ΑΛΕΞΑΝΔΡΕΙΟ Τ.Ε.Ι. ΘΕΣΣΑΛΟΝΙΚΗΣ
ΣΧΟΛΗ ΤΕΧΝΟΛΟΓΙΚΩΝ ΕΦΑΡΜΟΓΩΝ
ΤΜΗΜΑ ΠΛΗΡΟΦΟΡΙΚΗΣ

ΠΤΥΧΙΑΚΗ ΕΡΓΑΣΙΑ

<< ΜΕΛΕΤΗ ΤΗΣ ΑΡΧΙΤΕΚΤΟΝΙΚΗΣ CUDA ΚΑΙ ΠΑΡΑΛΛΗΛΟΣ ΠΡΟΓΡΑΜΜΑΤΙΣΜΟΣ ΚΑΡΤΩΝ GPU ΤΗΣ NVIDIA >>

Της φοιτήτριας

ΚΕΛΑΪΔΗ ΑΙΜΙΛΙΑ

Αρ. Μητρώου: 04/2519

Επιβλέπων καθηγητής

ΔΙΑΜΑΝΤΑΡΑΣ ΚΩΝ/ΝΟΣ

Θεσσαλονίκη 2012

ΠΡΟΛΟΓΟΣ

Η εκπόνηση της πτυχιακής εργασίας μου αποτέλεσε μια πολύ ενδιαφέρουσα εμπειρία και μια πρώτης τάξεως ευκαιρία για μένα να εμβαθύνω σε ζητήματα παράλληλου προγραμματισμού, ο οποίος -ούτως ή άλλως- βρίσκεται στο προσκήνιο εδώ και αρκετά χρόνια.

Υπάρχει πληθώρα προγραμματιστικών εργαλείων για τη συγγραφή παράλληλου κώδικα, η συγκεκριμένη όμως εργασία στοχεύει στην ανάλυση του μοντέλου CUDA που αναπτύχθηκε από τη Nvidia το 2006. Η πλατφόρμα CUDA χρησιμοποιείται για την ανάπτυξη εφαρμογών, που απαιτούν τη συνεργατική εκτέλεση κώδικα μεταξύ των συστημάτων κεντρικής επεξεργασίας και των καρτών γραφικών.

Πρόκειται επομένως, για την μελέτη ενός μοντέλου παράλληλης επεξεργασίας, που αφορά τη συνδυασμένη εκτέλεση κάποιων τμημάτων κώδικα από τους πυρήνες των κεντρικών επεξεργαστών και ορισμένων άλλων τμημάτων από τους πυρήνες των επεξεργαστών της κάρτας γραφικών. Απαιτείται, αφενός η κατάλληλη αλγοριθμική δομή με όσο το δυνατόν μεγαλύτερη έκθεση της παραλληλίας και αφετέρου η επαρκής γνώση της αρχιτεκτονικής της κάρτας, που θα εκτελέσει τον αλγόριθμο.

Για την αποδοτικότερη εκτέλεση ενός παράλληλου αλγορίθμου πρέπει να είμαστε σε θέση να εφαρμόσουμε τις κατάλληλες προγραμματιστικές τεχνικές, που θα ελαχιστοποιούν τη σειριακή εκτέλεση των εντολών. Η εργασία μου καλύπτει και αυτήν τη θεματική ενότητα, καθώς η συγγραφή κώδικα για παράλληλη εκτέλεση χωρίς ταυτόχρονη προσπάθεια για βελτίωση της απόδοσης δεν έχει ουσιαστικά πρακτική σημασία.

Η αναφορά στους τομείς χρήσης του μοντέλου προγραμματισμού CUDA σε πληθώρα επιστημονικών εφαρμογών, τις μεγάλες επιστημονικές προκλήσεις, όπως ονομάζονται χαρακτηριστικά, δεν αποτελεί απλώς το επισφράγισμα της συγγραφής της παρούσης εργασίας, αλλά το έναυσμα για περαιτέρω ενασχόληση με τον τόσο συναρπαστικό κόσμο της παράλληλης υπολογιστικής.

ΠΕΡΙΛΗΨΗ

Κατά την ανάπτυξη και ολοκλήρωση της πτυχιακής εργασίας μου ερευνήθηκαν ποικίλα θέματα. Αρχικά, έγινε μια εισαγωγή στην αρχιτεκτονική των παραδοσιακών επεξεργαστικών συστημάτων και στη συνέχεια αναπτύχθηκαν θέματα, που περιλαμβάνουν την οργάνωση παράλληλων συστημάτων. Από τα πιο σημαντικά θέματα που μελετήθηκαν ήταν η αρχιτεκτονική μιας κάρτας γραφικών και ο τρόπος με τον οποίο γίνεται η επεξεργασία των δεδομένων σε αυτήν.

Το σημαντικότερο τμήμα της πτυχιακής μου εργασίας αφορά τον παράλληλο προγραμματισμό και πιο συγκεκριμένα την πλατφόρμα παράλληλου προγραμματισμού CUDA της NVIDIA. Το πακέτο CUDA είναι, επί του παρόντος, το πιο διαδεδομένο API για εφαρμογές που περιλαμβάνουν εκτέλεση προγραμμάτων με συνεργασία μεταξύ κάρτας γραφικών και κεντρικού συστήματος επεξεργασίας. Ως γλώσσα αναφοράς της πλατφόρμας CUDA χρησιμοποιήθηκε κατ' επιλογήν η γλώσσα C. Επίσης, μελετήθηκαν και διάφορα παραδείγματα αλγορίθμων γραμμένα σε αυτήν τη γλώσσα.

Καθώς τα εργαλεία προγραμματισμού της CUDA παράγουν κώδικα, ο οποίος προσπαθεί να εκμεταλλευτεί όσο το δυνατόν πιο αποδοτικά την αρχιτεκτονική μιας κάρτας γραφικών, πρέπει να έχουμε τη δυνατότητα να διαχειριστούμε με κατάλληλο τρόπο τους πόρους της κάρτας και τις δυνατότητες της παράλληλης εκτέλεσης του κώδικα. Για να εκμεταλλευτούμε τις τεράστιες υπολογιστικές δυνατότητες, που προσφέρονται από την ανάπτυξη συστημάτων παράλληλης επεξεργασίας, πρέπει να είμαστε ακόμη σε θέση να αξιοποιήσουμε κατάλληλες προγραμματιστικές τεχνικές, που θα εκθέτουν όσο το δυνατό μεγαλύτερο ποσοστό παραλληλίας των εφαρμογών. Προς αυτήν την κατεύθυνση μελετήθηκαν ζητήματα και τεχνικές βελτιστοποίησης κώδικα, για παράλληλη εκτέλεση στους επεξεργαστές μιας κάρτας γραφικών

Μετά τη μελέτη της CUDA ακολουθεί μια σειρά από συμπεράσματα και εμπειρίες που αποκόμισα από την εκπόνηση της εργασίας μου, καθώς επίσης και μια σύντομη αναφορά στη σημασία που έχει η χρήση της πλατφόρμας CUDA σε μεγάλο πλήθος επιστημονικών εφαρμογών.

ΕΥΧΑΡΙΣΤΙΕΣ

Μέσα στο πλαίσιο της εκπόνησης της πτυχιακής εργασίας μου θεωρώ, ότι είναι απαραίτητο να απευθύνω τις θερμές μου ευχαριστίες προς τον καθηγητή του μαθήματος "Προηγμένων Αρχιτεκτονικών και Παράλληλων Συστημάτων" του ΑΤΕΙ Θεσσαλονίκης και επιβλέποντα καθηγητή μου, Δρ. Κωνσταντίνο Διαμαντάρα, ο οποίος με βοήθησε ιδιαίτερα με τις πολύτιμες συμβουλές του.

Ως επιβλέπων καθηγητής και έχοντας ο ίδιος μεγάλη εμπειρία και γνώση σε ζητήματα επεξεργασίας, αρχιτεκτονικής και οργάνωσης υπολογιστών, κατάφερε με τις παρεμβάσεις του να με βοηθήσει να κατανοήσω πολύ σημαντικά θέματα, που αφορούν τον παράλληλο προγραμματισμό και την οργάνωση ενός παράλληλου συστήματος γενικότερα. Η ανάπτυξη και η ολοκλήρωση της παρούσης εργασίας δεν θα ήταν δυνατή χωρίς την πολύτιμη συνεργασία και προτροπή του ίδιου.

Επίσης, θα ήθελα να τον ευχαριστήσω για την προσφορά πολύτιμης σχετικής βιβλιογραφίας και οπωσδήποτε για την ευκαιρία που μου έδωσε -μέσω του συγκεκριμένου θέματος της πτυχιακής εργασίας- να εμβαθύνω σε θέματα παράλληλης υπολογιστικής και προγραμματισμού. Από την ενασχόλησή μου με την παρούσα εργασία, εκτιμώ ότι ο παράλληλος προγραμματισμός είναι ένας τομέας που βρίσκεται ακόμα υπό ανάπτυξη και οτι στο εγγύς μέλλον θα είναι σε θέση να επηρεάσει και να μεταβάλλει πολλές προσεγγίσεις, οι οποίες χρησιμοποιούνται μέχρι σήμερα στην υπολογιστική επιστήμη.

Τέλος, θα ήθελα να ευχαριστήσω ιδιαίτερα τα μέλη της οικογένειάς μου, για την υποστήριξη και τη βοήθειά τους, καθώς βρίσκονταν πάντα δίπλα μου κατά τη διάρκεια των σπουδών μου και κατά την εκπόνηση της πτυχιακής μου εργασίας.

ΠΕΡΙΕΧΟΜΕΝΑ

Ευρετήριο σχημάτων

Ευρετήριο πινάκων

Κεφάλαιο 1

Εισαγωγή

Υποκεφάλαιο 1.1
ΚΙΝΗΤΡΟ ΠΤΥΧΙΑΚΗΣ ΕΡΓΑΣΙΑΣ

Η σημασία της παράλληλης υπολογιστικής έχει αρχίσει να διαφαίνεται εδώ και αρκετό καιρό. Η τάση αυτή ενισχύθηκε πλέον οριστικά, μετά την εμπορική διάθεση των πολυπύρηνων επεξεργαστών, οι οποίοι ανέδειξαν ακόμη περισσότερο τους λόγους για τους οποίους η παράλληλη επεξεργασία υπερτερεί κατά πολύ έναντι της απλής παραδοσιακής επεξεργασίας.

Ένας από τους κύριους λόγους, για τους οποίους η παράλληλη επεξεργασία χρησιμοποιείται ευρέως στις μέρες μας αφορά την πολυπλοκότητα των αλγορίθμων και το τεράστιο πλήθος δεδομένων, που καλείται να διαχειριστεί. Εφαρμογές, όπως για παράδειγμα η χαρτογράφηση των ωκεανών ή η μελέτη ουράνιων σωμάτων απαιτούν την εκτέλεση χρονοβόρων και περίπλοκων υπολογισμών, σε πολύ μεγάλο πλήθος δεδομένων. Το γεγονός αυτό καθιστά την παραδοσιακή επεξεργασία εξαιρετικά χρονοβόρα και ασύμφορη. Αντίθετα, χάρη στη μείωση του κόστους κυκλωμάτων και υλικού γενικότερα, η χρήση ενός παράλληλου συστήματος καθίσταται ιδιαίτερα συμφέρουσα πρόταση.

Η παράλληλη επεξεργασία παρ' όλα αυτά δεν βρίσκει εφαρμογή μόνο σε προβλήματα συγκεκριμένων επιστημονικών πεδίων, αλλά μπορεί να χρησιμοποιηθεί και για την ταχύτατη εκτέλεση αλγορίθμων, οι οποίοι θα εκτελούνται συνεργατικά. Μέσω συγκεκριμένων προγραμματιστικών εργαλείων –το μοντέλο CUDA είναι ένα από αυτά– μπορεί να συνταχθεί κώδικας του οποίου τα σειριακά μέρη θα εκτελούνται από τους πολυπύρηνους επεξεργαστές του κεντρικού συστήματος, ενώ τα παράλληλα τμήματα θα εκτελούνται, επίσης ταχύτατα, από τους επεξεργαστές μιας ή περισσότερων καρτών γραφικών, που θα είναι συνδεδεμένες με το κεντρικό σύστημα.

Το πακέτο CUDA είναι ένα πολλά υποσχόμενο μοντέλο για παράλληλο προγραμματισμό γενικού σκοπού σε κάρτες γραφικών. Παραδοσιακά βέβαια, οι κάρτες γραφικών διαχειρίζονται και παράγουν δεδομένα γραφικών, αλλά με τη βοήθεια του μοντέλου αυτού μπορούν να χρησιμοποιηθούν και για πληθώρα άλλων προβλημάτων, που ουδεμία σχέση έχουν με αλγορίθμους γραφικής απεικόνισης (εξ ου και ο όρος προγραμματισμός γενικού σκοπού).

Κυρίαρχη πρόκληση στο σημείο αυτό αποτελεί η ενασχόληση και η περαιτέρω εξοικείωση με το συγκεκριμένο εργαλείο προγραμματισμού, τη γνώση και τη διαχείριση των πόρων μιας κάρτας γραφικών, καθώς επίσης και την αποτελεσματική τους αξιοποίηση για την παραγωγή υψηλής ποιότητας κώδικα.

Υποκεφάλαιο 1.2
ΣΤΟΧΟΣ ΠΤΥΧΙΑΚΗΣ ΕΡΓΑΣΙΑΣ

Η παρούσα πτυχιακή εργασία έχει σαν σκοπό να παρουσιάσει σε ικανοποιητικό βαθμό τις γενικές έννοιες που χρησιμοποιούνται στην παράλληλη υπολογιστική, όπως τα νήματα και οι πυρήνες. Στη συνέχεια, μελετά τον τρόπο με τον οποίο δομείται ένα σύστημα που υποστηρίζει το μοντέλο CUDA. Αυτό το τμήμα είναι πολύ βασικό προκειμένου να μπορέσει να γίνει κατανοητός ο τρόπος εκτέλεσης των αλγορίθμων σε μια κάρτα γραφικών.

Δεδομένου ότι η γλώσσα C αποτελεί μια ευρέως διαδεδομένη γλώσσα προγραμματισμού, θεωρείται κατά ένα μεγάλο ποσοστό γνωστή και επομένως στην πτυχιακή εργασία δεν γίνεται μια εκ νέου ανάλυση της γλώσσας αυτής. Ως εκ τούτου, αναφέρονται μόνο οι καινούργιες έννοιες/προσθήκες που αφορούν τις επεκτάσεις της C οι οποίες χρησιμοποιούνται στη CUDA. Το πιο σημαντικό τμήμα της πτυχιακής καλύπτει τις θεματικές ενότητες που αναλύουν τις βασικές συναρτήσεις και λειτουργίες του μοντέλου CUDA και τις τεχνικές με τις οποίες μπορεί κάποιος να συντάξει αποδοτικότερο κώδικα.

Υποκεφάλαιο 1.3
ΟΡΓΑΝΩΣΗ ΚΕΙΜΕΝΟΥ

Στη συνέχεια της πτυχιακής εργασίας, που ξεκινά με το Κεφάλαιο 2, γίνεται μια εισαγωγική αναφορά στα απλά συστήματα μονοεπεξεργαστών που αποτελούν τον πυλώνα της σημερινής υπολογιστικής και με κανένα τρόπο δε θα μπορούσαν να παραληφθούν. Το Κεφάλαιο 3 αναλύει τις βασικές αρχιτεκτονικές ενός παράλληλου συστήματος ενώ στο Κεφάλαιο 4 περιγράφονται τα διάφορα μοντέλα παράλληλου προγραμματισμού και γίνεται η τοποθέτηση του μοντέλου CUDA στην κατάλληλη κατηγορία αναφορικά με την αρχιτεκτονική που χρησιμοποιεί ως πλατφόρμα. Στα Κεφάλαια 5 και 6 μελετάται εκτενώς η πλατφόρμα προγραμματισμού CUDA, ενώ στο Κεφάλαιο 7 περιγράφονται οι τεχνικές βελτιστοποίησης κώδικα γραμμένου σε CUDA. Το Κεφάλαιο 8 τέλος, αναφέρεται στην εξαγωγή συμπερασμάτων και προτάσεων ή προβλημάτων που συνάντησα κατά τη φάση της εκπόνησης της πτυχιακής μου εργασίας.

Κεφάλαιο 2

Απλή και Μαζική Επεξεργασία

Υποκεφάλαιο 2.1
ΒΑΣΙΚΑ ΜΕΡΗ Η/Υ – ΕΠΙΓΡΑΜΜΑΤΙΚΗ ΑΝΑΔΡΟΜΗ

Η ιστορία της επιστήμης των υπολογιστών ξεκινάει από το 1945,πολύ αργότερα σε σχέση με άλλες επιστήμες, αλλά παρόλα αυτά η εξέλιξή της είναι ραγδαία και συνεχίζεται με αυτόν το ρυθμό, ειδικά τις τελευταίες δεκαετίες. Τα βήματα της εξέλιξης αυτής μπορούν να συνοψισθούν ως εξής :

- Οι υπολογιστές, που κατασκευάζονταν τις δύο πρώτες δεκαετίες (1950-1960) ήταν - συγκριτικά με τα σημερινά υπολογιστικά συστήματα- σαφώς μεγαλύτεροι και με πολύ μικρότερες υπολογιστικές δυνατότητες. Για τους υπολογισμούς τους χρησιμοποιούσαν διακόπτες ή τρανζίστορς (ειδικές ημιαγώγιμες διατάξεις) μαζί με αντιστάσεις και πυκνωτές που τοποθετούνταν πάνω σε εκτυπωμένα κυκλώματα.

- Οι υπολογιστές της επόμενης γενιάς (1964-1970) αποτελούνται από κυκλώματα που ενσωματώνουν την τεχνολογία SSI, δηλαδή η μικρής κλίμακας ολοκλήρωση ψηφιακών κυκλωμάτων. Χάρη σε αυτήν την τεχνολογία μπορούσαν να τοποθετηθούν ηλεκτρονικές πύλες, διάφορα άλλα βοηθητικά στοιχεία και οι μεταξύ τους συνδέσεις, όλα πάνω σε μια λεπτή φέτα ημιαγώγιμου υλικού δημιουργώντας έτσι ολοκληρωμένα κυκλώματα (chips). Πλέον, οι υπολογιστές αποτελούνται από ολοκληρωμένα κυκλώματα, που και αυτά με τη σειρά τους αποτελούνται από 8-100 διακεκριμένα ψηφιακά στοιχεία.

- Τα υπολογιστικά συστήματα 4ης και 5ης γενιάς (από το 1970 και έπειτα) χαρακτηρίζονται από την εισαγωγή νέων τεχνολογιών ολοκλήρωσης ψηφιακών κυκλωμάτων, όπως η LSI, VLSI και ULSI, τεχνολογίες που επιτρέπουν την κατασκευή κυκλωμάτων με ακόμη μεγαλύτερη κλίμακα ολοκλήρωσης. Ένα πολύ σημαντικό γεγονός, που έθεσε τις βάσεις για την εξέλιξη των μικροϋπολογιστικών συστημάτων είναι η εμφάνιση, το 1971, του πρώτου LSI γενικού σκοπού -του πρώτου μικροεπεξεργαστή μέσα σε chip- από την εταιρεία Intel. Επρόκειτο για τον Intel 4004, τον πρώτο εμπορικά διαθέσιμο μικροεπεξεργαστή, με 4 bits εύρος διαύλου, συχνότητα ρολογιού στα 740kHz και επίδοση 0.07 MIPS (Million Instructions Per Second).

Όλα τα υπολογιστικά συστήματα (ανεξαρτήτως τεχνολογίας και γενιάς) είναι κατασκευασμένα με βάση ένα συγκεκριμένο μοντέλο αρχιτεκτονικής, το μοντέλο **von Neumann,** που περιγράφει έναν υπολογιστή αποθηκευμένου προγράμματος. Με βάση αυτήν την αρχιτεκτονική, ένα υπολογιστικό σύστημα αποτελείται από τις εξής τέσσερις βασικές μονάδες:

- Την **Κεντρική Μονάδα Επεξεργασίας (CPU)** που αποτελεί τον παραδοσιακό επεξεργαστή που χρησιμοποιείται στους προσωπικούς υπολογιστές και ο οποίος ελέγχει τη σωστή λειτουργία του συστήματος και εκτελεί εντολές, οι οποίες επενεργούν πάνω στα δεδομένα.

- την **Κεντρική Μνήμη (Main Memory)**, εκεί όπου γίνεται η αποθήκευση των εντολών ελέγχου (προγράμματος) και των δεδομένων τους.

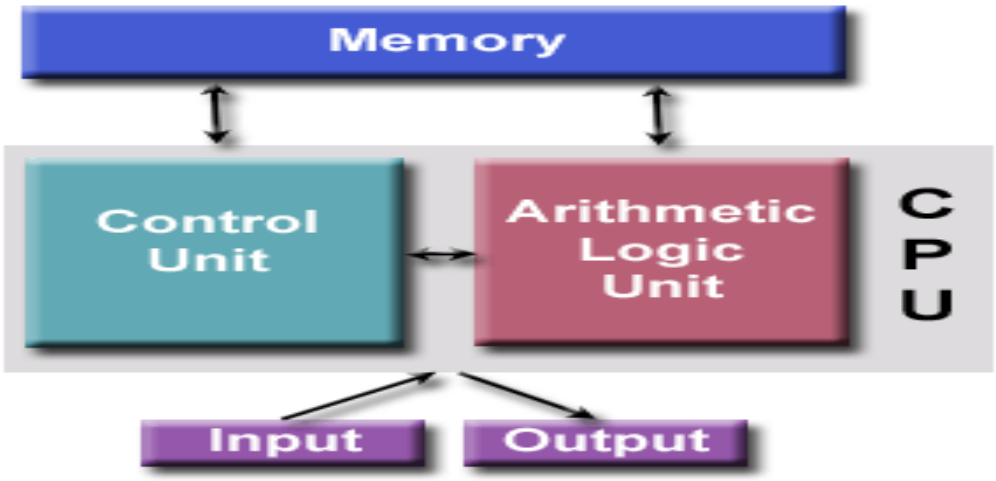
- τις **συσκευές I/O (I/O Components)** όπως για παράδειγμα οι σκληροί δίσκοι, το πληκτρολόγιο, η οθόνη , η κάρτα δικτύου κτλ. Αυτές οι μονάδες είναι υπεύθυνες για τη μεταφορά δεδομένων από τον υπολογιστή προς το εξωτερικό του περιβάλλον και αντίστροφα.

- το **δίαυλο συστήματος (System Bus)** που αποτελεί το μηχανισμό μεταφοράς δεδομένων και επικοινωνίας μεταξύ των τριών παραπάνω οντοτήτων.

Μια μεγαλύτερη **μητρική πλακέτα (motherboard chip)** είναι αυτή που ενσωματώνει όλα αυτά τα δομικά στοιχεία και καθιστά δυνατή τη μεταξύ τους επικοινωνία, αλλά και την επικοινωνία αυτών με τα υπόλοιπα μέρη, που αποτελούν το υπολογιστικό μας σύστημα.

Όλα αυτά τα στοιχεία που αναφέρθηκαν παραπάνω αποτελούνται και αυτά με τη σειρά τους από άλλες υπομονάδες, οι οποίες αναλαμβάνουν την εκτέλεση πιο συγκεκριμένων λειτουργιών. Επιγραμματικά μπορούμε να αναφερθούμε στην **Κεντρική Μονάδα Επεξεργασίας** οι οποία περιλαμβάνει:

- Την **Αριθμητική και Λογική Μονάδα (ALU)** οι οποία εκτελεί τις αριθμητικές και λογικές πράξεις.
- Τους **καταχωρητές (registers)** που χρησιμοποιούνται για την αποθήκευση και την ανάκτηση δεδομένων. Οι καταχωρητές έχουν πολύ μικρή χωρητικότητα, αλλά διαθέτουν πολύ μικρούς χρόνους προσπέλασης δεδομένων.
- Τη **Μονάδα Ελέγχου (Control Unit)** που μεταφράζει τις εντολές προγράμματος σε εσωτερικά σήματα προς την ALU
- Τον **Μετρητή Προγράμματος (Program Counter)**, ο καταχωρητής που αποθηκεύει κάθε φορά τη διεύθυνση μνήμης στην οποία υπάρχει η επόμενη προς εκτέλεση εντολή.
- Την **Μονάδα Εκτέλεσης Πράξεων Κινητής Υποδιαστολής (FPU)**
- Την **τοπική μνήμη** (η οποία αποτελείται από ένα σύνολο καταχωρητών και την κρυφή μνήμη cache)



Σχήμα 2.1.1 : Βασικές μονάδες ενός Η/Υ- μοντέλο von Neumann

Το σημαντικότερο στοιχείο σε αυτό το μοντέλο υπολογιστικών συστημάτων είναι ότι η επεξεργασία εντολών και δεδομένων ακολουθεί πάντα μια συγκεκριμένη ακολουθία βημάτων, το **βασικό κύκλο εντολής,** όπως αναφέρεται χαρακτηριστικά.

Ο **βασικός κύκλος εντολής** περιγράφει τη διαδικασία, βάσει της οποίας γίνεται η εκτέλεση μιας εντολής και περιλαμβάνει τα παρακάτω βήματα-φάσεις:

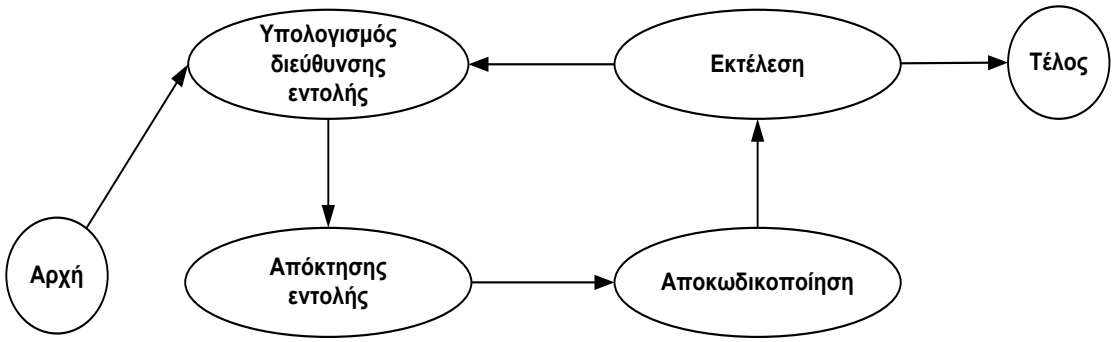
Βήμα 1: Φάση προσκόμισης εντολής: ο επεξεργαστής παραλαμβάνει την εντολή που θέλει να εκτελέσει από τη μνήμη (όπου βρίσκονται αποθηκευμένες οι προς εκτέλεση εντολές)

Βήμα 2: Φάση αποκωδικοποίησης εντολής:

Βήμα 3: Φάση εκτέλεσης εντολής: πριν παραλάβει την επόμενη εντολή ελέγχει αν υπάρχει κάποια άλλη διεργασία, που πρέπει να εκτελεστεί πριν από την εντολή.

Βήμα 4: Ενημέρωση του Μετρητή Προγράμματος και επανάληψη της διαδικασίας ξεκινώντας από το Βήμα 1.

0011



Σχήμα 2.1.2: Βασικός κύκλος εντολής

Το μοντέλο αρχιτεκτονικής υπολογιστή του **Neumann** αποτελεί τη βάση πάνω στην οποία αναπτύσσονται, κατά κύριο λόγο, τα υπολογιστικά συστήματα μέχρι και σήμερα.

Παράλληλα, και σε συνδυασμό με το μοντέλο αυτό, η εξελικτική πορεία της αρχιτεκτονικής υπολογιστών έχει στηριχθεί σε τρεις βασικές αρχές. Ο λόγος για τον οποίο αξίζει να αναπτύσσουμε αρχιτεκτονικές με βάση αυτές τις αρχές είναι **η αύξηση της υπολογιστικής επίδοσης των μηχανών**.

Οι αρχές αυτές είναι:

- Το **pipelining**, δηλαδή η εκτέλεση των βημάτων πολλαπλών εντολών ανά κύκλο μηχανής, από μία κεντρική επεξεργαστική μονάδα ακόμα και σε διαφορετικές φάσεις εκτέλεσης. Ουσιαστικά, πρόκειται για την εκμετάλλευση της δυνατότητας για παράλληλης εκτέλεσης εντολών.
- Ο **παραλληλισμός**, ο οποίος με τη χρήση πολλών επεξεργαστικών μονάδων επιτρέπει την εκτέλεση πολλαπλών εντολών
- Η **αρχή της τοπικότητας των κλήσεων** στην οποία στηρίζεται το μοντέλο της κρυφής (cache memory) και της εικονικής μνήμης (virtual memory).

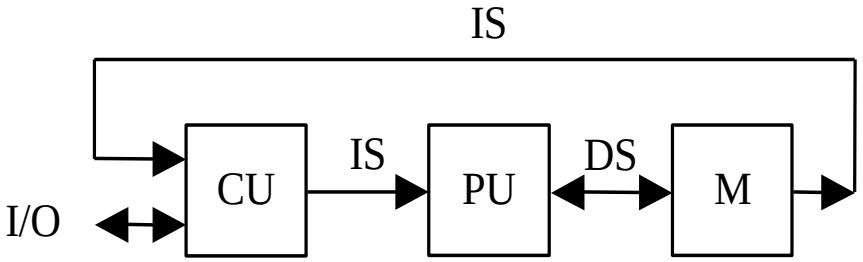
Υποκεφάλαιο 2.2
ΒΑΣΙΚΕΣ ΑΡΧΙΤΕΚΤΟΝΙΚΕΣ Η/Υ – ΤΑΞΙΝΟΜΗΣΗ ΚΑΤΑ FLYNN

Σχετικά με την υπολογιστική αρχιτεκτονική, η βασικότερη ταξινόμηση είναι αυτή του Flynn. Η κατηγοριοποίηση αυτή γίνεται με βάση το είδος και το πλήθος των εντολών, δηλαδή των ακολουθιών εντολών (Instruction Streams) και των προς επεξεργασία δεδομένων, δηλαδή των ακολουθιών δεδομένων (Data Streams) .

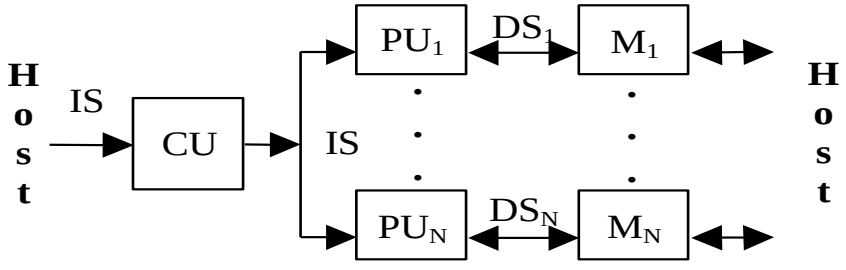
Οι τέσσερις κατηγορίες διάκρισης των υπολογιστών κατά τον Flynn είναι επιγραμματικά:

- **SISD (Single Instruction Stream – Single Data Stream)**
- **SIMD (Single Instruction Stream – Multiple Data Stream)**
- **MISD (Multiple Instruction Stream – Single Data Stream)**
- **MIMD (Multiple Instruction Stream – Multiple Data Stream)**

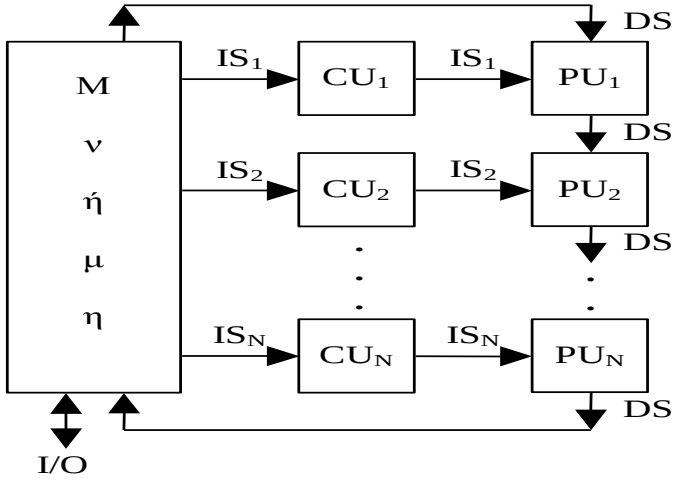
Τα μοντέλα SIMD και MIMD αποτελούν τα βασικά μοντέλα που περιγράφουν τα παράλληλα συστήματα γι' αυτό και θα αναλυθούν διεξοδικά. Το μοντέλο MISD δεν έχει υλοποιηθεί στην πράξη (τουλάχιστον μέχρι στιγμής) οπότε δεν αποτελεί αυτόνομο υπολογιστικό σύστημα. Παρόλα αυτά, η φιλοσοφία του χρησιμοποιείται ευρέως στην τεχνική pipeline μέσα σε υπομονάδες κεντρικών επεξεργαστών (για παράδειγμα στο pipeline εντολών). Σχετικά με το μοντέλο SISD, αυτό πρόκειται για το μοντέλο υπολογιστή της κλασικής επεξεργασίας, το οποίο ακολουθεί το βασικό κύκλο εντολής, όπως περιγράφηκε πιο πάνω. Για τους λόγους που προαναφέρθηκαν, τα μοντέλα SISD και MISD δεν θα αναλυθούν περαιτέρω.



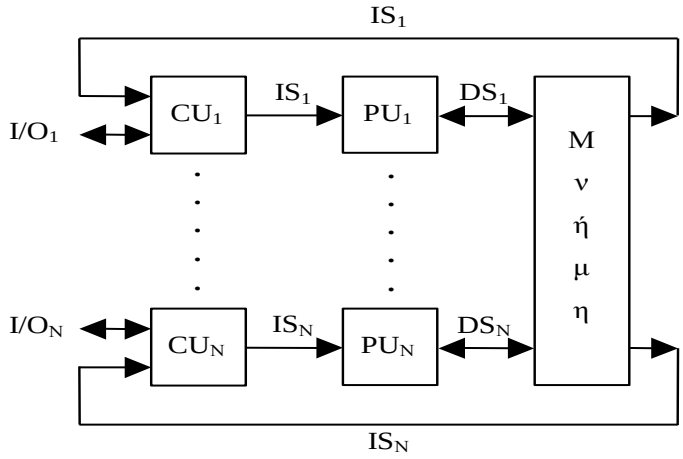
Σχήμα 2.2.1 : Μοντέλο SISD



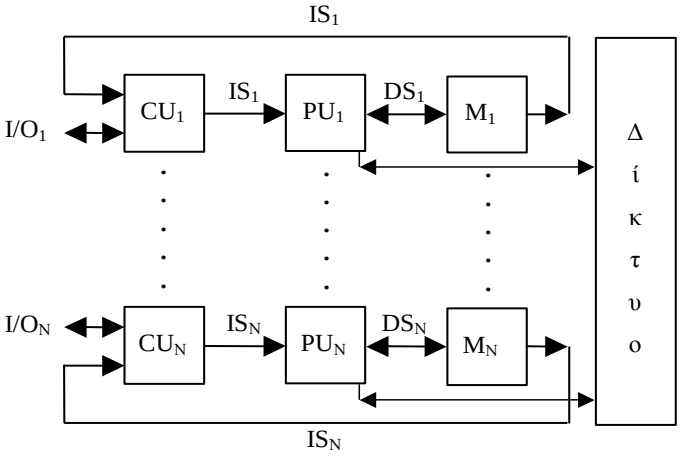
Σχήμα 2.2.2 : Μοντέλο SIMD



Σχήμα 2.2.3 : Μοντέλο MISD



Σχήμα 2.2.4 : Μοντέλο MIMD με κοινό δίαυλο



Σχήμα 2.2.5 : Μοντέλο MIMD με δίκτυο

Υποκεφάλαιο 2.3

ΠΕΡΙΣΣΟΤΕΡΑ ΓΙΑ ΤΑ ΜΟΝΤΕΛΑ SIMD ΚΑΙ MIMD

Το βασικότερο στοιχείο του μοντέλου **SIMD** είναι το οτι υπάρχουν πολλές μονάδες επεξεργασίας (ή αλλιώς επεξεργαστικά στοιχεία) οι οποίες εκτελούν την ίδια εντολή σε διαφορετικά δεδομένα. Κάθε επεξεργαστική μονάδα παραλαμβάνει την προς εκτέλεση εντολή από την Μονάδα Ελέγχου του συστήματος, ενώ τα δεδομένα πάνω στα οποία θα εκτελεστεί η εντολή τα παίρνει από την τοπική της μνήμη. Βασικό στοιχείο είναι, ότι κάθε επεξεργαστική μονάδα διαθέτει τοπική μνήμη για παραλαβή και αποθήκευση δεδομένων. Το μοντέλο αυτό περιγράφει υπολογιστές, οι οποίοι δεν αποτελούν αυτόνομα υπολογιστικά συστήματα, είναι προσκολλημένοι (attached) σε κάποιον κύριο υπολογιστή (πχ. ένα PC) από τον οποίο λαμβάνουν τις προς εκτέλεση εντολές. Είναι επίσης αρκετά απλοί, δεδομένου ότι δεν χειρίζονται τα δεδομένα μέσω μονάδων I/O και επομένως δεν χρησιμοποιούν ούτε λειτουργικό σύστημα. Ο κύριος ρόλος τους είναι συνήθως να εκτελούν συγκεκριμένες λειτουργίες πάνω σε πολλαπλά δεδομένα παρόμοιας φύσης (όπως πχ. αλγορίθμους για επεξεργασία εικόνας, μετασχηματισμούς Fourrier) γι' αυτό και χρησιμοποιούνται ως επιταχυντές.

Χαρακτηριστικό του μοντέλου **MIMD** είναι ότι πολλές εντολές εκτελούνται, η κάθε μία σε διαφορετικά δεδομένα. Συγκεκριμένα, υπάρχουν πολλές μονάδες ελέγχου, καθεμία από τις οποίες εκδίδει διαφορετικές εντολές, που παραλαμβάνουν με τη σειρά τους οι πολλαπλές επεξεργαστικές μονάδες. Φυσικά, σε κάθε μονάδα ελέγχου αντιστοιχίζεται και μία μονάδα επεξεργασίας. Σχετικά με τα στοιχεία μνήμης υπάρχουν δύο επιλογές: είτε κάθε επεξεργαστικό στοιχείο διαθέτει τη δική του τοπική μνήμη και άρα η ανταλλαγή δεδομένων γίνεται με τη χρήση κάποιου δικτύου διασύνδεσης, είτε υπάρχει μια κοινή μνήμη διαθέσιμη σε όλους για επικοινωνία. Το μοντέλο αυτό προσομοιάζει τη λειτουργία των πολυεπεξεργαστικών συστημάτων, συστήματα τα οποία είναι γενικού σκοπού και μπορούν να λειτουργήσουν και ως αυτόνομοι υπολογιστές. Παράδειγμα τέτοιων συστημάτων είναι τα clusters, που περιλαμβάνουν πολλές μονάδες επεξεργασίας, οι οποίες συνδέονται μεταξύ τους σε δίκτυο. Το μοντέλο MIMD είναι αυτό που τα τελευταία χρόνια υιοθετείται σε πολλές περιπτώσεις κυρίως για δύο λόγους:

- Πρώτον, διότι προσφέρει ευελιξία ως προς τον τρόπο χρήσης του (είτε ως μηχανή απλής χρήσης είτε ως πολυπρογραμματιζόμενη μηχανή για ταυτόχρονη εκτέλεση διεργασιών).
- Δεύτερον, διότι για την υλοποίησή του δεν απαιτείται πολύπλοκος ή ακριβός εξοπλισμός. Μπορούμε να φτιάξουμε ένα πολυεπεξεργαστικό σύστημα ενσωματώνοντας πολλούς παραδοσιακούς επεξεργαστές με χαμηλό κόστος ο καθένας.

Υποκεφάλαιο 2.4

ΤΑΞΙΝΟΜΗΣΗ MIMD

Ενότητα 2.4.1

ΤΑΞΙΝΟΜΗΣΗ MIMD ΜΕ ΒΑΣΗ ΤΗΝ ΑΡΧΙΤΕΚΤΟΝΙΚΗ ΜΝΗΜΗΣ

Οι MIMD μηχανές είναι αυτές, που κατά κύριο λόγο χρησιμοποιούνται για παράλληλη επεξεργασία, και επομένως το μοντέλο MIMD είναι αυτό που χρησιμοποιείται κατά κύριο λόγο για να περιγράψει παράλληλα υπολογιστικά μοντέλα.

Μπορούμε να χωρίσουμε τις μηχανές MIMD σε δύο κατηγορίες, ανάλογα με τον τρόπο οργάνωσης της μνήμης του συστήματός τους. Οι κατηγορίες αυτές είναι:

- Οι **πολυεπεξεργαστές**, δηλαδή οι υπολογιστές με **κοινή (shared)** μνήμη.
- Οι **πολυυπολογιστές**, δηλαδή οι υπολογιστές με **κατανεμημένη (distributed)** μνήμη.

Ενότητα 2.4.2

ΤΑΞΙΝΟΜΗΣΗ MIMD ΜΕ ΒΑΣΗ ΤΗΝ ΟΡΓΑΝΩΣΗ ΜΝΗΜΗΣ

Πολυεπεξεργαστές: το μοντέλο αυτό περιγράφει συστήματα, που αποτελούνται από πολλαπλούς επεξεργαστές, οι οποίοι εργάζονται ανεξάρτητα ο ένας από τον άλλον, αλλά

έχουν πρόσβαση σε έναν καθολικό φυσικών διευθύνσεων, που είναι κοινός για όλους. Ο αριθμός των επεξεργαστών σ' αυτήν την περίπτωση δεν είναι μεγάλος (συνήθως μέχρι 16 επεξεργαστές), ενώ οι λειτουργίες που εκτελούνται στη μνήμη και αφορούν την τροποποίηση δεδομένων (λειτουργία write) είναι ορατές από όλους τους επεξεργαστές.

