστεί, αλλά προσφέρει περισσότερο έλεγχο πάνω στην κάρτα γραφικών. Η διεπαφή εκτέλεσης είναι υψηλού επιπέδου και βρίσκεται πάνω από την διεπαφή οδηγών.



Εικόνα 6-2: CUDA Application Programming Interfaces

Για το περιβάλλον που πραγματοποιήθηκε η υλοποίηση χρειαστήκαμε τα παρακάτω εργαλεία:

- Μια κάρτα γραφικών η οποία υποστηρίζει τον προγραμματισμό σε CUDA.
- Ένα οδηγό λογισμικού NVIDIA για την υποστήριξη της κάρτας γραφικών.
- Το λογισμικό CUDA Development toolkit.
- Το λογισμικό ενός compiler C.

Το λογισμικό CUDA Development Toolkit μας παρέχει ένα προγραμματιστικό περιβάλλον για την ανάπτυξη εφαρμογών βασισμένες στην γλώσσα C ή C++. Οι εφαρμογές αυτές είναι υλοποιημένες με τέτοιο τρόπο ώστε να τρέχουν πάνω στο υλικό μιας κάρτα γραφικών. Το λογισμικό αυτό περιέχει έναν compiler για τις NVIDIA κάρτες γραφικών, βιβλιοθήκες μαθηματικών και εργαλεία για την βελτιστοποίηση των εφαρμογών μας.

Η CUDA C μας παρέχει έναν εύκολο τρόπο να παράγουμε εφαρμογές οι οποίες θα τρέξουν πάνω στην κάρτα γραφικών μέσω της χρήση της ευρέως διαδεδομένης γλώσσα C. Αποτελείται από σύνολο προεκτάσεων της γλώσσας C και ένα σύνολο από βιβλιοθήκες. Η σημαντικότερη προέκταση που προστίθεται πάνω στην C είναι η δυνατότητα ορισμού της υπορουτίνας του υπολογιστικού πυρήνα(kernel), όπως και η χρήση ορισμένων λέξεων για την ρύθμιση του πλέγματος. Οι πυρήνες μπορούν να γραφτούν με ένα σετ εντολών που ονομάζεται PTX. Είναι όμως αποτελεσματικότερο να χρησιμοποιούμε μια γλώσσα υψηλότερου επιπέδου, όπως η C. Στις δυο αυτές περιπτώσεις, οι kernels πρέπει να μεταγλωττιστούν σε δυαδικό κώδικα από τον μεταγλωττιστή nvcc για να εκτελεστεί από την συσκευή. Ο nvcc(NVIDIA's CUDA Compiler) είναι βασισμένος πάνω στον ανοιχτού λογισμικού compiler LLVM.

Η συσκευή στην οποία πραγματοποιήθηκε η υλοποίηση είναι αυτή με τα παρακάτω χαρακτηριστικά:

Device	"GeForce 930M"
CUDA Driver Version / Runtime Version	8.0 / 8.0
CUDA Capability Major/Minor version number	5
Total amount of global memory	2048 MBytes (2147483648 bytes)
(3) Multiprocessors, (128) CUDA Cores/MP	384 CUDA Cores
GPU Max Clock rate	941 MHz (0.94 GHz)
Memory Clock rate	900 Mhz
Memory Bus Width	64-bit
L2 Cache Size	1048576 bytes
Maximum Texture Dimension Size (x,y,z)	1D=(65536), 2D=(65536, 65536), 3D=(4096, 4096, 4096)
Maximum Layered 1D Texture Size, (num) layers	1D=(16384), 2048 layers
Maximum Layered 2D Texture Size, (num) layers	2D=(16384, 16384), 2048 layers
Total amount of constant memory	65536 bytes
Total amount of shared memory per block	49152 bytes
Total number of registers available per block	65536
Warp size	32
Maximum number of threads per multiprocessor	2048
Maximum number of threads per block	1024
Max dimension size of a thread block (x,y,z)	(1024, 1024, 64)
Max dimension size of a grid size (x,y,z)	(2147483647, 65535, 65535)
Maximum memory pitch	2147483647 bytes
Texture alignment	512 bytes
Concurrent copy and kernel execution	Yes with 1 copy engine(s)
Run time limit on kernels	No
Integrated GPU sharing Host Memory	No
Support host page-locked memory mapping	Yes
Alignment requirement for Surfaces	Yes
Device has ECC support	Disabled
CUDA Device Driver Mode (TCC or WDDM)	WDDM (Windows Display Driver Model)
Device supports Unified Addressing (UVA)	Yes
Device PCI Domain ID / Bus ID / location ID	0 / 1 / 0
CUDA Device Driver Mode (TCC or WDDM)	WDDM (Windows Display Driver Model)
Device supports Unified Addressing (UVA)	Yes
Device PCI Domain ID / Bus ID / location ID	0 / 1 / 0

Εικόνα 6-3: Χαρακτηριστικά συσκευής συστήματος

Το κεντρικό σύστημα το οποίο φιλοξενεί την κάρτα γραφικών είχε τα παρακάτω χαρακτηριστικά.

OS Name	Microsoft Windows 10 Pro
OS Version	10.0.14393 N/A Build 14393
OS Manufacturer	Microsoft Corporation
OS Configuration	Standalone Workstation
OS Build Type	Multiprocessor Free
System Manufacturer	TOSHIBA
System Model	SATELLITE P50-C
System Type	x64-based PC
Processor(s)	Intel® Core™ i7-6500U CPU @2.50GHz
Total Physical Memory	8.00 GB

Εικόνα 6-4: Χαρακτηριστικά κεντρικού συστήματος

6.1.1 Ροή μεταγλώττισης

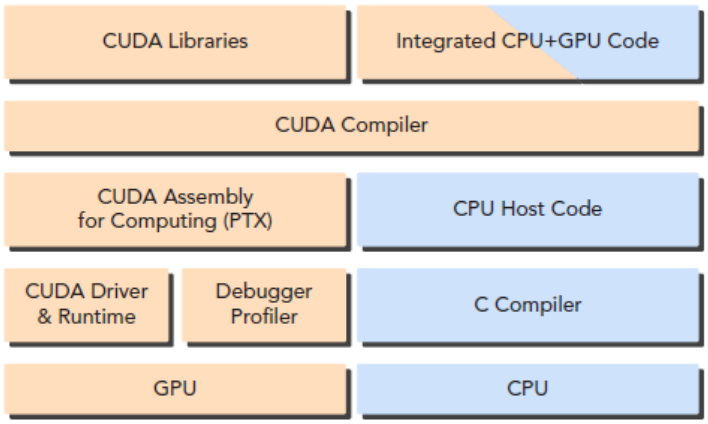
Ένα πρόγραμμα CUDA είναι ένας συνδυασμός δύο κομματιών:

- Το κομμάτι του κώδικα που θα τρέξει από την CPU.
- Το κομμάτι του κώδικα που θα τρέξει από την GPU.

Τα πηγαία αρχεία που παρέχονται στον nvcc για μεταγλώττιση περιέχουν κώδικα ο οποίος ένα μέρος του θα τρέξει πάνω στη CPU και ένα άλλο μέρος του θα τρέξει πάνω στην κάρτα γραφικών. Μια βασική λειτουργία του μεταγλωττιστή είναι να διαχωρίσει τον κώδικα της κάρτας γραφικών από αυτόν της CPU και στην συνέχεια να πραγματοποιήσει τις παρακάτω λειτουργίες:

- Να μεταγλωττίσει τον κώδικα της κάρτας γραφικών σε δυαδική μορφή.
- Να αλλάξει τον κώδικα της CPU ο οποίος περικλείεται σε <<<…..>>>, και να τον αντικαταστήσει με τους kernel που έχουν ήδη μεταγλωττιστεί.