- Τα **πλεονεκτήματα** που προκύπτουν από αυτήν την οργάνωση μνήμης είναι:

  - Η μνήμη είναι ενιαία και κοινή προς όλους και επομένως έχουμε να κάνουμε με ένα προγραμματιστικό μοντέλο, που είναι φιλικό προς τον χρήστη, διότι δεν διαφέρει από τα παραδοσιακά μοντέλα ακολουθιακού προγραμματισμού.

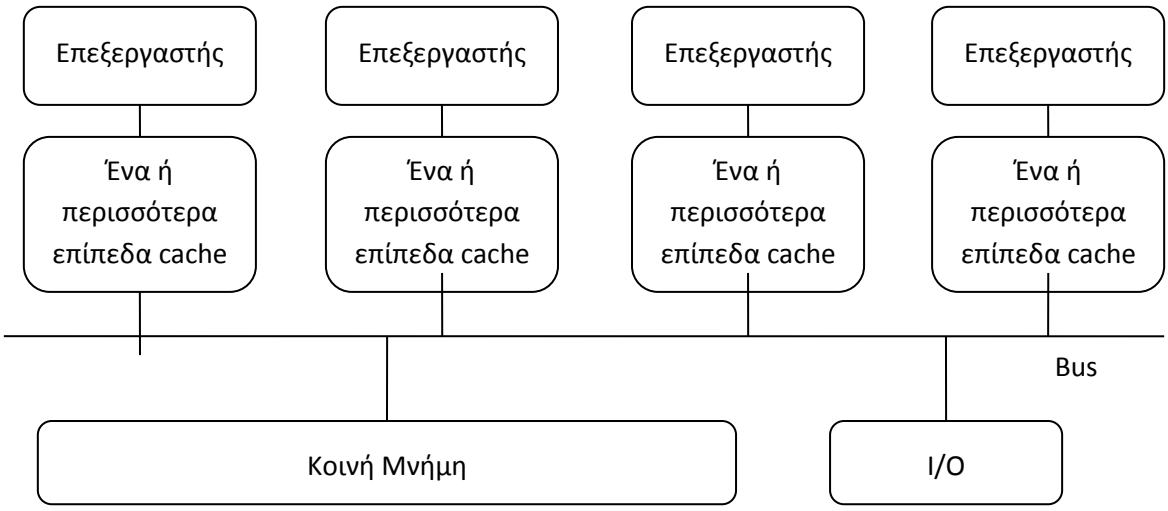
  - Η επικοινωνία και ο συγχρονισμός των δεδομένων γίνεται εύκολα, αφού όλοι οι επεξεργαστές έχουν πρόσβαση στον κοινό πόρο μνήμης.

- Τα **μειονεκτήματα** αυτής της προσέγγισης είναι:

  - Η έλλειψη κλιμάκωσης (scalability) στον αριθμό των επεξεργαστών. Η προσθήκη κεντρικών επεξεργαστών στο σύστημα αυξάνει την αιτήσεις για πρόσβαση στη μνήμη και άρα και την κυκλοφορία στον δίαυλο μνήμης, αυξάνονται επομένως και οι προσπελάσεις στην κρυφή μνήμη.

  - Η προσπάθεια για την επίλυση του παραπάνω προβλήματος απαιτεί τη χρήση ειδικού υλικού, όπως για παράδειγμα μνήμη με μεγάλο εύρος ζώνης διαύλου, ή ειδικά διασυνδετικά δίκτυα, κάτι που αυξάνει σημαντικά το κόστος και την αναβάθμιση και συντήρηση του συστήματος.

  - Ο προγραμματισμός που απαιτεί πολλές φορές τη χρήση ειδικών τεχνικών.

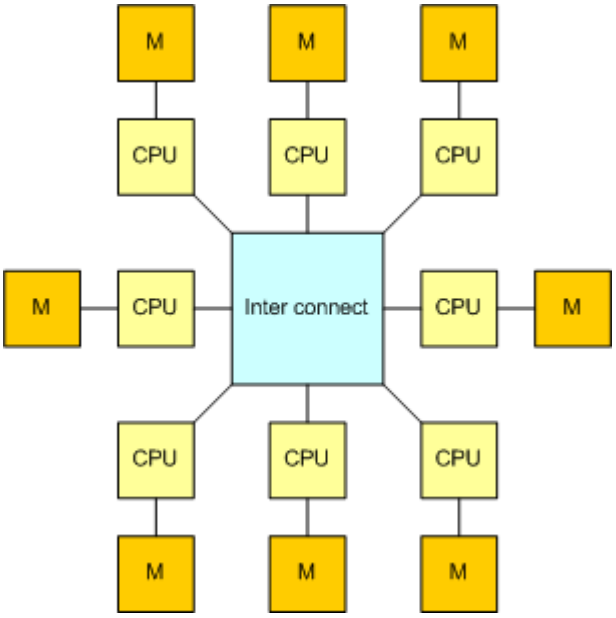


Σχήμα 2.4.1 : Μοντέλο πολυεπεξεργαστών πάνω σε κοινό δίαυλο

Ο τρόπος διασύνδεσης των πολυεπεξεργαστών μεταξύ τους, αλλά και με τη μνήμη γίνεται συνήθως με δίκτυα τύπου bus, αλλά υπάρχουν και διαφορετικοί τύποι που μπορούν να χρησιμοποιηθούν, όπως πολλαπλά bus ή δίκτυα διακοπτών crossbar .

- Από την άποψη του τρόπου προσπέλασης στη μνήμη, η αρχιτεκτονική αυτή ονομάζεται και μοντέλο **Ομοιόμορφης Προσπέλασης Μνήμης (UMA** – Uniform Memory Access). Η ονομασία του οφείλεται ακριβώς στο ότι η μοιραζόμενη μνήμη είναι προσβάσιμη από όλους τους επεξεργαστές, οπότε ο μέσος χρόνος προσπέλασης μνήμης είναι ο ίδιος, ανεξάρτητα από τη διεύθυνση στην οποία είναι αποθηκευμένο το δεδομένο.

Πολυυπολογιστές: στο μοντέλο αυτό ανήκουν τα συστήματα στα οποία κάθε επεξεργαστής διαθέτει τη δική του τοπική μνήμη και η επικοινωνία με τις απομακρυσμένες μνήμες (των άλλων επεξεργαστών) επιτυγχάνεται μέσω του δικτύου (συνήθως εις γνώση του απομακρυσμένου επεξεργαστή). Οι επεξεργαστές τροποποιούν τα δεδομένα τους στη μνήμη, χωρίς αυτό να επιφέρει αλλαγές στα δεδομένα των υπόλοιπων επεξεργαστών (ακόμη και αν έχουν αποθηκεύσει αντίγραφο του ίδιου δεδομένου) ενώ κάθε επεξεργαστής διαθέτει τη δική του κρυφή μνήμη και επομένως δεν τίθεται θέμα συνοχής κρυφής μνήμης. Η χρονική στιγμή της προσπέλασης δεδομένων σε μια απομακρυσμένη μνήμη, το ποια δεδομένα θα προσπελαστούν και ο τρόπος με τον οποίο θα συγχρονιστούν οι παράλληλες εργασίες καθορίζεται προγραμματιστικά και μόνο. Στο μοντέλο αυτό μπορεί να χρησιμοποιηθούν διάφορες τεχνολογίες δικτύωσης, από το γνωστό μας Ethernet μέχρι άλλου τύπου δίκτυα, ενώ ο τρόπος με τον οποίο επιτυγχάνεται η επικοινωνία των επεξεργαστών μπορεί να είναι είτε καθολικός είτε σημειακός



Σχήμα 2.4.2 : Μοντέλο πολυυπολογιστών πάνω σε δίκτυο διασύνδεσης

- Τα **πλεονεκτήματα** αυτού του μοντέλου οργάνωσης μνήμης είναι:
    - Η εύκολη κλιμάκωση όσον αφορά την προσθήκη περισσότερων επεξεργαστών και την αύξηση του μεγέθους της μνήμης / επεξεργαστή.
    - Η μείωση του χρόνου προσπέλασης στη μνήμη, διότι πλέον η πρόσβαση στα δεδομένα γίνεται απρόσκοπτα και ταχύτατα μέσω τοπικών μνημών χωρίς την ύπαρξη κεντρικού διαύλου και άρα χωρίς την εισαγωγή καθυστέρησης λόγω διαιτησίας.

- Η αύξηση του bandwidth, το οποίο μετράται σε bits/second, του όγκου δηλαδή των δεδομένων που μπορούν να προσπελαστούν σε ένα συγκεκριμένο χρονικό διάστημα.

- Η δυνατότητα χρήσης συμβατών υπολογιστών και τεχνολογιών δικτύου, μειώνοντας σημαντικά το απαιτούμενο κόστος για την κατασκευή, αναβάθμιση και συντήρηση του συστήματος.

➢ Τα **μειονεκτήματα** που παρουσιάζει το μοντέλο είναι:

- Επιβάρυνση του δικτύου με την επικοινωνία μεταξύ των επεξεργαστών μέσω μηνυμάτων. Το γεγονός αυτό, επιβάλλει μια πρόσθετη καθυστέρηση, αφενός λόγω της πολυπλοκότητας του ίδιου του δικτύου και αφετέρου λόγω της καθυστέρησης μετάδοσης του σήματος (του μηνύματος επικοινωνίας) στο δίκτυο από τον αποστολέα στον παραλήπτη.

- Χρήση σύνθετων προγραμματιστικών μοντέλων, προκειμένου να είναι δυνατή η διεπεξεργαστική επικοινωνία και ανταλλαγή δεδομένων.

- Μη ομοιόμορφοι χρόνοι προσπέλασης μνήμης.

- Αδρομερής προγραμματισμός, και επομένως η αποτελεσματική κλιμάκωση μπορεί να επιτευχθεί μόνο για μεγάλο πλήθος δεδομένων και για εφαρμογές όπου η επικοινωνία μεταξύ των επεξεργαστών μπορεί να περιοριστεί όσο το δυνατόν περισσότερο.

Ο τρόπος κατανομής της μνήμης στους πολλαπλούς επεξεργαστές καθορίζεται από 2 παραμέτρους:

- τη **χωρική** κατανομή, η οποία σχετίζεται με τη φυσική θέση της μνήμης σε σχέση με τους επεξεργαστές, δηλαδή το πού είναι τοποθετημένη φυσικά μέσα στο εκάστοτε σύστημα.

- τη **λογική** κατανομή που σχετίζεται με τον τρόπο με τον οποίο κάθε επεξεργαστής "βλέπει" και προσπελαύνει τη μνήμη του συστήματος.

Ειδικά στην περίπτωση του μοντέλου των πολυυπολογιστών που εξ ορισμού συνεπάγεται τη χρήση **χωρικά κατανεμημένης μνήμης** (εννοώντας ότι κάθε στοιχείο μνήμης βρίσκεται εγκατεστημένο -onboard- στην πλακέτα όπου βρίσκεται και κάθε επεξεργαστής) μπορούμε να διακρίνουμε δύο περιπτώσεις :

- Ο κάθε επεξεργαστής να μπορεί να προσπελαύνει θέσεις μνήμης που περιέχονται μόνο μέσα στην δική του τοπική μνήμη (ιδιωτική μνήμη) οπότε σε αυτήν την περίπτωση κάνουμε χρήση μνήμης η οποία είναι **λογικά κατανεμημένη**.

- Οι επεξεργαστές του συστήματος να μπορούν να προσπελαύνουν οποιαδήποτε διεύθυνση μνήμης, σαν να επρόκειτο για έναν ενιαίο χώρο διευθύνσεων, ακόμα και αν τα στοιχεία της μνήμης συστήματος είναι στην πραγματικότητα χωρικά κατανεμημένα. Σ’ αυτήν την περίπτωση μπορούμε να μιλάμε για μνήμη η οποία είναι **λογικά κοινή**.

Με βάση τις παραπάνω περιπτώσεις διακρίνουμε δύο βασικές προσεγγίσεις του μοντέλου πολυυπολογιστή, οι οποίες επιγραμματικά είναι:

1. Το μοντέλο της χωρικά και λογικά κατανεμημένης μνήμης όπου κάθε επεξεργαστής έχει πρόσβαση μόνο στα δεδομένα της τοπικής του μνήμης. Σε περίπτωση, που κάποιος επεξεργαστής θελήσει να αποκτήσει πρόσβαση στην απομακρυσμένη μνήμη κάποιου άλλου επεξεργαστή, θα πρέπει να του γνωστοποιήσει το αίτημα του μέσω μηνύματος. Γι' αυτόν ακριβώς το λόγο το μοντέλο αυτό ονομάζεται και **μοντέλο ανταλλαγής μηνυμάτων (message passing)**.

2. Το μοντέλο της χωρικά κατανεμημένης, αλλά λογικά κοινής μνήμης, που συνοψίζεται με τη φράση μοντέλο **Κοινής Κατανεμημένης Μνήμης (DSM-Distributed Shared Memory).** Οι μνήμες σ'αυτήν την περίπτωση είναι χωρικά κατανεμημένες, αλλά αντιμετωπίζονται από τους επεξεργαστές ως ένας ενιαίος χώρος διευθύνσεων. Επομένως, κάθε επεξεργαστής μπορεί να εκδίδει εντολές πρόσβασης σε οποιαδήποτε πεδία διευθύνσεων (τα οποία χωρικά μπορεί να βρίσκονται είτε τοπικά, είτε απομακρυσμένα), τις οποίες χειρίζεται μια μονάδα διαχείρισης μνήμης υλοποιημένη με τη βοήθεια κατάλληλου υλικού.
   Από προγραμματιστικής άποψης, οι **πολυυπολογιστές Κοινής Κατανεμημένης Μνήμης** δεν διαφέρουν από το μοντέλο των πολυεπεξεργαστών κοινής μνήμης, διότι σ' αυτήν την περίπτωση έχουμε τη δυνατότητα προσπέλασης οποιασδήποτε διεύθυνσης (είτε τοπικής, είτε απομακρυσμένης).

Η παράμετρος η οποία αλλάζει σε αυτό το μοντέλο, αφορά το χρόνο προσπέλασης της μνήμης, ο οποίος είναι διαφορετικός για μια εντολή προσπέλασης διεύθυνσης που εξυπηρετείται από μία χωρικά τοπική μνήμη και διαφορετικός για μια εντολή που εξυπηρετείται από μία χωρικά απομακρυσμένη μνήμη. Για παράδειγμα, αν ένας επεξεργαστής A θελήσει να ανακτήσει το περιεχόμενο της μεταβλητής b που βρίσκεται στην τοπική του μνήμη, θα το κάνει σε πολύ μικρότερο χρόνο από αυτόν που θα χρειαστεί για να ανακτήσει το περιεχόμενο της μεταβλητής c που βρίσκεται στη μνήμη του επεξεργαστή B. Γι' αυτόν το λόγο οι πολυυπολογιστές DSM ονομάζονται μηχανές **Μη-Ομοιόμορφης Προσπέλασης Μνήμης (NUMA**- Non-Uniform Memory Access) σε αντιδιαστολή με τους πολυεπεξεργαστές **UMA** που αναφέρθηκαν στην παραπάνω ενότητα.

Υποκεφάλαιο 2.5

ΙΕΡΑΡΧΙΑ ΜΝΗΜΗΣ ΣΕ ΑΠΛΑ ΕΠΕΞΕΡΓΑΣΤΙΚΑ ΣΥΣΤΗΜΑΤΑ

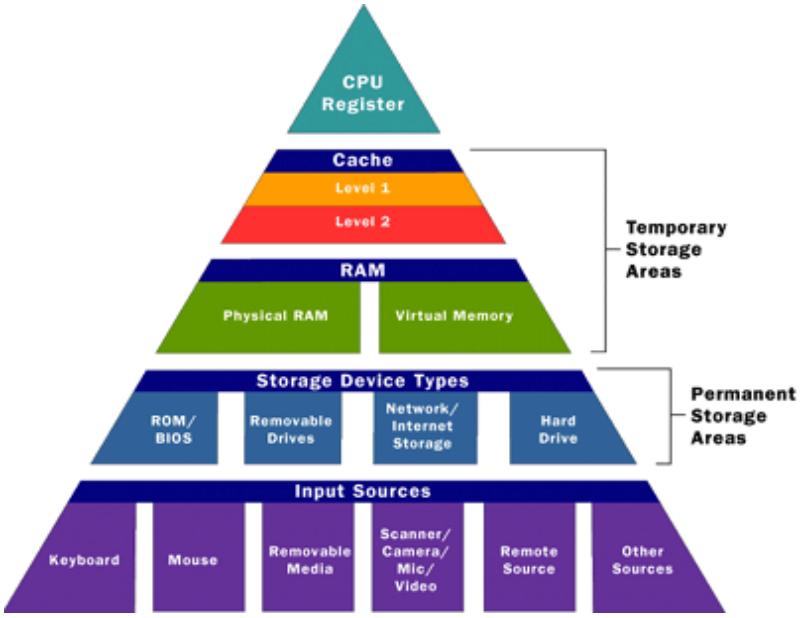
Θεωρητικά, θα μπορούσαν να κατασκευαστούν μνήμες, οι οποίες θα χαρακτηρίζονταν από μεγάλη χωρητικότητα και μικρό χρόνο προσπέλασης. Πρακτικά όμως, η κατασκευή ενός τύπου μνήμης, που θα πληροί αυτές τις προδιαγραφές κοστίζει τόσο πολύ, ώστε η εμπορική του διάθεση να καθίσταται ασύμφορη. Αυτό το συμπέρασμα γίνεται εύκολα κατανοητό αν σκεφτεί κανείς, ότι καθώς προσπαθούμε να αυξήσουμε τη χωρητικότητα μιας μνήμης, ταυτόχρονα αυξάνουμε το πλήθος των τρανζίστορ ανά bit, μεγαλώνουμε την πυκνότητα των τρανζίστορ και άρα μεγαλώνουμε το χρόνο προσπέλασης δεδομένων στη μνήμη, διότι αλλάζουμε την τεχνολογία που χρησιμοποιείται για την κατασκευή της.

Οι μνήμες ημιαγωγών είναι οι ταχύτερες μνήμες, αλλά όσο περισσότερο αυξάνεται η χωρητικότητά τους, τόσο πιο αργές αυτές καθίστανται. Όσον αφορά τα κυκλώματα μνήμης, είναι προφανές ότι η ταχύτητα είναι αντιστρόφως ανάλογη της χωρητικότητας. Και στην περίπτωση ακόμα, που γινόταν προσπάθεια κατασκευής μιας μνήμης ημιαγωγών με μεγάλη χωρητικότητα, αλλά ταυτόχρονα μικρό χρόνο προσπέλασης, η συγκεκριμένη τεχνολογία κατασκευής θα απαιτούσε πολύ μεγάλο κόστος ανάπτυξης για να καταστεί αποτελεσματική.

Γι' αυτόν το λόγο, η διάθεση μνήμης με πολύ μεγάλη χωρητικότητα (αναφορικά με τα σημερινά δεδομένα) παρέχεται μέχρι στιγμής από άλλα είδη αποθήκευσης (για παράδειγμα CD, DVD, Flash μνήμες κτλ.), που χαρακτηρίζονται από διαφορετική τεχνολογία σε σχέση με αυτήν που χρησιμοποιείται στις μνήμες ημιαγωγών.

Επομένως, υπάρχει μια αδυναμία κατασκευής μεγάλων και ταυτόχρονα γρήγορων κυκλωμάτων μνήμης. Παρόλα αυτά, μέσα σε ένα υπολογιστικό σύστημα απαιτείται η συνεργασία του επεξεργαστή με τη μνήμη, καθώς ένας επεξεργαστής δεν θα μπορούσε να λειτουργήσει αυτόνομα, διότι δεν είναι κύκλωμα αποθήκευσης δεδομένων. Στο σημείο αυτό εντοπίζεται ο βασικός προβληματισμός, διότι ναι μεν η ταχύτητα εκτέλεσης εντολών στον επεξεργαστή αυξάνεται εκθετικά σε σχέση με τον χρόνο, αλλά ο ρυθμός αύξησης της ταχύτητας των μνημών είναι γραμμικός. Με βάση αυτό το στοιχείο, είναι πολύ εύκολο να κατανοήσει κανείς ότι υπάρχει μία τεράστια διαφορά ταχυτήτων ανάμεσα στον επεξεργαστή και στη μνήμη.

Η επίλυση του προβλήματος που προέρχεται από αυτήν τη διαφορά ταχυτήτων καθίσταται δυνατή με την οργάνωση της μνήμης κατά ένα συγκεκριμένο ιεραρχικό μοντέλο. Η ιεράρχηση αυτή γίνεται σε μια πυραμίδα, τη βάση της οποίας αποτελούν οι μνήμες μαζικής αποθήκευσης, που χαρακτηρίζονται από μεγάλη χωρητικότητα, μεγάλους χρόνους προσπέλασης και χαμηλό κόστος . Στη συνέχεια, και ένα επίπεδο παραπάνω, τοποθετείται η κύρια μνήμη (μνήμη ημιαγωγών) η οποία είναι ταχύτερη μνήμη μικρότερης χωρητικότητας και σαφώς ακριβότερη σε σχέση με τις μαζικές μνήμες. Στο επόμενο επίπεδο συναντούμε τις πολύ γρήγορες μνήμες cache, με μέγεθος της τάξης των λίγων MB, και καταλήγουμε με το τελευταίο επίπεδο, στο οποίο βρίσκονται οι καταχωρητές που τοποθετούνται μέσα στην μονάδα επεξεργασίας, έχουν ελάχιστη χωρητικότητα και είναι ταχύτατοι. Χρησιμοποιώντας το συγκεκριμένο μοντέλο ιεραρχικής μνήμης μέσα σε ένα υπολογιστικό σύστημα, καθίσταται δυνατός ο πολλαπλασιασμός της ταχύτητας τροφοδότησης του επεξεργαστή με δεδομένα και η ελαχιστοποίηση του χρόνου εκείνου κατά τον οποίο ο επεξεργαστής παραμένει αδρανής αναμένοντας δεδομένα από τις μονάδες μνήμης.



Σχήμα 2.5.1 : Ιεραρχικό μοντέλο μνήμης

Υποκεφάλαιο 2.6

ΒΑΣΙΚΑ ΣΤΟΙΧΕΙΑ ΣΧΕΔΙΑΣΗΣ ΕΠΕΞΕΡΓΑΣΤΩΝ

Το βασικότερο στοιχείο ενός υπολογιστικού συστήματος είναι σαφέστατα η κεντρική μονάδα επεξεργασίας ή αλλιώς κεντρικός επεξεργαστής (CPU). Ο επεξεργαστής αυτός αποτελείται από διάφορες υπομονάδες (ALU, FPU, καταχωρητές) οι οποίες συνεργάζονται μεταξύ τους για την εκτέλεση εντολών προγράμματος και την εξαγωγή αποτελεσμάτων.

Η αρχιτεκτονική σχεδίαση ενός επεξεργαστή καθορίζεται από τα εξής τρία **χαρακτηριστικά**:

1. τη σχεδίαση της εσωτερικής μνήμης
2. τη σχεδίαση του συνόλου εντολών
3. τη χρήση τεχνικών pipelining και παραλληλισμού

Αναλύουμε τα **χαρακτηριστικά** αυτά:

1. Όσον αφορά τη σχεδίαση της εσωτερικής μνήμης του συστήματος, αυτή ακολουθεί το μοντέλο ιεραρχίας που περιγράψαμε παραπάνω και το οποίο περιλαμβάνει:
   - Τους καταχωρητές που βρίσκονται μέσα στην κεντρική μονάδα επεξεργασίας
   - Τη μνήμη cache επιπέδου 1 (L1 cache)
   - Τη μνήμη cache επιπέδου 2 (L2 cache)
   - Τη μνήμη cache επιπέδου 3 (L3 cache)

2. Η σχεδίαση του συνόλου των εντολών (Instruction Set) και του τρόπου υλοποίησής τους από έναν επεξεργαστή είναι αυτή που καθορίζει την πολυπλοκότητα της αρχιτεκτονικής του.

Σ' αυτήν την περίπτωση διακρίνουμε δύο βασικές κατηγορίες επεξεργαστών:

- Την οικογένεια επεξεργαστών με **πολύπλοκο σύνολο εντολών** ή **CISC (Complex Instruction Set Computers)** οι οποίοι δημιουργήθηκαν με σκοπό να παρέχουν ένα πλούσιο ρεπερτόριο εντολών πάνω στο υλικό του επεξεργαστή (200-400 εντολές). Οι συμβατικοί επεξεργαστές, όπως οι Intel Pentium, VAX/8600 και οι Intel Core ανήκουν σε αυτήν την κατηγορία. Η όλη προσπάθεια εστιαζόταν στην υλοποίηση περισσότερων και πολυπλοκότερων πράξεων, μέσω του υλικού (το οποίο είναι σαφώς ταχύτερο σε σχέση με το λογισμικό) παρά μέσω του λογισμικού του επεξεργαστή. Αυτοί οι επεξεργαστές χρησιμοποιούν

μικροπρογραμματισμό στο κύκλωμα ελέγχου εντολών (σε πρώτη φάση τουλάχιστον, διότι σε μεταγενέστερες υλοποιήσεις χρησιμοποιείται καλωδιωμένη λογική προς αποφυγή του μικροπρογραμματισμού) και έχουν εντολές που εκτελούνται σε διάφορους κύκλους μηχανής. Για παράδειγμα, άλλες εντολές εκτελούνται σε έναν κύκλο μηχανής και άλλες χρειάζονται δύο ή και πολύ περισσότερους.

- Την οικογένεια επεξεργαστών με **μειωμένο σύνολο εντολών ή RISC (Reduced Instruction Set Computers)** όπως για παράδειγμα οι επεξεργαστές Intel i860, SPARC, IBM RS/6000**.** Η εμφάνισή τους οφείλεται στο οτι αμφισβητήθηκε η πρακτική σημασία της υιοθέτησης πολύπλοκων συνόλων εντολών. Παρατηρηθήκε επίσης, ότι όσο περισσότερο το υλικό επιφορτιζόταν με πρόσθετες και περίπλοκες λειτουργίες - εντολές τόσο πιο αργά τις εκτελούσε. Επομένως, υιοθετήθηκε μια εντελώς διαφορετική προσέγγιση σχεδιασμού υλικού, περιλαμβάνοντας ένα σύνολο από βασικές μόνο εντολές. Βασικό χαρακτηριστικό των RISC είναι το ότι όλες οι εντολές απαιτούν περίπου τον ίδιο αριθμό κύκλων μηχανής για να εκτελεστούν (περίπου ένας - δύο κύκλοι μηχανής ανά εντολή).

Υπάρχει φυσικά και το υπολογίσιμο κόστος, που προέρχεται από την εκτέλεση μιας περισσότερο πολύπλοκης εντολής σε σχέση με τις υπόλοιπες, παρόλα αυτά έχει αποδειχθεί στατιστικά, ότι η εκτέλεση τέτοιων εντολών είναι σπάνια και επομένως το κόστος σε τέτοιες περιπτώσεις είναι αμελητέο. Οι RISC επεξεργαστές έχουν εν γένει ταχύτερο κύκλο μηχανής συγκρινόμενοι με τους CISC.

3. Τεχνικές παραλληλισμού και pipelining. Με την τεχνική **pipeline** ο κύκλος απόκτησης – εκτέλεσης μιας μηχανής χωρίζεται σε Ν φάσεις. Το υλικό του επεξεργαστή χωρίζεται και αυτό με τη σειρά του σε Ν κομμάτια καθένα από τα οποία αναλαμβάνει την εκτέλεση μιας από τις Ν φάσεις της εντολής. Με αυτόν τον τρόπο μπορούμε να εκμεταλλευτούμε με αποδοτικότερο τρόπο τις επεξεργαστικές δυνατότητες μιας κεντρικής μονάδας επεξεργασίας και να την κρατούμε απασχολημένη για περισσότερο χρόνο, κερδίζοντας παράλληλα και από άποψη ταχύτητας εκτέλεσης των εντολών. Το pipeling έχει σαν στόχο την επίτευξη (όσο το δυνατόν μεγαλύτερου βαθμού) παραλληλίας στην εκτέλεση των Ν εντολών. Μια ειδική κατηγορία επεξεργαστών RISC που κάνουν χρήση πολλαπλών pipelines είναι:

- οι **superscalar** επεξεργαστές, οι οποίοι μπορούν να εκτελούν ταυτόχρονα στον ίδιο κύκλο μηχανής (μέσα στον μοναδικό πυρήνα) παραπάνω από μια εντολές, ενώ η δρομολόγηση πολλαπλών εντολών στις pipeline πραγματοποιείται από το υλικό του επεξεργαστή. Με αυτόν τον τρόπο η ταχύτητα εκτέλεσης των εντολών αυξάνεται παρόλο που ο κύκλος μηχανής παραμένει ίδιος σε σχέση με έναν παραδοσιακό επεξεργαστή **RISC.**

- οι επεξεργαστές **VLIW (Very Long Instruction Word)** οι οποίοι υλοποιούν μια διαφορετική σχεδιαστική άποψη και έχουν πολύ μεγάλο μήκος εντολής (256 ή 1024 bits). Σ' αυτήν την περίπτωση κάθε εντολή ουσιαστικά αποτελείται από άλλες μικρότερες, οι οποίες μπορούν να εκτελεστούν ταυτόχρονα. Το μοντέλο αυτό προσομοιάζει με το μοντέλο superscalar, με τη βασική τους διαφορά να εντοπίζεται στο ότι οι VLIW χρησιμοποιούν μικροκώδικα, που φορτώνεται σε μια ROM και μεταφράζεται προκειμένου να χρησιμοποιηθεί στη δρομολόγηση εντολών.

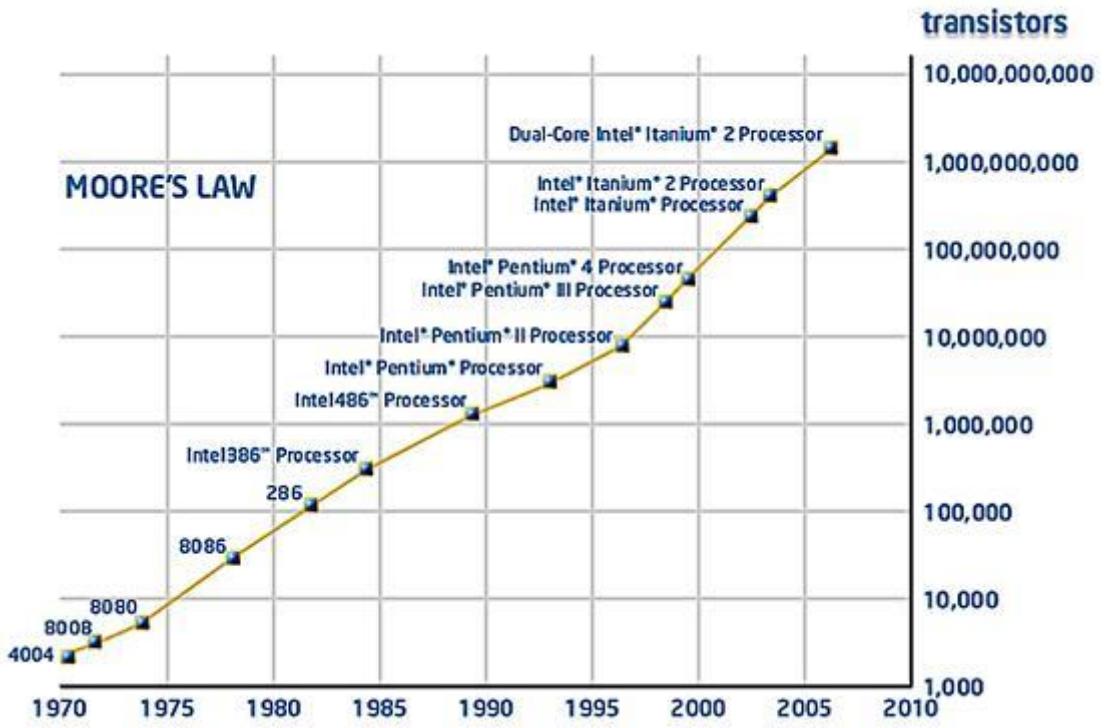
Υποκεφάλαιο 2.7

ΕΞΕΛΙΞΗ ΣΤΟΝ ΤΟΜΕΑ ΤΩΝ ΠΑΡΑΔΟΣΙΑΚΩΝ ΕΠΕΞΕΡΓΑΣΤΩΝ

Από τη στιγμή της εμφάνισης των υπολογιστικών συστημάτων ξεκίνησε -και συνεχίζεται και μέχρι σήμερα- μια διαρκής προσπάθεια για τη βελτίωση και την εξέλιξή τους. Αυτή η προσπάθεια στηρίχθηκε σε δύο βασικούς άξονες: την εξέλιξης της αρχιτεκτονικής και την εξέλιξη της κατασκευής ολοκληρωμένων κυκλωμάτων. Αυτό ουσιαστικά μεταφράζεται ως προσπάθεια αφενός βελτίωσης του συνόλου των εντολών, που εκτελεί ένας επεξεργαστής και αφετέρου αύξησης του αριθμού των ψηφιακών στοιχείων (πυκνότητας) ανά τρανζίστορ.

Όσον αφορά αυτήν την παράμετρο, ξεκινήσαμε με τη χρήση τη βασικής τεχνολογίας ολοκλήρωσης κυκλωμάτων στους υπολογιστές 3ης γενιάς (SSI), η οποία αργότερα εξελίχθηκε και έδωσε τη θέση της σε τεχνολογίες ολοκλήρωσης μεγαλύτερης κλίμακας (LSI, VLSI, ULSI) στους υπολογιστές 4ης και 5ης γενιάς. Ειδικά, σε ό,τι αφορά την εξέλιξη της τεχνολογίας ολοκλήρωσης των ψηφιακών κυκλωμάτων (έχει διατυπωθεί ως νόμος του Moore) η πυκνότητά τους αυξάνεται εκθετικά σε σχέση με τον χρόνο και διπλασιάζεται περίπου κάθε 2 χρόνια.

Πίνακας 2.7.1 : Γράφημα απεικόνισης του νόμου Moore



Σχετικά με την παράμετρο της βελτίωσης του συνόλου εντολών πρέπει να αναφέρουμε την εμφάνιση, το 1985, της RISC αρχιτεκτονικής που περιγράψαμε παραπάνω. Αυτή η νέα θεώρηση εισήγαγε μια εντελώς διαφορετική σχεδίαση του συνόλου των εντολών μιας μηχανής. Παρόλα αυτά, η ανάπτυξη εντολών RISC εγκαταλείφθηκε λίγο αργότερα, διότι το όφελος από την αύξηση της πυκνότητας των τρανζίστορ (η οποία σημειωνόταν με αλματώδεις ρυθμούς) υπερκάλυπτε το κόστος και την προσπάθεια που απαιτούσε η ανάπτυξη πολύπλοκων συνόλων εντολών.

Παράλληλα με τη μείωση του μεγέθους των τρανζίστορ, έχουμε τη συνεχή προσθήκη τρανζίστορ μέσα στο κύκλωμα του κεντρικού επεξεργαστή. Αυτό σημαίνει πρόσθετες υπολογιστικές δυνατότητες, μείωση του κόστους επεξεργασίας και ταυτόχρονη βελτίωση της αξιοπιστίας. Επιπλέον, εντείνεται και η προσπάθεια για την αύξηση της συχνότητας χρονισμού του κεντρικού επεξεργαστή.

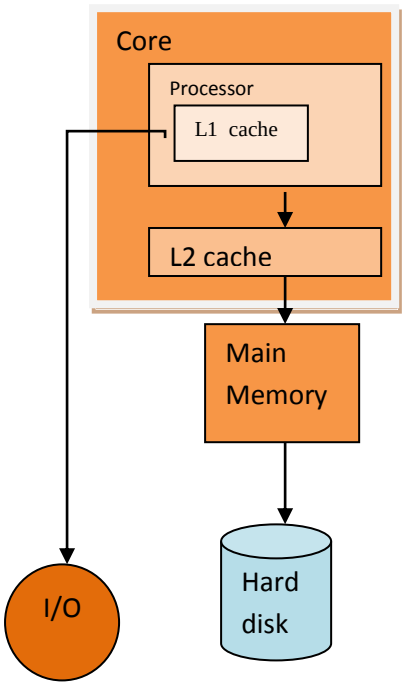
Όπως είναι φυσικό, υπάρχει πάντα η απαίτηση για αύξηση της απόδοσης των υπολογιστικών συστημάτων. Δεδομένης της επιθυμίας, ειδικά τα τελευταία χρόνια, για ανάπτυξη ταχύτερων και περιπλοκότερων εφαρμογών, η απαίτηση για επεξεργαστικά

συστήματα που θα μπορούν να εκτελούν αυτές τις εφαρμογές αποτελεσματικά και σε μικρό χρόνο γίνεται όλο και εντονότερη.

Κάποια στιγμή, όλες αυτές οι τεχνολογίες, αν και ήταν καινοτόμες, έφτασαν σε κάποιο σημείο πέρα από το οποίο αδυνατούσαν πλέον να συνεισφέρουν σε περαιτέρω βελτίωση της απόδοσης. Το γεγονός αυτό ήταν απόλυτα αναμενόμενο, αν σκεφτεί κανείς ότι όλες οι μέχρι τότε τεχνικές αύξησης της απόδοσης στηρίζονταν είτε στην εξέλιξη του υλικού ή στη βελτίωση του συνόλου εντολών, τομείς που κάποια στιγμή της εξέλιξής τους συναντούν φυσικά εμπόδια. Αρχικά, το πρόβλημα εντοπίστηκε στην πυκνότητα των τρανζίστορ μέσα στο ολοκληρωμένο κύκλωμα, η οποία προφανώς δεν μπορούσε να αυξάνεται απεριόριστα και μάλιστα χωρίς την αντίστοιχη κατανάλωση ενέργειας που συνεπάγεται. Επιπλέον, η βελτίωση του συνόλου εντολών, όπως και της ταχύτητας εκτέλεσης εντολών (π.χ. μέσω τεχνικών όπως το pipeline) είχε φτάσει και αυτή στο όριό της, διότι παρεμποδιζόταν από το μεγάλο χρόνο προσπέλασης της Κεντρικής Μνήμης. Επιπλέον, η πορεία αύξησης της συχνότητας ρολογιού δεν μπορούσε να συνεχιστεί επ' άπειρον, οπότε και αυτή από ένα σημείο και μετά είχε παραμείνει στάσιμη στην τάξη των 3 GHz περίπου.

Μέχρι τότε, η βελτίωση της υπολογιστικής απόδοσης στόχευε σε βελτιώσεις που αφορούσαν την αρχιτεκτονική του επεξεργαστή ή τον τρόπο διασύνδεσης των μονάδων του. Από αυτό το κρίσιμο σημείο κι έπειτα άρχισε να γίνεται αντιληπτό ότι η αύξηση της υπολογιστικής επίδοσης δεν μπορούσε να συντελεστεί μέσω των τεχνικών που σχετίζονταν με την εξέλιξη του υλικού ή την αύξηση της ταχύτητας εκτέλεσης των εντολών. Άρχισαν επομένως να συντελούνται βήματα προς μια διαφορετική κατεύθυνση, αυτήν της εκτέλεσης πολλαπλών εντολών από πολλαπλούς επεξεργαστές μέσα σε ένα υπολογιστικό σύστημα. Μάλιστα, οι επεξεργαστές αυτοί θα μπορούσαν να είναι είτε συμμετρικοί (ίδιοι) είτε και διαφορετικής αρχιτεκτονικής ο καθένας (για παράδειγμα στην περίπτωση που αποτελούν μέρος ενός cluster). Το γεγονός αυτό σήμαινε αφενός τη μεταστροφή απο τη φιλοσοφία των μονοεπεξεργαστικών συστημάτων σε συστήματα με πολλαπλούς επεξεργαστές, και αφετέρου την μεταστροφή από τον παραλληλισμό σε επίπεδο εντολών (ILP) στον παραλληλισμό σε επίπεδο νημάτων (TLP). Τα παράλληλα νήματα θα μπορούσαν να εκτελούνται από τις πολλαπλές μονάδες επεξεργασίας μέσα στο σύστημα. Στην ουσία, από το σημείο αυτό και έπειτα, ξεκινάει η εισαγωγή πολλαπλών πυρήνων επεξεργασίας (από 2 μέχρι 16 πυρήνες) μέσα στο ολοκληρωμένο κύκλωμα των κεντρικών μονάδων επεξεργασίας. Στα χρόνια που ακολούθησαν καθιερώθηκε η προσθήκη πολλαπλών πυρήνων μέσα στους επεξεργαστές. Τυπικό παράδειγμα ενός τέτοιου επεξεργαστή είναι ο τετραπύρηνος Intel Core i7.

Ένα γενικό παράδειγμα οργάνωσης του πυρήνα ενός πολυπύρηνου επεξεργαστή φαίνεται στην παρακάτω εικόνα.



Σχήμα 2.7.2 : Γενικό μοντέλο οργάνωσης πυρήνα
ενός πολυπύρηνου επεξεργαστή.

ΥΛΟΠΟΙΗΣΕΙΣ :

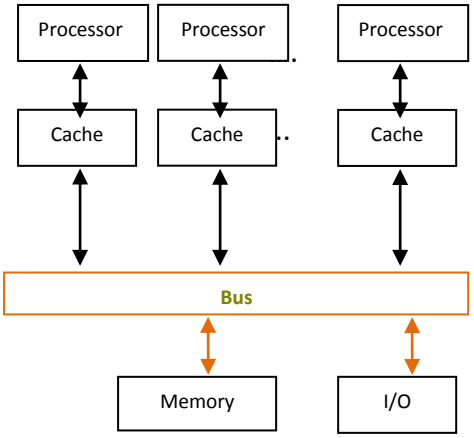
Γενικά, υπάρχουν πολλές υλοποιήσεις πολυπύρηνων επεξεργαστών ανάλογα με τον εκάστοτε κατασκευαστή. Σε όλες όμως τις υλοποιήσεις, βασική προϋπόθεση αποτελεί η επικοινωνία μεταξύ των πυρήνων. Η επικοινωνία αυτή επιτυγχάνεται:

- είτε μέσω του κοινού δίαυλου επικοινωνίας τύπου bus
- είτε με τη χρήση ενός δικτύου διασύνδεσης.