Ο παραγόμενος κώδικας της CPU εξάγεται σαν C κώδικας που παραδίδεται για περαιτέρω μεταγλώττιση από κάποιο άλλο εργαλείο ή καλείται απευθείας από τον nvcc ο μεταγλωττιστής της CPU για να μεταγλωττιστής στο τελευταίο στάδιο της επεξεργασίας.



Εικόνα 6-5: Διαχωρισμός κώδικα CPU-GPU

6.2 Φόρτωση ηχητικών αρχείων

Τα δεδομένα που χρησιμοποιούμε για την ανίχνευση ενός pattern είναι δυο αρχεία. Το ένα αρχείο περιλαμβάνει τον ήχο μιας ταινίας που μέσα σε αυτό θα ανιχνεύσουμε το pattern. Το δεύτερο αρχείο περιέχει το αρχείο του pattern που θέλουμε να ανιχνεύσουμε. Τα δυο αρχεία αυτά τα έχουμε επεξεργαστεί έτσι ώστε να εξάγουμε από αυτά το Audio Spectrum Envelop. Το ηχητικό αρχείο περνάει από επεξεργασία μέσο μίας τεχνικής κινούμενου παραθύρου. Το προτεινόμενο μέγεθος του παράθυρου είναι 46.4ms με μηδενική επικάλυψη μεταξύ των διαδοχικώς παραθύρων. Σε κάθε frame , εξάγεται ο ηχητικός φάκελος φάσματος MPEG7(). Το ASE είναι ένα πολυδιάστατο φασματικό διάνυσμα που βασίζεται στην ιδέα ότι οι γειτονική συντελεστές του μετασχηματισμού Fourier, μπορούν να ομαδοποιηθούν για να δημιουργήσουν ένα κάδο, δημιουργώντας έτσι ένα πιο απλό τρόπο αναπαράστασης του φάσματος του σήματος

Το ASE υπολογίζεται με τον εξής τρόπο:

- Γίνεται δειγματοληψία του σήματος με συχνότητα Fs = 22050Hz και το σπάμε σε μη επικαλύψιμα frames. Σε κάθε frame υπολογίζεται ο μετασχηματισμός Fourier.
- Παίρνοντας την συχνότητα 1000Hz σαν σημείο εκκίνησης, το φάσμα χωρίζεται σε μπάντες οι οποίες είναι λογαριθμικά κατανεμημένες μεταξύ των 62.5Hz και 11025Hz. Η φασματική ανάλυση r της κάθε μπάντας είναι r = 2^j = 2^(-2) = 0.25 οκτάβες…….
- Αφού έχουμε φτιάξει τις μπάντες, το άθροισμα των σταθερών ισχύς του φάσματος σε κάθε μπάντα αποτελεί την σταθερά ASE. Στο τέλος οι σταθερές DFT κάτω από 62.5Hz αθροίζονται για να δώσουν άλλη μια ASE σταθερά.

Στην μελέτη μας επειδή η συχνότητα δειγματοληψίας είναι 22050Hz, ο πραγματικός αριθμός των μπαντών είναι 31. Έτσι, ο υπολογισμός του ASE μας δίνει ένα διάνυσμα 31 διαστάσεων για κάθε frame. Αν $X_a=\{x_1, x_2, ..., x_F\}$ είναι ένα διάνυσμα ASE και F είναι ο αριθμός των συχνοτήτων. Στη συνέχεια βρίσκουμε την πρώτη παράγωγο του διανύσματος. Αυτό είναι μια τεχνική που εφαρμόζεται συχνά σε ηχητικά δείγματα, για να απεικονίσουμε την εξάρτηση μεταξύ γειτονικών frame. Έτσι, στην δικιά μας περίπτωση η πρώτη παράγωγος δίνεται από την σχέση $dx_i=G(x_{i+1}-x_i)$, όπου G=0.375 είναι ένας παράγοντας ενίσχυσης. Το αποτέλεσμα αυτής της διαδικασίας είναι ότι τα διανύσματα που περιγράφουν

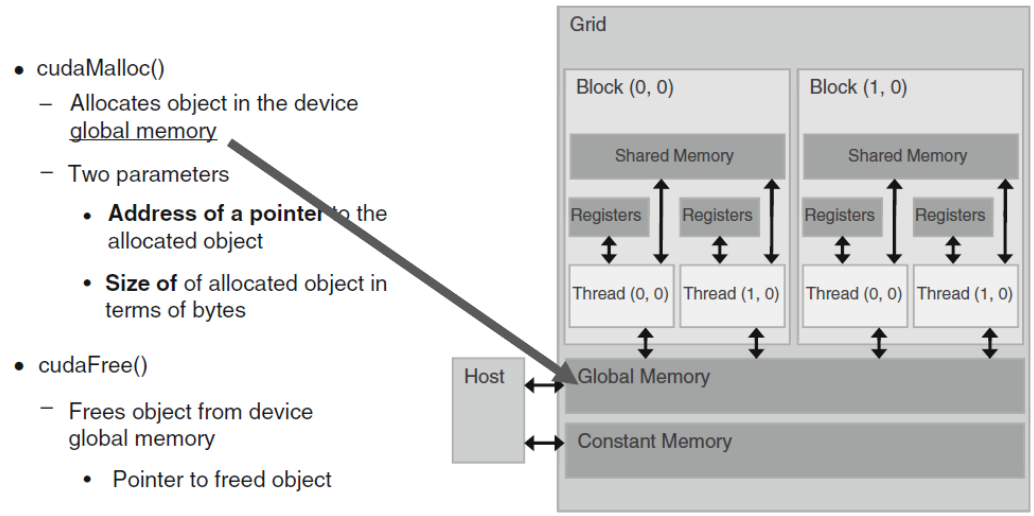
το αρχείο του ήχου έχουν 62 διαστάσεις. Η ίδια διαδικασία πραγματοποιείται και για το αρχείο ήχου του προτύπου που θέλουμε να ανιχνεύσουμε.

Η πρώτη μας εργασία είναι να ανεβάσουμε στην εφαρμογή μας τα δύο αρχεία, το πρώτο με τον ήχο της ταινίας και το δεύτερο με το pattern που θέλουμε να ανιχνεύσουμε. Τα αρχεία αυτά αποθηκεύονται σε δύο πίνακες με στοιχεία τύπου float. Αρχικά τα αρχεία αυτά αποθηκεύονται στην μνήμη της κεντρικής μονάδας(host). Έτσι, τα δεδομένα μας αυτά πρέπει να μεταφερθούν στην μνήμη της κάρτας γραφικών. Η δέσμευση της μνήμης, όπου θα αποθηκεύσουμε τα δεδομένα καλείται από την CPU μέσω της εντολής *cudaMalloc(void** devPtr, size_t size)*.

Παράμετροι:

- devPtr: Δείκτης που δείχνει στην μνήμη που θα δεσμευτεί.
- size: Μέγεθος μνήμη που θα δεσμευτεί σε bytes.

Η εντολή *cudaMalloc* δεσμεύει έναν συγκεκριμένο γραμμικό μέρος της μνήμης και επιστρέφει έναν δείκτη που δείχνει στην δεσμευμένη αυτή μνήμη. Η συμπεριφορά της είναι παρόμοια με την εντολή *malloc()* της C. Η μνήμη είναι κατάλληλα ευθυγραμμισμένη για οποιοδήποτε είδος μεταβλητής. Η εντολή επιστρέφει *cudaErrorMemoryAllocation* σε περίπτωση που αποτύχει η δέσμευση της μνήμης. Η εντολή παίρνει σαν παραμέτρους το μέγεθος της μνήμης που θέλουμε να δεσμεύσουμε και μια μεταβλητή δείκτη που θα δείχνει στην μνήμη η οποία θα δεσμευτεί. Η *cudaMalloc* μπορεί, επίσης να χρησιμοποιηθεί και από τον κώδικα που εκτελείται από την κάρτα γραφικών κατά την εκτέλεση του προγράμματος.