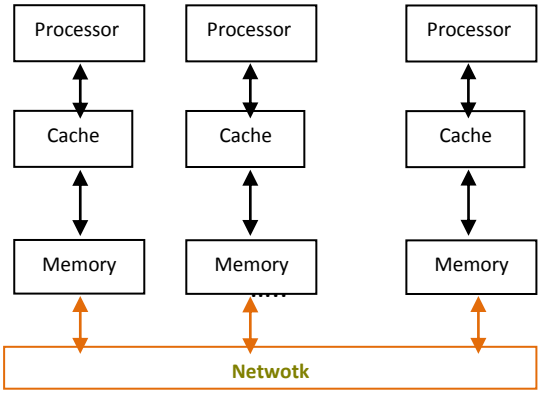
Η χρήση του κοινού διαύλου προϋποθέτει την υλοποίηση του μοντέλου κοινής μνήμης, ενώ η προσέγγιση του δικτύου διασύνδεσης χρησιμοποιεί το μοντέλο της κατανεμημένης μνήμης.

Η υλοποίηση του μοντέλου κοινού διαύλου (bus) αντιμετωπίζει περιορισμούς επεκτασιμότητας. Από ένα σημείο κι έπειτα η απόδοσή του διαύλου μειώνεται. Για παράδειγμα με προσθήκη 32 πυρήνων πάνω στο δίαυλο, παρατηρούμε αυξημένο

υπολογιστικό φόρτο, που προκύπτει από την επικοινωνία και τον ανταγωνισμό που υπάρχει μεταξύ των πυρήνων. Πάντως, οι επεξεργαστές αυτοί κατάφεραν να επιλύσουν τα προβλήματα σχετικά με την ταχύτητα εκτέλεσης των προγραμμάτων, μειώνοντας ταυτόχρονα την κατανάλωση ισχύος και γι' αυτόν το λόγο έχουν πλέον αντικαταστήσει πλήρως όλους τους παραδοσιακούς μονοεπεξεργαστές.



Σχήμα 2.7.3 : Πολυπύρηνοι επεξεργαστές υλοποιημένοι με κοινό δίαυλο



Σχήμα 2.7.4 : Πολυπύρηνοι επεξεργαστές υλοποιημένοι με δίκτυο διασύνδεσης

Κεφάλαιο 3

Παράλληλη Επεξεργασία

Υποκεφάλαιο 3.1

ΕΙΣΑΓΩΓΗ ΣΤΗΝ ΠΑΡΑΛΛΗΛΗ ΕΠΕΞΕΡΓΑΣΙΑ ΚΑΙ ΤΙΣ ΑΡΧΙΤΕΚΤΟΝΙΚΕΣ ΤΗΣ

Τεχνικές παραλληλοποίησης εκτέλεσης εντολών είχαν χρησιμοποιηθεί σε πολύ μεγάλο βαθμό στο παρελθόν και συνεχίζουν να χρησιμοποιούνται μέχρι και σήμερα σε συνδυασμό με άλλες τεχνικές βελτιστοποίησης απόδοσης. Η παράλληλη επεξεργασία ωστόσο, είναι ένα σχετικά καινούργιο αντικείμενο μελέτης στην υπολογιστική επιστήμη .

Βασικό της χαρακτηριστικό της παράλληλης επεξεργασίας και ταυτόχρονα η ειδοποιός διαφορά της σε σχέση με άλλες τεχνικές / υλοποιήσεις συστημάτων είναι ότι στοχεύει στη βελτίωση της ταχύτητας επεξεργασίας **χωρίς την τροποποίηση της τεχνολογίας του υλικού.**

Μιλώντας γενικά, ένας υπολογιστής θεωρείται παράλληλος αν αποτελείται από πολλές επεξεργαστικές μονάδες, οι οποίες συνεργάζονται στενά για τη λύση του ίδιου προβλήματος σε χρόνο μικρότερο από το χρόνο που θα χρειαζόταν ένας επεξεργαστής μόνος του για να λύσει το ίδιο πρόβλημα. Οι επεξεργαστές αυτοί λέγονται *σφιχτά συνδεδεμένοι* (*tightly coupled*). (αναφορά σημειώσεις Διαμαντάρας)

Τα τελευταία δέκα χρόνια η παράλληλη επεξεργασία άρχισε να κάνει αισθητή την παρουσία της στον χώρο της αγοράς με την εμπορική διάθεση παράλληλων συστημάτων υπολογιστών, που αποτελούνται από διπλούς ή τετραπλούς επεξεργαστές Pentium. Το κόστος αυτών των συστημάτων ήταν μικρό και από αυτήν την άποψη μπορούσαν να χρησιμοποιηθούν και για προσωπική χρήση από ιδιώτες (για παράδειγμα ως υπολογιστές γραφείου).

Η καθαρή παράλληλη επεξεργασία απαιτεί στενή συνεργασία μεταξύ των επεξεργαστών για την εξαγωγή αποτελεσμάτων σε μικρότερο χρόνο συγκριτικά με το χρόνο που θα απαιτούσε μια παραδοσιακή μονάδα επεξεργασίας για το ίδιο αποτέλεσμα. Για να γίνει εφικτή αυτού του είδους η συνεργασία πρέπει οι επεξεργαστές να βρίσκονται φυσικά τοποθετημένοι πολύ κοντά ο ένας με τον άλλον, ακόμα και πάνω στην ίδια κάρτα (πλακέτα), γεγονός το οποίο βελτιώνει κατά πολύ την ταχύτητα της μεταξύ τους επικοινωνίας.

Ορισμένοι επεξεργαστές υλοποιούνται προσανατολισμένοι στην εκτέλεση συγκεκριμένων ή μιας κατηγορίας αλγορίθμων και ονομάζονται γι’ αυτό το λόγο επεξεργαστές **ειδικού**

σκοπού. Αυτοί διακρίνονται από τους επεξεργαστές **γενικού σκοπού,** οι οποίοι υλοποιούνται έτσι ώστε να εκτελούν οποιαδήποτε εφαρμογή.

Ένα βασικό ζήτημα, το οποίο χρειάζεται περαιτέρω ανάλυση είναι το επικοινωνιακό μοντέλο που θα ακολουθηθεί για την κατασκευή ενός παράλληλου συστήματος, με τρόπο που θα καθιστά το όλο σύστημα αποδοτικό. Οι βασικοί μηχανισμοί επικοινωνίας που υλοποιούνται στα παράλληλα συστήματα είναι δύο:

- Επικοινωνία με τη χρήση **κοινής μνήμης** (βασισμένη στο κατάλληλο υλικό)
- Επικοινωνία με **ανταλλαγή μηνυμάτων** (βασισμένη στο κατάλληλο λογισμικό)

  - Τα πλεονεκτήματα μιας υλοποίησης που χρησιμοποιεί **κοινή μνήμη** είναι:
    - Συμβατότητα με το γνωστό μοντέλο επικοινωνίας που συναντούμε και σε έναν παραδοσιακό επεξεργαστή.
    - Ευκολία στον προγραμματισμό εφαρμογών.
    - Ταχύτερη επικοινωνία.
    - Χρήση κρυφής μνήμης που μειώνει αισθητά τις απαιτήσεις για πρόσβαση στη μνήμη.

  - Αντιστοίχως, η υλοποίηση συστημάτων με **ανταλλαγή μηνυμάτων** προσφέρει τα εξής:
    - Απλούστερη καλωδιωμένη λογική, διότι σ’ αυτήν την περίπτωση δεν γίνεται χρήση πολύπλοκων κυκλωμάτων, που κανονικά είναι υπεύθυνα για τη διατήρηση της συνέπειας μεταξύ της κοινής μνήμης και της μνήμης cache.
    - Μεγαλύτερη δυνατότητα επεκτασιμότητα του συστήματος για τον απλό λόγο, ότι απλώς προσθέτουμε περισσότερους επεξεργαστές, οι οποίοι είναι ανεξάρτητοι ο ένας από τον άλλο.

Μέχρι τη στιγμή της συγγραφής της παρούσας εργασίας, το μεγαλύτερο μέρος των παράλληλων συστημάτων ακολουθούν την αρχιτεκτονική κοινής μνήμης με δίκτυο bus, έχει αρχίσει όμως να διαφαίνεται η μεταστροφή προς την αρχιτεκτονική κατανεμημένης μνήμης, όπου η επικοινωνία πραγματοποιείται με ανταλλαγή μηνυμάτων.

Υποκεφάλαιο 3.2

ΕΣΑΓΩΓΗ ΣΤΟΝ ΠΑΡΑΛΛΗΛΟ ΠΡΟΓΡΑΜΜΑΤΙΣΜΟ

Όπως αναφέρθηκε και προηγουμένως, βασικός στόχος του μοντέλου της παράλληλης επεξεργασίας είναι η εκτέλεση προγραμμάτων σε μικρότερο χρόνο από εκείνον που θα χρειαζόταν ένας μονοεπεξεργαστής ενός παραδοσιακού υπολογιστικού συστήματος. Για να μπορέσει να επιτευχθεί αυτός ο στόχος, δηλαδή να γίνει εφικτή η εκμετάλλευση της επεξεργαστικής παραλληλίας που προσφέρεται, θα πρέπει τα προς εκτέλεση προγράμματα να έχουν την κατάλληλη δομή. Σ' αυτήν την περίπτωση θα πρέπει οι προγραμματιστές να χρησιμοποιήσουν εργαλεία παράλληλου προγραμματισμού, ο οποίος όμως διαφέρει σε αρκετά σημεία από τον συμβατικό προγραμματισμό ενός μονοεπεξεργαστή.

Ένα πολύ σημαντικό ζήτημα, που αφορά τον προγραμματισμό για παράλληλη συστήματα είναι η επικοινωνία. Θα πρέπει στο σημείο αυτό να λάβουμε υπόψη μας, ότι τα προς εκτέλεση προγράμματα χωρίζονται σε μικρότερα τμήματα κώδικα (διεργασίες) τα οποία εκτελούνται κάποια συγκεκριμένη χρονική στιγμή από έναν εκ των επεξεργαστών.

Με τον όρο διεργασία εννοούμε ένα στιγμιότυπο προγράμματος που εκτελείται. Με τον όρο επικοινωνία επομένως, αναφερόμαστε στην επικοινωνία μεταξύ αυτών των διεργασιών.

Η διαδιεργασική επικοινωνία είναι απαραίτητη για τους εξής λόγους :

- Την **ορθή χρήση των κοινών πόρων** ενός συστήματος (κοινά δεδομένα ή κοινές περιοχές μνήμης). Στους πολυεπεξεργαστές που κάνουν χρήση κοινής μνήμης χρησιμοποιούνται τεχνικές, όπως οι σηματοφορείς και οι κλειδιαριές για την προστασία των αγαθών, όπου η πρόσβαση είναι δυνατή άμεσα από όλους. Στους πολυυπολογιστές η διαδικασία πρόσβασης γίνεται με ανταλλαγή μηνυμάτων μεταξύ των επεξεργαστών.

- Το **συγχρονισμό των διεργασιών**, που αποτελεί προϋπόθεση για τη σωστή εκτέλεση παράλληλου κώδικα σε ένα σύστημα.

- Την **ανταλλαγή δεδομένων**, η οποία στο μεν μοντέλο των πολυεπεξεργαστών συντελείται μέσω της κοινής μνήμης, στο δε μοντέλο πολυυπολογιστών χρησιμοποιείται η ανταλλαγή μηνυμάτων.

Στο εύλογο, ίσως, ερώτημα σχετικά με το πόση θα είναι η αύξηση της ταχύτητας εκτέλεσης ενός παράλληλου αλγορίθμου αν προσθέσουμε απεριόριστα νέους επεξεργαστές στο σύστημά μας η απάντηση είναι η εξής:

- ❖ Η **ιδανική** περίπτωση θα ήταν η επιτάχυνση του παράλληλου προγράμματος να αποτελεί γραμμική συνάρτηση του αριθμού των επεξεργαστών που προσθέτουμε στο σύστημα, οπότε στην περίπτωση αυτή θα μιλούσαμε για **τέλεια επεκτασιμότητα** (scalability) της παράλληλης αρχιτεκτονικής.
- ✓ Στην **πραγματικότητα** όμως, η επιτάχυνση ενός παράλληλου προγράμματος δεν μπορεί να είναι απεριόριστη. Από έναν συγκεκριμένο αριθμό επεξεργαστών και έπειτα, όσες και αν είναι οι προσθήκες επεξεργαστών στο σύστημα, αυτές δεν θα επιφέρουν **καμία μεταβολή** στην ταχύτητα εκτέλεσης του προγράμματος.

Αυτό γίνεται εύκολα κατανοητό αν σκεφτεί κανείς ότι:

- Υπάρχει σημαντική επιβάρυνση του χρόνου εκτέλεσης από τον χρόνο που δαπανάται για την επικοινωνία μεταξύ των επεξεργαστών.

- Είναι επίπονο, και τις περισσότερες φορές (πλην συγκεκριμένων εφαρμογών) αδύνατο να τεμαχιστεί ένας αλγόριθμος σε Ν ισοδύναμα παράλληλα τμήματα.

- Ένας αλγόριθμος δεν είναι ποτέ εξ ολοκλήρου παράλληλος. Εμπεριέχει σίγουρα ένα ποσοστό σειριακού κώδικα, το οποίο δεν μπορεί να τεμαχιστεί ώστε να εκτελεστεί με παράλληλο τρόπο. Η μέγιστη επιτάχυνση που μπορούμε να επιτύχουμε στην εκτέλεση του αλγορίθμου σχετίζεται άμεσα με το σειριακό κομμάτι του αλγορίθμου. Αυτή η τελευταία παράμετρος περιγράφει ένα βασικό πρόβλημα το οποίο ορίζεται από τον **νόμο του Amdahl**.

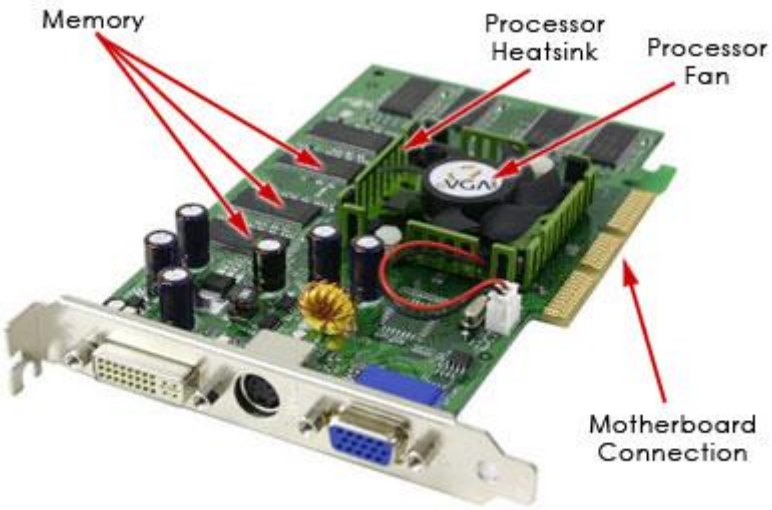
Η συνολική επιτάχυνση σύμφωνα με τον νόμο του Amdahl είναι ίση με:

$$\mathbf{T_{σειριακηςΜηχανης} / T_{παραλληληςΜηχανης} = N / [1 + (N-1)s]}$$

Για να αναφέρουμε ένα παράδειγμα, ας θεωρήσουμε οτι το 10% ενός αλγόριθμου είναι σειριακές εντολές που δεν παραλληλίζονται (άρα s=0.1). Η συνολική μας επιτάχυνση από την παράλληλη εκτέλεση του συγκεκριμένου αλγορίθμου είναι ίση με 1/s = 1/10 = 0.1 ακόμα και στην περίπτωση που διαθέσουμε 1.000 επεξεργαστές που θα λειτουργούν παράλληλα.

Υποκεφάλαιο 3.3

ΚΑΡΤΑ ΓΡΑΦΙΚΩΝ- ΠΕΡΙΓΡΑΦΗ



Σχήμα 3.3.1: Στοιχεία μιας κάρτας γραφικών

Όλες οι πληροφορίες (για παράδειγμα εικόνες-γραφικά , κείμενο κλπ) οι οποίες εμφανίζονται στην οθόνη ενός υπολογιστικού συστήματος παράγονται από την επεξεργασία δεδομένων κατάλληλης μορφής στα κυκλώματα μιας κάρτας γραφικών.

Στην ουσία πρόκειται για μία πλακέτα που περιλαμβάνει τυπωμένα κυκλώματα όπως μνήμες, μονάδες επεξεργασίας γραφικών κ.ά, η οποία αντιλαμβάνεται την οθόνη σαν ένα πλήθος από εκατομμύρια κελιά (pixels) η θέση καθενός από τα οποία καθορίζεται από ένα ζεύγος συντεταγμένων. Περιλαμβάνει επίσης και κυκλώματα που υλοποιούν το BIOS και το οποίο αναλαμβάνει την αποθήκευση των ρυθμίσεων της κάρτας και την εκτέλεση διαγνωστικών τεστ κατά την εκκίνηση του συστήματος.

Η κάρτα γραφικών δεν μπορεί φυσικά να λειτουργήσει αυτόνομα μέσα σε ένα σύστημα αλλά βρίσκεται πάντα σε συνεργασία με τις υπόλοιπες μονάδες του υπολογιστή. Συνδέεται με :

- την μητρική κάρτα του υπολογιστή (μέσω της οποίας λαμβάνει τα δεδομένα προς επεξεργασία και την απαραίτητη τροφοδοσία, εκτός και αν η κάρτα έχει μεγαλύτερες απαιτήσεις τροφοδοσίας οπότε συνδέεται απευθείας με το τροφοδοτικό),
- την κεντρική μνήμη ,
- την κεντρική μονάδα επεξεργασίας (που διαθέτει τα δεδομένα για κάθε pixel)

- και την οθόνη.

Όταν ο κεντρικός επεξεργαστής του συστήματος (CPU) εκτελεί μια συγκεκριμένη εφαρμογή προκειμένου να αναπαραστήσει τα γραφικά στοιχεία της εφαρμογής, τροφοδοτεί την κάρτα γραφικών με τα καταλλήλως διαμορφωμένα δεδομένα. Η κάρτα γραφικών στη συνέχεια εκτελεί την επεξεργασία, την μετάφραση και την αναπαράσταση των δεδομένων με κατάλληλο τρόπο στην οθόνη. Η κάρτα γραφικών οργανώνει την οθόνη σε μορφή πλέγματος (raster), την χωρίζει σε pixel και στη συνέχεια προσθέτει τον κατάλληλο φωτισμό/σκίαση, υφή και χρώμα σε κάθε pixel. Η διαδικασία αυτή είναι σχετικά απλή ως προς την κατανόηση της και η υλοποίηση της πραγματοποιείται σε φάσεις (περάσματα). Οι φάσεις αυτές απαιτούν την εκτέλεση εκατομμύρια υπολογισμών / δευτερόλεπτο, οι οποίοι αυξάνονται όσο η κάρτα καλείται να επεξεργαστεί πολυπλοκότερα και μεγαλύτερα γραφικά. Όπως είναι λογικό, καθίσταται δύσκολο για ένα κεντρικό σύστημα επεξεργασίας να αναλάβει αυτή την πρόσθετη εργασία, δεδομένου ότι είναι ήδη επιφορτισμένο με την εκτέλεση άλλων εντολών.

Κατά τις δεκαετίες 1960-1980, το μοναδικό μέσο για την οπτική αναπαράσταση των δεδομένων ήταν οι εκτυπωτές. Η χρήση των οθονών ως μονάδες ενός υπολογιστικού συστήματος ξεκίνησε σχετικά αργά:

- Έχουμε την εμφάνιση του προσωπικού υπολογιστή της IBM το 1981, ο οποίος ενσωμάτωνε και την πρώτη κάρτα γραφικών (αρχική ονομασία των καρτών ήταν ο αγγλικός όρος video card) με την ονομασία MDA (Monochrome Display Adapter). Η κάρτα αυτή είχε τη δυνατότητα αναπαράστασης κειμένου με λευκό ή πράσινο χρώμα σε μαύρο φόντο.

- Ακολούθησαν οι **CGA (Color Graphics Adaptor),** κάρτες με δυνατότητα απεικόνισης οκτώ χρωμάτων και οι **EGA(Enchanced Graphics Adaptor)** απεικόνισης 64 χρωμάτων. Το ελάχιστο πρότυπο διασύνδεσης για τις κάρτες γραφικών που χρησιμοποιούνται στα σημερινά υπολογιστικά συστήματα είναι το **VGA (Video Graphics Array)** με δυνατότητες απεικόνισης 256 χρωμάτων. Με την εμφάνιση του προτύπου **Quantum Extended Graphics Array (QXGA)** είναι δυνατή η απεικόνιση χρωμάτων με ανάλυση μεγαλύτερης των 2040 x 1536 pixels.

- Από το 1995 και έπειτα οι κάρτες γραφικών συνεχίζουν να εξελίσσονται με υλοποιήσεις που μπορούν να απεικονίσουν δισδιάστατα και τρισδιάστατα γραφικά. Το 1997 η εταιρία 3dfx κυκλοφορεί την κάρτα με την ονομασία Voodoo Rush για 2D και λίγο αργότερα την Voodoo 2 για 3D γραφικά. Η Intel χρησιμοποίησε την τεχνολογία **AGP (Accelarated Graphics Port)** για τη σύνδεση μεταξύ της μητρικής κάρτας και της κάρτας γραφικών. Η συγκεκριμένη τεχνολογία αργότερα αντικαταστάθηκε από την γρηγορότερη PCI και τελικά την

PCIexpress .

Σχήμα 3.3.2 : Πολλαπλές κάρτες γραφικών συνδεδεμένες σε μία μητρική κάρτα

Υποκεφάλαιο 3.4

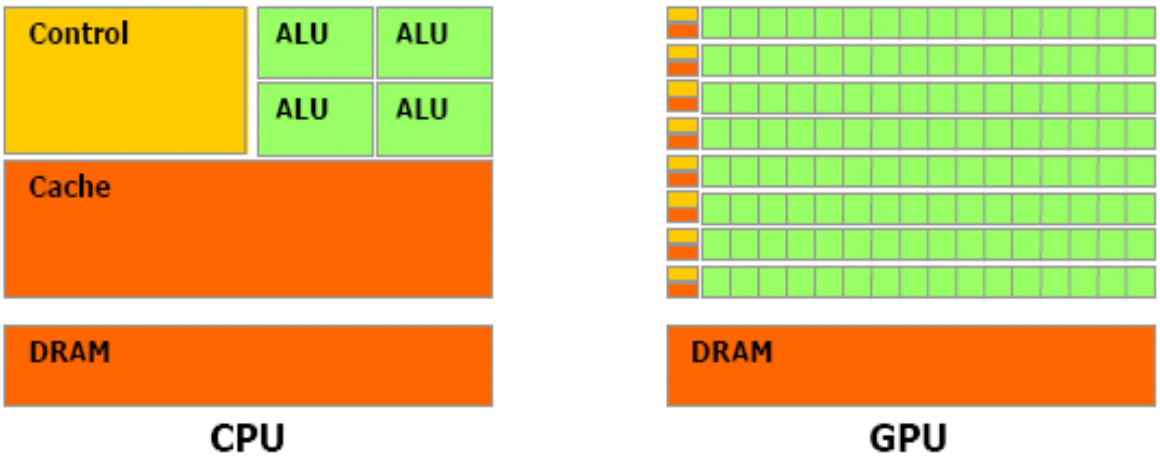
ΜΟΝΑΔΑ ΕΠΕΞΕΡΓΑΣΙΑΣ ΓΡΑΦΙΚΩΝ

Όπως αναφέρθηκε και παραπάνω, τη δεκαετία 1980-1990 οι οθόνες άρχισαν να αποτελούν μέρος των υπολογιστικών συστημάτων αυξάνοντας τις απαιτήσεις για επεξεργασία γραφικών. Ειδικά εκείνη την εποχή έγινε η εισαγωγή των γραφικών λειτουργικών συστημάτων στους προσωπικούς υπολογιστές (Windows) και από εκείνη τη στιγμή και έπειτα άρχισαν να εμφανίζονται κάρτες γραφικών, που σαν στόχο είχαν να υποβοηθοηθούν και να επιταχύνουν σημαντικά όλες τις δισδιάστατες γραφικές λειτουργίες του υπολογιστή. Μάλιστα, τη δεκαετία του ‘80 η Silicon Graphics ήταν η εταιρεία που ξεκίνησε να προωθεί τη χρήση γραφικών σε τρισδιάστατη μορφή για διάφορες εφαρμογές, που ακόμη βέβαια είχαν ένα μικρό αγοραστικό κοινό (στρατός, κυβερνητικές και επιστημονικές υπηρεσίες).

Μια κάρτα γραφικών περιλαμβάνει φυσικά πολλές υπομονάδες- κυκλώματα, όπως αναφέρθηκε και πρωτύτερα, αλλά το σημαντικότερο τμήμα της ήταν και θα παραμείνει η μονάδα επεξεργασίας γραφικών **(GPU-Graphics Processing Unit)**. Βασικό χαρακτηριστικό της GPU είναι το ότι είναι επιφορτισμένη με την επεξεργασία δεδομένων και την εκτέλεση υπολογισμών, που επενεργούν πάνω σε αυτά. Από αυτήν την άποψη, μια

GPU θα μπορούσαμε να πούμε οτι προσομοιάζει με τη λειτουργία ενός κεντρικού επεξεργαστή. Πέραν αυτής της ομοιότητας, η μονάδα επεξεργασίας γραφικών και ένας επεξεργαστής διαφέρουν σε ορισμένα σημεία :

11



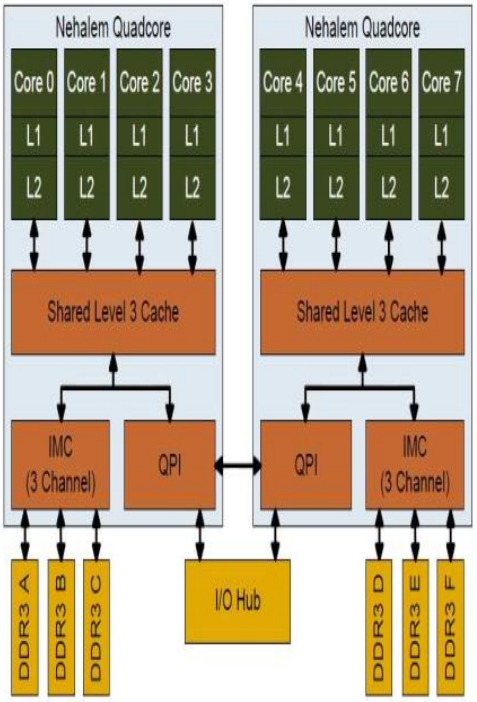
Σχήμα 3.4 : Σχεδιαστικές διαφορές μεταξύ CPU και GPU

- **Στη φύση των υπολογισμών τους οποίους καλούνται να εκτελέσουν**. Μια GPU είναι ένας επεξεργαστής με περισσότερο εξειδικευμένες λειτουργίες σε σχέση με μια CPU, και γι' αυτό το λόγο δεν μπορεί να την αντικαταστήσει. Οι πυρήνες μιας CPU (από τους 2 έως τους 16 πυρήνες που μπορεί να περιλαμβάνει μια πολυπύρηνη υλοποίηση) εκτελούν με όσο το δυνατόν αποδοτικό τρόπο προγράμματα, τα οποία είναι οργανωμένα σε νήματα που μπορούν να εκτελεστούν παράλληλα. Μια GPU αντίστοιχα, είναι ένας επεξεργαστής με πολύ μεγαλύτερο αριθμό πυρήνων και είναι ειδικά υλοποιημένος για να εκτελεί παράλληλα πολύπλοκους μαθηματικούς και γεωμετρικούς υπολογισμούς πάνω σε δεδομένα ίδιας φύσης. Επομένως, αυτό που καταφέρνουν οι πυρήνες ενός επεξεργαστή της κάρτας γραφικών είναι να εκτελούν με πολύ αποδοτικό τρόπο τον ίδιο αλγόριθμο σε ένα μεγάλο πλήθος (ή ακόμα και ένα τεράστιο πλήθος) δεδομένων. Αυτού του τύπου οι αλγόριθμοι θα απαιτούσαν μέρες ή και μήνες (ανάλογα με το πλήθος των δεδομένων και την πολυπλοκότητα του αλγορίθμου)

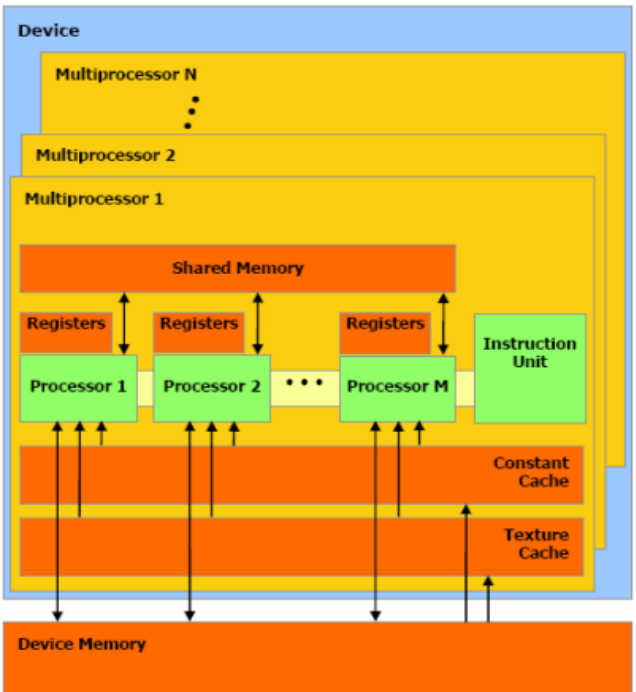
αν εκτελούνταν αποκλειστικά σε μια πολυπύρηνη CPU. Ιδανικοί αλγόριθμοι που προορίζονται για εκτέλεση στους πυρήνες μιας GPU είναι τα προγράμματα επεξεργασίας εικόνας και πολυμέσων, αλγόριθμοι κωδικοποίησης βίντεο και άλλα προγράμματα εκτός του πεδίου των πολυμέσων, όπως αλγόριθμοι επεξεργασίας σήματος και οι οικονομικές και βιολογικές εφαρμογές που έχουν τεράστιες υπολογιστικές απαιτήσεις .

- **Στη διαφορετική αρχιτεκτονική και σχεδίαση**. Εξαιτίας της διαφορετικής φύσης των υπολογισμών τους οποίους εκτελούν και για τους οποίους αποδίδουν σαφώς καλύτερα, οι δύο διαφορετικές μονάδες (CPU και GPU) σχεδιάζονται και υλοποιούνται με διαφορετικό τρόπο. Όσον αφορά τη CPU, το μεγαλύτερο τμήμα της αποτελείται από τις μονάδες ελέγχου και ένα μικρότερο κομμάτι αφήνεται για τα κυκλώματα εκτέλεσης υπολογισμών. Σε μια GPU αντίθετα, το μεγαλύτερο ποσοστό κατέχουν τα τρανζίστορ που είναι υπεύθυνα για την εκτέλεση πράξεων και αυτό λόγω των απαιτήσεων για πολλούς παράλληλους υπολογισμούς. Τα κυκλώματα για τις μνήμες cache και τις μονάδες ελέγχου αποτελούν το μικρότερο κομμάτι της υλοποίησης μιας GPU. Αυτή η προσέγγιση είναι δικαιολογημένη, αφού το ίδιο πρόγραμμα εκτελείται πάνω σε πολλά δεδομένα, οπότε υπάρχουν μειωμένες απαιτήσεις για κυκλώματα ελέγχου ροής εντολών. Επίσης, ο μεγάλος χρόνος προσπέλασης της μνήμης μπορεί να ισοσταθμιστεί από τον χρόνο που εξοικονομεί η παράλληλη εκτέλεση των υπολογισμών, οπότε δεν υπάρχουν αυξημένες απαιτήσεις ούτε και για μνήμες cache.

- **Στον διαφορετικό τρόπο οργάνωσης και διαχείρισης μνήμης.** Προς χάριν της κατανόησης του διαφορετικού τρόπου οργάνωσης της μνήμης ανάμεσα στους πολυπύρηνους επεξεργαστές και τις κάρτες γραφικών, γίνεται αμέσως παρακάτω η παράθεση δύο αρχιτεκτονικών μνήμης: των πολυπύρηνων επεξεργαστών Nehalem και μιας κάρτας γραφικών με υλοποίηση CUDA.


Nehalem Quadcore
Core 0
Core 1
Core 2
Core 3
Core 4
Core 5
Core 6
Core 7
L1
L2
Shared Level 3 Cache
IMC
(3 Channel)
QPI
I/O Hub
DDR3 A
DDR3 B
DDR3 C
DDR3 D
DDR3 E
DDR3 F

0011


Device
Multiprocessor N
Multiprocessor 2
Multiprocessor 1
Shared Memory
Registers
Processor 1
Processor 2
Processor M
Instruction Unit
Constant Cache
Texture Cache
Device Memory

Οι πολυπύρηνοι επεξεργαστές Bloomfield της Intel με την κωδική ονομασία Nehalem (οι οποίοι αποτελούν την εξέλιξη των κλασικών multicore επεξεργαστών όπως για παράδειγμα των 2-πύρηνων intel Core) διαθέτουν 4 πυρήνες με κάθε πυρήνα να έχει την δική του L1 και L2 cache (με μέγεθος μερικά KB) . Οι πυρήνες έχουν πρόσβαση στην κοινή μνήμη cache L3 (μεγέθους 8MB) και επικοινωνούν μεταξύ τους μέσω αυτής . Το κύκλωμα της κοινής cache υλοποιεί σηματοφόρους που εξασφαλίζουν την συνοχή των δεδομένων μεταξύ αυτής και των cache ανώτερων επιπέδων των πυρήνων.

Μέσα σε έναν πολυεπεξεργαστή κάρτας, ένα νήμα μπορεί να διαβάσει και να γράψει δεδομένα στον καταχωρητή, την τοπική, την κοινόχρηστη μνήμη καθώς επίσης και την καθολική μνήμη της συσκευής. Μπορεί όμως μόνο να διαβάσει από τη μνήμη σταθερών (constant memory) και την μνήμη υφών (texture memory). Πρόσβαση για ανάγνωση και καταχώρηση δεδομένων στην καθολική μνήμη της συσκευής έχει και το κεντρικό σύστημα host.

Υποκεφάλαιο 3.5

ΠΡΟΓΡΑΜΜΑΤΙΣΜΟΣ ΜΙΑΣ ΚΑΡΤΑΣ ΓΡΑΦΙΚΩΝ

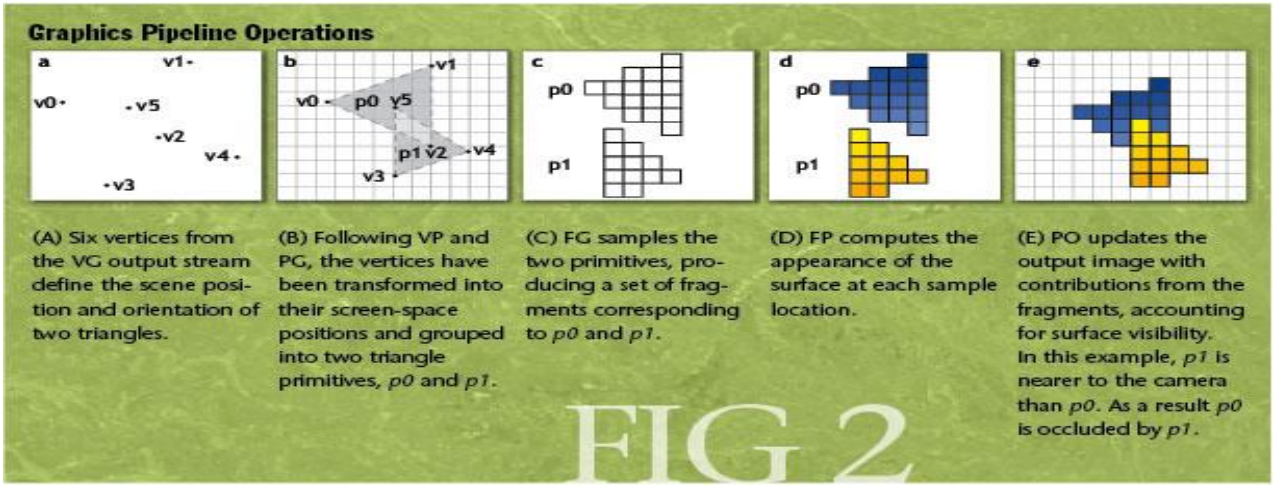
Από το 1980 μέχρι τα τέλη της δεκαετίας του '90, οι κάρτες γραφικών χρησιμοποιούσαν διοχετεύσεις σταθερών συναρτήσεων, των οποίων οι παράμετροι μπορούσαν να ρυθμιστούν, αλλά δεν μπορούσαν να προγραμματιστούν.

Για τον προγραμματισμό των διοχετεύσεων αυτών σταδιακά άρχισαν να αναπτύσσονται μερικά APIs όπως το Direct3D ή το OpenGL. Ειδικά το OpenGL καθιερώθηκε ως ανοικτό πρότυπο το1992 από την Silicon Graphics που το κατασκεύασε. Με αυτήν την κίνηση η εταιρεία ευελπιστούσε, ότι το OpenGLθα τυποποιηθεί διεθνώς ως ένα διασυστημικό πακέτο βιβλιοθηκών για τη συγγραφή γραφικών εφαρμογών σε ένα 3D περιβάλλον.

Από τα μέσα κυρίως της δεκαετίας του ’90 αυξάνονται οι απαιτήσεις των πελατών για εφαρμογές γραφικών σε 3D περιβάλλον και το γεγονός αυτό υποβοηθείται και από την εισαγωγή των PC Games, τα οποία χρησιμοποιούν τέτοιου είδους γραφικά. Επομένως, η σχεδίαση των GPU διαμορφώθηκε κατά κύριο λόγο από τη βιομηχανία των PC Games, η οποία εξελισσόταν με ταχείς ρυθμούς και ασκούσε πιέσεις για ολοένα και καλύτερες επιδόσεις. Η Nvidia με την έκδοση της κάρτας γραφικών GeForce 256 άρχισε να ξεπερνά τα υπολογιστικά όρια που υπήρχαν μέχρι τότε, καθώς για πρώτη φορά υπολογισμοί όπως π.χ. η σκίαση και ο φωτισμός ενός αντικειμένου δεν γινόταν στην CPU, αλλά απευθείας πάνω στα κυκλώματα της κάρτας γραφικών.

Αφού οι κάρτες γραφικών αποτελούνταν από πολυπύρηνους και πολυνηματικούς επεξεργαστές, με την εμφάνιση των πρώτων APIs εμφανίστηκε και η ιδέα της χρήσης της υπολογιστικής δύναμης των GPU και για άλλους υπολογισμούς που δεν σχετίζονται με γραφικά. Παρόλα αυτά, η ιδέα αυτή ήταν εξαιρετικά περίπλοκη για να είναι εφαρμόσιμη σε αυτό το στάδιο, και αυτό οφειλόταν στην πολυπλοκότητα των υπολογισμών που εκτελούνται μέσα σε μια ακολουθία εντολών μιας κάρτας γραφικών.

<u>Απλοποιημένη ακολουθία εκτέλεσης εντολών στην κάρτα γραφικών</u>



Σχήμα 3.5.1 : Ακολουθία εντολών στην κάρτα γραφικών

Οι λειτουργίες που επιτελούνται μέσα στη γραμμή εντολών της κάρτας είναι :

- **VG (vertex generation)**: Ένα αντικείμενο αναπαρίσταται ως ένα σύνολο κορυφών που αποτελούν απλά γεωμετρικά σχήματα. Τις πληροφορίες για τις ακριβείς συντεταγμένες των κορυφών αυτών η κάρτα τις παίρνει από τη μνήμη.

- **VP (vertex processing)**: Κάθε μία κορυφή υπόκειται σε επεξεργασία και παράγεται έτσι κατά αντιστοιχία μια άλλη κορυφή με διαφορετικές ιδιότητες από την πρώτη.