Εικόνα 6-6: Συναρτήσεις CUDA Api για την διαχείριση της Global Memory

Η εντολή αυτή συνήθως ακολουθείται και από την εντολή *cudaFree(void* devPtr)*. Η εντολή *cudaFree* ελευθερώνει την μνήμη στην οποία δείχνει ένας δείκτης, η οποία μνήμη είχε πριν δεσμευτεί από την εντολή *cudaMalloc*. Επίσης, η εντολή δεν μπορεί να χρησιμοποιηθεί στον κώδικα του host όταν η δέσμευση έχει γίνει από τον κώδικα της κάρτας γραφικών και αντίστροφα(βλέπε Εικόνα 6-7: Χρήση *cudaFree*).

	cudaMalloc() on Host	cudaMalloc() on Device
cudaFree() on Host	Supported	Not Supported
cudaFree() on Device	Not Supported	Supported
Allocation limit	Free device memory	cudaLimitMallocHeapSize

Εικόνα 6-7: Χρήση *cudaFree*

Αφού δεσμεύσουμε την επιθυμητή μνήμη πρέπει να μεταφέρουμε τα δεδομένα μας τα οποία είναι αποθηκευμένα στην μνήμη της κεντρική μονάδας. Η μεταφορά των δεδομένων γίνεται μέσο της εντολής

*cudaMemcpy(void * dst, const void * src, size_t count, enum cudaMemcpyKind kind)* της βιβλιοθήκης της CUDA.

Παράμετροι:

- dst: Διεύθυνση της μνήμης που θα γίνει η αντιγραφή.
- src: Διεύθυνση της μνήμης από όπου θα γίνει η αντιγραφή.
- count: Μέγεθος σε bytes των δεδομένων που θα αντιγραφούν.
- kind: Τύπος αντιγραφής δεδομένων.

Η εντολή *cudaMemcpy* αντιγράφει έναν αριθμό bytes από την περιοχή της μνήμης που δείχνει η παράμετρος της πηγής, στην περιοχή της μνήμης που δείχνει η παράμετρος του προορισμού. Μια άλλη παράμετρος που πρέπει να καθοριστεί είναι το είδος της αντιγραφής που θα πραγματοποιηθεί. Αυτή η παράμετρος μπορεί να πάρει τιμές, *cudaMemcpyHostToHost, cudaMemcpyHostToDevice, cudaMemcpyDeviceToHost*, ή *cudaMemcpyDeviceToDevice* και καθορίζει την κατεύθυνση της αντιγραφής. Οι περιοχές μνήμης μεταξύ των οποίων θα γίνει η αντιγραφή θα πρέπει να συμπίπτουν σε μέγεθος.

Έτσι με την χρήση της συνάρτησης *cudaMemcpy* αντιγράφουμε τα δεδομένα, δηλαδή το πρότυπο και το αρχείο ήχου της ταινίας, στην μνήμη της κάρτας γραφικών.

```
cudaMemcpy(ASE_Result_y_Dev, ASE_Result_y, NumFeatures * NumPatternFrames * sizeof(float), cudaMemcpyHostToDevice);
        cudaMemcpy(ASE_Result_x_Dev, ASE_Result_x, NumFeatures * NumMovieFrames * sizeof(float), cudaMemcpyHostToDevice);
```

Εικόνα 6-8: Αντιγραφή δεδομένων από την CPU στην GPU

6.3 Υπολογισμός πίνακα κόστους αντικατάστασης

Η κάρτα γραφικών αναλαμβάνει να εκτελέσει τον υπολογισμό του πίνακα κόστους αντικατάστασης μεταξύ των διανυσμάτων ASE των προτύπων και του αρχείου ήχου, με τον οποίο θα εκτελεστεί ο αλγόριθμος Smith Waterman.Στη περίπτωση των πρωτεϊνών ο πίνακας ομοιότητας είναι ένας πίνακας, όπου η κάθε αλλαγή ενός γράμματος με ένα άλλο, κοστολογείται με μια συγκεκριμένη τιμή. Επομένως, στην βιολογία έχουμε ένα κοινό πίνακας ομοιότητας για όλες τις συγκρίσεις μεταξύ της ακολουθίας που ψάχνουμε της βάσης δεδομένων.

Στη περίπτωση μας που έχουμε την ανίχνευση ενός ηχητικού προτύπου, ο πίνακας ομοιότητας αλλάζει μεταξύ κάθε ζευγαριού ηχητικών δειγμάτων που συγκρίνονται. Ο πίνακας κόστους αντικατάστασης μεταξύ των διανυσμάτων ASE των προτύπων και των διανυσμάτων ASE ενός μέρος(παράθυρο) του ήχου της ταινίας υπολογίζεται με τον παρακάτω τρόπο.

Ουσιαστικά το κόστος της αντικατάστασης ενός διανύσματος του προτύπου με ένα διάνυσμα της ταινίας δίνεται ως εξής:

- Πρώτα υπολογίζεται το συνημίτονο της γωνίας μεταξύ δύο διανυσμάτων από την παρακάτω σχέση.

$$S(i,j) = \frac{\sum_{k=1}^{L} x_i(k) y_j(k)}{\sqrt{\sum_{k=1}^{L} x_i^2(k)} * \sqrt{\sum_{k=1}^{L} y_j^2(k)}}$$

, όπου L = 62.

- Το S(i, j) στην συνέχεια πολλαπλασιάζεται με 100 και στρογγυλοποιείται στον κοντινότερο ακέραιο, παίρνοντας έτσι τιμές [-100, 100].

Αυτός ο υπολογισμός στην υλοποίηση μας πραγματοποιείται από κάθε νήμα ξεχωριστά, δηλαδή ένα νήμα υπολογίζει τον πίνακα ομοιότητας τον οποίο θα χρησιμοποιήσει στην συνέχεια για να εκτελέσει τον αλγόριθμο Smith Waterman. Το αρχείο ήχου του προτύπου που έχουμε χρησιμοποιήσει έχει συνολικά 108 frames και το παράθυρο της ταινίας για το οποίο τρέχουμε τον αλγόριθμο Smith Waterman περιλαμβάνει 200 frames. Ο πίνακας κόστους αντικατάστασης που δημιουργείται είναι ένας πίνακας μεγέθους 201x109, προσθέτοντας μια μηδενική γραμμή και μια μηδενική στήλη στην αρχή του πίνακα.

Έτσι αφού το κάθε frame παριστάνεται από ένα διάνυσμα, ο πίνακας ομοιότητας παριστάνει την ομοιότητα μεταξύ ενός frame του προτύπου και ενός frame της ταινίας. Η ομοιότητα εκφράζεται με τον τρόπο που περιεγράφηκε παραπάνω.

Η δημιουργία έτσι του πίνακα ομοιότητας πραγματοποιείται από την παρακάτω συνάρτηση *SimMatrix*.

```
__device__ void SimMatrix(float *Audio, float *Pattern, long LongWindow, long
NumPatternFrames, long NumFeatures, long Thresh, float Dt[][patternsPlusOne]) {

   long i, j, k;
   float num, den1, den2;

   int idx = blockIdx.x*blockDim.x + threadIdx.x;
   for (i = idx * LongWindow; i < (idx * LongWindow + LongWindow); i++) {
        den1 = 0.0;
   for (k = 0; k<NumFeatures; k++){
      den1 += (Audio[(i * NumFeatures) + k] * Audio[(i * NumFeatures) + k]);
   }
   for (j = 0; j<NumPatternFrames; j++) {
      num = 0.0; den2 = 0.0;
      for (k = 0; k<NumFeatures; k++){
         num += (Audio[(i * NumFeatures) + k] * Pattern[(j * NumFeatures) + k]);
         den2 += (Pattern[(j * NumFeatures) + k] * Pattern[(j * NumFeatures) + k]);
      }
      Dt[(i + 1) - idx * LongWindow][j + 1] = (100 * num) / (sqrt(den1)*sqrt(den2));
      if (Dt[(i + 1) - idx * LongWindow][j + 1]<Thresh)
         Dt[(i + 1) - idx * LongWindow][j + 1] = -Thresh;
      }
   }
}
```

6.4 Εφαρμογή Smith-Waterman(1η Υλοποίηση)

Ο αλγόριθμος Smith Waterman θα τρέξει πάνω σε αυτό τον πίνακα κόστους αντικατάστασης. Θα τρέξει όμως, και πάνω σε 100 τυχαίες παραλλαγές του πίνακα αυτού για να συγκρίνουμε την μέση τιμή αυτών των τυχαίων διανυσμάτων με την πραγματική σύγκριση. Από αυτή την σύγκριση θα διαπιστώσουμε αν το σκορ ομοιότητας για το πραγματικό πίνακα κόστους αντικατάστασης είναι μεγάλο και μπορούμε θεωρήσουμε ότι έχουμε την ανίχνευση του προτύπου. Για να πραγματοποιηθούν αυτές οι παραλλαγές πραγματοποιούμε μια τυχαία ανακατανομή στις στήλες του πίνακα. Για την δημιουργία της τυχαίας αυτής παραλλαγής χρησιμοποιούμε την συνάρτηση *curand()*. Η συνάρτηση αυτή ακολουθείται πάντα από την συνάρτηση *curand_init()* και μας επιστρέφει ένα σύνολο ψευδοτυχαίων αριθμών. Η συνάρτηση *curand()* παίρνει σαν παράμετρο μια κατάσταση. Αν αυτή η κατάσταση με την οποία καλούμε την *curand()*, τότε μας επιστρέφει τους ίδιους τυχαίους αριθμούς. Η συνάρτηση *curand_init()* αρχικοποιεί μια κατάσταση που καλείται από τον χρήστη δίνοντας ένα σπόρο, έναν αριθμό ακολουθίας και ένα όφσετ. Διαφορετικοί σπόροι διασφαλίζουν την παραγωγή διαφορετικών καταστάσεων και κατά συνέπεια διαφορετικών τυχαίων ακολουθιών αριθμών. Ο ίδιος σπόρος πάντα παράγει την ίδια αρχική κατάσταση και την ίδια ακολουθία αριθμών.

Σαν παραμέτρους αυτή η συνάρτηση παίρνει τον πίνακα με τα διανύσματα ASE ήχου της ταινίας, τον πίνακα με τα διανύσματα του pattern, το μέγεθος του παραθύρου του ήχου της ταινίας, των αριθμό των διανυσμάτων του pattern, το μέγεθος των διανυσμάτων και τον πίνακα όπου θα αποθηκευτεί ο πίνακας ομοιότητάς. Παρατηρούμε ότι το μέγεθος του πίνακα που θα αποθηκεύσουμε τις τιμές του πίνακα ομοιότητας δίνεται στατικά. Για να δεσμεύσουμε την μνήμη που θα αποθηκευτεί αυτός ο πίνακας εκτελούμε την παρακάτω εντολή από τον kernel:

```
float Dt[windowPlusOne][patternsPlusOne];
```

Η δέσμευση αυτού του πίνακα γίνεται στην local memory, που όπως έχουμε αναφέρει γίνεται στην αργή Global Memory. Επειδή ο πίνακας είναι μεγάλος ο compiler αναγνωρίζει ότι ο πίνακας αυτός δεν χωράει να αποθηκευτεί στους τοπικούς καταχωρητές και καταφεύγει στην δέσμευση της Global Memory. Οι τιμές των windowPlusOne και patternsPlusOne έχουν οριστεί στην αρχή του προγράμματος σαν σταθερές. Αυτός ο πίνακας θα ήταν βέλτιστο να αποθηκευτεί στην shared μνήμη της κάρτας γραφικών.

Επειδή όμως όπως αναφέραμε και παραπάνω το μέγεθος του είναι 201x109, ο αριθμός των bytes που χρειάζονται για να αποθηκευτεί είναι 201*109*sizeof(float) = 87636bytes. Αυτός ο αριθμός ξεπερνάει κατά πολύ το μέγεθος της κοινόχρηστης μνήμης που είναι 49152bytes.

Στην συνέχεια έχοντας υπολογίσει τον πίνακα ομοιότητας θα εκτελέσουμε τον αλγόριθμο Smith Waterman. Για τον υπολογισμό του αλγόριθμου θα χρειαστούμε τρεις νέους πίνακες τύπου float οι οποίοι θα έχουν μέγεθος 201x109. Η δέσμευση της μνήμης γίνεται πάλι στατικά μέσα στον kernel μέσω των εντολών:

float acc_cost[windowPlusOne][patternsPlusOne];

float hor_cost[windowPlusOne][patternsPlusOne];

float ver_cost[windowPlusOne][patternsPlusOne];

Οι τιμές windowPlusOne και patternsPlusOne δίνονται σαν σταθερές στην αρχή του προγράμματος.

Η εκτέλεση λοιπόν του αλγορίθμου για τον σύνολο των patterns διανυσμάτων και το σύνολο των διανυσμάτων ενός παραθύρου γίνεται μέσο της παρακάτω συνάρτησης της συσκευής:

```
__device__ int SW(float D[][patternsPlusOne], long M, long N,
        long penalty, float acc_cost[][patternsPlusOne], float hor_cost[][patternsPlusOne],
        float ver_cost[][patternsPlusOne]) {

        float diag_cost, temp;
        long i, j, indo;


        for (i = 1; i<M; i++) {
                for (j = 1; j<N; j++) {

                        diag_cost = acc_cost[i - 1][j - 1] + D[i][j];

                        ver_cost[i][j] = fmaxf(acc_cost[i - 1][j] - (100 + penalty),
ver_cost[i - 1][j] - penalty);
                        hor_cost[i][j] = fmaxf(acc_cost[i][j - 1] - (100 + penalty),
hor_cost[i][j - 1] - penalty);
                        temp = -100000;
                        temp = fmaxf(diag_cost, -100000);
                        temp = fmaxf(ver_cost[i][j], temp);
                        temp = fmaxf(hor_cost[i][j], temp);
                        acc_cost[i][j] = temp;

                        if (acc_cost[i][j] <= 0) {
                                acc_cost[i][j] = 0;
                        }
                }
        }
        return(0);
}
```

Χαρακτηριστικό μειονέκτημα της υλοποίησης αυτής είναι η χρήση των τριών μεγάλων πινάκων που αποθηκεύονται, μεν στατικά σαν local μεταβλητές, αλλά ουσιαστικά αποθηκεύεται σαν local memory πάνω στην global memory της κάρτας γραφικών. Όπως είδαμε προηγουμένως, όταν ο compiler διαπιστώσει ότι έχει να δεσμεύσει μνήμη για ένα νήμα και αυτή η μνήμη δεν χωράει στους αντίστοιχους καταχωρητές, τότε η δέσμευση γίνεται στην global memory. Η global memory λόγου του περιορισμένου εύρους ζώνης δημιουργεί σημαντικές καθυστερήσεις. Αυτή λοιπόν η επαναλαμβανόμενη επικοινωνία μεταξύ των νημάτων και της global memory έχει σαν αποτέλεσμα να μειώνονται σημαντικά οι επιδόσεις. Αυτό είναι από τα σημαντικότερα προβλήματα που έχουμε να αντιμετωπίσουμε, αφού ο αλγόριθμος για να εκτελεστεί χρειάζεται πολλές επικοινωνίες με την μνήμη και χρειάζεται μεγάλες περιοχές μνήμης για την αποθήκευση των δεδομένων.