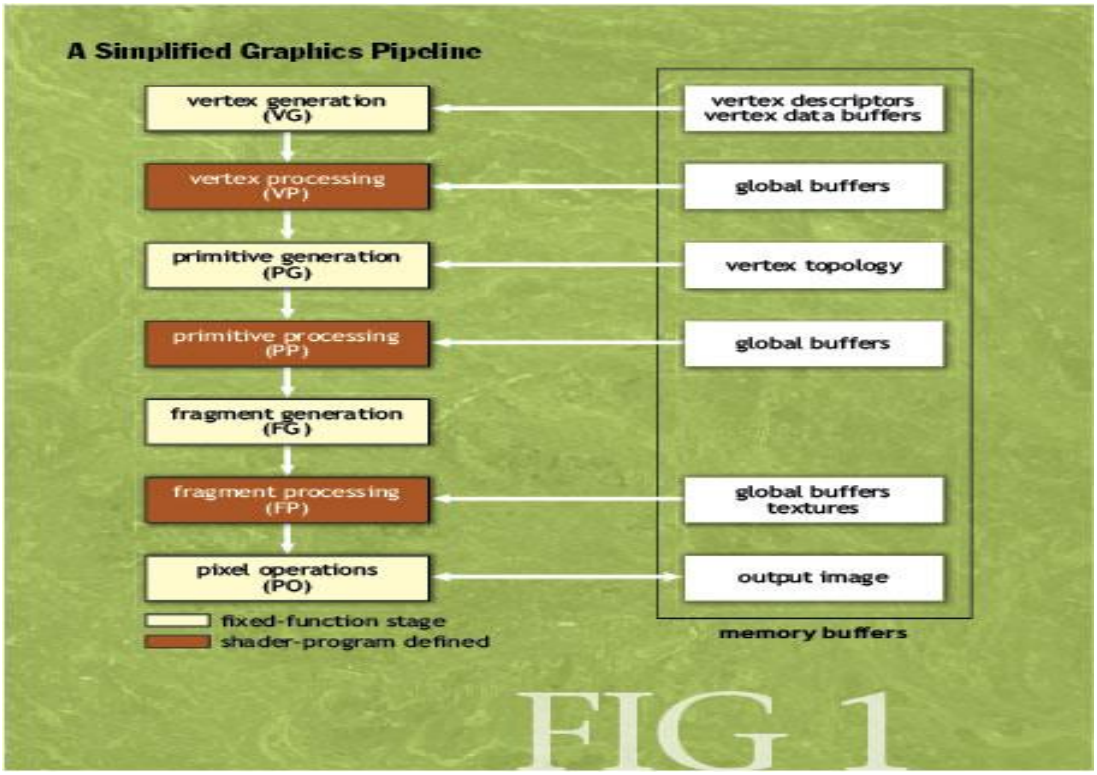
- **PG (primitive generation)**: Οι κορυφές ομαδοποιούνται για να σχηματίσουν τα γεωμετρικά σχήματα για περαιτέρω επεξεργασία (π.χ. τρίγωνα).

- **PP (primitive processing**): Τα απλά γεωμετρικά σχήματα υπόκεινται σε επεξεργασία για να αναλύονται λεπτομερέστερα τα γεωμετρικά σχήματα.

- **FG (fragment generation)**: Ονομάζεται και rasterization και κατά τη διαδικασία αυτή κάθε γεωμετρικό σχήμα που δημιουργήθηκε από την PP προβάλλεται στα pixels της οθόνης και καθορίζεται η εμφάνισή τους.

- **FP (fragment processing)**: Προστίθεται το εφέ του φωτισμού, το χρώμα της επιφάνειας και το πόσο διαφανής θα είναι ή όχι αυτή. Εδώ γίνεται η ρεαλιστική (όσο κατά το δυνατόν) απεικόνιση του αντικειμένου και χρησιμοποιούνται δεδομένα που λαμβάνονται από έναν ειδικό τύπο μνήμης της κάρτας που ονομάζεται texture μνήμη.

- **PO (pixel operations)**: Εδώ υπολογίζεται ο τρόπος με τον οποίο θα φαίνεται κάθε pixel ενός αντικειμένου σε σχέση με κάποιο άλλο ή άλλα αντικείμενα που βρίσκονται πιο κοντά στην εικονική κάμερα.



Σχήμα 3.5.2 : Ακολουθία εντολών κάρτας γραφικών και χρήση προσωρινής μνήμης (buffers)

Από τις παραπάνω αναφερθείσες λειτουργίες, οι λειτουργίες VP, PP και FP είναι προγραμματιζόμενες με τη χρήση συναρτήσεων που είναι γνωστές ως shaders. Η σύνταξη των εντολών των συναρτήσεων αυτών γίνεται με shading γλώσσες όπως η Cg της Nvidia, η GLSL της OpenGL και η HLSL της Microsoft. Ο κώδικας από αυτές τις γλώσσες μεταφράζεται σε bytecode και στη συνέχεια μετατρέπεται σε δυαδικό κώδικα ανάλογα με την κάρτα γραφικών όπου θα εκτελεστεί.

Οι κάρτες γραφικών είναι σχεδιασμένες να “χρωματίζουν” κάθε pixel της οθόνης με βάση τις πληροφορίες που παίρνουν από αριθμητικές προγραμματιζόμενες μεταβλητές που ονομάζονταν pixel shaders. Οι συναρτήσεις shaders συνδυάζουν τις συντεταγμένες (x,y) που χαρακτηρίζουν κάθε pixel μαζί με άλλα δεδομένα (χρώμα, υφή, συντεταγμένες και άλλες ιδιότητες). Καθώς τα μόνα APIs για GPU προγραμματισμό ήταν η OpenGL και η DirectX, ο μοναδικός τρόπος να εκτελεστεί ένας γενικού σκοπού αλγόριθμος στην GPU, ήταν να εκφραστεί ως ένας αλγόριθμος γραφικών, οπότε το πρόβλημα και η λύση του μετατρέπονταν πλέον από πρόβλημα γενικού σκοπού σε πρόβλημα γραφικής φύσης.

Αντί αυτού του τύπου των δεδομένων (υφή, χρώμα), θα μπορούσαν να περαστούν ως παράμετροι των συναρτήσεων shaders, διαφορετικού είδους δεδομένα. Στην ουσία δίνονται προς επεξεργασία τα δεδομένα που αφορούν τους προς επίλυση αλγόριθμους. Αυτή η τεχνική προγραμματισμού ονομάστηκε GPGPU (General Programming using a Graphics Processing Unit).

Αν και η ιδέα αυτή λειτούργησε, υπήρχαν παράγοντες που την καθιστούσαν ιδιαίτερα περίπλοκη για να μπορέσει να αποτελέσει βοήθημα για μαζικό προγραμματισμό σε κάρτες γραφικών. Οι λόγοι αυτοί ήταν :

- Οι περιορισμοί σχετικά με τον αριθμό των δεδομένων που θα μπορούσαν να δοθούν ως παράμετροι στα shaders .
- Ο τρόπος με τον οποίο θα αποθηκευτούν τα αποτελέσματα στη μνήμη (ειδικά αν πρόκειται για ένα εύρος από διευθύνσεις και όχι για συγκεκριμένες θέσεις μνήμης).
- Οι υπολογισμοί floating-point δεν είχαν την επιθυμητή ακρίβεια που απαιτούσαν οι υπολογισμοί.
- Η μετατροπή και η συγγραφή των υπολογισμών έπρεπε να γραφτεί σε shading γλώσσες από τους ίδιους τους προγραμματιστές -κάτι το οποίο προϋπέθετε και τις απαραίτητες δεξιότητες (αν υποθέσουμε ότι η εξοικείωση με το OpenGL / DirectX πρότυπο υπήρχε ήδη).

Ο συνδυασμός όλων αυτών των περιορισμών αποτελούσε ένα πολύ μεγάλο εμπόδιο για όλους τους επίδοξους προγραμματιστές και καθιστούσε δύσκολη τη συνεργασία μεταξύ των πολλαπλών επεξεργαστών της κάρτας γραφικών, οι οποίοι θα έπρεπε να επικοινωνούν μεταξύ τους για την εκτέλεση των υπολογισμών.

Καθώς δεν υπήρχαν ακόμη τα κατάλληλα εργαλεία, ο προγραμματισμός των καρτών γραφικών σε ικανοποιητικό επίπεδο ήταν δύσκολος. Το 2007, με την παρουσίαση της πλατφόρμας **CUDA (Compute Unified Device Architecture)** της **Nvidia** σημειώθηκε η πολυπόθητη αλλαγή που επέτρεψε τη συγγραφή κώδικα για επεξεργαστές στις κάρτες γραφικών και μάλιστα με τη χρήση εργαλείων προγραμματισμού σε γλώσσα C/C++.

Κεφάλαιο 4

Μοντέλα Παράλληλης Επεξεργασίας

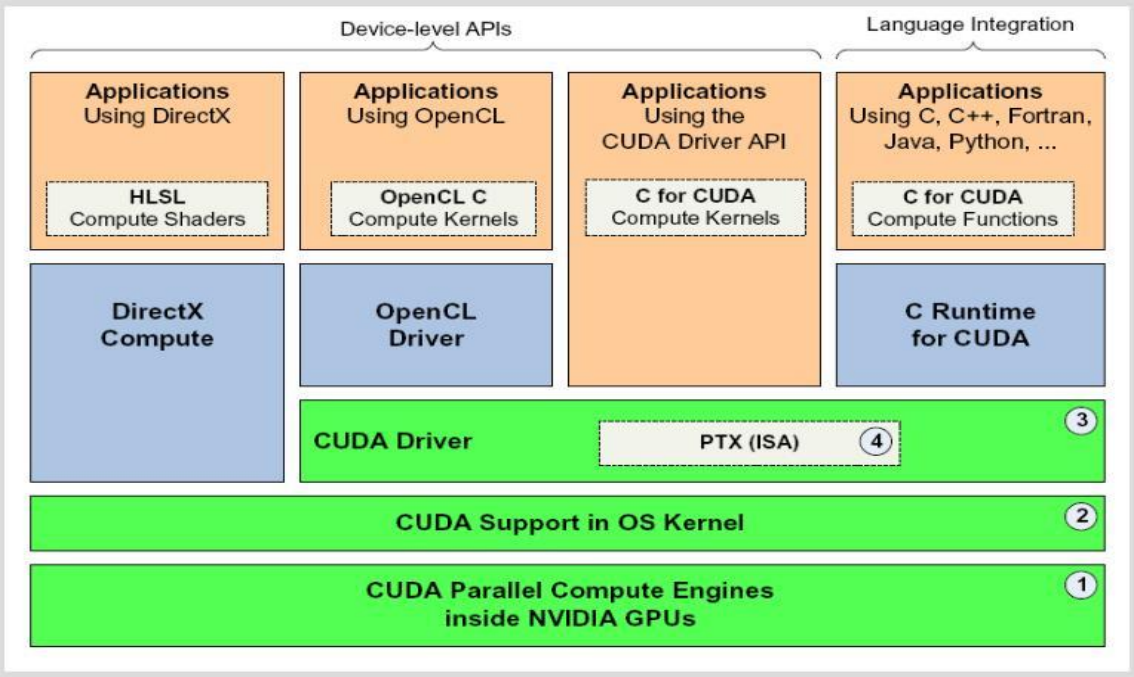
Υποκεφάλαιο 4.1

ΜΟΝΤΕΛΑ ΠΡΟΓΡΑΜΜΑΤΙΣΜΟΥ ΠΑΡΑΛΛΗΛΟΥ ΚΩΔΙΚΑ

Υπάρχουν πολλά μοντέλα για τη συγγραφή και την εκτέλεση παράλληλων προγραμμάτων. Τα μοντέλα αυτά χωρίζονται σε 2 βασικές κατηγορίες, ανάλογα με την οργάνωση της μνήμης των συστημάτων στα οποία βρίσκουν εφαρμογή, τα μοντέλα **κοινής** και τα μοντέλα **κατανεμημένης** μνήμης.

Τα μοντέλα κοινής μνήμης όπως το OpenCL, το DirectCompute και φυσικά το μοντέλο CUDA θα περιγραφούν σε ξεχωριστές ενότητες που θα αναπτυχθούν για το κάθε επιμέρους μοντέλο, έτσι ώστε να είναι δυνατή η εποικοδομητική μεταξύ τους αντιπαραβολή.

Όσον αφορά τα μοντέλα κατανεμημένης μνήμης, θα αναφερθούν κάποια βασικά στοιχεία τους χωρίς όμως επιπλέον ανάλυση, καθότι αυτά τα μοντέλα περιλαμβάνουν υλοποιήσεις διαφορετικές από αυτήν του μοντέλου CUDA.



Σχήμα 4.1.1 : Διαστρωμάτωση CUDA και άλλα προγραμματιστικές πλατφόρμες

- **Μοντέλα που χρησιμοποιούν κοινή μνήμη**

1. Ένα από αυτά τα μοντέλα είναι το **OpenMP** το οποίο χρησιμοποιείται σε συστήματα πολυεπεξεργαστών με κοινόχρηστη μνήμη, επομένως όλη η επικοινωνία γίνεται μέσω αυτής και δεν απαιτείται η ανταλλαγή μηνυμάτων. Βασικό μειονέκτημα του OpenMP είναι η έλλειψη scalability, καθώς δεν μπορεί να επεκταθεί σε περισσότερους από 200 κόμβους λόγω της επιβάρυνσης που προκαλείται από τη διαχείριση των νημάτων και τη διατήρηση της συνοχής της cache (cache coherence).

2. Άλλο μοντέλο της κατηγορίας αυτής, και μάλιστα αυτό το οποίο θα μελετήσουμε διεξοδικά σε σχέση με τα υπόλοιπα, είναι το μοντέλο προγραμματισμού **CUDA**. Η συγκεκριμένη πλατφόρμα, την οποία εισήγαγε η Nvidia το 2007, χρησιμοποιεί κοινή μνήμη για παράλληλη εκτέλεση αλγορίθμων σε επεξεργαστικά στοιχεία, που είναι στενά συνδεδεμένα μεταξύ τους (tightly-coupled). Για την επικοινωνία και τη ανταλλαγή δεδομένων μεταξύ της CPU και της GPU χρησιμοποιούνται παρόμοιες προγραμματιστικές τεχνικές με τη μεταβίβαση μηνυμάτων του μοντέλου MPI καθώς προς το παρόν υπάρχουν περιορισμένες δυνατότητες για χρήση της κοινόχρηστης μνήμης για τον συγκεκριμένο σκοπό. Το μοντέλο CUDA χαρακτηρίζεται από απλή διαχείριση των νημάτων -κάτι που συνεπάγεται χαμηλή επιβάρυνση στους κόμβους - και επιπλέον δεν απαιτεί τη χρήση ειδικών κυκλωμάτων για τη διατήρηση της συνοχής της cache. Με αυτόν τον τρόπο αποκτά δυνατότητες για επεκτασιμότητα σε μεγάλο αριθμό κόμβων.

3. Στην ίδια κατηγορία ανήκει και το **OpenCL** το οποίο προέκυψε από την συνεργασία εταιρειών όπως η Apple , Nvidia, Samsung, AMD/ATI, Intel, Google και πολλές άλλες, ως μια εναλλακτική πλατφόρμα απέναντι στο μοντέλο **CUDA**. Η αρχιτεκτονική του προσομοιάζει με την αρχιτεκτονική του μοντέλου της **CUDA** και οι όποιες διαφορές υπάρχουν, έχουν να κάνουν κυρίως με θέματα ονοματολογίας. Το βασικό του χαρακτηριστικό είναι η φορητότητα των εφαρμογών του μεταξύ ετερογενών συστημάτων CPU και GPU. Αυτό άμεσα συνεπάγεται και αυξημένη πολυπλοκότητα στους αλγόριθμους που υλοποιούν το **OpenCL.**

4. Τέλος, θα πρέπει να αναφέρουμε και το DirectCompute, το μοντέλο προγραμματισμού για κάρτες γραφικών που ανέπτυξε η Microsoft και το οποίο ενσωματώνεται στο DirectX API. Η συγγραφή προγραμμάτων βασίζεται στην .hlsl που αποτελεί γλώσσα προγραμματισμού για συναρτήσεις shader, ενώ χρησιμοποιεί αντίστοιχες έννοιες προγραμματισμού με αυτές των υπολοίπων μοντέλων.

- **Μοντέλα που χρησιμοποιούν κατανεμημένη μνήμη.**

1. Το μοντέλο OpenMP περιλαμβάνει συναρτήσεις ή οδηγίες προς των μεταφραστή και μεταβλητές περιβάλλοντος, οι οποίες καλούνται από διάφορες γλώσσες υψηλού επιπέδου όπως η C, C++, Fortran. Ένα πρόγραμμα OpenMP ξεκινάει με

```
#include <omp.h>
 #include <stdio.h>
 main(int argc, char *argv[]) {
int myThreadId, numThreads, var;
/* Σειριακός κώδικας */.

/* Αρχή παράλληλου τμήματος. Δημιούργησε με fork ένα πλήθος νημάτων
Καθόρισε ποιές μεταβλητές θα είναι ιδιωτικές σε κάθε νήμα
 ή κοινές για όλα τα νήματα */
12 #pragma omp parallel private(myThreadId, var) shared(numThreads)
 { /* Παράλληλο τμήμα - εκτελείται απ' όλα τα νήματα */
 myThreadId = omp_get_thread_num(); /* get my thread-id (private) */
 numThreads = omp_get_num_threads(); /* συνολικό πλήθος νημάτων */
 if (myThreadId == 0) {
... /* εκτελείται από το νήμα 0 */
 } else if (myThredId == 1) {
 ... /* εκτελείται από το νήμα 1 */
} else if (myThredId == 2) {
 ... /* εκτελείται από το νήμα 2
 }
 } /* Τέλος παράλληλου τμήματος. Όλα τα νήματα καταστρέφονται
εκτός από ένα που συνεχίζει */
 /* Συνεχίζεται ο σειριακός κώδικας */
 }
```

Κώδικας 4.1.2: OpenMP

μια σειριακή διεργασία δημιουργώντας ένα αρχικό νήμα. Στη συνέχεια η κεντρική διεργασία μπορεί να παράγει περισσότερα νήματα μέσω της εντολής fork και δίνοντας σε κάθε ένα από αυτά έναν μοναδικό αριθμό. Τα νήματα αυτά λειτουργούν παράλληλα και μπορούν να συνενωθούν σε οποιοδήποτε στάδιο της εκτέλεσης μέσω της εντολής join.

2. Το **MPI (Message Passing Interface)** που χρησιμοποιείται σε υπολογιστικά συστήματα συστοιχιών (computer clusters). **Συστοιχίες** ονομάζονται τα συστήματα εκείνα που επιτρέπουν την εκτέλεση απαιτητικών παράλληλων εφαρμογών, χρησιμοποιούν εξελιγμένους μηχανισμούς για τον διαμοιρασμό των πόρων όλων των διαθέσιμων κόμβων και είναι εύκολα επεκτάσιμα. Το **MPI** είναι μια πλατφόρμα στην οποία όλη η κοινή χρήση των δεδομένων και η επικοινωνία

μεταξυ των κόμβων επιτυγχάνεται μέσω της μεταβίβασης μηνυμάτων, διότι δεν υπάρχει κοινόχρηστη μνήμη. Όπως μπορεί να αντιληφθεί κάποιος, η συγγραφή ενός προγράμματος που θα προβλέπει και θα διαχειρίζεται την ανάγκη για μεταφορά των μηνυμάτων μεταξύ των κόμβων είναι μια αρκετά επίπονη και χρονοβόρα διαδικασία.

```
#include "stdio.h"
#include "mpi.h"
int main(int argc, char *argv[]) {
int rank, size;
int sum,startval,endval,accum;
int i,j;
MPI_Status status;
MPI_Init(&argc,&argv);
MPI_Comm_size(MPI_COMM_WORLD, &size);
MPI Comm rank(MPI COMM WORLD, &rank);
sum = 0; // zero sum for accumulation
startval = 1000*rank/size+1;
endval = 1000*(rank+1)/size;

for(i = startval; i <= endval; i++)
sum = sum + i;
if(rank != 0)
MPI_Send(&sum,1,MPI_INT,0,1,MPI_COMM_WORLD);
else
for(j = 1; j < size; j++) {
MPI_Recv(&accum,1,MPI_INT,j,1,MPI_COMM_WORLD, &status);
sum = sum + accum;
}
if(rank == 0)
printf("The sum from 1 to 1000 is: %i\n",sum);
MPI_Finalize();
}
```

Κώδικας 4.1.3: MPI

Όπως φαίνεται στο παραπάνω παράδειγμα κώδικα, οι επεξεργαστές της συστοιχίας ανάλογα με το rank στο οποίο ανήκουν εκτελούν είτε τη συνάρτηση MPI_Send, είτε την MPI_Recv. Όσον αφορά το rank των επεξεργαστών, αυτός ο επεξεργαστής που κάνει την εκκίνηση του MPI προγράμματος έχει rank 0 (οπότε εκτελεί την MPI_Send) ενώ όλοι οι υπόλοιποι έχουν τα υπόλοιπα rank από 1 έως n (και άρα εκτελούν το τμήμα κώδικα που αντιστοιχεί στην MPI_Recv) .

1. Άλλο μοντέλο παράλληλης επεξεργασίας με κατανεμημένη μνήμη είναι το **Hadoop** της Apache. Το μοντέλο αυτό χρησιμοποιεί μια ενιαία, γραμμένη σε γλώσσα Java, πλατφόρμα προγραμματισμού για παράλληλη επεξεργασία πολύ μεγάλου όγκου δεδομένων σε συστοιχίες υπολογιστικών συστημάτων. Το μεγαλύτερο μέρος της επεξεργασίας αφορά το κομμάτι του **MapReduce**, μια διασύνδεση η οποία τεμαχίζει τον κύριο όγκο των δεδομένων σε μικρότερα ανεξάρτητα κομμάτια και αναθέτει το κάθε ένα από αυτά στους κόμβους της συστοιχίας. Το μοντέλο **MapReduce** στηρίζεται σε δύο διεργασίες, οι οποίες εκτελούνται για κάθε κόμβο ξεχωριστά:

    - **master JobTracker**. Η διεργασία αυτή προγραμματίζει τους προς εκτέλεση υπολογισμούς, ελέγχει και επανεκτελεί τις διεργασίες που απέτυχαν να ολοκληρωθούν.
    - **slave TaskTracker**. Αυτή η διεργασία εκτελεί τους υπολογισμούς που αναθέτει ο JobTracker στους κόμβους.

Το μοντέλο αυτό λειτουργεί οργανώνοντας τα δεδομένα εισόδου σε ζευγάρια τιμών της μορφής < κλειδί, τιμή > και παράγει επίσης δεδομένα οργανωμένα με τον ίδιο τρόπο. Για κάθε ζευγάρι τιμών εισόδου υπάρχει και ένα ζευγάρι τιμών (συνήθως διαφορετικού τύπου) το οποίο παράγεται ως αποτέλεσμα. Οι τιμές key και value πρέπει να είναι Serializable και γι' αυτό υλοποιούν τη διεπαφή Writable και επιπλέον οι κλάσεις της key υλοποιούν τη διεπαφή WritableComparable για να μπορεί να γίνει ταξινόμηση στις τιμές της.

Παρακάτω δίνεται το παράδειγμα του κώδικα WordCount.java το οποίο κάνει χρήση των συναρτήσεων MapReduce:

```
package org.myorg;

import java.io.IOException;
import java.util.*;

import org.apache.hadoop.fs.Path;
import org.apache.hadoop.conf.*;
import org.apache.hadoop.io.*;
import org.apache.hadoop.mapred.*;
import org.apache.hadoop.util.*;

public class WordCount {

public static class Map extends MapReduceBase
implements Mapper<LongWritable, Text, Text, IntWritable> {
            private final static IntWritable one = new IntWritable(1);
            private Text word = new Text();

      public void map(LongWritable key, Text value, OutputCollector<Text,
            IntWritable> output, Reporter reporter) throws IOException {
              String line = value.toString();
              StringTokenizer tokenizer = new StringTokenizer(line);
                  while (tokenizer.hasMoreTokens()) {
                        word.set(tokenizer.nextToken());
                        output.collect(word, one);
                  }
       }
 }


public static class Reduce extends MapReduceBase  implements Reducer<Text,
IntWritable, Text, IntWritable> {

            public void reduce(Text key, Iterator<IntWritable> values,
            OutputCollector<Text,IntWritable> output, Reporter reporter)
            throws IOException {
```

```
                    int sum = 0;

                    while (values.hasNext()) {
                          sum += values.next().get();
                    }
                    output.collect(key, new IntWritable(sum));
              }
}




public static void main(String[] args) throws Exception {
JobConf conf = new JobConf(WordCount.class);
conf.setJobName("wordcount");

conf.setOutputKeyClass(Text.class);
conf.setOutputValueClass(IntWritable.class);

conf.setMapperClass(Map.class);
conf.setCombinerClass(Reduce.class);
conf.setReducerClass(Reduce.class);

conf.setInputFormat(TextInputFormat.class);
conf.setOutputFormat(TextOutputFormat.class);

FileInputFormat.setInputPaths(conf, new Path(args[0]));
FileOutputFormat.setOutputPath(conf, new Path(args[1]));

JobClient.runJob(conf);
   }
}
```

Κώδικας 4.1.4 : Map - Reduce

Υποκεφάλαιο 4.2

ΕΙΔΙΚΟΤΕΡΑ ΓΙΑ ΤΑ ΜΟΝΤΕΛΑ- OpenCL

Στο σημείο αυτό θα αναπτυχθούν ειδικότερα το μοντέλο OpenCL το οποίο αποτελεί μια εναλλακτική λύση απέναντι στο μοντέλο CUDA και περιλαμβάνει κάποιες ομοιότητες σε σχέση με αυτό. Το μοντέλο OpenCL περιλαμβάνει προγραμματισμό σε χαμηλό επίπεδο και για αυτόν το λόγο στο τέλος της ενότητας παρατίθεται και ένα απόσπασμα κώδικα.

Το OpenCL παρουσιάστηκε το 2008 και είναι το πρώτο μοντέλο προγραμματισμού ανοικτού προτύπου για την ανάπτυξη εφαρμογών, που εκτελούνται σε ετερογενή παράλληλα συστήματα, τα οποία περιλαμβάνουν πολυπύρηνους επεξεργαστές, κάρτες γραφικών και άλλα επεξεργαστικά συστήματα. Βασικό χαρακτηριστικό του OpenCL είναι ότι παρέχει λειτουργικότητα ανεξάρτητα από το υλικό στο οποίο θα εκτελεστεί. Αυτή η προσέγγιση ήταν αναμενόμενη, καθώς το πρότυπο OpenCL αναπτύχθηκε με τη συνεργασία πολλών εταιρειών του τομέα της πληροφορικής. Το πρότυπο αυτό σχεδιάστηκε για να προσφέρει εύκολα μεταφέρσιμες εφαρμογές και αυτό ακριβώς το πετυχαίνει υιοθετώντας αφηρημένα μοντέλα εκτέλεσης και μνήμης. Το γεγονός, ότι είναι ένα πρότυπο ανεξάρτητο από το υλικό πάνω στο οποίο τρέχει, επιτρέπει την συγγραφή προγραμμάτων που μπορούν να εκτελεστούν σε ένα μεγάλο μέρος συστημάτων που υποστηρίζουν διαφορετικά APIs.

Για τη συγγραφή των κομματιών του παράλληλου κώδικα στο πρότυπο αυτό χρησιμοποιείται η γλώσσα προγραμματισμού C99 και άλλα πρόσθετα APIs τα οποία αφού πρώτα προσδιορίσουν την πλατφόρμα πάνω στην οποία θα εκτελεστεί ο κώδικας, στη συνέχεια τον εκτελούν χρησιμοποιώντας τις κατάλληλες συναρτήσεις. Όσον αφορά τον παραλληλισμό, το OpenCL παρέχει παραλληλοποίηση, τόσο σε επίπεδο διεργασιών (task-based parallelism) όσο και σε επίπεδο δεδομένων (data-based parallelism)

Ομοιότητες και Διαφορές μεταξύ CUDA και OpenCL

Ομοιότητες:

- Το OpenCL αντιμετωπίσει το υλικό ιεραρχικά θεωρώντας ότι υπάρχει μια κύρια διεργασία, η οποία εκτελείται στον host (host process) και οι υπόλοιπες δευτερεύουσες διεργασίες που εκτελούνται στις συσκευές (device processes).
- Εκτέλεση που βασίζεται στην παραλληλία δεδομένων.
- Σύνθετη ιεραρχία μνήμης.

Διαφορές :

- Το OpenCL είναι ένα διασυστημικό πρότυπο οπότε μπορεί να υποστηρίζει ετερογενή συστήματα (όπως π.χ. GPU, CPU) αντιμετωπίζοντάς τα όλα αυτά σαν μια ενιαία πλατφόρμα.

- Λόγω του παραπάνω γεγονότος, χρησιμοποιεί ένα πολύπλοκο μοντέλο για την διαχείριση των συσκευών και τη μεταγλώττιση και την εκκίνηση του πυρήνα.

Πίνακας 4.2.1 : Εννοιολογικές αντιστοιχίες μεταξύ OpenCL και CUDA

Εννοια παραλληλίας στο OpenCL	**Ισοδύναμη έννοια στην CUDA**
kernel	kernel
__global__	__global
host	host
device	device
NDRange	Grid
work group item	thread
work groups	block
threadIdx.x	get_local_id(0)
BlockIdx.x	get_group_id(0)
BlockDim.x	get_work_size(0)

```
  // create a compute context with GPU device
  context = clCreateContextFromType(NULL, CL_DEVICE_TYPE_GPU, NULL, NULL,
NULL);

  // create a command queue
  queue = clCreateCommandQueue(context, NULL, 0, NULL);

  // allocate the buffer memory objects
  memobjs[0] = clCreateBuffer(context, CL_MEM_READ_ONLY |
CL_MEM_COPY_HOST_PTR, sizeof(float)*2*num_entries, srcA, NULL);
  memobjs[1] = clCreateBuffer(context, CL_MEM_READ_WRITE,
sizeof(float)*2*num_entries, NULL, NULL);

  // create the compute program
  program = clCreateProgramWithSource(context, 1, &fft1D_1024_kernel_src,
NULL, NULL);

 // build the compute program executable
  clBuildProgram(program, 0, NULL, NULL, NULL, NULL);

  // create the compute kernel
  kernel = clCreateKernel(program, "fft1D_1024", NULL);
```

```
  // set the args values
  clSetKernelArg(kernel, 0, sizeof(cl_mem), (void *)&memobjs[0]);
  clSetKernelArg(kernel, 1, sizeof(cl_mem), (void *)&memobjs[1]);
  clSetKernelArg(kernel, 2, sizeof(float)*(local_work_size[0]+1)*16,
NULL);
  clSetKernelArg(kernel, 3, sizeof(float)*(local_work_size[0]+1)*16,
NULL);

  // create N-D range object with work-item dimensions and execute kernel
  global_work_size[0] = num_entries;
  local_work_size[0] = 64;
  clEnqueueNDRangeKernel(queue, kernel, 1, NULL, global_work_size,
local_work_size, 0, NULL, NULL);
```

Κώδικας 4.2.2 : OpenCL

Υποκεφάλαιο 4.3

ΕΙΔΙΚΟΤΕΡΑ ΓΙΑ ΤΑ ΜΟΝΤΕΛΑ - DirectCompute

Το μοντέλο προγραμματισμού DirectCompute αποτελεί το μοντέλο το οποίο παρουσίασε η εταιρεία Microsoft για παράλληλο προγραμματισμό γενικού σκοπού σε GPU για συστήματα που τρέχουν MS Windows Vista ή MS Windows 7. Η συγκεκριμένη πλατφόρμα είναι κομμάτι του πακέτου API της DirectX (το οποίο δημιουργήθηκε επίσης από τη Microsoft) και παρέχει ένα περιβάλλον για συγγραφή και compile παράλληλου κώδικα χαμηλού επιπέδου.

Το DirectCompute θα υποστηρίζεται από όλα τα GPU chip που έχουν εγκατεστημένη την DirectX11 ενώ υποστηρίζεται ήδη από μερικά chip που τρέχουν με γραφικά DirectX10. Ως περιβάλλον, προσανατολίζεται κυρίως στον προγραμματισμό διαδραστικών εφαρμογών πραγματικού χρόνου (interactive, real-time applications) και γι' αυτόν το λόγο είναι άμεσα συνδεδεμένο με τα APIs του πακέτου Direct3D. Γενικά, όπως τουλάχιστον διαφαίνεται μέχρι στιγμής, το συγκεκριμένο API θα μπορεί να χρησιμοποιηθεί σε προβλήματα στα

οποία μπορεί να γίνει χρήση παράλληλου κώδικα, όπως οι προσομοιώσεις, η κρυπτογράφηση, η επεξεργασία video και εικόνας. Χρησιμοποιείται επίσης στον τομέα των PC games σε εφαρμογές rendering, physics κ.ά.

Η γλώσσα προγραμματισμού που χρησιμοποιείται για τη συγγραφή κώδικα στο DirectCompute είναι η HLSL, μια γλώσσα η οποία χρησιμοποιείται ήδη για τη δημιουργία γραφικών σε πολλές gaming εφαρμογές. Η σύνταξη της συγκεκριμένης γλώσσας είναι παρόμοια με αυτήν της C ως προς τα βασικά της στοιχεία τα οποία είναι:

1. Οι εντολές που αποτελούν οδηγίες προς τον compiler (παράδειγμα οι εντολές #define, #ifdef, κ.ά.).
2. Οι βασικοί τύποι δεδομένων (bool, int, float).
3. Η χρήση τελεστών, μεταβλητών και συναρτήσεων.

Πέρα από τις βασικές ομοιότητες, η HLSL έχει και βασικές διαφορές σε σχέση με την C, όπως παράδειγμα:

1. Δεν είναι δυνατή η χρήση δεικτών
2. Υπάρχουν ενσωματωμένες μεταβλητές και τύποι δεδομένων (matrix, float4)
3. Υπάρχουν intrinsic συναρτήσεις (mul, normalize)

Ο κώδικας, που είναι γραμμένος σε hlsl, περνάει μέσα από έναν compiler (συνήθως είναι κάποια βιβλιοθήκη του πακέτου D3D ή και κάποιο άλλο API) και μετατρέπεται σε ένα κομμάτι κώδικα το οποίο δεν είναι απευθείας έτοιμο για εκτέλεση και το οποίο με τη σειρά του συνδέεται με τον ανάλογο driver προκειμένου να παράγει το τελικό σύνολο εκτελέσιμων εντολών, που είναι αντίστοιχο με το hardware που θα χρησιμοποιηθεί για την υλοποίηση.

Η διαδικασία με την οποία γίνεται η συγγραφή και η χρήση παράλληλου κώδικα στην πλατφόρμα του DirectCompute είναι σχετικά απλή και αποτελείται από τα παρακάτω στάδια τα οποία θα αναφερθούν επιγραμματικά:

Βήμα 1. Αρχικοποιούμε το interface του DirectCompute.

Βήμα 2. Συντάσσουμε τον κώδικα με τη hlsl

Βήμα 3. Κάνουμε compile τον κώδικα που συντάξαμε με έναν DirectX compiler.

Βήμα 4. Φορτώνουμε τον compiled κώδικα στην GPU.

Βήμα 5. Δημιουργούμε έναν buffer μέσα στον οποίο θα περάσουμε τα δεδομένα μας και δημιουργούμε επίσης και μια όψη (view) για να έχουμε πρόσβαση σε αυτά.

Βήμα 6. Εκτελούμε τον κώδικα στην GPU.

Βήμα 7. Αντιγράφουμε τα αποτελέσματα της εκτέλεσης πίσω στην CPU.

```
 // Get a CUDA capable adapter

   std::vector<IDXGIAdapter1*> vAdapters;

   IDXGIFactory1* factory;

   CreateDXGIFactory1(__uuidof(IDXGIFactory1), (void**)&factory);

   IDXGIAdapter1 * pAdapter = 0;

UINT i=0;

   while(factory->EnumAdapters1(i, &pAdapter) != DXGI_ERROR_NOT_FOUND)

   {

       vAdapters.push_back(pAdapter);

       ++i;

   }

g_driverType = D3D_DRIVER_TYPE_UNKNOWN;

   hr = D3D11CreateDevice( vAdapters[devNum], g_driverType, NULL,
```

```
createDeviceFlags, levelsWanted,

               numLevelsWanted, D3D11_SDK_VERSION, &g_pD3DDevice,
&g_D3DFeatureLevel, &g_pD3DContext );

// now dispatch ("run") the compute shader, with a set of 16x16 groups.

       g_pD3DContext->Dispatch( 16, 16, 1 );

pd3dImmediateContext->CSSetShader( ... );

    pd3dImmediateContext->CSSetConstantBuffers( ...);

    pd3dImmediateContext->CSSetShaderResources( ...);  // CS input

    // CS output

    pd3dImmediateContext->CSSetUnorderedAccessViews( ...);

 pd3dImmediateContext->Dispatch( dimx, dimy, 1 );
```

Κώδικας 4. 3 : DirectCompute

Κεφάλαιο 5

Βασικά στοιχεία του μοντέλου CUDA

Υποκεφάλαιο 5.1

ΑΡΧΙΤΕΚΤΟΝΙΚΗ ΜΙΑΣ ΚΑΡΤΑΣ ΓΡΑΦΙΚΩΝ ΠΟΥ ΥΠΟΣΤΗΡΙΖΕΙ CUDA

Πριν ξεκινήσουμε να αναλύουμε το μοντέλο της CUDA, θε πρέπει να γίνει μια σύντομη αναφορά στον τρόπο με τον οποίο οργανώνεται μια GPU που υποστηρίζει το μοντέλο αυτό. Μια GPU CUDA με υπολογιστική δυνατότητα 2.x οργανώνεται με βάση τα παρακάτω χαρακτηριστικά :

- **Επεξεργαστικά στοιχεία .**
  - **SM**
  - **SP**

Μια συστοιχία από πολυεπεξεργαστές συνεχούς ροής γνωστοί ως **SM** (SM - Streaming Multiprocessors) οι οποίοι διαχειρίζονται έναν τεράστιο αριθμό νημάτων. Βασικό δομικό στοιχείο κάθε SM είναι οι **SP** (Streaming Processors) ή αλλιώς επεξεργαστές συνεχούς ροής που μοιράζονται τη μονάδα ελέγχου (control unit) και την κρυφή μνήμη των εντολών. Σε αντιστοιχία με το μοντέλο της πολυπύρηνης αρχιτεκτονικής οι SP αποτελούν τους επεξεργαστές και οι οποίοι διαχειρίζονται έναν τεράστιο αριθμό νημάτων. Ο αριθμός των SM διαφέρει από τη μία γενιά GPU CUDA στην άλλη.

Κάθε SP διαθέτει μία μονάδα πολλαπλασιασμού και μία μονάδα πρόσθεσης. Επίσης περιέχει και εξειδικευμένες μονάδες που εκτελούν floating point λειτουργίες όπως πχ. υπολογισμούς τετραγωνικών ριζών κ.ά. Μέσα στα SP τα νήματα κάθε εφαρμογής πρέπει να οργανώνονται μαζικά και να εκτελούνται με αντίστοιχο τρόπο, ώστε να επιτυγχάνεται όσο το δυνατόν μεγαλύτερος βαθμός παραλληλίας.

- **Στοιχεία μνήμης**

  - **Καθολική μνήμη (ανάγνωση και εγγραφή)**
  - **Μνήμη Σταθερών**
  - **Μνήμη υφών (μόνο ανάγνωση)**
  - **Κοινόχρηστη μνήμη (48 KB/ MP)**
  - **Τοπική μνήμη**
  - **Καταχωρητές (8192- 16384 32-bit / MP)**

Μια GPU διαθέτει μέχρι 4 GB DRAM μνήμης τύπου GDDR (διπλού αριθμού δεδομένων γραφικών – Graphics Double Data Rate) η οποία αποτελεί την **καθολική μνήμη** της κάρτας και είναι ανεξάρτητη από την κεντρική μνήμη του υπολογιστή κεντρικής επεξεργασίας. Η διαφορά αυτού του τύπου μνήμης σε σχέση με την κεντρική μνήμη που βρίσκεται στην μητρική κάρτα του host, εντοπίζεται στο ότι η μνήμη στην κάρτα γραφικών χρησιμοποιείται ως buffer για frames γραφικών -όσον αφορά τις εφαρμογές γραφικών- ενώ λειτουργεί σαν off-chip μνήμη (με μεγάλο εύρος ζώνης και αρκετά μεγάλο λανθάνοντα χρόνο) όταν πρόκειται για άλλες υπολογιστικές εργασίες. Ο μεγάλος λανθάνων χρόνος της καθολικής μνήμης αντισταθμίζεται μερικώς με την προσθήκη μνήμης cache στις κάρτες με υπολογιστική δυνατότητα 2.x .

Η μνήμη σταθερών αποτελεί και αυτή μια αργή μνήμη στην οποία προστέθηκε cache και χρησιμοποιείται από τα νήματα μόνο για ανάγνωση. Παρέχει την δυνατότητα χρήσης της εντολής LoaD Uniform (LDU).

Η μνήμη υφών χρησιμοποιείται επίσης για ανάγνωση από τα νήματα και εξυπηρετεί ειδικά πρότυπα προσπέλασης δεδομένων.

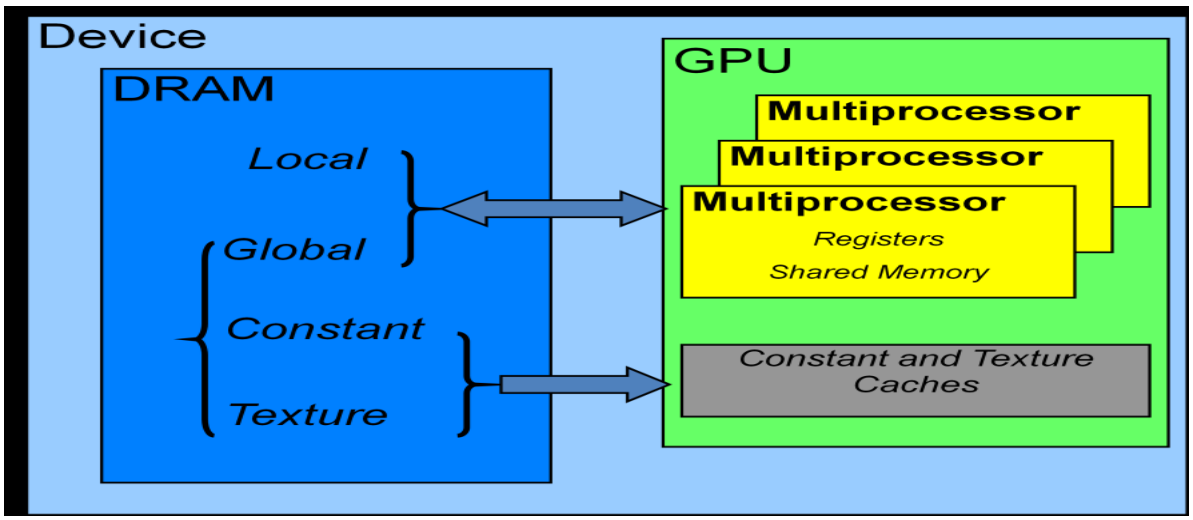
Η κοινόχρηστη μνήμη είναι ένα γρήγορο στοιχείο μνήμης, επειδή όμως είναι χωρισμένη σε υποστοιχεία μνήμης (banks) χρειάζεται προσοχή ώστε να μην υπάρχουν συγκρούσεις μεταξύ αυτών. Χρησιμοποιείται από τα νήματα εντός του ίδιου μπλοκ για ανταλλαγή δεδομένων.

Η τοπική μνήμη είναι αργή μνήμη που δεν διαθέτει cache, χρησιμοποιεί αυτόματο συγκερασμό (coalesce) προσπελάσεων για ανάγνωση και εγγραφή δεδομένων. Η κύρια χρήση της αφορά την αποθήκευση δεδομένων, που δεν μπορούν να αποθηκευθούν στους καταχωρητές (λόγω έλλειψης χώρου).

Οι καταχωρητές αποτελούν τα πιο γρήγορα στοιχεία μνήμης και χρησιμοποιούνται από τα νήματα για ανάγνωση και εγγραφή δεδομένων (κάθε νήμα διαθέτει τον δικό του καταχωρητή).

- **Εύρος ζώνης επικοινωνίας και εύρος ζώνης μνήμης.** Το **εύρος ζώνης επικοινωνίας** μιας κάρτας γραφικών αφορά το διαθέσιμο εύρος για ανταλλαγή δεδομένων μεταξύ της κάρτας και του κεντρικού συστήματος επεξεργασίας. Το **εύρος ζώνης μνήμης** αποτελεί το διαθέσιμο εύρος για ανταλλαγή δεδομένων μεταξύ της καθολικής μνήμης και των υπόλοιπων στοιχείων μέσα στα όρια της κάρτας γραφικών. Το εύρος ζώνης επικοινωνίας είναι πολύ μικρότερο από το εύρος ζώνης μνήμης. Για παράδειγμα στην κάρτα G80 που εισήγαγε την αρχιτεκτονική CUDA το εύρος ζώνης επικοινωνίας με την CPU ήταν 8 GB/s ενώ το εύρος ζώνης μνήμης 86,4 GB/s αλλά αναμένεται να αυξηθεί στο μέλλον.

- 00**11Ιεραρχία μνήμης.** Η αρχιτεκτονική CUDA ορίζει οτι όλα τα νήματα καλούνται από τον κώδικα που εκτελείται στο κεντρικό σύστημα, αλλά εκτελούνται σε μια ξεχωριστή κάρτα γραφικών, που διατηρεί τα δικά της ανεξάρτητα κυκλώματα μνήμης. Τα πιο γρήγορα στοιχεία μνήμης αποτελούν οι καταχωρητές και η κοινόχρηστη μνήμη που βρίσκονται τοποθετημένα στα κυκλώματα των SM και διαθέτουν πολύ μικρή χωρητικότητα, της τάξης των μερικών KB. Τη μεγαλύτερη χωρητικότητα (της τάξης των GB) αλλά ταυτόχρονα και τους μεγαλύτερους χρόνους προσπέλασης διαθέτει η καθολική μνήμη, η οποία βρίσκεται πάνω στο κύκλωμα της κάρτας γραφικών, αλλά εκτός ορίων των SM.



Σχήμα 5.1 : Οργάνωση της μνήμης σε κάρτα γραφικών

Υποκεφάλαιο 5.2

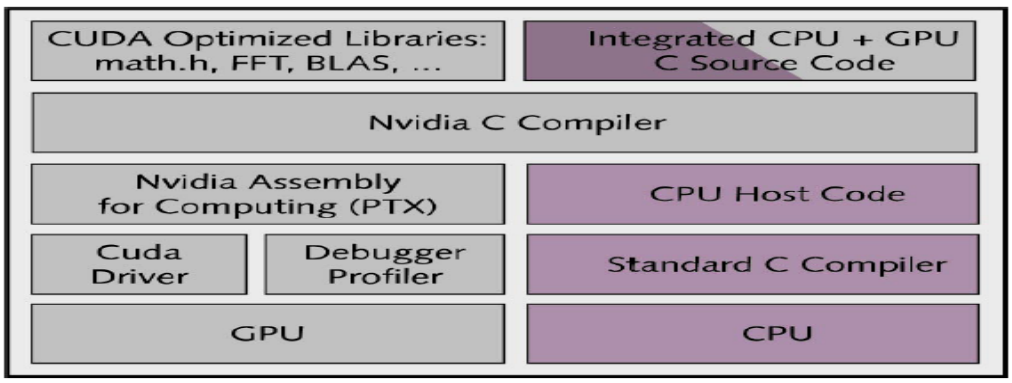
ΒΑΣΙΚΑ ΣΤΟΙΧΕΙΑ ΜΟΝΤΕΛΟΥ CUDA

Το Νοέμβριο του 2006 η Nvidia παρουσίασε το καινοτόμο μοντέλο προγραμματισμού CUDA το οποίο παρέχει:

1. APIs για συγγραφή και εκτέλεση υπολογιστικών προγραμμάτων.
2. Υποστήριξη γλωσσών προγραμματισμού υψηλού επιπέδου (CUDA C).
3. Μεταφραστή για την μετάφραση των προγραμμάτων (nvcc compiler).
4. Ο στόχος της αρχιτεκτονικής αυτής ήταν να παρέχει εργαλεία προγραμματισμού εφαρμογών γενικού σκοπού που θα μπορούν να εκτελούνται στα κυκλώματα των καρτών γραφικών. Με την έννοια γενικού σκοπού προγράμματα, εννοούμε προγράμματα διαφορετικής φύσης από τις εφαρμογές επεξεργασίας γραφικών.

Εκτός από τα παραπάνω, παρέχεται επίσης και η δυνατότητα συμπληρωματικής-βοηθητικής χρήσης πρόσθετων βιβλιοθηκών όπως:

- CUFFT, που αποτελεί βιβλιοθήκη για ταχύτατους μετασχηματισμούς Fourrier.
- CUBLAS, βιβλιοθήκη υποστήριξης συναρτήσεων γραμμικής άλγεβρας.
- NPP, βιβλιοθήκη συναρτήσεων για επεξεργασία δεδομένων με CUDA.
- Nsight ή άλλα προγραμματιστικά εργαλεία για αποσφαλμάτωση και βελτιστοποίηση κώδικα.



Σχήμα 5.2.1: Προγραμματιστικά εργαλεία σε ένα σύστημα CUDA

Η αρχιτεκτονική CUDA είναι **ετερογενούς** φύσης. Αυτό σημαίνει ότι ο κώδικας εκτελείται από δύο συστήματα ταυτόχρονα, από το host σύστημα, που αποτελείται από έναν ή περισσότερους επεξεργαστές και μία ή περισσότερες κάρτες γραφικών με πολλές GPU.

Κάθε αλγόριθμος, ο οποίος προορίζεται για εκτέλεση σε ένα ετερογενές σύστημα δεν είναι δυνατό να εκτελείται εξ ολοκλήρου σε ένα σύστημα μόνο. Πρέπει να οργανώνεται σε τμήματα, από τα οποία κάποια θα εκτελούνται στο host σύστημα και κάποια στην κάρτα γραφικών. Αυτή η διαδικασία χρειάζεται, διότι κάθε σύστημα αποτελείται από διαφορετικά οργανωμένο υλικό και λειτουργεί αποδοτικά κάτω από διαφορετικού είδους κώδικα.

Μια κάρτα γραφικών λειτουργεί αποδοτικά κάτω από συγκεκριμένες συνθήκες, όπως :

- Υπολογισμοί που επενεργούν την ίδια χρονική στιγμή, παράλληλα, πάνω σε μεγάλο πλήθος από δεδομένα. Κλασικό παράδειγμα τέτοιας περίπτωσης είναι οι αιρθμητικοί υπολογισμοί σε δεδομένα πινάκων.

- Κάποιες περιπτώσεις προσπέλασης μνήμης διευκολύνουν το υλικό της κάρτας γραφικών να συνενώνει τις προσπελάσεις στη μνήμη και να εκτελεί λειτουργίες για ανάγνωση-γράψιμο πάνω σε πολλά δεδομένα ταυτόχρονα, με μία μόνο εντολή / λειτουργία.

- Υπολογισμοί οι οποίοι απαιτούν όσο το δυνατό λιγότερη αντιγραφή δεδομένων από την κάρτα προς τον host και αντίστροφα, καθώς αυτή η διαδικασία κοστίζει σε χρόνο, αλλά και απόδοση.

◦ Αν πολλαπλοί πυρήνες εκτελούν λειτουργίες στα ίδια δεδομένα, τότε τα αποτελέσματα των υπολογισμών του κάθε πυρήνα θα πρέπει να αποθηκεύονται προσωρινά στη μνήμη της συσκευής μέχρι να τα χρησιμοποιήσει ο επόμενος πυρήνας που θα κληθεί. Η αντιγραφή των αποτελεσμάτων και μετά η επαναπροσκόμισή τους στην κάρτα αποτελεί χαμένο χρόνο.

◦ Σε περίπτωση που οι υπολογισμοί είναι υπερβολικά περίπλοκοι και η εκτέλεσή τους σε μια κάρτα CUDA κρίνεται απαραίτητη για τη βελτίωση της απόδοσης, η σπατάλη χρόνου για τη μεταφορά των δεδομένων από το ένα σύστημα στο άλλο μπορεί να ισοσταθμιστεί από την ταχύτητα εκτέλεσης των υπολογισμών.

Η συγγραφή προγραμμάτων με το μοντέλο CUDA περιλαμβάνει αρκετές δυνατότητες ως προς τη γλώσσα σύνταξης. Καταρχήν, ως γλώσσα σύνταξης μπορεί να χρησιμοποιηθεί κάποια γλώσσα υψηλού επιπέδου όπως πχ. C, C++, Python οι οποίες είναι ευρέως διαδεδομένες γλώσσες προγραμματισμού. Υπάρχει και η υποστήριξη για γλώσσες προγραμματισμού όπως η java, αλλά το τμήμα αυτό βρίσκεται ακόμη υπό δοκιμή και θα μπορούσε κάποιος ενδεχομένως να αντιμετωπίσει ορισμένα δύσκολα σημεία. Εκτός από

γλώσσες υψηλού επιπέδου (πχ. CUDA C) μπορεί να χρησιμοποιηθεί και μια άλλη προγραμματιστική διεπαφή, το Driver API, το οποίο αντιστοιχεί σε χαμηλότερου επιπέδου προγραμματισμό. Ο κώδικας γραμμένος με τις βιβλιοθήκες του Driver API είναι γενικά πιο περίπλοκος στη σύνταξη και στην κατανόηση, προσφέρει όμως περισσότερο έλεγχο στις παραμέτρους των προγραμματιζόμενων συναρτήσεων της κάρτας γραφικών.

Το μοντέλο της CUDA βασίζεται στην ενοποιημένη γραμμή εντολών που χρησιμοποιείται από τις κάρτες για την επεξεργασία και την απεικόνιση γραφικών.

CUDA APIs

Το μοντέλο CUDA προσφέρει δύο προγραμματιστικές διεπαφές:

- CUDA Driver API που είναι προγραμματιστική διεπαφή χαμηλού επιπέδου που υλοποιείται μέσω της βιβλιοθήκης nvcuda (πρόθεμα των στοιχείων είναι το cu) και αποτελεί τη βάση και για άλλες υλοποιήσεις υψηλότερου επιπέδου.
- C Runtime API η οποία είναι διεπαφή υψηλού επιπέδου, υλοποιείται χρησιμοποιώντας την δυναμική βιβλιοθήκη cudart και το πρόθεμα cuda ενώ έχει ως βάση της υλοποίησης παραμένει το CUDA Driver API.

Συνολικά οι βιβλιοθήκες και οι διεπαφές του μοντέλου CUDA περιλαμβάνουν συναρτήσεις για τις εξής λειτουργίες :

- Διαχείριση της κάρτας γραφικών.
- Διαχείριση του context.
- Διαχείριση μνήμης.
- Έλεγχος εκτέλεσης εντολών.
- Διαχείριση αναφορών υφής (texture references).
- Διαχείριση των code modules.
- Διαλειτουργικότητα με τις βιβλιοθήκες της OpenGL και της Direct3D.

Διαφορές CUDA Driver API – C Runtime CUDA

Οι δύο προγραμματιστικές διεπαφές της CUDA είναι αμοιβαίως αποκλειόμενες. Αν και μπορούν δύο αλγόριθμοι που βασίζονται σε διαφορετικές διεπαφές να συνεργαστούν, εντούτοις αυτό γίνεται σε μικρό βαθμό και με συγκεκριμένες μόνο συναρτήσεις.

- **C Runtime** περιβάλλον της CUDA είναι αυτό που χρησιμοποιείται στο μεγαλύτερο βαθμό και παρέχει μεγαλύτερη ευκολία στον προγραμματισμό της κάρτας μέσω implicit initialization, context and module management.

- **CUDA Driver API** είναι μια διεπαφή, που απαιτεί τη συγγραφή περισσότερου και πιο περίπλοκου κώδικα. Η αποσφαλμάτωση είναι δυσκολότερη σε αυτήν την περίπτωση, αλλά προσφέρει καλύτερο έλεγχο των λειτουργιών της κάρτας, κάτι που αποτελεί σημαντικό πλεονέκτημα. Συγκεκριμένα, ο καθορισμός των παραμέτρων που δίνονται στους πυρήνες και η εκκίνησή τους είναι δυσκολότερη σε αυτήν την περίπτωση, διότι οι αρχικοποίηση γίνεται απευθείας μέσω συναρτήσεων και όχι έμμεσα μέσω ειδικής σύνταξης.

Πολύ σημαντικές έννοιες οι οποίες πρέπει να αναφερθούν είναι η διεργασία και το νήμα.

- Η διεργασία είναι ένα στιγμιότυπο προγράμματος, που εκτελείται μαζί με τα απαραίτητα συστατικά που χρειάζεται για την εκτέλεση, όπως ο μετρητής προγράμματος, οι καταχωρητές, το τμήμα της μνήμης που περιέχει τον κώδικα, απαραίτητοι πόροι του λειτουργικού συστήματος κτλ.

- Το νήμα είναι μια διαδοχική σειρά εκτέλεσης εντολών μέσα σε μια διεργασία. Για παράδειγμα μέσα σε μια if-else δομή, ένα νήμα θα μπορούσε να εκτελέσει την ακολουθία εντολών που προκύπτουν από το if και ένα άλλο νήμα την ακολουθία που προκύπτει από το else.

Όσον αφορά το μοντέλο της CUDA, θα πρέπει να αναφέρουμε τις βασικές έννοιες στις οποίες στηρίζεται:

1. Τους **πυρήνες (kernels)** που σηματοδοτούν τα κομμάτια κώδικα που θα εκτελεστούν στην κάρτα γραφικών.

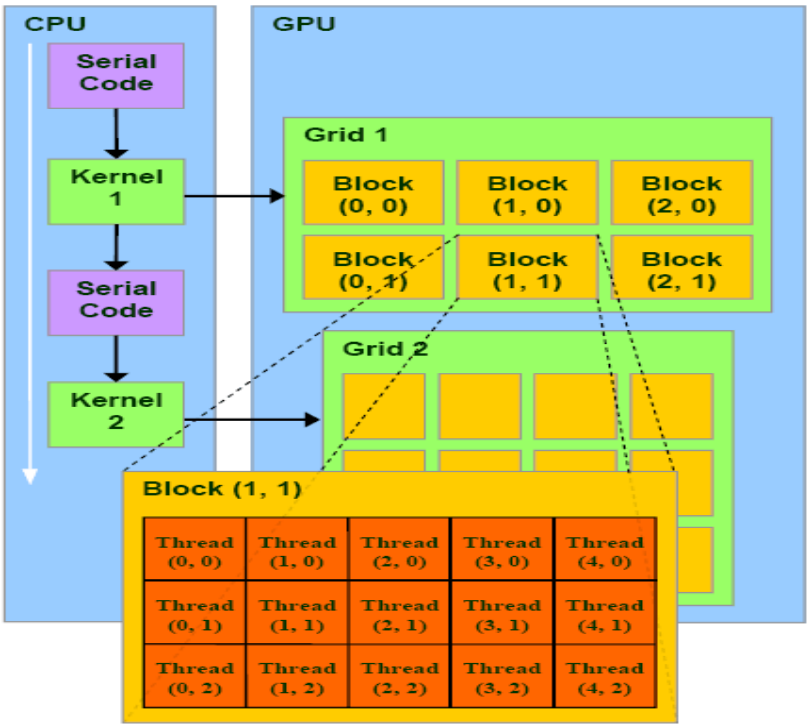
2. Τα **νήματα (threads) εκτέλεσης** στα οποία διασπάται ο κώδικας προκειμένου να μπορεί να εκτελεστεί παράλληλα από τις μονάδες επεξεργασίας της κάρτας.

3. Τα **μπλοκ (block)** που ομαδοποιούν τα νήματα της εκτέλεσης.

4. Τα **πλέγματα (grids)** που αποτελούν τις ομάδες στις οποίες οργανώνονται τα μπλοκ.

Χρήση των εννοιών στο μοντέλο της CUDA

Ο κώδικας που συντάσσεται με τα προγραμματιστικά εργαλεία του CUDA είναι ενιαίος και περιέχει εντολές, που ανάλογα με το πρόθεμά τους, εκτελούνται είτε στην κάρτα γραφικών, είτε στους πυρήνες του κεντρικού επεξεργαστή.



Σχήμα 5.2.2: Οργάνωση νημάτων σε κάρτα γραφικών

Γίνεται επομένως, διαχωρισμός του ενιαίου κώδικα, ο οποίος χωρίζεται στον κώδικα προς εκτέλεση στον host και τον κώδικα προς εκτέλεση στη device του συστήματος.

Για να είναι εφικτή η παράλληλη εκτέλεση των διεργασιών σε μια συσκευή, με υποστήριξη CUDA, θα πρέπει αυτές να οργανώνονται με ένα συγκεκριμένο τρόπο. Βάσει του μοντέλου CUDA εισάγονται ορισμένες νέες έννοιες που αφορούν την ομαδοποίηση των διεργασιών και την κατάλληλη διαχείρισή τους.

1. Οι **πυρήνες (kernel)** είναι συναρτήσεις γραμμένες στη γλώσσα C και καθορίζουν τα τμήματα κώδικα, που θα εκτελεστούν στη device και δημιουργούν **νήματα** τα οποία εκτελούνται παράλληλα όσες φορές καθορίσουμε εμείς μέσα από τις παραμέτρους που τους δίνουμε.

2. Η δυνατότητα παράλληλης εκτέλεσης των νημάτων δίνεται μέσα από την οργάνωσή τους σε ομάδες δηλαδή τα **μπλοκ (block)** εννοώντας ότι τα νήματα τα οποία βρίσκονται μέσα στα μπλοκ εκτελούνται παράλληλα. Σε κάθε περίπτωση, το κάθε μπλοκ μπορεί να φιλοξενήσει μέχρι ένα συγκεκριμένο αριθμό νημάτων, ο οποίος καθορίζεται πάντα από τις δυνατότητες της εκάστοτε κάρτας εκτέλεσης. Συγκεκριμένα, πρόκειται για την ιδιότητα maxThreadsPerBlock, που ορίζει το μέγιστο αριθμό νημάτων που μπορεί να περιέχει ένα μπλοκ και η οποία στα τελευταία μοντέλα είναι ίση με 512 νήματα. Τα μπλοκ επίσης, οργανώνονται σε συστοιχίες για να αποτελέσουν ένα ή περισσότερα πλέγματα, ο αριθμός των οποίων εξαρτάται από το πλήθος των δεδομένων μας και τον αριθμών των διαθέσιμων επεξεργαστών στην κάρτα γραφικών.

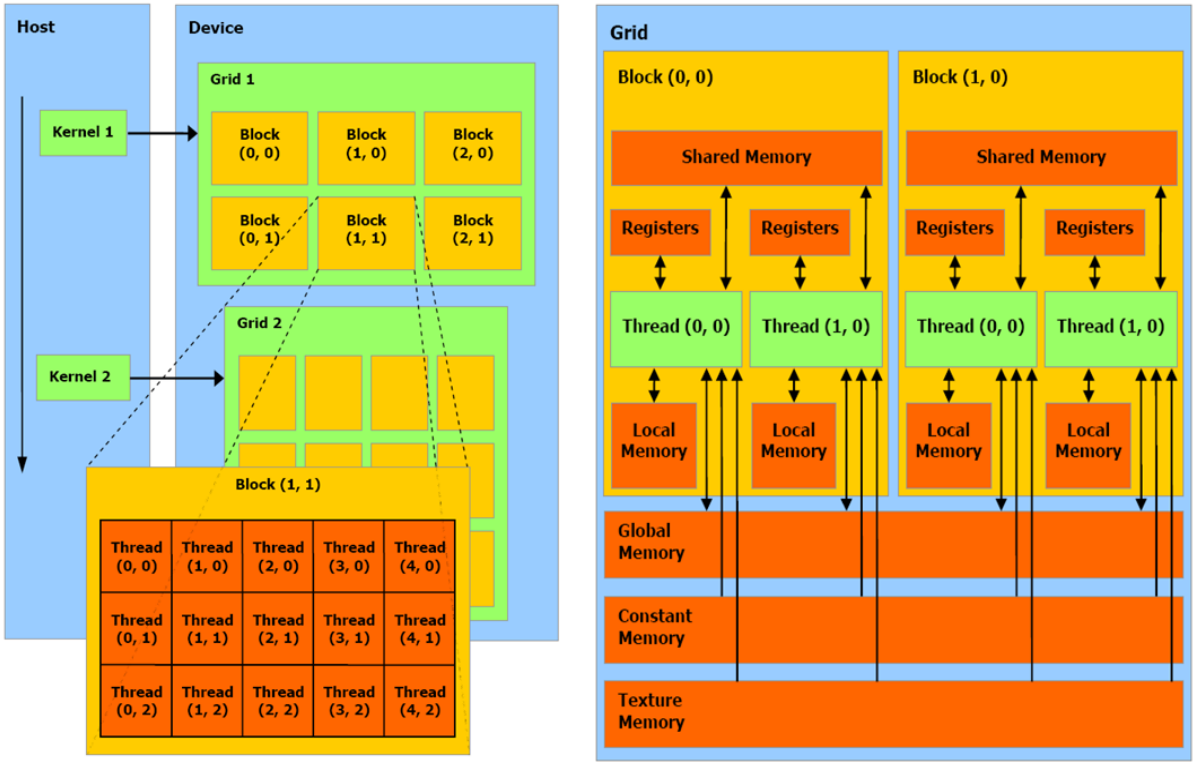
3. Τα μπλοκ επίσης, οργανώνονται σε συστοιχίες για να αποτελέσουν ένα ή περισσότερα **πλέγματα (grid)**, ο αριθμός των οποίων εξαρτάται από το πλήθος των δεδομένων μας και τον αριθμών των διαθέσιμων επεξεργαστών στην κάρτα γραφικών.

Για τον προσδιορισμό όλων των εννοιών που χρησιμοποιούνται για προγραμματισμό σε CUDA, δηλαδή των νημάτων, των μπλοκ και των πλεγμάτων μέσα σε ένα περιβάλλον εκτέλεσης, χρησιμοποιούνται ορισμένες **ενσωματωμένες μεταβλητές**. Με τη βοήθεια αυτών των μεταβλητών ο προγραμματιστής μπορεί να καθορίσει επακριβώς τις διαστάσεις ενός μπλοκ ή ενός πλέγματος και να έχει πρόσβαση στους δείκτες που προσδιορίζουν μοναδικά κάθε μπλοκ ή κάθε νήμα μέσα σε ένα πλέγμα.

Αυτές οι μεταβλητές είναι:

- **gridDim** (τύπου dim3 που περιέχει τις διαστάσεις ενός πλέγματος)
- **blockIdx** (τύπου unint3 που προσδιορίζει κάθε μπλοκ μέσα σε ένα πλέγμα)
- **blockDim** (τύπου dim3 και προσδιορίζει τις διαστάσεις ενός μπλοκ)
- **threadIdx** (τύπου unint3 που προσδιορίζει μονοσήμαντα ένα νήμα μέσα στο πλέγμα)
- **warpSize** (τύπου int που προσδιορίζει το μέγεθος ενός σμήνους σε νήματα)

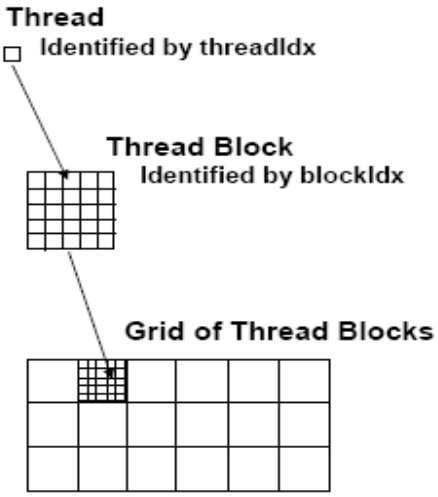
Σημειώνεται, ότι δεν μπορούμε να εξαγάγουμε τη διεύθυνση αποθήκευσης στη μνήμη αυτών των μεταβλητών, αλλά ούτε και να κάνουμε ανάθεση τιμής σε αυτές. Χρησιμοποιώντας όμως αυτές τις μεταβλητές μπορούμε να έχουμε πρόσβαση σε χρήσιμες ιδιότητες της κάρτας γραφικών.



Σχήμα 5.2.3: Ιεραρχία μνήμης ανά μπλοκ και πλέγμα σε κάρτα γραφικών

- **<u>Προσδιορισμός ταυτότητας νήματος</u>**.

0011Ο αριθμός των νημάτων που μπορεί να περιέχει ένα μπλοκ είναι περιορισμένος κυρίως λόγω του ότι όλα τα νήματα στο ίδιο μπλοκ κάνουν χρήση της κοινόχρηστης μνήμης. Η ταυτότητα κάθε νήματος είναι προσδιορίσιμη άμεσα, με τη χρήση του διανύσματος **threadIdx** που περιέχει τις 3 συντεταγμένες ενός νήματος, ανάλογα αν το νήμα αυτό ανήκει σε ένα μονοδιάστατο / δισδιάστατο / τρισδιάστατο μπλοκ. Για παράδειγμα, για να πάρουμε τις συντεταγμένες ενός νήματος δύο διαστάσεων αναφερόμαστε σε αυτό ως εξής:



Σχήμα 5.2.4: Το νήμα και η τοποθέτησή του σε ένα πλέγμα εκτέλεσης

→ int x = threadIdx.x;

→ int y = threadIdx.y;

Φυσικά, στην περίπτωση που θέλουμε να προσδιορίσουμε μονοσήμαντα ένα νήμα, αλλά έχουμε περισσότερα του ενός μπλοκ και έχοντας υπόψη ότι κάθε μπλοκ μπορεί να διαθέτει μέχρι 3 διαστάσεις, θα χρησιμοποιήσουμε τον παρακάτω τύπο:

→int x= threadIdx.x + blockIdx.x * blockDim.x;

- **Προσδιορισμός μπλοκ.**

Γενικά, ο αριθμός των μπλοκ που μπορούμε να χρησιμοποιήσουμε διαφέρει ανάλογα με το πλήθος των δεδομένων που διαχειριζόμαστε, αλλά και από το πλήθος των διαθέσιμων επεξεργαστών. Για να προσδιορίσουμε τη θέση ενός συγκεκριμένου μπλοκ μέσα σε μια ομάδα από μπλοκ (πλέγμα) δύο διαστάσεων, για παράδειγμα, χρησιμοποιούμε τις εξής μεταβλητές :

→ int x = blockIdx.x;
→ int y = blockIdx.y;

Αντιστοίχως, για να προσδιορίσουμε τις διαστάσεις ενός μπλοκ, αναφερόμαστε σε αυτές ως εξής:

→ int x = blockDim.x;
→ int y = blockDim.y;

Ένα μπλοκ νημάτων μπορεί να είναι μονοδιάστατο, δισδιάστατο ή τρισδιάστατο.

- **Προσδιορισμός πλέγματος.**

Ένα πλέγμα μπορεί να έχει δύο διαστάσεις το πολύ και τις διαστάσεις του τις περνάμε ως παραμέτρους στη συνάρτηση πυρήνα με τρόπο που θα εξεταστεί παρακάτω. Κάθε πυρήνας εκτέλεσης δημιουργεί ένα πλέγμα εκτέλεσης, του οποίου οι διαστάσεις προσδιορίζονται με παρόμοιο τρόπο, όπως με τα μπλοκ και τα νήματα με τις μεταβλητές τύπου dim3:

→ gridDim.x

→ gridDim.y

Ένα πλέγμα μπορεί να είναι είτε μονοδιάστατο, είτε δισδιάστατο.

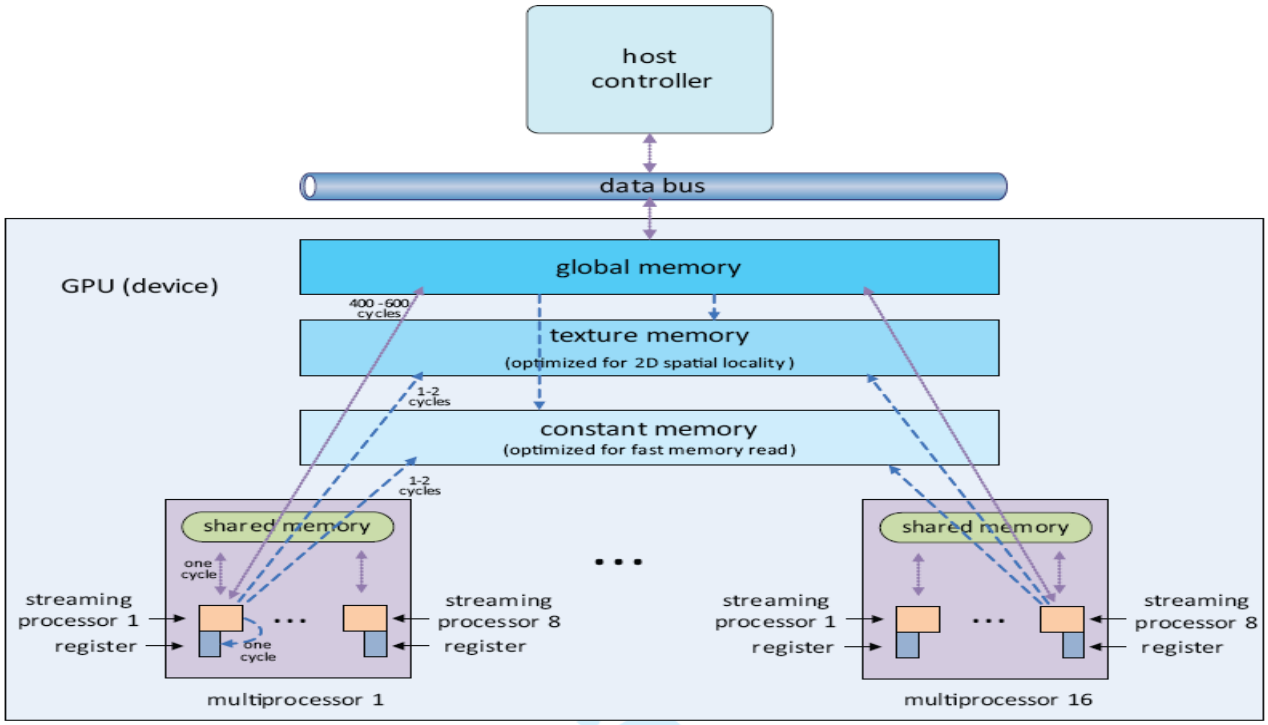
- **Προσδιορισμός νημάτων ανά warp**

  Γίνεται μέσω της μεταβλητής **warpSize** η οποία είναι τύπου int και καθορίζει από πόσα νήματα αποτελείται ένα σμήνος.

  → int warpSize;

Λογική Ιεραρχία Μνήμης στο μοντέλο CUDA

0011



Σχήμα 5.2.5: Λογική οργάνωση μνήμης

Καταρχήν, πρέπει να επισημάνουμε, ότι η μνήμη της κάρτας γραφικών (και συγκεκριμένα μνήμη τύπου GDDR) είναι ανεξάρτητη από τη μνήμη του host συστήματος. Η μνήμη της συσκευής αναφέρεται ως **καθολική μνήμη (global memory)** είτε ως **device memory** και σ' αυτήν τη μνήμη έχουν πρόσβαση όλα τα νήματα ανεξάρτητα από τα μπλοκ ή το πλέγμα στο οποίο ανήκουν. Εννοιολογικά, η μνήμη αυτή χωρίζεται σε κάποια τμήματα τα οποία εξυπηρετούν τους GPU επεξεργαστές ανάλογα με τα δεδομένα που θα προσπελάσουν. Αυτά τα λογικά τμήματα είναι:

- **Μνήμη σταθερών (constant memory)** όπου αποθηκεύονται οι τιμές που παραμένουν σταθερές κατά τη διάρκεια των εκτελούμενων διεργασιών καθώς και τα ορίσματα των πυρήνων προς εκτέλεση.

- **Μνήμη υφών (texture memory)** όπου γίνεται η προσπέλαση δεδομένων υφής.

Η καθολική μνήμη της συσκευής μπορεί να δεσμευτεί είτε ως **γραμμική μνήμη (linear memory)** είτε ως **CUDA πίνακας (CUDA array)**. Η γραμμική μνήμη αποτελείται από ένα χώρο διευθύνσεων των 32 bit όσον αφορά τις κάρτες γραφικών με υπολογιστική δυνατότητα 1.x ενώ για τις κάρτες με υπολογιστική δυνατότητα από 2.x και άνω αυτός ο χώρος αντιστοιχεί σε 64 bit. Οι πίνακες CUDA είναι τμήματα μνήμης που έχουν βέλτιστη απόδοση στη μεταφορά δεδομένων από και προς την μνήμη υφών **(texture memory).**

Βασικές συναρτήσεις διαχείρισης της γραμμικής μνήμης είναι οι:

- cudaMalloc()
- cudaFree()
- cudaMemcpy()

Εκτός από την καθολική μνήμη υπάρχουν και άλλα επίπεδα μνήμης, στα οποία εφαρμόζεται συγκεκριμένη ιεραρχία και στα οποία έχουν πρόσβαση συγκεκριμένα τμήματα κώδικα .

Ειδικότερα υπάρχει :

➢ Η **τοπική μνήμη (local memory) σε επίπεδο νήματος**, στην οποία έχει πρόσβαση μόνο ένα συγκεκριμένο νήμα. Πρόκειται για καταχωρητές με πολύ μικρή χωρητικότητα που βρίσκονται τοποθετημένοι μέσα στα SP και τους οποίους χρησιμοποιεί κάθε νήμα για αποθήκεση προσωρινών δεδομένων.

- Η **κοινή μνήμη σε επίπεδο μπλοκ (shared memory)**, η οποία μπορεί να προσπελαστεί από όλα τα νήματα που εκτελούνται μέσα στο ίδιο μπλοκ. Οι κυριότεροι τρόποι δέσμευσης κοινόχρηστης μνήμης είναι δύο, ο δυναμικός και ο στατικός. Η στατική δέσμευση γίνεται ως εξής: __shared__int a[128] και η δυναμική δέσμευση που εκτελείται κατά την εκκίνηση του πυρήνα: extern__shared_float b[].

Πίνακας 5.2.6 : Συνοπτική παρουσίαση των στοιχείων μνήμης

Memory	Location	Cached	Access	Scope	Lifetime
Register	On-chip	N/A	R/W	Thread	Thread
Local	Off-chip	No	R/W	Thread	Thread
Shared	On-chip	N/A	R/W	Block	Block
Global	Off-chip	No	R/W	Global	Application
Constant	Off-chip	Yes	R	Global	Application
Texture	Off-chip	Yes	R	Global	Application

Μνήμη υφών (texture memory)

Η μνήμη υφών είναι μια κατηγορία μνήμης που επιτρέπει την προσπέλαση δεδομένων αποκλειστικά για ανάγνωση και προσφέρει μικρούς χρόνους προσπέλασης εκμεταλλευόμενη τη χωρική τοπικότητά τους. Πρόκειται για ειδική μνήμη RAM της οποίας η προσπέλαση γίνεται από τους πυρήνες του host προγράμματος μέσω συναρτήσεων που ονομάζονται texture fetches.

Μια τέτοια συνάρτηση περιλαμβάνει Χ παραμέτρους , η πρώτη από τις οποίες ονομάζεται texture reference και ορίζει το τμήμα της μνήμης υφών, το οποίο θα προσπελαστεί. Μια texture reference πρέπει να αντιστοιχίζεται με συγκεκριμένο τμήμα της μνήμης (το τμήμα αυτό ονομάζεται texture) προκειμένου να μπορεί να χρησιμοποιηθεί από κάποιον πυρήνα. Επίσης, μια texture reference καθορίζει και τη διάσταση ενός texture (ορίζεται ως μονοδιάστατος, δισδιάστατος ή τρισδιάστατος πίνακας).

Άλλα χαρακτηριστικά, που καθορίζονται από μια αναφορά texture, είναι ο τύπος των δεδομένων εισόδου και εξόδου, ο τρόπος με τον οποίο αυτά θα μεταφραστούν και το είδος της επεξεργασίας τους.

Τα textures μπορεί να αναφέρονται είτε σε γραμμική μνήμη, είτε σε πίνακες CUDA. Οι πίνακες CUDA μπορούν να προσπελαστούν μόνο από τους πυρήνες που τρέχουν στον host και είναι μονοδιάστατα, δισδιάστατα ή τρισδιάστατα τμήματα μνήμης, που αποτελούνται από στοιχεία. Κάθε ένα από τα στοιχεία αυτά περιέχει μία, δύο ή τέσσερις μεταβλητές που περιέχουν signed ή unsigned τιμές από 8-μπιτους, 16-μπιτους ή 32-μπιτους ακεραίους και επίσης 16-μπιτους ή 32-μπιτους αριθμούς κινητής υποδιαστολής.

```
//dilwsi metavlitis texIn ws texture reference

texture<float> texIn;

HANDLE_ERROR( cudaMalloc( (void**)&data.dev_inSrc,
imageSize ) );

HANDLE_ERROR( cudaBindTexture( NULL, texIn,
data.dev_inSrc,
imageSize ) );

__global__ void copy_const_kernel( float *iptr ) {
// map from threadIdx/BlockIdx to pixel position
int x = threadIdx.x + blockIdx.x * blockDim.x;
int y = threadIdx.y + blockIdx.y * blockDim.y;
int offset = x + y * blockDim.x * gridDim.x;
float c = tex1Dfetch(texConstSrc,offset);
if (c != 0)
iptr[offset] = c;
}
```

Κώδικας 5.2.6: Προσπέλαση στη μνήμη υφών

```
Sphere *s;

__constant__ Sphere s[SPHERES];

int main( void ) {
CPUBitmap bitmap( DIM, DIM );
unsigned char *dev_bitmap;
// allocate memory on the GPU for the output bitmap
HANDLE_ERROR( cudaMalloc( (void**)&dev_bitmap,
                    bitmap.image_size() ) );

// allocate temp memory, initialize it, copy to constant
// memory on the GPU, and then free our temp memory
Sphere *temp_s = (Sphere*)malloc( sizeof(Sphere) * SPHERES );
for (int i=0; i<SPHERES; i++) {
temp_s[i].r = rnd( 1.0f );
temp_s[i].g = rnd( 1.0f );
temp_s[i].b = rnd( 1.0f );
temp_s[i].x = rnd( 1000.0f ) - 500;
temp_s[i].y = rnd( 1000.0f ) - 500;
temp_s[i].z = rnd( 1000.0f ) - 500;
temp_s[i].radius = rnd( 100.0f ) + 20;
}
HANDLE_ERROR( cudaMemcpyToSymbol( s, temp_s,
sizeof(Sphere) * SPHERES) );
free( temp_s );
// generate a bitmap from our sphere data
dim3 grids(DIM/16,DIM/16);
dim3 threads(16,16);
kernel<<<grids,threads>>>( dev_bitmap );
// copy our bitmap back from the GPU for display
HANDLE_ERROR( cudaMemcpy( bitmap.get_ptr(), dev_bitmap,
bitmap.image_size(),
cudaMemcpyDeviceToHost ) );
bitmap.display_and_exit();
// free our memory
cudaFree( dev_bitmap );

}
```

Κώδικας 5.2.7

Μνήμη σταθερών (constant memory)

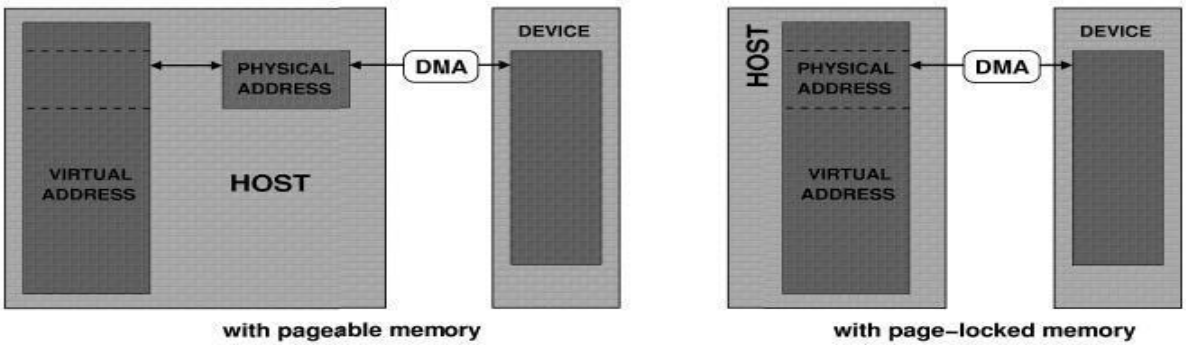
Τα δεδομένα τα οποία πρόκειται να μείνουν αμετάβλητα κατά τη διάρκεια εκτέλεσης ενός πυρήνα ή οι παράμετροι του πυρήνα αποθηκεύονται σε αυτήν τη μνήμη. Οποιοδήποτε δεδομένο αποθηκεύεται στη μνήμη σταθερών (64KB) μέσω της λέξης-κλειδί __constant__ μπορεί να προσπελαστεί αποκλειστικά και μόνο για ανάγνωση.