Στη συνέχεια το ίδιο νήμα θα εκτελέσει τον ίδιο αλγόριθμο για 100 τυχαίες παραλλαγές του πίνακα ομοιότητας. Έτσι το κάθε νήμα μεταβάλει τυχαία τον πίνακα ομοιότητας και τρέχει ξανά τον αλγόριθμου Smith Waterman. Συνολικά το νήμα θα τρέξει τον αλγόριθμο 101 φορές, μια φορά για τον πραγματικό πίνακα κόστους αντικατάστασης και άλλες 100 φορές για τυχαίες παραλλαγές αυτού.

Η υλοποίηση αυτή τρέχει σε χρόνο 11771.368164 ms. Η αντίστοιχη ακολουθιακή υλοποίηση ολοκληρώνεται σε χρόνο 60800.000000. Παρουσιάζεται λοιπόν μια επιτάχυνση της τάξης 5.2x.

6.5 Ελαχιστοποίηση χρήσης μνήμης(2η Υλοποίηση)

Στα προηγούμενα πραγματοποιήσαμε δύο καινούργιες υλοποιήσεις στις οποίες πραγματοποιήσαμε από μια αλλαγή. Στην μία υλοποίηση αλλάξαμε την δομή, έτσι ώστε να έχουμε περισσότερη παραλληλία και κάθε νήμα να έχει λιγότερο φόρτο. Στην άλλη υλοποίηση χρησιμοποιήσαμε δυναμική δέσμευση μνήμης. Είδαμε όμως ότι αυτές οι δύο υλοποιήσεις δεν προσφέρανε στην ταχύτητα της εφαρμογής, αντιθέτως παίξανε αρνητικό ρόλο.

Έτσι, επιστρέψαμε στην αρχική μας υλοποίηση και προσπαθήσαμε να μειώσουμε όσο το δυνατόν τις προσπελάσεις στην μνήμη. Παρατηρώντας την λογική του αλγόριθμου S-W παρατηρούμε ότι οι πίνακες E,F και H είναι δισδιάστατοι πίνακες, αλλά ο αλγόριθμος χρησιμοποιεί μόνο μια γραμμή των πινάκων F, H και μόνο μια τιμή του πίνακα E. Αυτό είναι ιδιαίτερα σημαντικό γιατί ο τετραγωνικός χώρος που χρειαζόμαστε για τους πίνακες E,F και H να εξαντλήσει την μνήμη που χρειαζόμαστε στην global memory.

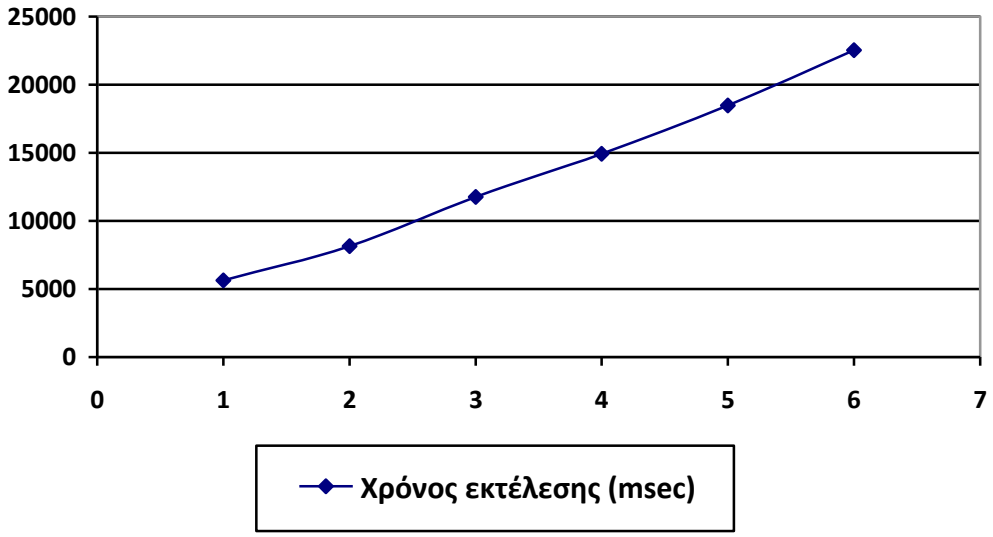
Χρησιμοποιώντας την δομή της πρώτης υλοποίησης, στην οποία ένα νήμα τρέχει τον αλγόριθμο SW για το κανονικό πίνακα ομοιότητας και για τις 100 παραλλαγές, πραγματοποιήσαμε τις παρακάτω αλλαγές για την βελτιστοποίηση του προγράμματος.

- **Αφαίρεση του πίνακα E[i,j].** Ο πίνακας κόστους υπολογίζεται ανά γραμμή και κάθε στοιχείο της γραμμή υπολογίζεται από αριστερά προς το δεξιά. Με αυτόν τον τρόπο μπορούμε να αποφύγουμε την χρήση του πίνακα E[i, j], χρησιμοποιώντας στη θέση του μια μεταβλητή η οποία θα μεταφέρει αυτό το κόστος στην επόμενη επανάληψη το οποίο θα αντικαθιστάτε με τον νέο υπολογισμό. Επομένως, αποφεύγουμε την χρήση ενός πίνακα που θα έπρεπε να αποθηκεύσουμε στην global memory και την επαναλαμβανόμενη επικοινωνία με αυτόν που επιφέρει καθυστέρηση στον αλγόριθμο. Στην θέση του χρησιμοποιήσαμε μια μεταβλητή η οποία αποθηκεύεται σε ένα καταχωρητή το οποίο είναι πολύ πιο γρήγορο από την global memory.
- **Υπολογισμός τυχαίων αριθμών.** Ο υπολογισμός των τυχαίων αριθμών για την δημιουργία των τυχαίων παραλλαγών του πίνακα ομοιότητας είναι μια ακόμα αλλαγή που πραγματοποιήθηκε σε αυτή την υλοποίηση. Στις δυο προηγούμενες υλοποιήσεις η δημιουργία των τυχαίων αριθμών που χρησιμοποιούνται για το νέο indexing των τυχαίων παραλλαγών πραγματοποιήθηκε στην κάρτα γραφικών μέσω την συνάρτησης curand. Σε αυτήν την υλοποίηση πραγματοποιήσαμε την παραγωγή των τυχαίων αριθμών την πραγματοποιήσαμε στον Host και στην συνέχεια τους μεταφέραμε στην GPU. Μετρώντας αυτή την διαδικασία παρατηρήσαμε ότι είναι ταχύτερη.
- **Μείωση των προσβάσεων στην Global Memory.** Στον υπολογισμό στου SW προσπαθήσαμε να κάνουμε χρήση όσο το δυνατών περισσότερο των καταχωρητές. Όλες οι προσβάσεις στην Global Memory πραγματοποιήθηκαν μόνο μια φορά και στη συνέχεια τα δεδομένα αποθηκεύτηκαν σε μεταβλητές ώστε να έχουμε μεγαλύτερη ταχύτητα διαβάσματος και εγγραφής.

Με τις αλλαγές παρατηρήσαμε επιτάχυνση σε σχέση με την προηγούμενη υλοποίηση. Στην πρώτη μας υλοποίηση η ανίχνευση ενός εφέ σε μια ταινία πραγματοποιήθηκε σε χρόνο 11771ms. Σε αυτή την υλοποίηση ο χρόνος ανίχνευσης ενός εφέ σε μια ταινία ήταν 5625 ms. Το σειριακό πρόγραμμα πραγματοποίησε την ανίχνευση σε 60800 ms. Επομένως, σε σχέση με το σειριακό έχουμε επιτάχυνση της τάξης 10x. Σε αυτή την υλοποίηση τρέξαμε τον αλγόριθμο και για παραπάνω δεδομένα. Τρέξαμε λοιπόν τον αλγόριθμο για περισσότερα εφέ. Η ανίχνευση των παραπάνω εφέ πραγματοποιήθηκε παράλληλα, δηλαδή για την ανίχνευση των επιπρόσθετων εφέ ενεργοποιήσαμε τα απαραίτητα επιπλέον νήματα, που είναι αναγκαία για τον υπολογισμό των πράξεων.

Αριθμός Εφέ	Χρόνος Εκτέλεσης
1	5625.165039 ms
2	8136.344238 ms
3	11764.581055 ms
4	14927.651367 ms
5	18464.052734 ms
6	22540.164063 ms

Στο παρακάτω διάγραμμα παρουσιάζεται ο χρόνος εκτέλεσης του προγράμματος σε σχέση με τον αριθμό των προτύπων που ανιχνεύουμε. Παρατηρούμε ότι ο χρόνος εκτέλεσης με την αύξηση των εφέ δεν διπλασιάζεται, όπως πραγματοποιείται σε ένα σειριακό σύστημα, αλλά ούτε μένει και σταθερός, που θα πραγματοποιούταν, αν το όφελος από την παραλληλία ήταν απόλυτο.



Έτσι σε σχέση με το σειριακό πρόγραμμα η επιτάχυνση παρουσιάζεται στο παρακάτω πίνακα:

Αριθμός Εφέ	Επιτάχυνση
1	10.8x
2	14.9x
3	15.5x
4	16.3x
5	16.5x
6	16.2x

6.6 Υλοποιήσεις με μειωμένες επιδόσεις

Στην παρακάτω ενότητα παρουσιάζουμε κάποιες από τις υλοποιήσεις που πραγματοποιήσαμε, αλλά δεν βελτίωσαν τις επιδόσεις της εφαρμογής. Αρχικά, μια από αυτές τις τροποποιήσεις, ήταν η μεγαλύτερη παραλληλοποίηση του αλγόριθμου S-W. Λόγω, όμως των πολλών ενεργοποιημένων νημάτων και της αναγκαίας και συχνής επικοινωνίας με την μνήμη δεν έχει το επιθυμητό αποτέλεσμα. Μια ακόμα αλλαγή που πραγματοποιήσαμε και είχε αρνητικό αποτέλεσμα στις επιδόσεις ήταν η δυναμική δέσμευση της μνήμης στο πεδίο του κάθε νήματος. Τέλος, ούτε η χρήση της κοινόχρηστης μνήμης είχε τα επιθυμητά αποτελέσματα όπως αναλύεται παρακάτω.

6.6.1 Αλλαγή της δομής και επέκταση της παραλληλοποίησης

Στην αρχική υλοποίηση το κάθε νήμα εκτέλεσε το Smith Waterman με πίνακα κόστους αντικατάστασης που έχει προκύψει για ένα παράθυρο του αρχείου ήχου της ταινίας και για το πρότυπο που θέλουμε να ανιχνεύσουμε. Στην συνέχεια το ίδιο νήμα δημιουργεί μια τυχαία σειρά δεικτιοδότησης για τα στοιχεία μιας στήλες και ανακατανέμει τα στοιχεία του πίνακα ομοιότητας ώστε να παραχθούν 100 τυχαίες παραλλαγές του πίνακα αντικατάστασης κόστους. Κατόπιν εκτελεί τον Smith Waterman για τις 100 τυχαίες παραλλαγές του πίνακα ομοιότητας.

Για να μειώσουμε τον φόρτο εργασίας του κάθε νήματος στην υλοποίηση αυτή αλλάξαμε την δομή του προγράμματος, έτσι ώστε να έχει μεγαλύτερη παραλληλία. Διαφοροποιήσαμε την δομή μας έτσι ώστε η εκτέλεση του αλγόριθμου για τις 100 αυτές παραλλαγές να πραγματοποιούνται από ένα νήμα η κάθε μία. Ένα ολόκληρο μπλοκ, λοιπόν στην συγκεκριμένη υλοποιήσει θα αναλάβει την εκτέλεση του αλγόριθμου για το πρότυπο και για ένα κομμάτι του ήχου της ταινίας. Το πρώτο νήμα του μπλοκ αναλαμβάνει να τρέξει τον αλγόριθμο για τον πραγματικό πίνακα ομοιότητας. Τα υπόλοιπα 100 νήματα του μπλοκ, αναλαμβάνουν να τρέξουν τον αλγόριθμο για τις 100 παραλλαγές του πίνακα ομοιότητας. Έτσι κάθε ένα thread εκτελεί τον αλγόριθμος 1 φορά και όχι 101 φορές όπως προηγουμένως. Ο kernel εκτελείται έτσι ώστε κάθε block αυτού να έχει 101 threads και υπολογίζει τους πίνακες κόστους μεταξύ ενός παραθύρου και των patterns. Παρακάτω παραθέτουμε τον κώδικα που υλοποίησε αυτή την αλλαγή:

```
if (threadIdx.x == 0){
        isPermutation = 0;
        SW(Dt, LongWindow + 1, NumPatternFrames + 1, 33, acc_cost, hor_cost, ver_cost,
isPermutation, p, s);
}else{
        isPermutation = 1;
        SW(Dt, II, JJ, 33, acc_cost, hor_cost, ver_cost,isPermutation, p, s);
}
for (i = 0; i < II; i++){
        for (j = 0; j < JJ; j++){
                if (acc_cost[i][j] > maxv_dev[idx]){
                        maxv_dev[idx] = acc_cost[i][j];
                }
        }
}
```

Παρατηρούμε λοιπόν ότι το νήμα με threadIdx.x = 0 θα εκτελέσει τον Smith Waterman για τον πραγματικό πίνακα ομοιότητας και τα υπόλοιπά threads του block θα εκτελέσουν τον Smith Waterman για τις τυχαίες παραλλαγές του Smith Waterman. Παρ' ότι αυτή η δομή του προγράμματος παρουσιάζει μεγαλύτερη παραλληλοποίηση και θα περιμέναμε να έχουμε ταχύτερο χρόνο εκτέλεσης, το πρόγραμμα είναι πιο αργό από την προηγούμενη υλοποίηση. Ο χρόνος που τρέχει το πρόγραμμα είναι 16067.883789 ms. Ο λόγος που πρέπει να παρατηρείται αυτό είναι λόγο της μεγάλης επικοινωνίας που υπάρχει μεταξύ της global memory και των threads. Από τη στιγμή που η μνήμη δεσμεύεται για παράδειγμα μέσο της στατικής εντολής

float acc_cost[windowPlusOne][patternsPlusOne]

,αυτός ο πίνακας αποθηκεύεται στην local memory του κάθε thread η οποία βρίσκεται στην global memory.

6.6.2 Δυναμική δέσμευση μνήμης

Στις προηγούμενες υλοποιήσεις θεωρήσαμε ότι το μέγεθος του παραθύρου του ήχου της ταινίας, ο αριθμός των διανυσμάτων που περιγράφουν το πρότυπο και ο αριθμός των παραλλαγών που θα τρέξει ο αλγόριθμος δίνονται στατικά από τον χρήστη και όχι δυναμικά. Για να εκτελέσουμε το πρόγραμμα δυναμικά για αυτές τις τιμές θα πρέπει να κάνουμε χρήση της εντολής malloc μέσα στον kernel. Η εντολή malloc είναι μια εντολή που χρησιμοποιείται στην γλώσσα C και μπορεί να χρησιμοποιηθεί και στην κάρτα γραφικών. Δεσμεύει ένα κομμάτι μνήμης και επιστρέφει ένα δείκτη στην αρχή του μπλοκ. Το κομμάτι της μνήμης δεν αρχικοποιείται και πρέπει να αρχικοποιηθεί από τον χρήστη. Η υλοποίηση, λοιπόν που πραγματοποιήσαμε με την δυναμική δέσμευση της μνήμης είχε σαν αποτέλεσμα να έχουμε πολύ μεγάλη καθυστέρηση. Η υλοποίηση αυτή ήταν πιο αργή, ακόμα και από την σειριακή υλοποίηση. Ο

λόγος που είχαμε αυτή την καθυστέρηση οφείλεται στο γεγονός ότι η μνήμη αυτή δεσμεύτηκε στην Global Memory και έτσι είχαμε μεγάλες καθυστερήσεις στην επικοινωνία με την αργή αυτή μνήμη.