Αυτός ο περιορισμός αντισταθμίζεται από το γεγονός, ότι η αποθήκευση στη μνήμη σταθερών παρέχει μια σειρά από πλεονεκτήματα έναντι της αποθήκευσης στην καθολική μνήμη.

1. Αν γειτονικά νήματα απαιτήσουν πρόσβαση στην ίδια θέση μνήμης σταθερών, η GPU παράγει μόνο μία αίτηση για ανάγνωση σαν να επρόκειτο για αίτηση από ένα νήμα μόνο και στη συνέχεια κοινοποιεί μέσω broadcast το δεδομένο και στα υπόλοιπα νήματα.
2. Η μνήμη σταθερών είναι μια cached μνήμη, οπότε συνεχόμενες αιτήσεις για πρόσβαση στην ίδια θέση μνήμης δεν θα επιβαρύνουν το σύστημα με επιπλέον αιτήσεις διατηρώντας διαθέσιμο μεγαλύτερο εύρος ζώνης μνήμης.

Η αποθήκευση δεδομένων στη μνήμη σταθερών και η προσπέλασή τους εγκυμονεί κινδύνους σε περίπτωση που τα γειτονικά νήματα δεν θέλουν να προσπελάσουν την ίδια, αλλά διαφορετική θέση μνήμης το καθένα. Σ αυτήν την περίπτωση κάθε πρόσβαση εκτελείται σειριακά αυξάνοντας κατά πολύ τον χρόνο προσπέλασης, ο οποίος μετατρέπεται σε σειριακές προσβάσεις x χρόνο για κάθε αίτημα πρόσβασης.

Page-Locked (Pinned) ή απευθείας ανάθεση μνήμης στο κεντρικό σύστημα



Σχήμα 5.2.8 : Pinned (Page-locked) και pageable μνήμη

Το περιβάλλον εκτέλεσης της CUDA μας δίνει τη δυνατότητα να δεσμεύσουμε απευθείας χώρο στην εικονική μνήμη του host. Οι λόγοι για τους οποίους μια τέτοια δέσμευση μνήμης επιφέρει πλεονεκτήματα είναι:

- Η μεταφορά δεδομένων από τη page-locked μνήμη στη μνήμη της κάρτας γραφικών μπορεί να εκτελείται ταυτόχρονα με την εκτέλεση συναρτήσεων πυρήνα για ορισμένες κάρτες γραφικών.
- Το τμήμα της μνήμης που δεσμεύτηκε στον host μπορεί να αντιστοιχιστεί (map)

απευθείας στο χώρο διευθύνσεων της μνήμη στην κάρτα, οπότε απαλείφεται η ανάγκη για αντιγραφή δεδομένων από τη μία μνήμη στην άλλη.

- Τα συστήματα με δίαυλο FSB έχουν μεγαλύτερο bandwidth για μεταφορά δεδομένων που σημαίνει μεγαλύτερο εύρος ζώνης επικοινωνίας μεταξύ page-locked μνήμης και μνήμης στην κάρτα.

Ασύγχρονη εκτέλεση

Για την διευκόλυνση της ταυτόχρονης εργασίας του κεντρικού συστήματος και της κάρτας, γίνεται η χρήση ασύγχρονων συναρτήσεων, οι οποίες επιστρέφουν τον έλεγχο του συστήματος πίσω στον host ακόμα κι αν οι εργασίες που ανατέθηκαν στην κάρτα δεν έχουν ολοκληρωθεί.

Οι ασύγχρονες αυτές συναρτήσεις είναι:

- Οι πυρήνες
- Συναρτήσεις που εκτελούν μεταφορά δεδομένων μεταξύ της μνήμης του host και τις κάρτας.
- Συναρτήσεις μεταφοράς δεδομένων μεταξύ στοιχείων μνήμης μέσα στην ίδια την κάρτα.
- Συναρτήσεις που κάνουν set την μνήμη.

Ταυτόχρονη εκτέλεση πυρήνων και μεταφορά δεδομένων

- Συσκευές με υπολογιστική ικανότητα έκδοσης 2.0 και άνω μπορούν να εκτελούν μέχρι τέσσερις πυρήνες ταυτόχρονα. Οι πυρήνες που χρησιμοποιούν πολλές αναφορές σε μνήμη texture ή γενικά κάνουν εκτεταμένη χρήση της τοπικής μνήμης είναι δυσκολότερο να εκτελούνται ταυτόχρονα με άλλους πυρήνες.

- Συσκευές με υπολογιστική ικανότητα έκδοσης 2.0 μπορούν επίσης να εκτελούν μεταφορές δεδομένων από την page-locked μνήμη του host προς τη μνήμη της συσκευής και αντίστροφα.

Ρεύματα (Streams)

Οι προς εκτέλεση εντολές οργανώνονται με τη μορφή ρευμάτων και επομένως η ταυτόχρονη εκτέλεσή τους εξασφαλίζεται από τη διαχείριση των ρευμάτων με ανάλογο τρόπο. Ο τρόπος με τον οποίο εκτελούνται οι εντολές μέσα στα ρεύματα δεν είναι σίγουρος, μπορεί να εκτελούνται με διαφορετική σειρά ανά ρεύμα, είτε να εκτελούνται ταυτόχρονα

(αυτό επιφέρει απροσδιοριστία στον τρόπο με τον οποίο εκτελείται η επικοινωνία μεταξύ των πυρήνων).

Γεγονότα

Το περιβάλλον εκτέλεσης του μοντέλου CUDA επιτρέπει την ασύγχρονη καταγραφή γεγονότων από την εφαρμογή για οποιοδήποτε σημείο μέσα στον κώδικα. Το γεγονός αυτό βοηθάει στον ακριβή χρονισμό των εργασιών και τη διαχείρισή τους με αποτελεσματικό τρόπο. Ένα γεγονός δημιουργείται όταν όλες οι διεργασίες ή οι εντολές σε ένα δεδομένο ρεύμα έχουν ολοκληρωθεί. Από αυτήν τη χρονική στιγμή και έπειτα είναι δυνατή η καταγραφή του.

Σύγχρονες κλήσεις

Ενώ οι ασύγχρονες κλήσεις επιστρέφουν τον έλεγχο στο κεντρικό σύστημα ακόμα και πριν τελειώσουν οι διεργασίες που έχουν ανατεθεί στην συσκευή , οι σύγχρονες διεργασίες δεν επιστρέφουν τον έλεγχο στο νήμα διαχείρισης του κεντρικού συστήματος αν οι διεργασίες στη συσκευή δεν έχουν εκτελεστεί πλήρως.

Συνεργασία με άλλες διασυνδέσεις προγραμματισμού γραφικών

Η συνεργασία με άλλες διασυνδέσεις περιλαμβάνει δύο περιπτώσεις:

1. είτε η CUDA να **διαβάσει** δεδομένα που προέρχονται από υπολογισμούς των OpenGL και Direct3D.
2. είτε η CUDA να **γράψει** δεδομένα στη μνήμη και στη συνέχεια αυτά να χρησιμοποιηθούν στις πλατφόρμες OpenGL και Direct3D.

Διαχείριση σφαλμάτων

Όλες οι συναρτήσεις σε περίπτωση σφάλματος επιστρέφουν έναν κωδικό λάθους. Στην περίπτωση που εκτελείται μια ασύγχρονη συνάρτηση η κατάσταση διαφοροποιείται ελαφρώς, μιας και οι συναρτήσεις αυτές επιστρέφουν τον έλεγχο στην κυρίως διεργασία του host χωρίς να έχουν ολοκληρώσει την εκτέλεσή τους. Έτσι, οι κωδικοί σφαλμάτων επιστρέφονται μεν, αλλά σε διαφορετική χρονική στιγμή από αυτήν που συμβαίνει το

σφάλμα και από συναρτήσεις που ακολουθούν και δεν συνδέονται με την ίδια τη συνάρτηση που προκάλεσε το σφάλμα.

Αποσφαλμάτωση με χρήση του Device Emulator Mode

Η διασύνδεση CUDA-GDB χρησιμοποιείται για αποσφαλμάτωση κώδικα σε κάρτες με υπολογιστική ικανότητα 1.0 και άνω. Σημαντικό βοήθημα αποσφαλμάτωσης κώδικα αποτελεί η λειτουργία προσομοίησης, η οποία ενσωματώνεται στον μεταφραστή και στο περιβάλλον εκτέλεσης και λειτουργεί ακόμα και σε συσκευές, που δεν μπορούν να τρέξουν CUDA. Σ' αυτήν την περίπτωση η μετάφραση και η εκτέλεση του κώδικα που κανονικά εκτελείται στην κάρτα γίνεται στο κεντρικό σύστημα επεξεργασίας χρησιμοποιώντας εργαλεία αποσφαλμάτωσης που διαθέτει ο host.

ΕΝΟΤΗΤΑ 5.2.1

ΓΛΩΣΣΑ ΠΡΟΓΡΑΜΜΑΤΙΣΜΟΥ – ΠΑΡΟΥΣΙΑΣΗ

Η CUDA C αποτελεί μια επέκταση της standard C (ANSI C) και περιλαμβάνει:

- Τις γνωστές εντολές της παραδοσιακής γλώσσας C.
- Επιπλέον συντακτικό για εντολές που υλοποιούν τους πυρήνες, τα μπλοκ και τα πλέγματα, δηλαδή όλες τις έννοιες.

Η προσθήκη των επεκτάσεων στην C παρέχει στους προγραμματιστές τα εργαλεία που χρειάζονται για να διαχειριστούν τα βασικά σημεία τα οποία αποτελούν και το θεμέλιο του μοντέλου της CUDA, δηλαδή την ιεραρχία των μπλοκ νημάτων, τη διαχείριση των κοινόχρηστων μονάδων μνήμης και του συγχρονισμού των διεργασιών.

Τα βασικά αυτά σημεία της αρχιτεκτονικής παρέχουν ένα προγραμματιστικό πλαίσιο μέσα στο οποίο ο εκάστοτε κώδικας οργανώνεται σε κομμάτια που χαρακτηρίζονται από χονδροειδή παραλληλισμό και μπορούν να εκτελεστούν ανεξάρτητα το ένα από το άλλο. Τα κομμάτια αυτά περιέχουν στη συνέχεια άλλα μικρότερα τμήματα κώδικα οργανωμένα με

λεπτό παραλληλισμό και παραλληλισμό νημάτων και εκτελούνται με συνεργασία μεταξύ των παράλληλων επεξεργαστών.

Δηλαδή, τα νήματα μπορούν να εκτελούνται παράλληλα και ανεξάρτητα το ένα με το άλλο, αλλά έχουν επίσης και τη δυνατότητα να συνεργάζονται, αν ο αλγόριθμος προς υλοποίηση σε CUDA είναι δομημένος με κατάλληλο τρόπο. Σε αυτό το σημείο ακριβώς είναι που παρατηρούμε το κέρδος, που προσφέρει η παράλληλη εκτέλεση ενός αλγορίθμου.

Υποκεφάλαιο 5.3

ΣΤΟΙΧΕΙΑ ΤΗΣ ΠΡΟΓΡΑΜΜΑΤΙΣΤΙΚΗΣ ΔΙΕΠΑΦΗΣ ΤΗΣ CUDA C

- **Προσδιοριστικά μεταβλητών**

  Πρόκειται για λέξεις κλειδιά, που προσδιορίζουν το πού αποθηκεύονται και πού εκτελούνται οι μεταβλητές και άλλα στοιχεία:

  - **__device__int GlobalVar:** ορίζεται μια μεταβλητή GlobalVar η οποία αποθηκεύεται στην καθολική μνήμη της κάρτας γραφικών και έχει διάρκεια ζωής όσο εκτελείται η εφαρμογή. Έχουν πρόσβαση σε αυτήν όλα τα νήματα του πλέγματος, καθώς επίσης και το κεντρικό σύστημα του host μέσω των βιβλιοθηκών του runtime.
  - **__shared__int SharedVar:** δηλώνει μια μεταβλητή η οποία αποθηκεύεται στην κοινόχρηστη μνήμη στην οποία έχουν προσπέλαση αποκλειστικά και μόνο τα νήματα εντός του ίδιου μπλοκ. Έχει διάρκεια ζωής όσο το μπλοκ βρίσκεται υπό εκτέλεση.
  - **__constant_int ConVar:** δηλώνει μια μεταβλητή η οποία αποθηκεύεται μέσα στη μνήμη σταθερών και παραμένει εκεί μέχρι να τελειώσει η εκτέλεση της συγκεκριμένης εφαρμογής. Σ' αυτήν τη μεταβλητή έχουν πρόσβαση όλα τα νήματα του πλέγματος, καθώς επίσης και το host σύστημα μέσω του runtime.

- **Προσδιοριστικά συναρτήσεων**

  Πρόκειται για λέξεις – κλειδιά που προσδιορίζουν αφενός πού θα εκτελεστούν οι συναρτήσεις και αφετέρου από πού θα κληθούν (είτε από το σύστημα host, είτε από την κάρτα γραφικών).

- **__device__void SimpleFunc(...):** Δηλώνεται μια συνάρτηση, η οποία εκτελείται στην κάρτα γραφικών και καλείται αποκλειστικά και μόνο από αυτήν.

- **__global__ void KernelFunc(...):** Πρόκειται για τη δήλωση μιας συνάρτησης – πυρήνα η οποία καλείται αποκλειστικά και μόνο από το host σύστημα κεντρικής επεξεργασίας και εκτελείται στην κάρτα γραφικών.

- **__host__ void SimpleFunc(...):** Δηλώνεται μια συνάρτηση που εκτελείται στον host και μπορεί να κληθεί μόνο από τον host.

- **__host__and__device__ void SimpleFunc(...):** Δηλώνεται μια συνάρτηση η οποία εκτελείται και στη συσκευή της κάρτας γραφικών και στο κεντρικό host σύστημα.

- **__noinline__void NoInlineFunc(...):** Προτρέπει τον μεταφραστή, εφόσον υπάρχει η δυνατότητα, να μη συμπεριλάβει τη συγκεκριμένη συνάρτηση ως inline (ενσωματωμένη) μέσα στον κώδικα που θα εκτελεστεί στην κάρτα γραφικών.

- **Κλήση ενός πυρήνα από τον κώδικα του host για εκτέλεση στην κάρτα γραφικών :**

  - **KernelFunc <<< 500, 128 >>>**

Την παραπάνω γραμμή τη συναντάμε μέσα σε τμήματα host κώδικα. Σηματοδοτεί την κλήση μιας συνάρτησης για να γίνει η εκτέλεσή της στην κάρτα γραφικών.

- **Διανυσματικοί τύποι:**

Υπάρχουν ειδικοί διανυσματικοί τύποι, οι οποίοι παράγονται από τους αντίστοιχους βασικούς τύπους, τους ακέραιους και τους τύπους κινητής υποδιαστολής. Κάθε διανυσματικός τύπος περιγράφεται από τέσσερις συντεταγμένες στις οποίες παρέχεται πρόσβαση μέσω τεσσάρων πεδίων, των x, y, z, w. Για τη δημιουργία και τη χρήση των διανυσματικών τύπων καλούνται οι κατάλληλες συναρτήσεις δημιουργίας (δομητές) που έχουν τη μορφή **make_<type name>** και μέσω των οποίων γίνεται και η αρχικοποίηση των x, y, z, w πεδίων.

  - **int2 make_int2(int x, int y);**

Για παράδειγμα, στην παραπάνω γραμμή κατασκευάζεται ένα διάνυσμα τύπου int2 με τιμή (x,y)

Όσον αφορά τον κώδικα στο σύστημα host, οι απαιτήσεις για δέσμευση μνήμης των διανυσματικών τύπων είναι ίδιες με τις απαιτήσεις μνήμης των αντίστοιχων βασικών τύπων. Πάντως, το γεγονός αυτό δεν ισχύει απολύτως και για τον κώδικα που εκτελείται στην κάρτα γραφικών.

Ένας διανυσματικός τύπος που χρησιμοποιείται ευρέως στο μοντέλο CUDA είναι ο dim3. Πρόκειται για έναν διανυσματικό ακέραιο, ο οποίος βασίζεται στον τύπο uint3 και χρησιμοποιείται για να προσδιορίζει διαστάσεις. Όποια μεταβλητή ορίζεται ως τύπου dim3 και δεν αρχικοποιείται από τον προγραμματιστή παίρνει εξ ορισμού την τιμή 1.

Οι διανυσματικοί τύποι που υποστηρίζονται είναι :

→ [u]char1, [u]char2, [u]char3, [u]char4

→ [u]short1, [u]short2, [u]short3, [u]short4

→ [u]int1, [u]int2, [u]int3, [u]int4

→ [u]long1, [u]long2, [u]long3, [u]long4

→ longlong1, longlong2

→ float1, float2, float3, float4

→ double1, double2

- **<u>Συναρτήσεις συγχρονισμού</u> :**

  - **void __syncthreads():** Περιμένει μέχρις ότου όλα τα νήματα του ίδιου μπλοκ φτάσουν στο σημείο του κώδικα, όπου έχει οριστεί η syncthreads(). Επίσης, θα πρέπει όλες οι προσπελάσεις στην καθολική και την κοινόχρηστη μνήμη που έγιναν από τα νήματα του μπλοκ μέχρι εκείνο το σημείο να είναι ορατές σε όλα τα νήματα του μπλοκ.

- **<u>Memory Fence Συναρτήσεις:</u>**

Οι συναρτήσεις αυτές επιτρέπουν τη διαχείριση των προσπελάσεων στην καθολική και την κοινόχρηστη μνήμη.

  - **void__threadfence()**

- **void__threadfence_block()**

- **<u>Ενδεικτικές συναρτήσεις του CUDA Runtime</u>:**

Διαχείρηση κάρτας

- cudaGetDeviceCount() , cudaGetDeviceProperties()

Διαχείρηση γραμμικής μνήμης στην κάρτα

- cudaMalloc(), cudaFree() ,
- cudaMemcpy() με παραμέτρους:
- cudaMemcpyHostToHost / cudaMemcpyHostToDevice
- cudaMemcpyDeviceToHost / cudaMemcpyDeviceToDevice

Συνεργασία με άλλες προγραμματιστικές πλατφόρμες

- cudaGLMapBufferObject() , cudaD3DMapResources()

Διαχείριση μνήμης texture στην κάρτα

- cudaBindTexture() , cudaBindTextureToArray()

Paged-Locked Memory in Host

- cudaMallocHost() , cudaFreeHost()

Κοινόχρηστη μνήμη

- Στατική δέσμευση με τη δήλωση __shared__int a[64]
- Δυναμική δέσμευση με τη δήλωση extern __shared__float b[]

Διαχείριση σφαλμάτων

- cudaError_t cudaGetLastError(void)
- const char* cudaGetErrorString (cudaError_t error)

Μετάφραση στο περιβάλλον CUDA

- Βασική εντολή για κλήση του συμβολομεταφραστή είναι η **nvcc flags filename.cu**

Flags

- o όνομα_παραγόμενου_αρχείου
- g πληροφορίες για αποσφαλμάτωση στη μεριά του host
- G αποσφαλμάτωση στην κάρτα
- deviceemu προσομοίωση στον host
- use_fast_math χρήση της ταχύτερης έκδοσης μαθηματικής συνάρτησης που είναι ενσωματωμένη στο περιβάλλον εκτέλεσης της κάρτας γραφικών.
- arch ματάφραση για συγκεκριμένη αρχιτεκτονική GPU
- #pragma unroll n ξετυλίγει τη δομή επανάληψης n φορές

Πίνακας 5.3.1 : Μαθηματικές συναρτήσεις του μοντέλου CUDA

Operator/Function	**Device Function**
x/y	__fdividef(x,y)
sinf(x)	__sinf(x)
cosf(x)	__cosf(x)
tanf(x)	__tanf(x)
sincosf(x,sptr,cptr)	__sincosf(x,sptr,cptr)
logf(x)	__logf(x)
log2f(x)	__log2f(x)
log10f(x)	__log10f(x)
expf(x)	__expf(x)
exp10f(x)	__exp10f(x)
powf(x,y)	__powf(x,y)

Ατομικές συναρτήσεις

Οι ατομικές συναρτήσεις εκτελούνται κάθε φορά από ένα νήμα αποκλειστικά και επομένως κατά τη διάρκεια εκτέλεσης της εντολής αποκλείεται η παρέμβαση από οποιοδήποτε άλλο νήμα. Από τη στιγμή που κάποιο νήμα θα ξεκινήσει να εκτελεί την ατομική εντολή, κανένα άλλο νήμα δεν θα μπορέσει να προσπελάσει τη θέση μνήμης στην οποία αναφέρεται η εντολή.

Μια ατομική συνάρτηση εκτελεί τρία βήματα: ανάγνωση, τροποποίηση και αποθήκευση αλλαγών σε μια θέση στην καθολική ή την κοινόχρηστη μνήμη. Όλες οι ατομικές λειτουργίες επενεργούν αποκλειστικά σε signed και unsigned ακέραια δεδομένα. Εξαίρεση αποτελεί η εντολή atomicAdd() για τις κάρτες γραφικών με υπολογιστική δυνατότητα 2.0 και την λειτουργία atomicExch() που ισχύει για όλες τις κάρτες και μπορεί να διαχειρίζεται και δεδομένα κινητής υποδιαστολής μονής ακρίβειας. Μερικές ατομικές συναρτήσεις αναφέρονται ενδεικτικά παρακάτω:

- **atomicAdd(), atomicSub()**
- **atomicExch(), atomicMin(), atomicMax()**

Συναρτήσεις για textures

Οι συναρτήσεις που πραγματοποιούν προσπελάσεις στη μνήμη υφών δέχονται σαν παραμέτρους αναφορές προς αντικείμενα που αποθηκεύονται στη μνήμη υφών. Οι αναφορές αυτές περιλαμβάνουν έναν συνδυασμό από αμετάβλητα (γνωστά κατά το χρόνο της συμβολομετάφρασης) και μεταβλητά ορίσματα (γνωστά κατά το χρόνο εκτέλεσης).

Αυτός ο συνδυασμός ορισμάτων προσδιορίζει τον τρόπο με τον οποίο θα μεταφραστούν οι συντεταγμένες του αντικειμένου, την επεξεργασία κατά τη διάρκεια εκτέλεσης της συνάρτησης προσπέλασης και την τιμή επιστροφής της συνάρτησης.

- **Tex1D(), tex2D(), tex3D()**

Χρονισμός

- **clock_t clock();**

Η κλήση της παραπάνω συνάρτησης επιστρέφει την τιμή του χρονιστή για ένα συγκεκριμένο SM. Η τιμή του κυκλώματος χρονιστή αυξάνεται κατά ένα σε κάθε κύκλο ρολογιού. Μετρώντας την τιμή αυτή στην αρχή της εκτέλεσης κάθε πυρήνα και στο τέλος του, αφαιρώντας την τελική τιμή από την αρχική και κάνοντας το ίδιο για κάθε νήμα μπορούμε να προσδιορίσουμε το συνολικό χρόνο που χρειάστηκε η κάρτα γραφικών για να ολοκληρώσει το συγκεκριμένο νήμα εντολών. Πάντως, ο μετρητής αυτός δεν προσδιορίζει τον ακριβή χρόνο που χρειάστηκε η κάρτα γραφικών αποκλειστικά για την εκτέλεση του νήματος των εντολών. Ο συνολικός χρόνος είναι κατά πολύ μεγαλύτερος σε σχέση με τον χρόνο εκτέλεσης του νήματος, καθώς ο συνολικός χρόνος μετράται ανά νήμα και τα νήματα μπορεί να απέχουν μεταξύ τους χρονικά.

Συναρτήσεις Warp Voting

Οι συναρτήσεις αυτές υποστηρίζονται από τις κάρτες γραφικών υπολογιστικής δυνατότητας 1.2 και ανώτερη.

- **int __all(int predicate);**
- **int __any(int predicate);**
- **unsigned int __ballot(int predicate);**

- **Πρόσθεση όλων των στοιχείων ενός πίνακα a[] με το b - Παράδειγμα κώδικα**

CPU program	CUDA program

void increment_cpu (float *a , float b , int N)
 	
{
 	
for (int idx=0; idx<N ; idx++)
a[idx]=a[idx] + b;
 	
}
 	
void main()
 	
{
…….
increment_cpu(a, b ,N);
}
	__global__ void increment_gpu (float *a , float b , int N)
	
	{
	
	int idx = blockIdx.x * blockDim.x + threadIdx.x ;
	if(idx < N)
	a[idx]=a[idx] + b;
	
	}
	
	void main()
	
	{
	…….
	dim3 dimBlock (blocksize);
	dim3 dimGrid(ceil (N / (float) (blocksize)));
	
	increment_gpu<<<dimGrid, dimBlock>>> (a,b,N);
	
	}

Σχήμα 5.3.1 : Σύγκριση κώδικα για εκτέλεση σε κεντρικό σύστημα (host) και κάρτα γραφικών.

1

- **ΑΛΓΟΡΙΘΜΙΚΟ ΠΑΡΑΔΕΙΓΜΑ**

→Παράδειγμα απλού κώδικα ο οποίος αυξάνει κάθε στοιχείο ενός διανύσματος a με την τιμή b.

- **Αλγόριθμος που εκτελείται στην πλευρά του host υπολογιστή**

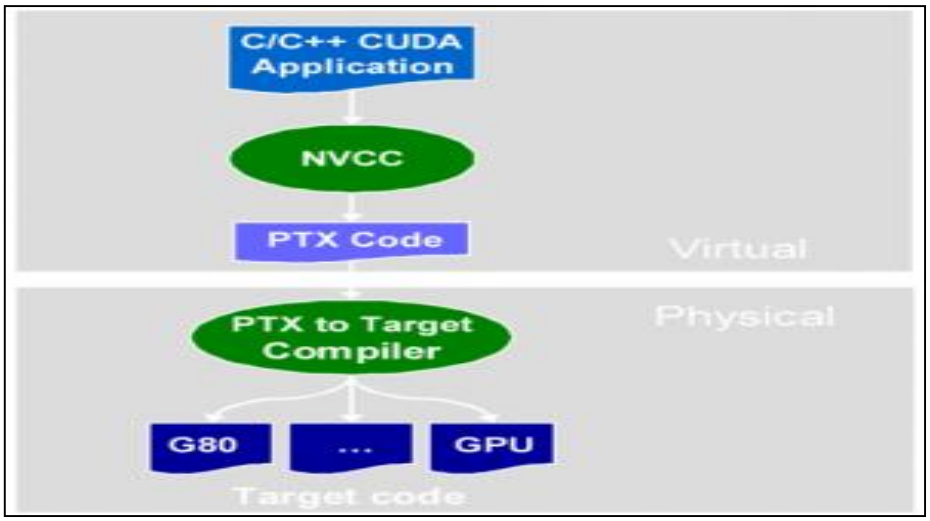
```
//allocate host memory
unsigned int numBytes = N * sizeof (float);
float* h_A= (float * ) malloc (numBytes);

//allocate memory in device
float* d_A = 0;
cudaMalloc((void**) & d_A , numBytes);

//copy data from host to device
cudaMemcpy(d_A, h_A, numBytes , cudaMemcpyHostToDevice);

//invoke the kernel
increment_gpu<<< N / blockSize , blockSize >>> (d_A , b);

//copy data from device back to host
cudaMemcpy (h_A , d_A , numBytes , cudaMemcpyDeviceToHost);

//free device memory
cudaFree(d_A);
```

- **Αλγόριθμος που εκτελείται στην κάρτα γραφικών**

```
__global__ void increment_gpu ( float *a , float b , int N)

{

        idx = blockIdx.x * blockDim.x + threadIdx.x ;
                if( idx < N )
                a[idx]=a[idx] + b;
}
```

Υποκεφάλαιο 5.4

PTX ΚΑΙ CUDA DRIVER API



Σχήμα 5.4.1 : Μετάφραση στο μοντέλο CUDA

PTX :

- ✓ Πρόκειται για την εικονική μηχανή παράλληλης εκτέλεσης νημάτων μέσω προγραμματισμού χαμηλού επιπέδου.
- ✓ Ως γλώσσα προγραμματισμού χαμηλού επιπέδου περιλαμβάνει ένα σταθερό προγραμματιστικό μοντέλο με την κατάλληλη αρχιτεκτονική συνόλου εντολών (**ISA**).

Ο κώδικας, ο οποίος έχει συνταχθεί με βάση την PTX, μεταφράζεται κατά τη στιγμή της εκτέλεσής του στην εκάστοτε κάρτα και προσαρμόζεται στο υλικό στο οποίο θα εκτελεστεί κάθε φορά. Ακόμα και οι εντολές που συντάσσονται για μεταφραστές CUDA C/C++ μετατρέπονται σε εντολές PTX και αυτές με τη σειρά τους βελτιστοποιούνται και μεταφράζονται σε εντολές προσαρμοσμένες στο υλικό της κάρτας γραφικών που εκτελεί τον αλγόριθμο.

```
.reg .b32 r1, r2;
.global .f32 array[N];
start: mov.b32 r1, %tid.x;
shl.b32 r1, r1, 2; // shift thread id by 2 bits
ld.global.b32 r2, array[r1]; // thread[tid] gets array[tid]
add.f32 r2, r2, 0.5; // add 1/2
```

Κώδικας 5.4.2 : Εντολές σε γλώσσα PTX

Στόχοι της PTX

- ✓ Η παροχή ενός ισχυρού σετ εντολών που θα προσαρμόζεται στα χαρακτηριστικά υλικού κάθε γενιάς κάρτας.
- ✓ Η αύξηση της απόδοσης των αλγορίθμων, που περνούν από τη διαδικασία της μετάφρασης, ώστε η εκτέλεσή τους να είναι συγκρίσιμη με την εκτέλεση του αλγορίθμου σε native κώδικα.
- ✓ Η παροχή ενός ανεξάρτητου συνόλου εντολών, το οποίο θα μπορεί να αποτελεί ενιαία βάση για διαφορετικές πλατφόρμες (CUDA C/C++)
- ✓ Η διευκόλυνση συγγραφής κώδικα για βιβλιοθήκες, πυρήνες και διαδικασίες ελέγχων αρχιτεκτονικής.

Κυρίαρχη έννοια της PTX αποτελούν οι πίνακες νημάτων **(CTA – Cooperative Thread Array)** οι οποίοι συγκεντρώνουν ένα σύνολο από νήματα, τα οποία εκτελούνται παράλληλα και τα οποία μπορούν να συνεργαστούν μεταξύ τους αν συγχρονιστούν προγραμματιστικά σε κατάλληλα σημεία εκτέλεσης. Κάθε νήμα χαρακτηρίζεται από ένα μοναδικό αριθμό, **tid**, ο οποίος προσδιορίζεται από ένα τρισδιάστατο διάνυσμα της μορφής (tid.x, tid.y, tid.z). Επίσης και οι πίνακες CTA προσδιορίζονται από ένα τρισδιάστατο διάνυσμα της μορφής (ntid.x, ntid.y, ntid.z) το οποίο καθορίζει τον αριθμό των νημάτων που περιέχει κάθε διάσταση του πίνακα νημάτων.

Η ιεραρχία της μνήμης είναι ίδια με αυτήν του μοντέλου της CUDA C, ξεκινώντας από τους καταχωρητές που είναι διαθέσιμοι για κάθε νήμα και φτάνοντας στην καθολική μνήμη που αποτελεί τη συνολική μνήμη της κάρτας που είναι προσβάσιμη από όλα τα νήματα.

Στην περίπτωση της PTX το συντακτικό των εντολών είναι σαφώς δυσκολότερο σε σχέση με την CUDA C αλλά προσφέρει καλύτερο προγραμματιστικό έλεγχο πάνω στις λειτουργίες και τις δυνατότητες της κάρτας, κάτι φυσικά που είναι αναμενόμενο, αφού πρόκειται για μια γλώσσα προγραμματισμού χαμηλού επιπέδου.

CUDA Driver API

Το περιβάλλον CUDA Driver API αποτελεί ένα πλαίσιο προγραμματισμού που στηρίζεται σε αντικείμενα και αδιαφανείς χειριστές, οι οποίοι τα χειρίζονται μέσω διαφόρων συναρτήσεων. Πρόκειται για μια προστακτική γλώσσα υψηλότερου επιπέδου σε σχέση με τη γλώσσα PTX αλλά χαμηλότερου επιπέδου σε σχέση με την CUDA C.

Η υλοποίηση του περιβάλλοντος αυτού γίνεται μέσω της βιβλιοθήκης nvcuda και το πρόθεμα για κάθε στοιχείο του CUDA Driver API είναι το cu. Πριν από την κλήση οποιασδήποτε συνάρτησης της πλατφόρμας αυτής πρέπει να κληθεί η συνάρτηση αρχικοποίησης cuInit(). Στη συνέχεια πρέπει να πρέπει να δημιουργηθεί ένα cuda Context το οποίο θα πρέπει να αντιστοιχιστεί στην κάρτα γραφικών και να δηλωθεί ως το τρέχον context του νήματος host.

Μέσα στα πλαίσια ενός CUDA context, οι πυρήνες φορτώνονται εξωτερικά στον κώδικα του κεντρικού νήματος επεξεργασίας ως εντολές PTX ή δυαδικά αντικείμενα και γι' αυτό το λόγο, οι πυρήνες που είναι γραμμένοι σε C πρέπει να μεταφράζονται χωριστά σε PTX εντολές ή δυαδικά αντικείμενα.

Πίνακας 5.4.3 : Αντικείμενα της γλώσσας PTX

Object	**Handle**	**Description**
Device	CUdevice	Κάρτα γραφικών με υποστήριξη CUDA
Context	CUcontext	Αντίστοιχη έννοια με μια CPU διεργασία
Module	CUmodule	Αντίστοιχη έννοια με μια δυναμική βιβλιοθήκη
Function	CUfunction	Πυρήνας
Heap memory	CUdeviceptr	Δείκτης αναφοράς προς την καθολική μνήμη
CUDA array	CUarray	Αντικείμενο για δεδομένα οργανωμένα σε μονοδιάστατο ή δισδιάστατο πίνακα . Η προσπέλαση γίνεται από αναφορές τύπου texture.
Texture reference	CUtexref	Αντικείμενο που καθορίζει την ερμηνεία των texture δεδομένων.

Προτιμότερο είναι σε περίπτωση που θέλει κάποιος οι εφαρμογές του να εκτελούνται και σε μελλοντικές γενιές καρτών, να συντάσσει κώδικα σε PTX και όχι σε δυαδικά αντικείμενα, διότι αυτά είναι μεταφράζονται σε γηγενή κώδικα προσαρμοσμένο στην αρχιτεκτονική μιας συγκεκριμένης κάρτας γραφικών.

Βασικά αντικείμενα του CUDA Driver API

- **Device:** Σε αυτήν την περίπτωση αναφερόμαστε στην κάρτα γραφικών, η οποία ενσωματώνει τις κατάλληλες βιβλιοθήκες συναρτήσεων για την υποστήριξη της CUDA. Ο χειριστής του αντικειμένου της κάρτας είναι ο CUdevice.

- **Context:** Υπάρχει εννοιολογική αναλογία με μια CPU διεργασία. Όλοι οι πόροι και οι λειτουργίες του Driver API ενσωματώνονται μέσα στο CUcontext, που αποτελεί τον χειριστή του αντικειμένου και μόλις το αντίστοιχο context τερματιστεί, οι πόροι αυτοί αποδεσμεύονται και γίνονται διαθέσιμοι σε άλλα contexts.

- **Module:** Πρόκειται για πακέτα κώδικα και δεδομένων, που φορτώνονται δυναμικά σε μια κάρτα γραφικών, προσομοιάζοντας για παράδειγμα τη λειτουργία των δυναμικών βιβλιοθηκών του λειτουργικού συστήματος των Windows. Όλα τα ονόματα των συναρτήσεων, των καθολικών μεταβλητών και των αναφορών στη μνήμη υφών είναι προσβάσιμα από τα Modules, προκειμένου ο κώδικας που γράφεται από τρίτους να μπορεί να εκτελεστεί στα διαφορετικά CUDA context.

- **Function:** Πρόκειται για το αντικείμενο που περιγράφει τον πυρήνα ο οποίος φορτώνεται για εκτέλεση στην κάρτα γραφικών. Χειριστής του αντικειμένου είναι ο CUfunction.

- **Heap Memory:** Δείκτης που αναφέρεται στην μνήμη της κάρτας η οποία μπορεί να δεσμευθεί ως γραμμική μνήμη μέσω συναρτήσεων όπως η cuMemAlloc() ή η cuMemAllocPitch() και να αποδεσμευθεί χρησιμοποιώντας τη συνάρτηση cuMemFree().

- **CUDA array:** Αδιαφανές αντικείμενο το οποίο χρησιμοποιείται για τη δέσμευση μονοδιάστατων ή τρισδιάστατων πινάκων στην κάρτα, με δεδομένα που προσπελαύνονται μέσω αναφορών τύπου texture (υφών).

- Texture reference: Το αντικείμενο αυτό προσδιορίζει τον τρόπο ερμηνείας των δεδομένων που αναφέρονται στη μνήμη τύπου texture.

http://timothylottes.blogspot.com/2010/05/cuda-driver-api-example.html

```
#include "EmptyTest.h"

#include "cuda.h"
#include <stdio.h>

int main(void) {
    // INITIALIZE RUNTIME API
    CUdevice cudaDevice;
    CUcontext cudaContext;
    CUevent cudaEvent[2];
    cuInit(0);
    cuDeviceGet(&cudaDevice, 0);
    cuCtxCreate(&cudaContext, CU_CTX_SCHED_SPIN | CU_CTX_MAP_HOST, cudaDevice);
    cuEventCreate(cudaEvent+0, CU_EVENT_DEFAULT);
    cuEventCreate(cudaEvent+1, CU_EVENT_DEFAULT);
    // LOAD KERNEL
    CUmodule module = (CUmodule)0;
    CUfunction function;
    CUdeviceptr clkDevice;
    unsigned int* clkHost;
    cuModuleLoad(&module, "Empty.cubin");
    cuModuleGetFunction(&function, module, "Empty");
    cuFuncSetBlockShape(function, THREADS_PER_CTA, 1, 1);
    cuFuncSetSharedSize(function, 0);
    // ALLOCATE SPACE FOR CLOCK RESULTS
    cuMemAlloc(&clkDevice, CTAS * 4);
    clkHost = (unsigned int*) malloc(CTAS * 4);
    // LAUNCH AND TIME KERNEL
    cuParamSetSize(function, 4);
    cuParamSeti(function, 0, clkDevice);
    cuStreamQuery(0);
    cuEventRecord(cudaEvent[0], 0);
    cuLaunchGrid(function, CTAS, 1);
    float ms;
    cuEventRecord(cudaEvent[1], 0);
    cuEventSynchronize(cudaEvent[1]);
    cuEventElapsedTime(&ms, cudaEvent[0], cudaEvent[1]);
    // READ CLOCKS
    cuMemcpyDtoH(clkHost, clkDevice, CTAS * 4);
    unsigned long long int avgClk = 0;
    unsigned long long int ctaTotal = 0;
    unsigned int maxClk = 0;
    unsigned int minClk = ~0;
    unsigned int cta;
    for(cta = 0; cta < CTAS; cta++) {
        unsigned int clk = clkHost[cta];
        if(clk > maxClk) maxClk = clk;
        if(clk < minClk) minClk = clk;
        avgClk += ((unsigned long long int)clk); }
    printf("Empty ::: cuEventElapsedTime()=%fms ::: 1st warp of CTA clock() -> min=%d,
avg=%d, max=%d ::: CTAS=%d\n",
        ms, minClk, (unsigned int)(avgClk/((unsigned long long int)CTAS)), maxClk,
CTAS); }
```

Κώδικας 5.4.4 : Κώδικας CUDA Driver API

Κεφάλαιο 6

Προγραμματιστική διεπαφή CUDA – Λεπτομέρειες υλοποίησης υλικού

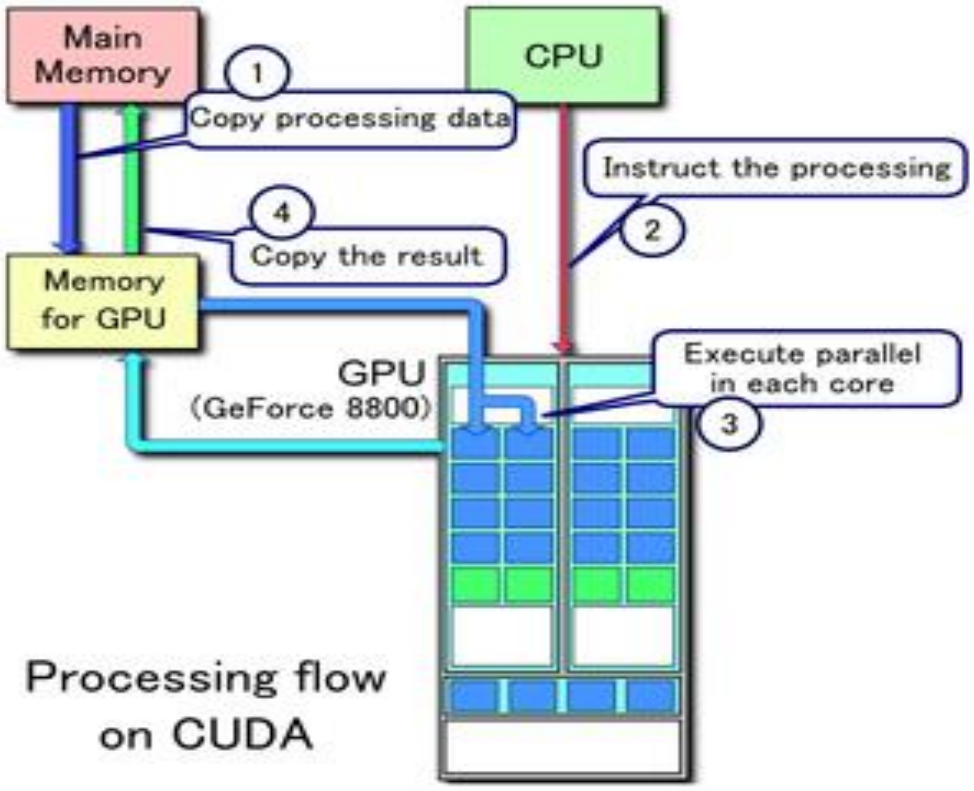
6.1 Υλοποίηση CUDA στο υλικό της κάρτας

6.2 Η προγραμματιστική διεπαφή της CUDA

Υποκεφάλαιο 6.1

ΥΛΟΠΟΙΗΣΗ CUDA ΣΤΟ ΥΛΙΚΟ ΤΗΣ ΚΑΡΤΑΣ

00**11**



Σχήμα 6.1: Ακολουθία εκτέλεσης του μοντέλου CUDA

Όπως έχει ήδη αναφερθεί, η αρχιτεκτονική του μοντέλου CUDA βασίζεται στους **Streaming Multiprocessors (SMs)** οι οποίοι μπορούν να εκτελούν εκατοντάδες νήματα ταυτόχρονα. Οι επεξεργαστές αυτοί υλοποιούν μια συγκεκριμένη αρχιτεκτονική χάρη στην οποία έχουν τη δυνατότητα να εκτελούν ένα μεγάλο αριθμό παράλληλων νημάτων.