Μια δεύτερη επιλογή για να έχουμε δυναμικό ορισμό μνήμης ήταν η δημιουργία των κατάλληλων πινάκων μέσο του host με την εντολή cudaMalloc στην global memory. Κάτι τέτοιο οδήγησε επίσης, στη δέσμευση μνήμης μόνο στην global memory της κάρτας γραφικών, με αποτέλεσμα να παρατηρούμε και σε αυτήν την υλοποίηση μεγάλη καθυστέρηση. Το γεγονός ότι χρειάζονται μεγάλοι πίνακες για την εκτέλεση του αλγορίθμου μας είναι το σημαντικότερο πρόβλημα της υλοποίησης του Smith-Waterman πάνω στην κάρτα γραφικών, αφού το μεγάλο μειονέκτημα της λειτουργίας της κάρτας γραφικών είναι η επικοινωνία με την μνήμη.

6.6.3 Χρήση κοινόχρηστη μνήμης

Ένα σημαντικό μειονέκτημα που έχουν μέχρι τώρα οι υλοποιήσεις μας ήταν ότι δεν κάναμε χρήση της κοινόχρηστης μνήμης(Shared Memory) της κάρτας γραφικών, η οποία είναι πολύ ταχύτερη από την global memory. Η κοινόχρηστη μνήμη είναι η ταχύτερη μνήμη που υπάρχει στην κάρτα γραφικών, μετά από τους καταχωρητές και η χρήση της προορίζεται για την συνεργασία των νημάτων ενός μπλοκ. Έτσι σε διεργασίες όπου τα νήματα ενός μπλοκ πρέπει να χρησιμοποιήσουν τα δεδομένα που υπάρχουν στην global memory πολλές φορές, προτιμάται η χρήση της κοινόχρηστης μνήμης. Για να γίνει χρήση της κοινόχρηστης μνήμης θα πρέπει τα δεδομένα από την global memory να μεταφερθούν στην shared memory, να γίνουν οι όποιες διεργασίες πάνω σε αυτή και την συνέχεια τα αποτελέσματα να μεταφερθούν πάλι στην global memory.

Γίνεται αντιληπτό ότι για να έχουμε όφελος από την χρήση της κοινόχρηστης μνήμης(shared memory), πρέπει στα δεδομένα που θα μεταφέρουμε σε αυτή, να πραγματοποιήσουμε πολλές διεργασίες. Πρέπει, δηλαδή τα δεδομένα αυτά να χρειάζεται να προσπελαστούν πολλές φορές, για να έχουμε όφελος από την μεταφορά τους στην κοινόχρηστη μνήμη. Αν για παράδειγμα κάποια δεδομένα που έχουμε στην global memory, χρειάζεται να προσπελαστούν μόνο μια φορά, δεν έχουμε όφελος να τα μεταφέρουμε στην κοινόχρηστη μνήμη. Στην περίπτωση μας τα δεδομένα που χρησιμοποιούνται επανειλημμένα είναι ο πίνακας κόστους αντικατάστασης. Ο πίνακας κόστους αντικατάστασής χρησιμοποιείται για να υπολογίζουμε τον πίνακα κόστους Smith – Waterman, για ένα κομμάτι του ήχου της ταινίας και για το πρότυπο που θέλουμε να ανιχνεύσουμε. Στη συνέχεια, τον ξανά χρησιμοποιούμε, αλλά με τυχαίες παραλλαγές των στοιχείων της κάθε στήλης και υπολογίζουμε τον πίνακας κόστους Smith – Waterman για αυτές τις τυχαίες παραλλαγές του αρχικού πίνακα. Επομένως, το κάθε στοιχείο του πίνακα θα διαβαστεί 101 φορές, 1 για τον πραγματικό πίνακα και 100 φορές για τις τυχαίες παραλλαγές του.

Η ταχύτερη υλοποίηση που έχουμε πραγματοποιήσει μέχρι στιγμής είναι αυτή στην οποία ένα νήμα πραγματοποιεί την υλοποίηση του αλγόριθμου Smith-Waterman για ένα κομμάτι ήχου της ταινία και του προτύπου, και των 100 παραλλαγών του πίνακα ομοιότητας. Έτσι ένα thread πραγματοποιεί 101 φορές τον αλγόριθμό Smith-Waterman. Ο πίνακας ομοιότητας λόγο του μεγάλου μεγέθους του δεν χωράει να αποθηκευτεί στην shared memory και έτσι αναγκαστικά αποθηκεύτηκε στην global memory.

Το μέγεθος του πίνακα ομοιότητας είναι αρκετά μεγάλο και όπως είδαμε παραπάνω δεν χωράει ολόκληρος στην κοινόχρηστη μνήμη. Για να καταφέρουμε να χρησιμοποιήσουμε την κοινόχρηστη μνήμη σπάσαμε τον πίνακα ομοιότητας σε δυο tiles έτσι ώστε να μπορεί να αποθηκευτεί σε αυτή. Έτσι ο αλγόριθμος θα τρέξει για το πρώτο μισό του πραγματικού πίνακα ομοιότητας και για τις 100 παραλλαγές, και στην συνέχεια θα τρέξει για το δεύτερο μισό του πίνακα. Σε αυτή την υλοποίηση αλλάξαμε και την δομή του προγράμματος έτσι ώστε ο αλγόριθμος να τρέχει για ένα κομμάτι ήχου και για το πρότυπο, από ένα ολόκληρο μπλοκ νημάτων και όχι από ένα μόνο νήμα. Το μπλοκ των νημάτων αποτελείται από 101 νήματα. Το πρώτο νήμα τρέχει τον αλγόριθμο για τον πραγματικό πίνακα ομοιότητας ενώ τα υπόλοιπα 100 thread τον τρέχουν για τις υπόλοιπες 100 παραλλαγές του πίνακα.

Μετά την υλοποίηση αυτή διαπιστώσαμε ότι ο χρόνος δεν βελτιώθηκε. Οι τυχαίες παραλλαγές του πίνακα ομοιότητας παράγονται δημιουργώντας μια τυχαία ανακατανομή στις στήλες του πίνακα ομοιότητας. Αυτή η τυχαία ανακατανομή γίνεται αλλάζοντας το indexing του αλγόριθμου SW. Για να δημιουργηθεί αυτό το τυχαίο Indexing, πρέπει να δημιουργηθούν, μέσα σε ένα πίνακα 201 στοιχείων αριθμοί από το 0 έως το 200 σε τυχαία κατανομή μέσα στον πίνακα. Επομένως, για τις 100 παραλλαγές του πίνακα ομοιότητας δημιουργούνται 100 τέτοιοι πίνακες 200 στοιχείων. Η επικοινωνία αυτή με την global memory για να πραγματοποιηθεί το indexing είναι πιθανώς ο λόγος που δεν παρατηρούμε την επιτάχυνση που περιμέναμε από την δομή αυτή της υλοποίησης.

7. Συμπεράσματα

Στην παρούσα διπλωματική αναπτύξαμε μια εφαρμογή, βασισμένη στην πλατφόρμα της CUDA, η οποία ανιχνεύει ηχητικά εφέ σε αρχεία ήχου ταινιών, με την χρήση του αλγόριθμου Smith - Waterman. Αρχικά παρουσιάσαμε τους λόγους για τους οποίους οι κάρτες γραφικών παρουσιάζουν μεγάλη εξέλιξη σε σχέση με τις CPU. Επίσης, παραθέτουμε πως οι κάρτες γραφικών μπορούν να μας βοηθήσουν σε προβλήματα που απαιτούν πολλού πόρους, λόγω των μεγάλων δυνατοτήτων σε παραλληλία. Στη συνέχεια μελετήσαμε την αρχιτεκτονική της κάρτας γραφικών του μοντέλου της CUDA. Παρουσιάσαμε τα βασικά μέρη της και το ρόλο που παίζουν για την βελτιστοποίησή των επιδόσεων. Μελετήσαμε το μοντέλο προγραμματισμού της CUDA, όπου ο προγραμματισμός πρέπει να γίνεται με γνώμονα την παραλληλοποίηση του εκάστοτέ προβλήματος και όχι με τον καθιερωμένο σειριακό τρόπο που προγραμματίζαμε μέχρι πρόσφατα.