Αυτή η αρχιτεκτονική ονομάζεται **SIMT (Single Instruction, Multiple Thread)** χρησιμοποιεί πολυνημάτωση και διοχετεύσεις εντολών, οι οποίες, αντίθετα με τις διοχετεύσεις τις CPU, εκτελούνται με αυστηρή σειρά χωρίς υποθετική εκτέλεση.

Ένας SM δημιουργεί, χειρίζεται και προγραμματίζει την εκτέλεση των νημάτων, αφού πρώτα τα οργανώσει σε ομάδες των 32 νημάτων που ονομάζονται **warp (σμήνη).** Ο ξενόγλωσσος όρος warp έναντι του αντίστοιχου ελληνικού όρου (σμήνος) είναι πιο δόκιμος και επομένως θα χρησιμοποιείται και στη συνέχεια υποκεφαλαίου. Τα νήματα που αποτελούν ένα warp ξεκινούν από την ίδια διεύθυνση προγράμματος, αλλά το καθένα από αυτά διαθέτει τον δικό του καταχωρητή και μετρητή εντολής, έτσι ώστε να μπορεί να ακολουθεί διαφορετικό μονοπάτι εκτέλεσης σε σχέση με τα υπόλοιπα νήματα. Ως δόκιμοι όροι χρησιμοποιούνται επίσης και το **half-warp** (το μισό warp) και το **quarter-warp** (τέταρτο του warp).

Η διαδικασία ανάθεσης νημάτων στους SM είναι απλή:

1. Όταν ένας πυρήνας καλείται, δημιουργεί ένα **πλέγμα** (grid).

2. Το πλέγμα οργανώνεται σε μπλοκ νημάτων και τα μπλοκ αυτά στη συνέχεια διανέμονται στους SM, ανάλογα με την διαθεσιμότητα του κάθε SM και τις δυνατότητες να φιλοξενήσουν ένα ή περισσότερα μπλοκ νημάτων.

3. Στη συνέχεια οι SM οργανώνουν τα μπλοκ νημάτων σε **warps,** ξεκινώντας με το πρώτο warp που αρχίζει με το νήμα 0 και συνεχίζοντας με τα υπόλοιπα νήματα κατά αύξοντα αριθμό. Τα warp που σχηματίζονται προγραμματίζονται για εκτέλεση από έναν **warp-scheduler.**

Τα 32 νήματα τα οποία συμμετέχουν σε ένα warp εκτελούν την ίδια εντολή και έτσι η μέγιστη παραλληλία παρατηρείται στις περιπτώσεις όπου όλα τα νήματα ακολουθούν το ίδιο μονοπάτι εκτέλεσης. Σε διαφορετική περίπτωση, η πορεία εκτέλεσης που ακολουθεί κάθε νήμα εκτελείται σειριακά αναιρώντας τις παράλληλες δυνατότητες του επεξεργαστή. Κάθε warp εκτελείται τελείως παράλληλα και ανεξάρτητα σε σχέση με τα υπόλοιπα warp, ασχέτως αν εκτελούν τον ίδιο ή διαφορετικά κομμάτια κώδικα.

Το περιεχόμενο εκτέλεσης (καταχωρητές, μετρητές προγράμματος κλπ) για κάθε warp που εκτελείται σε έναν SM αποθηκεύεται στην μνήμη του SM (στους καταχωρητές και την κοινόχρηστη μνήμη) και παραμένει εκεί μέχρι ότου ολοκληρωθεί η εκτέλεση του συγκεκριμένου warp. Με αυτόν τον τρόπο είναι εύκολη η εναλλαγή από το ένα warp στο άλλο προκειμένου να μην υπάρχει μεγάλο διάστημα κατά το οποίο ένας SM παραμένει αδρανής. Λόγω των περιορισμένων καταχωρητών και του συγκεκριμένου μεγέθους της κοινόχρηστης μνήμης στους SM, ο αριθμός των μπλοκ και των warp που μπορούν να φιλοξενηθούν από κάθε SM είναι συγκεκριμένος για κάθε πυρήνα και συνδέεται άμεσα με την υπολογιστική δυνατότητα της κάθε κάρτας μέσω συγκεκριμένων μαθηματικών τύπων.

Υποκεφάλαιο 6.2

Η ΠΡΟΓΡΑΜΜΑΤΙΣΤΙΚΗ ΔΙΕΠΑΦΗ ΤΗΣ CUDA

- **Device Memory**

- **cudaMalloc (void **devPtr, size_t size)**

Με τη συνάρτηση αυτή κάνουμε δέσμευση γραμμικής μνήμης στην κάρτα γραφικών. Η συνάρτηση δεσμεύει γραμμική μνήμη size σε bytes και επιστρέφει έναν δείκτη που αναφέρεται στην δεσμευμένη μνήμη.

- **cudaFree (void *devPtr)**

Με τη συνάρτηση αυτή κάνουμε αποδέσμευση της μνήμης στην οποία αντιστοιχεί ο δείκτης devPtr, ο οποίος επιστρέφεται από την κλήση μιας cudaMalloc().

- **cudaMemcpy (void *dst, const void * src, size_t count, enum cudaMemcpyKind kind)**

Πολύ βασική συνάρτηση που αντιγράφει δεδομένα που ορίζονται από τον δείκτη count σε bytes από μια περιοχή μνήμης (καθορίζεται από τον δείκτη src) σε άλλη (καθορίζεται από το δείκτη dst). Η παράμετρος kind περιγράφει το είδος των περιοχών μνήμης που εμπλέκονται στην αντιγραφή δεδομένων και καθορίζει την κατεύθυνση της αντιγραφής. Μπορεί να λάβει τις εξής τιμές:

- ✓ cudaMemcpyHostToHost
- ✓ cudaMemcpyHostToDevice
- ✓ cudaMemcpyDeviceToHost
- ✓ cudaMemcpyDeviceToDevice

- **cudaMallocPitch (void ** devPtr, size_t * pitch, size_t width, size_t height)**

Κάνει δέσμευση ενός δισδιάστατου πίνακα στην κάρτα με μέγεθος width * height bytes γραμμικής μνήμης και επιστρέφει τον αντίστοιχο δείκτη.

Παράμετροι :

- ✓ devPtr – Δείκτης προς την δεσμευμένη μνήμη.
- ✓ pitch – μέγεθος προς δέσμευση.
- ✓ width – πλάτος δέσμευσης (σε bytes).
- ✓ height – ύψος δέσμευσης.

- **cudaMalloc3D (struct cudaPitchedPtr * pitchedDevPtr, struct cudaExtent extent)**

Κάνει δέσμευση ενός τρισδιάστατου πίνακα στην κάρτα με τουλάχιστον width * height * depth bytes και επιστρέφει τον δείκτη cudaPitchedPtr.

Παράμετροι :

- ✓ pitchedDevPtr – Δείκτης προς τον δεσμευμένο πίνακα.
- ✓ extent – Το επιθυμητό μέγεθος μνήμης προς δέσμευση.

- **cudaMemcpy2D (void *dst, size_t dpitch, const void *src, size_t spitch, size_t width, size_t height, enum cudaMemcpyKind kind)**

Αντιγράφει έναν δισδιάστατο πίνακα από μια περιοχή της μνήμης (καθορίζεται από τον δείκτη src) σε μια άλλη (καθορίζεται από τον δείκτη dst). Η παράμετρος kind καθορίζει την

περιοχή μνήμης καθώς και την κατεύθυνση από και προς την οποία θα γίνει η αντιγραφή και μπορεί να πάρει τις εξής τιμές:

- ✓ cudaMemcpyHostToHost
- ✓ cudaMemcpyHostToDevice
- ✓ cudaMemcpyDeviceToHost
- ✓ cudaMemcpyDeviceToDevice

Οι παράμετροι dpitch και spitch αναφέρονται στο πλάτος της μνήμης σε bytes στην οποία έχει γίνει η δέσμευση των πινάκων που αντιστοιχούν στους δείκτες src kai dst. Οι περιοχές της μνήμης που ορίζονται από τα dpitch και spitch δεν μπορούν να επικαλύπτονται. Η παράμετρος width δεν πρέπει να ξεπερνά τις τιμές που έχουν δοθεί στις παραμέτρους dpitch και spitch. Σε περίπτωση που οι δείκτες dst και src αναφέρονται σε περιοχές μνήμης που δεν συμπίπτουν με τις περιοχές που ορίζονται από την κατεύθυνση αντιγραφής, προκύπτουν απρόσμενα αποτελέσματα.

Παράμετροι :

dst – Μνήμη προορισμού.
dpitch - Pitch της μνήμης προορισμού.
src – Διεύθυνση της μνήμης από όπου θα γίνει η αντιγραφή.
spitch - Pitch της μνήμης προέλευσης.
width – πλάτος του πίνακα που θα αντιγραφεί (εκφρασμένο σε στήλες, bytes).
height – ύψος του πίνακα που θα αντιγραφεί (εκφρασμένο σε γραμμές, bytes).
kind – Τύπος αντιγραφής.

- **cudaMemcpy3D (const struct cudaMemcpy3DParms * p)**

Παράμετροι :

p – Δομή με τις παραμέτρους της αντιγραφής:

```
struct cudaExtent {
size_t width;
size_t height;
size_t depth;
};
struct cudaExtent make_cudaExtent(size_t w, size_t h, size_t d);
struct cudaPos {
size_t x;
size_t y;
size_t z;
};
struct cudaPos make_cudaPos(size_t x, size_t y, size_t z);
struct cudaMemcpy3DParms {
struct cudaArray *srcArray;
struct cudaPos srcPos;
```

```
struct cudaPitchedPtr srcPtr;
struct cudaArray *dstArray;
struct cudaPos dstPos;
struct cudaPitchedPtr dstPtr;
struct cudaExtent extent;
enum cudaMemcpyKind kind;
};
```

Η συνάρτηση αυτή αντιγράφει δεδομένα μεταξύ δύο τρισδιάστατων αντικειμένων. Τα αντικείμενα αυτά μπορεί να είναι η μνήμη του κεντρικού συστήματος, η μνήμη της κάρτας γραφικών ή ένας πίνακας τύπου CUDA.

Η παράμετρος cudaMemcpy3DParms αρχικοποιείται με τη τιμή 0 (cudaMemcpy3DParms myParms = {0};) και καθορίζει τη διεύθυνση προορισμού, προέλευσης, το μέγεθος και το είδος της αντιγραφής που θα εκτελεστεί.

Η δομή struct την οποία δέχεται ως παράμετρος η συνάρτηση cudaMemcpy3D() πρέπει να καθορίζει έναν πίνακα προέλευσης ή έναν δείκτη προέλευσης, καθώς και έναν δείκτη προορισμού ή έναν πίνακα προορισμού.
Το πέρασμα περισσότερων παραμέτρων ως προορισμό ή/και πηγή επιστρέφει σφάλμα.

Τα srcPos και dstPos πεδία είναι προαιρετικά πεδία και περιέχουν δείκτες που δηλώνουν την αρχή της διευθυνσιοδότησης μέσα στα αντικείμενα προέλευσης και προορισμού
Για τους πίνακες CUDA, οι θέσεις αυτές πρέπει να έχουν το εύρος [0, 2048) για κάθε διάσταση.

Το extent πεδίο καθορίζει τις διαστάσεις της αντιγραφόμενης περιοχής ανάλογα με τα στοιχεία που αυτή περιέχει. Αν λαμβάνει μέρος στην αντιγραφή ένας πίνακας τύπου CUDA τότε το πεδίο extend καθορίζεται από τα στοιχεία του πίνακα. Αν δεν πρόκειται για έναν πίνακα CUDA τότε το συγκεκριμένο πεδίο καθορίζεται από τα στοιχεία που αποτελούν τον χαρακτήρα.

Το πεδίο kind καθορίζει την κατεύθυνση της αντιγραφής και οι τιμές που μπορεί να πάρει είναι:

- ✓ cudaMemcpyHostToHost
- ✓ cudaMemcpyHostToDevice
- ✓ cudaMemcpyDeviceToHost
- ✓ cudaMemcpyDeviceToDevice.

Αν και ο προορισμός και η προέλευση αναφέρονται σε πίνακες, τότε σε περίπτωση που δεν έχουν το ίδιο αριθμό στοιχείων η συνάρτηση θα επιστρέψει σφάλμα.

Οι περιοχές της μνήμης προέλευσης και προορισμού δεν μπορούν να επικαλύπτονται, διαφορετικά προκύπτουν απροσδιόριστα αποτελέσματα.

Το αντικείμενο προέλευσης πρέπει να βρίσκεται στην διεύθυνση, όπως αυτή καθορίζεται από τον δείκτη srcPos και το πεδίο extent. Το αντικείμενο προορισμού πρέπει να βρίσκεται αποκλειστικά στη διεύθυνση που υποδεικνύει ο δείκτης dstPos και το πεδίο extent.

Σε περίπτωση που το μέγεθος των δεικτών είναι μεγαλύτερο από το επιτρεπόμενο, η συνάρτηση cudaMemcpy3D() επιστρέφει σφάλμα.

- **cudaGetSymbolAddress (void ** devPtr, const char * symbol)**

Παράμετροι:

devPtr – επιστρέφει τον δείκτη προς την λέξη που αναζητούμε
symbol – καθολική μεταβλητή ή συμβολοσειρά που αναζητούμε

Επιστρέφει στον δείκτη devPtr την ακριβή διεύθυνση μνήμης στην κάρτα (καθολική μνήμη ή μνήμη σταθερών) στην οποία είναι αποθηκευμένη η λέξη που αναζητούμε.
Αν το αλφαριθμητικό αυτό δεν είναι δηλωμένο με το συγκεκριμένο όνομα ή αν δεν βρίσκεται είτε στην καθολική μνήμη, είτε στη μνήμη σταθερών, τότε ο δείκτης devPtr δεν εμφανίζει καμία αλλαγή στην τιμή του και επιστρέφεται σφάλμα του τύπου cudaErrorInvalidSymbol.
Αν υπάρχουν πολλαπλά ταιριάσματα για το συγκεκριμένο αλφαριθμητικό, τότε επιστρέφεται σφάλμα cudaErrorDuplicateVariableName.

- **cudaGetSymbolSize (size_t * size, const char * symbol)**

Παράμετροι:
size – το μέγεθος του αντικειμένου στο οποίο αναφέρεται το συγκεκριμένο αλφαριθμητικό
symbol - καθολική μεταβλητή ή συμβολοσειρά που αναζητούμε

Επιστρέφει τον δείκτη *size ο οποίος περιέχει το μέγεθος του αλφαριθμητικού. Αν το αλφαριθμητικό δεν μπορεί να βρεθεί (για τους λόγους που έχουν αναφερθεί παραπάνω), ο δείκτης *size διατηρεί την προηγούμενη τιμή του και η συνάρτηση επιστρέφει σφάλμα cudaErrorInvalidSymbol.

```
// Host code
float* devPtr;
int pitch;
cudaMallocPitch((void**)&devPtr, &pitch,
width * sizeof(float), height);
MyKernel<<<100, 512>>>(devPtr, pitch);
// Device code
__global__ void MyKernel(float* devPtr, int pitch)
{
for (int r = 0; r < height; ++r) {
```

```
float* row = (float*)((char*)devPtr + r * pitch);
for (int c = 0; c < width; ++c) {
float element = row[c];
}
}
}
```

- **Multiple Devices**

Σε ένα σύστημα μπορεί να είναι συνδεδεμένες προς χρήση περισσότερες από μία κάρτες. Το περιβάλλον CUDA διαθέτει συναρτήσεις για την απαρίθμησή τους, την ανάκτηση των ιδιοτήτων της κάθε κάρτας και την επιλογή κάποιας ή κάποιων εξ αυτών για την εκτέλεση υπολογισμών.

- **cudaGetDeviceCount (int * count)**

Παράμετροι :

count – Επιστρέφει το πλήθος των καρτών υπολογιστικής δυνατότητας από 1.0 και ανώτερη.

Η συνάρτηση επιστρέφει το πλήθος των διαθέσιμων καρτών προς εκτέλεση με υπολογιστική δυνατότητα μεγαλύτερη ή ίση της 1.0. Σε περίπτωση που δεν βρεθεί αντίστοιχη κάρτα γραφικών, η συνάρτηση θα επιστρέψει σφάλμα cudaErrorNoDevice.
Σε περίπτωση που δεν έχει φορτωθεί ο κατάλληλος οδηγός εκτέλεσης για την αναζήτηση με το συγκεκριμένο κριτήριο, επιστρέφεται το σφάλμα cudaErrorInsufficientDriver.

- **cudaGetDeviceProperties (struct cudaDeviceProp * prop, int device)**

Παράμετροι:

prop – ιδιότητες της κάρτας που επιστρέφονται από την κλήση της συνάρτησης αυτής
device –ο αναγνωριστικός αριθμός της κάρτας για την οποία ζητούμε την εμφάνιση των ιδιοτήτων

Η συνάρτηση επιστρέφει τις ιδιότητες μιας συγκεκριμένες κάρτας γραφικών.

- **cudaSetDevice (int device)**

Παράμετροι :

device – Η κάρτα γραφικών στην οποία το νήμα του κεντρικού συστήματος θα εκτελέσει τον κώδικα.

Καθορίζει την κάρτα στην οποία αναφέρεται το νήμα του κεντρικού συστήματος. Οι συναρτήσεις cudaMalloc(), cudaMallocPitch() και η συνάρτηση cudaMallocArray() θα εκτελέσουν δέσμευση μνήμης στη συγκεκριμένη συσκευή, ενώ οι συναρτήσεις cudaMallocHost(), cudaHostAlloc(), cudaHostRegister() θα δεσμεύσουν μνήμη στο κεντρικό σύστημα με διάρκεια ζωής αντίστοιχη του κώδικα που εκτελείται στην κάρτα. Όλα τα γεγονότα και τα ρεύματα που θα δημιουργηθούν, καθώς και οι πυρήνες προς εκτέλεση, θα εκτελεστούν στην κάρτα αυτή.

Η συνάρτηση αυτή μπορεί να κληθεί από οποιοδήποτε νήμα του κεντρικού συστήματος, οποτεδήποτε κριθεί αναγκαίο και μπορεί να αναφέρεται σε οποιαδήποτε κάρτα.

Η κάρτα γραφικών η οποία συνδέεται άμεσα με το νήμα εκτέλεσης του κεντρικού συστήματος έχει ως ID την τιμή 0 όταν κληθεί να εκτελέσει υπολογισμούς. Η τιμή 0 μπορεί βεβαίως να αντιστοιχεί σε οποιαδήποτε κάρτα γραφικών.

```
int deviceCount;
cudaGetDeviceCount(&deviceCount);
int device;
for (device = 0; device < deviceCount; ++device) {
cudaDeviceProp deviceProp;
cudaGetDeviceProperties(&deviceProp, device);
if (dev == 0) {
if (deviceProp.major == 9999 && deviceProp.minor == 9999)
printf("There is no device supporting CUDA.\n");
else if (deviceCount == 1)
printf("There is 1 device supporting CUDA\n");
else
printf("There are %d devices supporting CUDA\n",
deviceCount);
}
}
```

- **Texture Memory**

Τμήματα από τη μνήμη υφών διαβάζονται και στη συνέχεια δίνονται ως παράμετροι μέσα σε συναρτήσεις διαχείρισης υφών με τη μορφή αναφορών. Μια αναφορά προς ένα αντικείμενο με ιδιότητες υφής έχει ιδιότητες, κάποιες από τις οποίες είναι αμετάβλητες και

πρέπει να είναι γνωστές εκ των προτέρων στον συμβολομεταφραστή. Άλλες ιδιότητες μπορούν να τροποποιηθούν κατά τη διάρκεια της εκτέλεσης.

Μια αναφορά αντικειμένου υφής γίνεται με τη δήλωσή της ως αναφορά τύπου texture ως εξής:

- **texture<Type, Dim, ReadMode> texRef**;

- **Page-locked host memory**

Το περιβάλλον εκτέλεσης παρέχει τη δυνατότητα δέσμευσης ενός τμήματος της μνήμης του κεντρικού συστήματος ως 'καρφωτή' μνήμη.

- **cudaHostAlloc (void ** pHost, size_t size, unsigned int flags)**

Παράμετροι :

pHost – Δείκτης προς τη μνήμη που δεσμεύθηκε.
size – Επιθυμητό μέγεθος μνήμης προς δέσμευση σε bytes.
flags – Ιδιότητες της μνήμης προς δέσμευση.

Δεσμεύει απευθείας μνήμη στο κεντρικό σύστημα και την κάνει διαθέσιμη προς χρήση στην κάρτα γραφικών, με ό,τι οφέλη αυτό συνεπάγεται. Παρ' όλα αυτά η χρήση της μνήμης αυτής συνιστάται να γίνεται με σύνεση, διαφορετικά πολύ σύντομα το κεντρικό σύστημα επεξεργασίας μπορεί να έχει ελάχιστα αποθέματα μνήμης για σελιδοποίηση.

Το πεδίο flags περιέχει διάφορες παραμέτρους, οι οποίες παίρνουν διαφορετικές τιμές, ανάλογα με το είδος της μνήμης που θα δεσμευθεί. Για παράδειγμα, η παράμετρος cudaHostAllocDefault παίρνει την τιμή 0 και αναγκάζει τη συνάρτηση cudaHostAlloc() να προσομοιώσει τη λειτουργία της συνάρτησης cudaMallocHost().

- **Φορητή μνήμη**

Το τμήμα της φορητής μνήμης που δεσμεύεται ως απευθείας μνήμη μπορεί εξ ορισμού να χρησιμοποιηθεί αποκλειστικά από το νήμα εκείνο που κάνει την αρχική δέσμευση. Για να μπορέσει η καρφωτή μνήμη να είναι διαθέσιμη σε όλα τα νήματα πρέπει να δεσμευτεί με ενεργοποιημένη την παράμετρο **cudaHostAllocPortable** στη συνάρτηση **cudaHostAlloc().**

- **Write-Combining memory**

Η απευθείας μνήμη (καρφωτή μνήμη) εξ ορισμού λειτουργεί ως λειτουργεί ως μνήμη cacheable, μπορεί όμως εναλλακτικά να δεσμευτεί ως write-combining μνήμη δίνοντας την κατάλληλη τιμή στην παράμετρο **cudaHostAllocWriteCombined** της συνάρτησης **cudaHostAlloc().** ΑΗ μνήμη τύπου write-combining αποδεσμεύει τις κρυφές μνήμες επιπέδου L1 και L2 και έτσι παρέχεται περισσότερος χώρος μνήμης για την εκτέλεση της εφαρμογής. Επιπλέον, δεν παρακολουθείται από κάποιο πρωτόκολλο snooping, όταν εκτελεί μεταφορές δεδομένων στον δίαυλο PCI Express, αυξάνοντας την επίδοση κατά 40%. Ως εκ τούτου, αποτελεί μια καλή επιλογή για τη χρήση προσωρινών τμημάτων αποθήκευσης (buffers), που θα γράφονται στην CPU και θα διαβάζονται από την κάρτα μέσω της απευθείας μνήμης ή μέσω της μεταφοράς δεδομένων μεταξύ κεντρικούς συστήματος και κάρτας γραφικών.

Παρ' όλα αυτά, εξαιτίας του ότι η μνήμη αυτή έχει μικρούς χρόνους προσπέλασης για ανάγνωση δεδομένων, θα πρέπει να χρησιμοποιείται μόνο στην περίπτωση που το κεντρικό σύστημα επεξεργασίας θέλει να εγγράψει δεδομένα σε αυτήν.

- **Mapped Memory**

Σε ορισμένες κάρτες γραφικών είναι δυνατή η δέσμευση μνήμης στο κεντρικό σύστημα (host) και στη συνέχεια η αντιστοίχησή του σε μια περιοχή μνήμης στην ίδια την κάρτα γραφικών. Αυτό επιτυγχάνεται με τη χρήση της παραμέτρου **cudaHostAllocMapped** στη συνάρτηση **cudaHostAlloc().** Το τμήμα της μνήμης αυτής επομένως αντιστοιχίζεται σε δύο διευθύνσεις, μία διεύθυνση που δηλώνει στο κεντρικό σύστημα επεξεργασίας και μία αντίστοιχη διεύθυνση στην κάρτα γραφικών.

Σε αυτήν την περίπτωση ο δείκτης της μνήμης, που βρίσκεται στη συσκευή, μπορεί να ανακτηθεί καλώντας τη συνάρτηση cudaHostGetDevicePointer().

- **Ταυτόχρονη Εκτέλεση με Ασύγχρονες μεθόδους**

Σε παραπάνω κεφάλαιο έχει γίνει αναφορά στις ασύγχρονες συναρτήσεις και τον σημαντικό ρόλο τους στην εκτέλεση νημάτων σε ένα κεντρικό και σε μια συσκευή με ταυτόχρονο τρόπο. Σε κάρτες γραφικών υπολογιστικής δυνατότητας 1.1 και ανώτερα, η αντιγραφή δεδομένων μεταξύ της μνήμης του κεντρικού συστήματος και της κάρτας γίνεται ταυτόχρονα με την εκτέλεση του πυρήνα. Το αν αυτή η δυνατότητα είναι διαθέσιμη σε κάθε περίπτωση ή όχι, μπορούμε να το διαπιστώσουμε μέσω της κλήσης της συνάρτησης

cudaGetDeviceProperties() και συγκεκριμένα με τον έλεγχο της ιδιότητας **deviceOverlap.** Αυτή η δυνατότητα παρέχεται προς το παρόν μόνο για αντιγραφή δεδομένων μνήμης που δεν περιλαμβάνουν πίνακες τύπου CUDA ή δισδιάστατους πίνακες, που δεσμεύθηκαν μέσω της **cudaMallocPitch()**.

- **Ταυτόχρονη εκτέλεση πολλαπλών πυρήνων και αντιγραφής δεδομένων**

Μερικές κάρτες γραφικών με υπολογιστική δυνατότητα 2.0 και πάνω υποστηρίζουν την ταυτόχρονη εκτέλεση πολλαπλών πυρήνων. Για να διαπιστώσει κανείς αν μια κάρτα γραφικών υποστηρίζει τη συγκεκριμένη ιδιότητα θα πρέπει να καλέσει τη συνάρτηση **cudaGetDeviceProperties()** και να εξετάσει την ιδιότητα **concurrentKernels.**

Πρέπει να ληφθεί υπόψη, ότι ο μέγιστος αριθμός των πυρήνων, που μπορούν να εκτελεστούν ταυτόχρονα σε μια κάρτα, είναι τέσσερις.

Πυρήνες, οι οποίοι καλούνται από host threads, που βρίσκονται σε διαφορετικά CUDA contexts, δεν μπορούν να εκτελεστούν συγχρόνως. Επίσης, πυρήνες οι οποίοι κάνουν εκτεταμένες προσπελάσεις στη μνήμη υφών ή στην τοπική μνήμη της κάρτας είναι πολύ πιθανό, ότι δε θα εκτελούνται ταυτόχρονα με τους άλλους πυρήνες.

Οι κάρτες γραφικών αυτής της κατηγορίας μπορούν να εκτελούν ταυτόχρονα μεταφορές δεδομένων από την καρφωτή μνήμη στην μνήμη της συσκευής και αντίστροφα.

- **Ρεύματα**

Η ταυτόχρονη εκτέλεση εντολών που ανήκουν σε διαφορετικές εφαρμογές επιτυγχάνεται με τη χρήση ρευμάτων. Τα ρεύματα αποτελούνται από ακολουθίες εντολών που εκτελούνται σειριακά. Σε περίπτωση συνύπαρξης πολλαπλών ρευμάτων μέσα στην ίδια κάρτα γραφικών, η εκτέλεση των εντολών μέσα στα ρεύματα δεν ακολουθεί μια προκαθορισμένη σειρά, αλλά γίνεται με τέτοιο τρόπο, ώστε να εξασφαλίζεται η αποτελεσματική ή ακόμη και, αν είναι δυνατή, η ταυτόχρονη εκτέλεση των ρευμάτων.

- **Δημιουργία ρευμάτων**

- **typedef struct CUstream_st_ cudaStream_t**

```
cudaStream_t stream[2];
```

```
for (int i = 0; i < 2; ++i)
cudaStreamCreate(&stream[i]);
float* hostPtr;
cudaMallocHost((void**)&hostPtr, 2 * size);

for (int i = 0; i < 2; ++i)
cudaMemcpyAsync(inputDevPtr + i * size, hostPtr + i * size,
size, cudaMemcpyHostToDevice, stream[i]);
for (int i = 0; i < 2; ++i)
MyKernel<<<100, 512, 0, stream[i]>>>
(outputDevPtr + i * size, inputDevPtr + i * size, size);
for (int i = 0; i < 2; ++i)
cudaMemcpyAsync(hostPtr + i * size, outputDevPtr + i * size,
size, cudaMemcpyDeviceToHost, stream[i]);
cudaThreadSynchronize();
```

- **Γεγονότα**

Κατά την εκτέλεση μιας εφαρμογής σε περιβάλλον CUDA, πραγματοποιείται η ασύγχρονη δημιουργία γεγονότων. Ένα γεγονός δημιουργείται, όταν η εκτέλεση όλων των εντολών μέσα σε ένα ρεύμα έχει ολοκληρωθεί. Τα γεγονότα που προέρχονται από την εκτέλεση εντολών στο ρεύμα με ID μηδέν καταγράφονται, αφού έχουν ολοκληρώσει την εκτέλεση των εντολών τους όλα τα υπόλοιπα ρεύματα.

- **cudaEventCreate (cudaEvent_t * event)**

Παράμετροι :

event – δημιουργεί ένα γεγονός

- **cudaEventCreate With Flags (cudaEvent_t * event, unsigned int flags)**

Η συνάρτηση αυτή δημιουργεί ένα αντικείμενο γεγονότος με τις παραμέτρους που έχει ορίσει ο προγραμματιστής.

Το πεδίο flags περιλαμβάνει τα εξής πεδία:

• **cudaEventDefault**: Παράμετρος που ενεργοποιείται εξ ορισμού κατά τη δημιουργία γεγονότων.

• **cudaEventBlockingSync**: Δηλώνει, ότι το γεγονός που θα δημιουργηθεί πρέπει να λειτουργεί ως blocking. Ένα κεντρικό νήμα το οποίο καλεί τη συνάρτηση cudaEventSynchronize() για συγχρονισμό γεγονότων, δεν θα επιστρέψει μέχρι να ολοκληρωθεί το γεγονός, που έχει δημιουργηθεί ως Blocking.

Γεγονότα τα οποία δεν έχουν αρχικοποιημένη τιμή για την παράμετρο cudaEventBlockingSync αποδίδουν καλύτερα με τη χρήση των συναρτήσεων cudaStreamWaitEvent() και cudaEventQuery().

- **cudaEventRecord (cudaEvent_t event, cudaStream_t stream = 0)**

Παράμετροι :

event – Γεγονός προς καταγραφή.
stream – Ρεύμα που θα πραγματοποιήσει την καταγραφή.

Η συνάρτηση καταγράφει ένα γεγονός. Αν το ρεύμα καταγραφής έχει ID διαφορετικό από το μηδέν, τότε το γεγονός αυτό θα καταγραφεί, αφού όλες οι προηγούμενες συναρτήσεις εντός του ρεύματος ολοκληρώσουν τη λειτουργία τους.

Αν το ρεύμα καταγραφής είναι το 0 η καταγραφή γίνεται αφού ολοκληρωθούν όλες οι συναρτήσεις που εκτελούν λειτουργίες εντός του συγκεκριμένου πλαισίου CUDA.

Η λειτουργία της καταγραφής είναι ασύγχρονη και επομένως χρησιμοποιούμε τις συναρτήσεις cudaEventQuery() ή/και cudaEventSynchronize() για να προσδιορίσουμε τον χρόνο ακριβής καταγραφής του γεγονότος.

Αν η συνάρτηση cudaEventRecord() είχε κληθεί και από προηγουμένως, η πιο πρόσφατη κλήση είναι αυτή που θα επικαλύψει την παλαιότερη.

- **<u>Αποδέσμευση γεγονότων</u>**

- **cudaEventDestroy (cudaEvent_t event)**

Παράμετροι :

event – Το γεγονός που θέλουμε να αποδεσμεύσουμε.

Αποδεσμεύει ένα καθορισμένο γεγονός. Σε περίπτωση που το γεγονός αυτό έχει καταγραφεί χωρίς να έχει ολοκληρωθεί μέχρι τη στιγμή που καλείται η συνάρτηση αυτή θα επιστρέψει αμέσως τον έλεγχο και όλα τα δεδομένα και οι πόροι που σχετίζονται με το γεγονός θα αποδεσμευθούν με την ολοκλήρωση του γεγονότος.

```
cudaEventDestroy(start);
cudaEventDestroy(stop);
cudaEvent_t start, stop;
cudaEventCreate(&start);
cudaEventCreate(&stop);
cudaEventRecord(start, 0);
for (int i = 0; i < 2; ++i)
cudaMemcpyAsync(inputDev + i * size, inputHost + i * size,
size, cudaMemcpyHostToDevice, stream[i]);
for (int i = 0; i < 2; ++i)
MyKernel<<<100, 512, 0, stream[i]>>>
(outputDev + i * size, inputDev + i * size, size);
for (int i = 0; i < 2; ++i)
cudaMemcpyAsync(outputHost + i * size, outputDev + i * size,
size, cudaMemcpyDeviceToHost, stream[i]);
cudaEventRecord(stop, 0);
cudaEventSynchronize(stop);
float elapsedTime;
cudaEventElapsedTime(&elapsedTime, start, stop);
```

- **Κλήσεις σύγχρονων συναρτήσεων**

Όταν γίνεται κλήση μιας σύγχρονης συνάρτησης, τότε αυτό συνεπάγεται αυτόματα, ότι ο έλεγχος δεν επιστρέφεται στο νήμα του κεντρικού συστήματος, μέχρι να ολοκληρωθούν οι διεργασίες που έχουν ανατεθεί στην κάρτα γραφικών. Από τη στιγμή που θα επιστραφεί ο έλεγχος στο κεντρικό νήμα, αυτό μπορεί να λειτουργήσει με πολλούς τρόπους, όπως για παράδειγμα να κάνει spin. Αυτό καθορίζεται από τις τιμές των αντίστοιχων παραμέτρων τις οποίες μπορούμε να αρχικοποιήσουμε με την συνάρτηση που ακολουθεί:

- **cudaSetDeviceFlags (unsigned int flags)**

Παράμετροι:

flags – Παράμετροι που καθορίζουν τη λειτουργία της κάρτας γραφικών.

Καθορίζονται τα flags που θα χρησιμοποιηθούν, όταν αρχικοποιηθεί η επιλεγμένη κάρτα γραφικών. Αν δεν έχει επιλεγεί κάποια κάρτα, τότε τα flags θα αντιστοιχιστούν σε οποιαδήποτε κάρτα γραφικών αρχικοποιηθεί. Η μόνη περίπτωση να μην μπορέσουν να αντιστοιχιστούν αυτές οι παράμετροι σε μια κάρτα γραφικών είναι στην περίπτωση που αυτές οι παράμετροι έχουν ήδη οριστεί, είτε μέσω της ίδιας της κάρτας, είτε μέσω του νήματος στο κεντρικός σύστημα επεξεργασίας.

Αν μια κάρτα γραφικών έχει ήδη ενεργοποιηθεί και έχει αρχικοποιηθεί, τότε η συνάρτηση θα επιστρέψει σφάλμα cudaErrorSetOnActiveProcess.
Σε αυτήν την περίπτωση είναι απαραίτητο να επαναφέρουμε στην αρχική κατάσταση την κάρτα μέσω της συνάρτησης **cudaDeviceReset()** .

Τα δύο τελευταία ψηφία της παραμέτρου flag μπορούν να χρησιμοποιηθούν για να ορίσουν τον τρόπο με τον οποίο το κεντρικό νήμα εκτέλεσης αλληλεπιδρά με το λειτουργικό σύστημα, καθώς αναμένει για αποτελέσματα από την κάρτα.

Οι καθορισμένες τιμές για τα τελευταία δύο ψηφία της παραμέτρου είναι:

- **cudaDeviceScheduleAuto**: Είναι εξ ορισμού ενεργοποιημένο σε περίπτωση που η παράμετρος flag έχει την τιμή μηδέν. Σε αυτήν την περίπτωση χρησιμοποιείται ένας ευριστικός αλγόριθμος, ο οποίος βασίζεται στο πλήθος των ενεργών CUDA πλαισίων εκτέλεσης στη διεργασία C και το πλήθος των λογικών επεξεργαστών P στο σύστημα. Αν C > P τότε η CUDAθα ξεκινήσει να εκτελεί άλλα νήματα που προέρχονται από το λειτουργικό σύστημα μέχρις ότου ολοκληρωθεί η εκτέλεση των εντολών στην κάρτα. Σε διαφορετική περίπτωση, η CUDA απλώς θε περιμένει τα αποτελέσματα της εκτέλεσης της κάρτας γραφικών.

- **cudaDeviceScheduleSpin**: Δίνει οδηγία στον επεξεργαστή να λειτουργήσει εν αναμονή των αποτελεσμάτων εκτέλεσης της κάρτας γραφικών. Το γεγονός αυτό μειώνει την καθυστέρηση όσο διαρκούν οι υπολογισμοί στην κάρτα, αλλά ενδεχομένως μειώνει την απόδοση του κεντρικού συστήματος όταν εκτελεί παράλληλα νήματα με αυτά της CUDA.

- **cudaDeviceScheduleYield**: Δίνει οδηγία στην CUDA να σταματήσει την εκτέλεση του νήματός της για όσο χρόνο εκτελούνται υπολογισμοί στην κάρτα. Αυτό αυξάνει τον κενό χρόνο αναμονής του κεντρικού συστήματος, αλλά αυξάνει επίσης και την απόδοσή του εκτελώντας διεργασίες παράλληλα με τη λειτουργία της κάρτας.

- **cudaDeviceScheduleBlockingSync**: Χρησιμοποιείται αντί του cudaDeviceBlockingSync και δηλώνει στην CUDA να σταματήσει την εκτέλεση του νήματος του κεντρικού συστήματος σε ένα σημείο για συγχρονισμό, όσο το νήμα της κάρτας εκτελεί υπολογισμούς.

- **cudaDeviceMapHost**: Πρέπει να είναι ενεργοποιημένη προκειμένου να μπορεί να γίνει δέσμευση απευθείας μνήμης στο κεντρικό σύστημα. Διαφορετικά, η προσπάθεια κλήσης της συνάρτησης **cudaHostGetDevicePointer()** θα επιστρέψει σφάλμα.

- **cudaDeviceLmemResizeToMax**: Δίνει οδηγία να μην μειωθεί το μέγεθος της κοινόχρηστης μνήμης από τη στιγμή που θα αυξηθεί η κοινόχρηστη μνήμη που διατίθεται στους πυρήνες.

- **Διαλειτουργικότητα με άλλες πλατφόρμες προγραμματισμού γραφικών**

Ο κώδικας που παράγεται από άλλα προγραμματιστικά εργαλεία, όπως πχ. το OpenGL ή το Direct3D μπορεί να αποθηκευθεί στη μνήμη μιας συσκευής CUDA προκειμένου να χρησιμοποιηθεί από αυτήν και επίσης μπορεί να εκτελεστεί και η αντίστροφη λειτουργία, δηλαδή, δεδομένα που παράγονται από προγραμματισμό σε CUDA μπορούν να διαβαστούν από άλλες πλατφόρμες κώδικα.