Ο αλγόριθμος Smith – Waterman είναι ένας δυναμικός αλγόριθμος που η λύση του αρχικού προβλήματος προκύπτει από την λύση των επιμέρους προβλημάτων στα οποία έχει αναλυθεί το αρχικό. Η λύση του προβλήματος προκύπτει από τον υπολογισμό του πίνακα του σκορ ομοιότητας. Το κάθε στοιχείου του πίνακα εξαρτάται από τα στοιχεία του πίνακα που βρίσκονται πάνω και αριστερά από αυτό το στοιχείο. Επομένως, τα στοιχεία του πίνακα δεν είναι ανεξάρτητα μεταξύ τους, αφού για να υπολογιστεί ένα στοιχείο θα πρέπει να υπολογιστούν πρώτα τα στοιχεία που βρίσκονται πάνω και αριστερά του. Η παραλληλία λοιπόν που επιλέξαμε δεν είναι αυτή που θα υπολόγιζε τα στοιχεία του πίνακα σκορ ομοιότητας παράλληλα, αφού αυτά τα στοιχεία δεν είναι ανεξάρτητα μεταξύ τους και επομένως δεν μπορούν να υπολογιστούν παράλληλα. Η δομή της παραλληλίας που επιλέξαμε είναι να ανατεθεί σε κάθε νήμα ο υπολογισμός του πίνακα σκορ ομοιότητας για το εφέ και για ένα κομμάτι της ταινίας. Έτσι, όλα μαζί τα νήματα που εκτελούνται παράλληλα υπολογίζουν τον πίνακα για το εφέ και για όλα τα κομμάτια της ταινίας.

Ένα βασικό μειονέκτημα της υλοποίησης του αλγόριθμου Smith – Waterman πάνω σε μια κάρτα γραφικών CUDA είναι οι πολλές επικοινωνίες που απαιτεί με την μνήμη. Οι κάρτες γραφικών CUDA παρουσιάζουν μεγάλες επιδόσεις και μεγάλες επιταχύνσεις όταν οι επικοινωνία με την μνήμη είναι ελάχιστη. Ο αλγόριθμος, για τον υπολογισμό ενός στοιχείου του πίνακα σκορ ομοιότητας, πρέπει να διαβάσεις δυο στοιχεία από τρεις πίνακας. Για παράδειγμα από τον πίνακα ομοιότητας πρέπει να διαβάσει το ακριβώς από πάνω στοιχείο από αυτό που υπολογίζουμε και το ακριβώς αριστερά. Αυτή η επικοινωνίας με την μνήμη έχει σαν αποτέλεσμα να μην μπορούμε να εκμεταλλευτούμε πλήρως τις δυνατότητες της κάρτας γραφικών.

Ένα άλλο μειονέκτημα ήταν η μη δυνατότητα χρήση της ταχύτερης κοινόχρηστης μνήμης. Στην υλοποίηση του Smith – Waterman στη βιολογία έχουμε μια σημαντική διαφορά με την περίπτωση που θέλουμε να ανιχνεύσουμε ηχητικά εφέ. Στην βιολογία ο πίνακας κόστους αντικατάστασης είναι κοινός για όλες τις συγκρίσεις των υποακολουθιών που πραγματοποιούμε. Για παράδειγμα ο πίνακας BLOSUM62 είναι ένας πίνακας κόστους αντικατάστασης που χρησιμοποιείται στην βιολογία και μας δίνει το κόστος να αντικαταστήσουμε ένα στοιχείο της ακολουθίας με ένα άλλο. Αυτός, ο πίνακας είναι κοινός για όλες τις συγκρίσεις που κάνουμε και μπορεί εύκολα να αποθηκευτεί στην κοινόχρηστη μνήμη και να χρησιμοποιηθεί από όλα τα νήματα. Στην περίπτωση μας όμως, ο πίνακας κόστους αντικατάστασης είναι διαφορετικός για κάθε ζευγάρι σύγκρισης και είναι πολύ μεγαλύτερος σε μέγεθος. Έτσι, από την μια θα χρησιμοποιηθεί μόνο από ένα νήμα για ένα ζευγάρι σύγκρισης και δεν έχουμε όφελος από την μεταφορά του στην κοινόχρηστη μνήμη. Αυτό μας αναγκάζει να τον αφήσουμε στην global memory όπου η επικοινωνία μαζί της είναι αργή με αποτέλεσμα να μην εκμεταλλευόμαστε πλήρως τις δυνατότητες της κάρτας γραφικών.

Παρ' όλους τους παραπάνω περιορισμούς η εφαρμογής μας παρουσίασε μια επιτάχυνσης της τάξεως 12x σε σχέση με την αντίστοιχη σειριακή υλοποίηση. Το κάθε νήμα εκτέλεσε τον αλγόριθμο Smith – Waterman για το εφέ και ένα κομμάτι της ταινίας. Έτσι όλα μαζί τα νήματα εκτέλεσαν τον αλγόριθμο για το εφέ και για όλα τα κομμάτια της ταινίας παράλληλα, σε αντίθεση με το σειριακό πρόγραμμα όπου εκτελείται ο αλγόριθμος Smith – Waterman για κάθε κομμάτι της ταινίας σειριακά.

8. Βιβλιογραφία

[1]. Mihalis Psarakis, Aggelos Pikrakis, Giannis Dendrinos, "FPGA-based Acceleration for Tracking Audio Effects in Movies", 2012 IEEE 20th International Symposium on Field-Programmable Custom Computing Machines
[2]. T. F .Smith, M. S. Waterman, Identification of Common Molecular Subsequences, J .Mol. Biol. (1981)
[3]. Yongchao Liu, Adrianto Wirawan, Bertil Schmidt, "CUDASW++ 3.0: accelerating Smith-Waterman protein database search by coupling CPU and GPU SIMD instructions", BMC Bioinformatics 2013
[4]. Balaji Venkatachalam, "Parallelizing the Smith-Waterman Local Alignment Algorithm using CUDA", February 28, 2012
[5]. Ali Khajeh-Saeed a, Stephen Poole b, J. Blair Perot a, "Acceleration of the Smith–Waterman algorithm using single and multiple graphics processors", Journal of Computational Physics 229 (2010)
[6]. Yongchao Liu, Douglas L Maskell and Bertil Schmidt, "CUDASW++: optimizing Smith-Waterman sequence database searches for CUDA-enabled graphics processing units", BMC Research Notes 2009
[7]. Wen-mei W. Hwu, "GPU Computing Gems Emerald Edition"
[8]. David B. Kirk and Wen-mei W. Hwu, "Programming Massively Parallel Processors"
[9]. John Cheng, Max Grossman, Ty McKercher, "Professional CUDA® C Programming"
[10]. Shane Cook, "CUDA Programming A Developer's Guide to Parallel Computing with GPUs", Morgan Kaufman Publications 2013
[11]. NNIDIA Whitepaper, "NVIDIA's Next Generation CUDATM Compute Architecture: FermiTM"
[12]. Nicholas Wilt, "The CUDA Handbook: A Comprehensive Guide to GPU Programming", Addison – Wesley
[13]. Rob Farber, "CUDA Application Design and Development", Morgan Kaufman Publications 2011
[14]. Jason Sanders, Edward Kandrot, "CUDA by Example: An Introduction to General-Purpose GPU Programming", Addison - Wesley