Τα δεδομένα, που παράγονται από εντολές εκτέλεσης από άλλες πλατφόρμες πρέπει πρώτα να καταχωρηθούν ως δεδομένα της CUDA με τη βοήθεια συγκεκριμένων συναρτήσεων. Ως αποτέλεσμα αυτής της καταχώρησης επιστρέφεται ένας δείκτης που αντιστοιχίζεται σε ένα αντικείμενο τύπου **struct cudaGraphicsResource.**

Κάθε δεδομένο καταχωρείται με τον παραπάνω τρόπο μια φορά συνήθως, καθότι δημιουργείται μεγάλο overhead από την κλήση των συναρτήσεων δήλωσης. Για την ακριβώς αντίθετη διαδικασία, δηλαδή να αποπροσδιορίσουμε ένα δεδομένο χρησιμοποιούμε την παρακάτω συνάρτηση:

- **cudaGraphicsUnregisterResource (cudaGraphicsResource_t resource)**

Παράμετροι:

resource – Δεδομένο προς αποπροσδιορισμό.

Η συνάρτηση αυτή αποπροσδιορίζει ένα δεδομένο, ώστε να μην μπορεί να προσπελαστεί από την πλατφόρμα CUDA.

Σε περίπτωση σφάλματος επιστρέφεται η εξαίρεση cudaErrorInvalidResourceHandle.

Από τη στιγμή που κάποιο δεδομένο έχει τη δυνατότητα να καθοριστεί ως δεδομένο του περιβάλλοντος CUDA, μπορούμε να το αντιστοιχήσουμε ή να το απόαντιστοιχήσουμε όσες φορές θεωρήσουμε ότι το χρειαζόμαστε χρησιμοποιώντας τις συναρτήσεις (cudaGraphicsRegisterResource και cudaGraphicUnregisterResource).

- **cudaGraphicsMapResources (int count, cudaGraphicsResource_t * resources, cudaStream_t stream = 0)**

Παράμετροι :

count – Πλήθος δεδομένων προς αντιστοίχηση.
resources – Τα δεδομένα προς αντιστοίχηση.

Ορίζει μια αντιστοιχία των δεδομένων γραφικών και των κατάλληλα διαμορφωμένων δεδομένων για πρόσβαση από τις συναρτήσεις της CUDA. Τα δεδομένα που αντιστοιχίζονται στην πλατφόρμα CUDA δεν θα πρέπει να προσπελαύνονται από τις βιβλιοθήκες γραφικών, διαφορετικά προκύπτουν απροσδιόριστα αποτελέσματα.

Η συνάρτηση ορίζει ότι οποιαδήποτε κλήση συνάρτησης γραφικών πριν από την **cudaGraphicsMapResources()** θα ολοκληρωθεί πριν από οποιαδήποτε άλλη διεργασία ζητήσει να εκτελεστεί στο ρεύμα εντολών.

Αν τα δεδομένα περιέχουν διπλοεγγραφές, τότε επιστρέφεται μια εξαίρεση τύπου cudaErrorInvalidResourceHandle.

- **cudaGraphicsUnmapResources (int count, cudaGraphicsResource_t * resources, cudaStream_t stream = 0)**

Παράμετροι :

count – Πλήθος δεδομένων για αντιστοίχηση στις βιβλιοθήκες γραφικών.
resources – Τα δεδομένα προς αλλαγή της αντιστοίχησης.
stream – Ρεύμα συγχρονισμού.

Εκτελεί την ακριβώς αντίστροφη διαδικασία σε σχέση με την προηγούμενη συνάρτηση.

Από τη στιγμή που ένα δεδομένο αποαντιστοιχηθεί δεν μπορεί να προσπελαστεί από την CUDA, μέχρι να αντιστοιχιστεί εκ νέου.

Η συνάρτηση ορίζει, ότι οποιαδήποτε κλήση συνάρτησης γραφικών πριν από την **cudaGraphicsUnmapResources()** θα ολοκληρωθεί πριν από οποιαδήποτε άλλη διεργασία ζητήσει να εκτελεστεί στο ρεύμα εντολών.

Αν τα δεδομένα περιέχουν διπλοεγγραφές, τότε επιστρέφεται μια εξαίρεση τύπου cudaErrorInvalidResourceHandle.

Αν το δεδομένο δεν είναι αντιστοιχισμένο στην CUDA και όμως αιτείται η αποαναφορά του, τότε επιστρέφεται το σφάλμα cudaErrorUnknown.

Ο τρόπος χρήσης των δεδομένων αυτών (write-only/read-only) καθορίζεται από την συνάρτηση **cudaGraphicsResourceSetMapFlags().**

- **cudaGraphicsResourceSetMapFlags (cudaGraphicsResource_t resource, unsigned int flags)**

Παράμετροι:

resource – Δεδομένο, που έχει δηλωθεί και στο οποίο θα εφαρμοστούν τα flags.
flags – παράμετροι για την αντιστοίχηση.

Καθορίζει τα flags για την αντιστοίχηση των δεδομένων, ενώ οποιαδήποτε αλλαγή σε αυτές τις παραμέτρους, θα επηρεάσει και όλες τις επόμενες αντιστοιχίες δεδομένων.

Οι παράμετροι αυτές μπορούν να πάρουν μια από τις παρακάτω τιμές:

• **cudaGraphicsMapFlagsNone**: Δεν προσδιορίζει κάποιο τρόπο με τον οποίο θα χρησιμοποιηθεί το δεδομένο, οπότε θεωρείται ότι τα νήματα της CUDA μπορούν να γράφουν και να διαβάζουν σε αυτό.

• **cudaGraphicsMapFlagsReadOnly**: Προσδιορίζει οτι το νήμα της CUDA δεν μπορεί να γράψει στο δεδομένο προς αντιστοίχηση.

• **cudaGraphicsMapFlagsWriteDiscard**: Προσδιορίζει αφενός ότι η CUDA δεν μπορεί να αναγνώσει το συγκεκριμένο δεδομένο (resource) και αφετέρου, ότι οποιαδήποτε νέα τιμή εισαχθεί θα εγγραφεί πάνω στις παλαιότερες τιμές του δεδομένου.

Αν το δεδομένο έχει ήδη αντιστοιχιστεί στην CUDA τότε επιστρέφεται σφάλμα cudaErrorUnknown.

Αν η flag δεν έχει μια από τις παραπάνω τιμές τότε επιστρέφεται σφάλμα cudaErrorInvalidValue.

Η πρόσβαση σε αυτά τα δεδομένα γίνεται μέσω της διεύθυνσης στη μνήμη της συσκευής, που επιστρέφεται από τους δείκτες **cudaGraphicsResourceGetMappedPointer()** για τα δεδομένα τύπου buffer.

- **cudaGraphicsResourceGetMappedPointer (void ** devPtr, size_t * size, cudaGraphicsResource_t resource)**

Παράμετροι:

devPtr – επιστρεφόμενος δείκτης που αντιστοιχίζεται με τον buffer μέσω του οποίου γίνεται η προσπέλαση σε ένα δεδομένο.
size – μέγεθος του Buffer μέσω του οποίου γίνεται η προσπέλαση
resource – δεδομένο που αντιστοιχίζεται με τον buffer και στο οποίο επιθυμούμε να έχουμε πρόσβαση.

Η συνάρτηση επιστρέφει ένα δείκτη devPtr μέσω του οποίου γίνεται η προσπέλαση του δεδομένου που έχει αντιστοιχιστεί. Επιστρέφει στη μεταβλητή _size το μέγεθος της μνήμης σε Bytes στο οποίο έχει πρόσβαση ο δείκτης.

Η τιμή του δείκτη devPtr μπορεί να τροποποιηθεί κάθε φορά που γίνεται αντιστοίχηση του δεδομένου. Αν η resource δεν είναι Buffer, τότε επιστρέφεται σφάλμα cudaErrorUnknown και το ίδιο σφάλμα επιστρέφεται επίσης αν το δεδομένο δεν έχει αντιστοιχιστεί.

- **cudaGraphicsSubResourceGetMappedArray (struct cudaArray ** array, cudaGraphicsResource_t resource, unsigned int arrayIndex, unsigned int mipLevel)**

Παράμετροι :

array – Επιστρεφόμενος δείκτης τύπου πίνακα CUDA, μέσω του οποίου μπορεί να γίνει προσπέλαση σε ένα υποστοιχείο του δεδομένου.
resource – Το δεδομένο προς αντιστοίχηση.
arrayIndex – δείκτης πίνακα για πίνακες υφών ή δείκτης τύπου cubemap face όπως προσδιορίζεται από την cudaGraphicsCubeFace για υφές cubemap μέσω του οποίου έχει πρόσβαση το υποστοιχείο.
mipLevel - Mipmap level για την πρόσβαση του υποστοιχείου.

Επιστρέφει σε έναν δείκτη _array μέσω του οποίου μπορεί να προσπελαστεί το υποστοιχείο του δεδομένου που έχει αντιστοιχηθεί. Το δεδομένο που έχει αντιστοιχηθεί αντιστοιχίζεται με τον δείκτη του πίνακα και το mipLevel. Η τιμή του δείκτη array μπορεί να αλλάζει κάθε φορά που το resource αντιστοιχίζεται εκ νέου.

Αν το δεδομένο δεν είναι τύπου υφής, τότε δεν μπορεί να προσπελαστεί μέσω ενός πίνακα και επιστρέφεται σφάλμα cudaErrorUnknown. Αν ο arrayIndex δεν είναι έγκυρος δείκτης πίνακα για το δεδομένο, τότε επιστρέφεται σφάλμα cudaErrorInvalidValue. Αν επίσης, το mipLevel δεν είναι έγκυρο, τότε επιστρέφεται σφάλμα cudaErrorInvalidValue. Αν το

δεδομένο δεν έχει αντιστοιχιστεί τότε επιστρέφεται επίσης σφάλμα τύπου cudaErrorUnknown.

- **Διαχείριση σφαλμάτων**

Όπως έχει αναφερθεί και σε προηγούμενο κεφάλαιο, η διαχείριση των σφαλμάτων που προκύπτουν από τις κλήσεις ασύγχρονων συναρτήσεων είναι αρκετά επίπονη εργασία. Αυτό συμβαίνει, διότι αφενός οι συναρτήσεις αυτές επιστρέφουν λάθη σε μεταγενέστερα χρονικά σημεία του προγράμματος, είτε επιστρέφουν τα σφάλματα εγκαίρως, αλλά με κωδικούς, που δεν υποδηλώνουν το είδος του σφάλματος. Οσον αφορά τις ασύγχρονες συναρτήσεις ο τρόπος ελέγχου σφαλμάτων καθίσταται αρκετά ευκολότερος αν μετά από την κλήση μιας ασύγχρονης συνάρτησης καλέσουμε τη συνάρτηση **cudaThreadSynchronize().**

- **cudaThreadSynchronize (void)**

Η ιδιαιτερότητα αυτής της συνάρτησης συνίσταται στο ότι το όνομά της δεν αντανακλά τη λειτουργία της γι' αυτό και αντί αυτής συνιστάται η χρήση της συνάρτησης **cudaDeviceSynchronize.**

- **cudaDeviceSynchronize (void)**

Σταματάει την εκτέλεση οποιασδήποτε λειτουργίας στην κάρτα μέχρι να ολοκληρωθούν όλες τις προηγούμενες διεργασίες. Η συνάρτηση επιστρέφει σφάλμα αν κάποια από τις προηγούμενες διεργασίες δεν μπορέσει να ολοκληρωθεί.
Αν έχει καθοριστεί ο φορέας σήματος cudaDeviceScheduleBlockingSync σε set, τότε και το νήμα του συστήματος Host θα σταματήσει μέχρι να ολοκληρωθούν όλοι οι υπολογισμοί στην κάρτα και ελέγξουμε τον κωδικό σφάλματος που επιστρέφει η η παραπάνω συνάρτηση.
Για κάθε κεντρικό νήμα, το περιβάλλον εκτέλεσης διατηρεί μια μεταβλητή που διατηρεί τον κωδικό σφάλματος. Το περιβάλλον εκτέλεσης εξ ορισμού δίνει στη μεταβλητή αυτή μια τιμή, η οποία ισούται αρχικά με την τιμή cudaSuccess και η οποία στη συνέχεια αλλάζει ανάλογα με τον αν αργότερα προκύψει σφάλμα ή όχι. Η τιμή που μπορεί να πάρει εξαρτάται από το είδος του σφάλματος που λαμβάνει χώρα κάθε φορά (π.χ. σφάλμα που προέρχεται από την κλήση ασύγχρονης συνάρτησης ή από ακατάλληλες παραμέτρους, που περνώνται σε μια συνάρτηση). Η τιμή της μεταβλητής δίνεται μέσω της κλήσης της **cudaGetLastError()** η οποία δίνει ξανά στη μεταβλητή αρχική τιμή ίση με cudaSuccess.

- **cudaGetLastError (void)**

Επιστρέφει το πιο πρόσφατο σφάλμα που παρήχθηκε από κλήσεις στο περιβάλλον εκτέλεσης στο ίδιο νήμα του κεντρικού συστήματος και του θέτει τιμή ίση με cudaSuccess.

- **<u>Διαλειτουργικότητα μεταξύ του περιβάλλοντος εκτέλεσης και του Driver API της CUDA.</u>**

Μια εφαρμογή μπορεί να συνδυάζει κώδικα από το περιβάλλον εκτέλεσης της CUDA και από το Driver API και πιο συγκεκριμένα μπορεί να περιέχει συναρτήσεις ορισμένες από το Driver API, οι οποίες να χρησιμοποιούν βιβλιοθήκες που ορίζονται από το περιβάλλον εκτέλεσης (για παράδειγμα CUBLAS, CUFFT)

Οι συναρτήσεις, μέσω των οποίων γίνεται η διαχείριση της κάρτας γραφικών και των ιδιοτήτων της, μπορεί να ορίζονται είτε μέσω του runtime API, είτε μέσω του Driver API χωρίς προβλήματα.

Υπάρχουν και χαρακτηριστικά, τα οποία δεν μπορούν να υποστηριχθούν από κώδικα που ενσωματώνει συναρτήσεις από τις δύο διαφορετικές πλατφόρμες. Αυτά είναι:

→ Προσομοίωση συσκευής (device emulation).

→ Διαχείριση στοίβας που περιέχει τα πλαίσια εκτέλεσης (Context stack manipulation) με τις συναρτήσεις **cuCtxPopCurrent().**

- **cuCtxPopCurrent (CUcontext * pctx)**

Παράμετροι :

pctx – Επιστρέφει χειριστή για το καινούργιο πλαίσιο.

Εκτοπίζει το τρέχον πλαίσιο εκτέλεσης της CUDA από το κεντρικό σύστημα επεξεργασίας και το νήμα της CPU και επαναφέρει τον χειριστή του παλαιότερου. Το πλαίσιο που μόλις εκτοπίστηκε μπορεί να χρησιμοποιηθεί από διαφορετικό κεντρικό σύστημα (CPU νήμα) καλώντας τη συνάρτηση **cuCtxPushCurrent().**

Αν το πλαίσιο βρισκόταν σε τρέχουσα κατάσταση στο νήμα της CPU πριν από την κλήση της **cuCtxCreate(),** τότε αυτό μετατρέπεται σε τρέχον ξανά.

- **cuCtxPushCurrent (CUcontext ctx)**

Παράμετροι :

ctx – Πλαίσιο εκτέλεσης προς τοποθέτηση.

Τοποθετεί την παράμετρο ctx του context στη στοίβα νημάτων της CPU με τα διαθέσιμα contexts. Το συγκεκριμένο context γίνεται πλέον το τρέχον context στην CPU, επηρεάζοντας όλες τις συναρτήσεις CUDA, οι οποίες λειτουργούν με βάση το τρέχον context.
Μπορούμε να καλώντας τις συναρτήσεις **cuCtxDestroy()** ή **cuCtxPopCurrent().**

Κεφάλαιο 7

Τεχνικές βελτιστοποίησης στο περιβάλλον εκτέλεσης CUDA

Υποκεφάλαιο 7.1

ΕΙΣΑΓΩΓΗ

Από τη στιγμή που κάποιος κατανοήσει τα βασικότερα θέματα προγραμματισμού στην πλατφόρμα CUDA και συντάξει τους πρώτους αλγορίθμους, θα πρέπει στη συνέχεια να εμβαθύνει στις τεχνικές με τις οποίες μπορεί ένας αλγόριθμος να γίνει αποδοτικότερος.

Οι στρατηγικές βελτιστοποίησης κώδικα εστιάζουν σε τρία επίπεδα:

- Επίπεδο εφαρμογής, εφαρμόζοντας παραλληλία σε όσο το δυνατόν μεγαλύτερο βαθμό
- Επίπεδο κάρτας γραφικών, χρησιμοποιώντας τεχνικές για γρηγορότερη προσπέλαση στη μνήμη.
- Επίπεδο πολυεπεξεργαστών, κάνοντας αποτελεσματική χρήση των εντολών.

Σε επίπεδο εφαρμογής, θα πρέπει ο προς εκτέλεση αλγόριθμος να είναι δομημένος με κατάλληλο τρόπο, ώστε να είναι εύκολα παρατηρήσιμα τα τμήματα κώδικα που μπορούν να εκτελεστούν με παράλληλο τρόπο. Πέρα από την παραλληλοποίηση, πρέπει να εφαρμοστεί και η κατάλληλη αντιστοίχηση του κώδικα στο υλικό που θα τον εκτελέσει. Η εφαρμογή να διαθέτει ενδεχομένως και τμήματα κώδικα που μπορούν να εκτελούνται ταυτόχρονα στην κάρτα γραφικών, είτε τμήματα κώδικα που μπορούν να εκτελούνται ταυτόχρονα στη μεριά του host και της κάρτας.

Οι αποδοτικότερες προσπελάσεις μνήμης αναφέρονται σαφώς στη μείωση της μεταφοράς δεδομένων μεταξύ host και device συστήματος στο ελάχιστο. Επίσης, η μείωση προσπελάσεων στην καθολική μνήμη (με περισσότερη χρήση της κοινόχρηστης μνήμης) είναι ένα σημαντικό βήμα προς αυτήν την κατεύθυνση. Γενικά, όπου αντιλαμβανόμαστε ότι η καθυστέρηση επιβαρύνει κατά πολύ την απόδοση, ίσως θα πρέπει να σκεφτούμε και το ενδεχόμενο επανεκτέλεσης υπολογισμών, αντί της ανάκτησης από τη μνήμη.

Σε επίπεδο εντολών και όπου αυτό είναι δυνατό να χρησιμοποιήσουμε δεδομένα μικρότερης ακρίβειας, θα πρέπει να χρησιμοποιούνται οι γρήγορες εκδόσεις των συναρτήσεων.

Οι παραπάνω τεχνικές, αλλά και άλλες περισσότερες θα αναλυθούν παρακάτω σε κατηγορίες προτεραιότητας με βάση τη βελτίωση της απόδοσης, που μπορούμε να επιτύχουμε μέσω της χρήσης τους.

Η βελτίωση της ταχύτητας εκτέλεσης ενός αλγορίθμου εξαρτάται από τον ποσοστό παραλληλίας του, σύμφωνα με τον νόμο του Amdahl. Η προσπάθεια επομένως πρέπει να εστιαστεί στην παραλληλοποίηση όσο το δυνατόν μεγαλύτερου μέρους του αλγορίθμου, προκειμένου να μπορεί αυτός να εκτελεστεί στους πυρήνες μιας κάρτας γραφικών. Αν κάποια τμήματα του αλγορίθμου δεν μπορούν να παραλληλοποιηθούν επαρκώς, ώστε να εκτελεστούν στην κάρτα γραφικών, θα ήταν προτιμότερο να εκτελούνται αποκλειστικά στο σύστημα host.

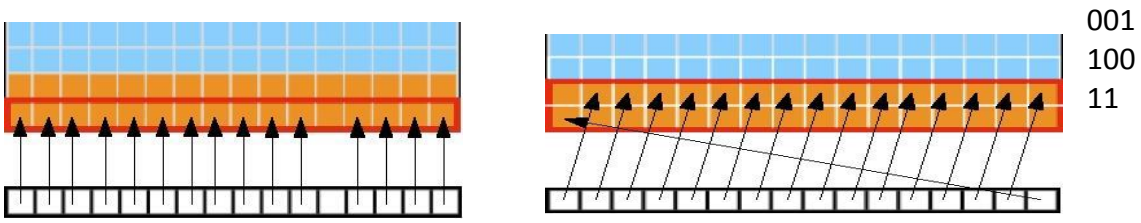
Υποκεφάλαιο 7.2

ΤΕΧΝΙΚΕΣ ΥΨΗΛΗΣ ΠΡΟΤΕΡΑΙΟΤΗΤΑΣ

- ***Παραλληλοποίηση του σειριακού κώδικα***: προσπαθούμε να διακρίνουμε τμήματα του αλγορίθμου τα οποία θα μπορούσαν να εκτελεστούν με παράλληλο τρόπο ή μετατρέπουμε τον αλγόριθμο έτσι, ώστε να εκθέσουμε τα τμήματα αυτά στα οποία μπορεί να εφαρμοστεί παράλληλη εκτέλεση.

- ***Ελαχιστοποίηση των μεταφορών δεδομένων μεταξύ host και device ακόμη και αν αυτό συνεπάγεται εκτέλεση πυρήνων στην κάρτα χωρίς αξιοσημείωτο όφελος:*** Το μέγιστο εύρος ζώνης για μεταφορά δεδομένων από την κάρτα γραφικών στους πυρήνες των επεξεργαστών της είναι πολύ μεγαλύτερο (141 Gbps στην GeForce GTX 280) από το εύρος επικοινωνίας μεταξύ του host και της κάρτας (8 Gbps μέσω του PCIe). Άρα είναι επιθυμητό να ελαχιστοποιήσουμε, όπου αυτό είναι δυνατό, τις ανταλλαγές δεδομένων μεταξύ της κάρτας και του συστήματος host, ακόμα και αν αυτό συνεπάγεται την εκτέλεση πυρήνων στην κάρτα χωρίς άμεσο όφελος. Για τις μεταφορές δεδομένων μεταξύ host και κάρτας μπορεί να χρησιμοποιηθεί και η καρφωτή μνήμη, που προσφέρει μεγαλύτερο εύρος ζώνης επικοινωνίας, με τρόπο που έχει αναλυθεί σε προηγούμενο κεφάλαιο.

- ***Οργάνωση και συνένωση των προσπελάσεων στην καθολική μνήμη:*** Η καθολική μνήμη της κάρτας πραγματοποιεί προσπελάσεις για εγγραφή και ανάγνωση δεδομένων με τη μορφή ενός παραθύρου που αντιστοιχεί σε ένα warp ή μισό warp (ανάλογα με την υπολογιστική δυνατότητα της κάρτας). Ο αριθμός των στοιχείων στα οποία γίνεται προσπέλαση είναι συγκεκριμένος και δομημένος και επομένως αν μπορέσουμε να τον εκμεταλλευτούμε συνενώνοντας πολλές προσπελάσεις σε μία κερδίζουμε άμεσα σε χρόνο και ταχύτητα εκτέλεσης.



Σχήμα 7.1: Προσπέλαση θέσεων μνήμης με και χωρίς συγκερασμό

- ***Μείωση των προσπελάσεων στην καθολική μνήμη όσο δυνατόν περισσότερο. Είναι προτιμότερο να γίνεται χρήση της κοινόχρηστης μνήμης:*** Οι προσπελάσεις της τοπικής ή της καθολικής μνήμης είναι αρκετά χρονοβόρες και συνήθως καταναλώνουν από 400 έως 600 κύκλους ρολογιού για να διαβάσουν ή να γράψουν δεδομένα. Το ακόλουθο παράδειγμα κώδικα επιδεικνύει μια ανάλογη προσπέλαση:

  →__shared__ float shared[32];
  →__device__ float device[32];
  →shared[threadIdx.x] = device[threadIdx.x];

  Η καθυστέρηση που επιβάλλει η μνήμη μπορεί γενικά να υπερκαλυφθεί αν καταφέρουμε να απασχολήσουμε τους πυρήνες τις κάρτας με την ταυτόχρονη εκτέλεση άλλων λειτουργιών (π.χ. αριθμητικούς υπολογισμούς) μέχρι να ολοκληρωθεί η προσπέλαση.

- ***Αποφυγή διακλαδώσεων εκτέλεσης στην εντολή που θα εκτελεστεί από ένα σμήνος (warp):*** Οι εντολές ελέγχου και επανάληψης (if, switch, do, while, for) οδηγούν πολλά νήματα μέσα σε ένα warp να ακολουθήσουν διαφορετικά μονοπάτια εκτέλεσης και καθιστούν την εκτέλεσή τους σειριακή. Αυτό έχει αντίκτυπο και στην γενικότερη απόδοση της κάρτας, λόγω του ότι δεσμεύονται νήματα για εκτέλεση σειριακών και όχι παράλληλων εντολών.

  Στις περιπτώσεις που η διαδρομή εκτέλεσης κάθε νήματος εξαρτάται από το

threadID, η συνθήκη ελέγχου θα πρέπει να συνταχθεί με τρόπο, ώστε να εξασφαλίζεται ότι θα διακλαδωθούν όσο το δυνατόν λιγότερα νήματα μέσα στο σμήνος. Για παράδειγμα θα μπορούσε να χρησιμοποιεί ως έλεγχος η συνθήκη (threadIdx / WSIZE), όπου WSIZE είναι το μέγεθος του warp. Σε αυτήν την περίπτωση δεν αποκλίνει κανένα νήμα, αφού η συνθήκη ελέγχου είναι απολύτως ευθυγραμμισμένη με το μέγεθος του σμήνους.

Υποκεφάλαιο 7.3

ΤΕΧΝΙΚΕΣ ΜΕΣΑΙΑΣ ΠΡΟΤΕΡΑΙΟΤΗΤΑΣ

- ***Οργάνωση της πρόσβασης στην κοινόχρηστη μνήμη με τέτοιο τρόπο, ώστε να μην υπάρχουν συγκρούσεις μεταξύ των banks και επομένως να μην υπάρχει ανάγκη για σειριακή εκτέλεση προσπελάσεων:*** Προκειμένου να μεγιστοποιηθεί η απόδοση των προσπελάσεων στην κοινόχρηστη μνήμη, αυτή χωρίζεται σε μικρότερα στοιχεία (banks) στα οποία μπορούν να έχουν ταυτόχρονη πρόσβαση τα νήματα.

Οταν παρ' όλα αυτά, προσπαθούν να έχουν πρόσβαση στην κοινόχρηστη μνήμη πολλαπλές διευθύνσεις, τότε έχουμε σύγκρουση. Για αυτόν το λόγο οι ταυτόχρονες προσπελάσεις στην κοινόχρηστη μνήμη διασπώνται σε επιμέρους προσπελάσεις μέσα από τις οποίες δεν προκύπτει σύγκρουση και έτσι οι προσπελάσεις αυτές εξυπηρετούνται πλέον σειριακά και όχι παράλληλα. Αυτό προκαλεί μείωση και στο συνολικό εύρος ζώνης επικοινωνίας κατά τόσες φορές όσες είναι και οι σειριακές προσπελάσεις που θα εκτελεστούν τελικώς.

- ***Χρήση της κοινόχρηστης μνήμης όποτε αντιλαμβανόμαστε ότι οι προσπελάσεις στην καθολική μνήμη φέρνουν αποτελέσματα, που δεν θα χρησιμοποιηθούν: (βλέπε Παράρτημα σελίδα 146).ΑΥΤΟ ΝΑ ΤΟ ΕΛΕΓΞΩ***

- ***Για να επικαλύπτεται ο κενός χρόνος που μπορεί να προκύψει από την πρόσβαση σε καταχωρητές, καλό είναι να διατηρείται η κάρτα απασχολημένη τουλάχιστον κατά ένα ποσοστό της τάξης του 25%:*** Σε περίπτωση που μια εντολή χρειαστεί για τους υπολογισμούς της ένα δεδομένο, που βρίσκεται

αποθηκευμένο σε έναν καταχωρητή, θα πρέπει να περιμένει. Η καθυστέρηση που εισάγεται σε αυτές τις περιπτώσεις είναι 24 κύκλοι, αλλά μπορούμε να εκμεταλλευτούμε αυτό το διάστημα αδράνειας αναθέτοντας στους πολυεπεξεργαστές περισσότερα νήματα για εκτέλεση. Για παράδειγμα στις κάρτες με υπολογιστική δυνατότητα 1.1 στους SM πρέπει να ανατίθενται τουλάχιστον 192 νήματα.

- ***Ο αριθμός των νημάτων ανά μπλοκ θα πρέπει να αποτελεί πολλαπλάσιο του 32, ώστε να είναι εύκολη η μαζική προσπέλαση μνήμης από τα νήματα και η αποτελεσματική διαχείρισή του***ς: Είναι πολύ σημαντική η επιλογή της διάστασης και του αριθμού των μπλοκ μέσα σε ένα πλέγμα, όπως επίσης και της διάστασης και του αριθμού των νημάτων μέσα σε ένα μπλοκ. Για παράδειγμα, αν φορτώσουμε σε μια κάρτα υπολογιστικής δυνατότητας 1.1 έναν πυρήνα με μέγιστο μέγεθος μπλοκ ίσο με 512 νήματα θα δεσμεύσουμε τον πολυεπεξεργαστή μόνο κατά ποσοστό 66%. Αυτό θα συμβεί διότι στη συγκεκριμένη κάρτα το μέγιστο δυνατό πλήθος νημάτων ανά SM φτάνει τα 768 νήματα, οπότε κάθε πολυεπεξεργαστής θα δεσμευτεί για ένα μόνο μπλοκ νημάτων. Αν όμως, φορτώναμε έναν πυρήνα με 256 νήματα, τότε θα απασχολούσαμε τον SM κατά ποσοστό 100% διότι κάθε SM θα διατηρούσε 2 μπλοκ νημάτων αντί του ενός.

- ***Στις περιπτώσεις, όπου η απώλεια σχετικής ακρίβειας των υπολογισμών δεν αποτελεί πρόβλημα, θα πρέπει να γίνεται χρήση των πιο γρήγορων και εξειδικευμένων εκδόσεων των μαθηματικών συναρτήσεων:*** Οι μαθηματικές συναρτήσεις του runtime περιβάλλοντος εκτέλεσης της CUDA χωρίζονται σε δύο κατηγορίες: τις συναρτήσεις με ονοματολογία της μορφής _functionName() και τις συναρτήσεις της μορφής functionName(). Οι συναρτήσεις της πρώτης κατηγορίας αντιστοιχίζονται απευθείας πάνω στο υλικό της κάρτας και έτσι εκτελούνται γρήγορα, αλλά με κάποιες απώλειες στην ακρίβεια των πράξεων. Η άλλη κατηγορία αφορά τις πιο αργές σε εκτέλεση συναρτήσεις, με μεγάλη ακρίβεια πράξεων.

- ***Στις δομές επανάληψης θα πρέπει να προτιμάται η χρήση signed και όχι των unsigned ακεραίων:*** Ο συμβολομεταφραστής μπορεί να εφαρμόσει ευκολότερα τεχνικές βελτιστοποίησης σε προσημασμένους (signed) ακεραίους από ό,τι σε μη προσημασμένους (unsigned). Αυτό συμβαίνει, διότι οι μη προσημασμένοι ακέραιοι, σε αντίθεση με τους προσημασμένους, καθορίζονται με σαφή τρόπο στο προγραμματιστικό περιβάλλον της γλώσσας C και δεν αφήνουν περιθώρια για βελτιώσεις. Στο παρακάτω παράδειγμα, θα αποτελούσε λογική λύση να δηλώσουμε το i ως unsigned ακέραιο, αφού αποτελεί μετρητή σε μια δομή επανάληψης και άρα αναμένουμε να είναι θετικός αριθμός σε κάθε περίπτωση.
-

→ for (i = 0; i < n; i++) {
out[i] = in[offset + stride*i];
}

Αν δηλωθεί όμως ως unsigned και η έκφραση stride * i προκαλέσει υπερχείλιση του 32 bit ακεραίου, ο compiler δεν θα μπορέσει να εφαρμόσει καμία τεχνική για να εξαλείψει το πρόβλημα. Αν αντίθετα ο i δηλωθεί ως signed, ο compiler έχει αυτή τη δυνατότητα.

Υποκεφάλαιο 7.4

ΤΕΧΝΙΚΕΣ ΧΑΜΗΛΗΣ ΠΡΟΤΕΡΑΙΟΤΗΤΑΣ

- ***Από την έκδοση 2.2 του CUDA Toolkit και έπειτα είναι δυνατή η χρήση zero-copy συναρτήσεων για κάρτες GPU που είναι ενσωματωμένες μέσα στο host σύστημα:*** Η λειτουργία zero-copy δίνει τη δυνατότητα στα νήματα να προσπελαύνουν άμεσα τη μνήμη του host, κάνοντας χρήση της καρφωτής μνήμης σύμφωνα με τον τρόπο που έχει ήδη αναλυθεί. Η λειτουργία zero-copy είναι ιδιαίτερα χρήσιμη σε περιπτώσεις, όπου αντικαθιστά τα ρεύματα μέσω των οποίων γίνεται η τυπική μεταφορά δεδομένων.

```
float *a_h, *a_map;
      ...
      cudaGetDeviceProperties(&prop, 0);
      if (!prop.canMapHostMemory)
      exit(0);


      cudaSetDeviceFlags(cudaDeviceMapHost);
      cudaHostAlloc((void **)&a_h, nBytes,
cudaHostAllocMapped);
      cudaHostGetDevicePointer((void **)&a_map, (void
*)a_h, 0);
      kernel<<<gridSize, blockSize>>>(a_map);
```

Κώδικας 7.4: Εντολές προσπέλασης μνήμης που έχει ανατεθεί απευθείας στο σύστημα host

Στον παραπάνω κώδικα, ο πυρήνας αναφέρεται στην καρφωτή μνήμη μέσω του δείκτη a_map.

- ***Για πυρήνες με λίστα από πολλούς παραμέτρους είναι καλό ορισμένες από αυτές να αποθηκεύονται στην μνήμη σταθερών για να μην δεσμεύεται μεγάλο μέρος της κοινόχρηστης μνήμης:*** Οι παράμετροι ή τα ορίσματα των πυρήνων αποθηκεύονται στην κοινόχρηστη μνήμη κατά την εκκίνησή τους. Για τους πυρήνες όμως, οι οποίοι διαθέτουν μια μεγάλη λίστα παραμέτρων θα ήταν προτιμότερο να αποθηκεύουμε τις παραμέτρους αυτές στη μνήμη των σταθερών.

- ***Αντί των λειτουργιών διαίρεσης και υπολοίπου μπορούν να χρησιμοποιηθούν λειτουργίες shift που είναι γρηγορότερες και απαιτούν λιγότερους υπολογιστικούς πόρους:*** Η διαίρεση ακεραίων και η συνάρτηση modulo για το υπόλοιπο ακέραιας διαίρεσης είναι δαπανηρές λειτουργίες από υπολογιστική άποψη. Όπου υπάρχει η δυνατότητα θα πρέπει είτε να αποφεύγονται, είτε να αντικαθιστώνται από bitwise συναρτήσεις. Για παράδειγμα, αν το n είναι μια δύναμη του 2, η πράξη (i/n) μπορεί να αντικατασταθεί από την πράξη (i >> log2(n)) ενώ η πράξη (i % n) αντιστοιχεί στην πράξη (i & (n-1)).

- ***Πρέπει να αποφεύγεται κατά το δυνατόν, η αυτόματη μετατροπή ακεραίων από double σε float:*** Γενικά, οι αυτόματες μετατροπές προσθέτουν κύκλους εντολών αυξάνοντας τον χρόνο εκτέλεσης οπότε καλό είναι να αποφεύγονται όταν αυτό είναι δυνατό. Όταν γνωρίζουμε, ότι θα χρειαστούμε αριθμούς κινητής υποδιαστολής μονής ακρίβειας, μπορούμε να τους δηλώσουμε απευθείας ως τέτοιους, προσθέτοντας στο τέλος το επίθεμα π.χ. 3.14159f, 1.0f.

- ***Πρέπει να αποφεύγονται κατά το δυνατόν οι δομές επανάληψης και οι εντολές ελέγχου στον προς εκτέλεση κώδικα. Αντί αυτών είναι προτιμότερο να γίνεται χρήση προεκτελούμενων εντολών:*** Για χάρη της βελτιστοποίησης του κώδικα είναι πολύ πιθανό ο συμβολομεταφραστής να “ξετυλίγει” τις δομές επανάληψης (το οποίο μπορεί να γίνει και από τον ίδιο τον προγραμματιστή με τη χρήση της #pragma unroll) ή να χρησιμοποιεί προεκτελούμενες εντολές αντί να εφαρμόζει τις εντολές ελέγχου διότι προκαλούν απόκλιση των νημάτων. Σε αυτήν την περίπτωση, κάθε εντολή ελέγχου αντιστοιχίζεται με ένα κωδικό κατάστασης που είναι μοναδικός για κάθε νήμα και παίρνει την τιμή true ή false ανάλογα με την αρχική συνθήκη ελέγχου. Κάθε εντολή ελέγχου προγραμματίζεται για εκτέλεση, αλλά μόνον οι εντολές με κωδικούς τιμής true εκτελούνται τελικά. Οι υπόλοιπες εντολές δεν έχουν τη δυνατότητα να γράψουν δεδομένα στη μνήμη ή να διαβάσουν διευθύνσεις και να αποτιμήσουν τελεστές πράξεων.

Υποκεφάλαιο 7.5

ΠΟΛΛΑΠΛΕΣ GPU

Ο προγραμματισμός εφαρμογών, που προορίζονται για εκτέλεση σε πολλαπλές κάρτες γραφικών, δεν αποτελεί κάτι το διαφορετικό σε σχέση με τον προγραμματισμό εφαρμογών για πολλαπλά νήματα ή sockets. Το πιο σημαντικό κομμάτι σε αυτήν την περίπτωση αφορά το ποια κάρτα γραφικών θα επιλέξουμε για την εκτέλεση του εκάστοτε νήματος. Συνήθως, η κάρτα που επιλέγεται για εκτέλεση είναι η κάρτα που δεν έχει συνδεθεί με κάποιο context και ως εκ τούτου δεν τρέχει ήδη κάποιο τμήμα κώδικα.

Για να μπορέσει να εκτελεστεί κάποιο τμήμα κώδικα σε μια κάρτα γραφικών θα πρέπει να δημιουργηθεί ένα πλαίσιο εντολών, το οποίο θα συνδέσει το νήμα του κεντρικού επεξεργαστή με την κάρτα γραφικών που θα αναλάβει την εκτέλεση. Μπορεί να δημιουργηθεί μόνο ένα τέτοιο πλαίσιο ανά νήμα κεντρικής επεξεργασίας το οποίο μπορεί επίσης να συνδέεται μόνο με μια κάρτα γραφικών τη φορά. Ένα τέτοιο πλαίσιο δημιουργείται κατά την πρώτη φορά που καλείται μια συνάρτηση αλλαγής κατάστασης, όπως για παράδειγμα μια συνάρτηση cudaMalloc() και καταστρέφεται μέσω της συνάρτησης cudaThreadExit() ή απλώς με τον τερματισμό της εκτέλεσης του αντίστοιχου νήματος κεντρικής επεξεργασίας.

Όσον αφορά το περιβάλλον CUDA, το CUDA Driver API επιτρέπει την πολλαπλή διαχείριση καρτών από ένα και μόνο κεντρικό νήμα, αλλά το CUDA runtime επιτρέπει σε κάθε νήμα του Host συστήματος να ελέγχει μόνο ένα context τη φορά. Επομένως, για να απασχολήσουμε n κάρτες γραφικών θα πρέπει απαραιτήτως να δημιουργήσουμε n host νήματα. Μια κάρτα γραφικών μπορεί να καλεί συναρτήσεις από ένα context κάθε φορά, αλλά μπορεί ταυτόχρονα να ανήκει σε πολλαπλά context, να συνδέεται δηλαδή με διαφορετικά host νήματα και να εκτελεί κώδικα από αυτά, όχι πάντως την ίδια χρονική στιγμή.

Ενότητα 7.5.1

ΕΠΙΛΟΓΗ ΚΑΡΤΑΣ ΓΡΑΦΙΚΩΝ

Η εξ ορισμού ανάθεση context από τη στιγμή της δημιουργίας του κι έπειτα εκτελείται στην κάρτα με αριθμό 0. Σε περίπτωση που η κάρτα αυτή δεν είναι διαθέσιμη ή δεν θέλουμε να ανατεθεί σε αυτήν η εκτέλεση του κώδικα θα πρέπει να το δηλώσουμε εμείς μέσω της συνάρτησης cudaSetDevice() η οποία υποδεικνύει ποια κάρτα θα αρχικοποιηθεί στη συνέχεια. Το παρακάτω κομμάτι κώδικα επιδεικνύει έναν τρόπο με τον οποίο μπορεί να γίνει επιλογή κάρτας γραφικών για την εκτέλεση.

```
int deviceCount;
cudaGetDeviceCount(&deviceCount);
int device;
for (device = 0; device < deviceCount; ++device) {
cudaDeviceProp deviceProp;

cudaGetDeviceProperties(&deviceProp, device);
if (dev == 0) {

       if (deviceProp.major == 9999 && deviceProp.minor == 9999)

              printf("There is no device supporting CUDA.\n");

       else if (deviceCount == 1)
              printf("There is 1 device supporting CUDA\n");

       else
              printf("There are %d devices supporting CUDA\n",
                     deviceCount);

       }
}
```

Από το CUDA 2.2 κι έπειτα υπάρχει η δυνατότητα μέσω λογισμικού σε linux πλατφόρμα να συνδέσουμε μια κάρτα με πολλαπλά νήματα host χωρίς να γίνει χρήση της συνάρτησης cudaSetDevice(). Στη συγκεκριμένη περίπτωση, ο διαχειριστής του συστήματος επιλέγει να χρησιμοποιήσει την λειτουργία exclusive μέσω του συστήματος SMI και στο οποίο αν η κάρτα με κωδικό αριθμό 0 είναι ήδη απασχολημένη με εκτέλεση κώδικα, αρχικοποιείται η αμέσως επόμενη διαθέσιμη κάρτα.

Ενότητα 7.5.2

ΕΠΙΚΟΙΝΩΝΙΑ ΝΗΜΑΤΩΝ

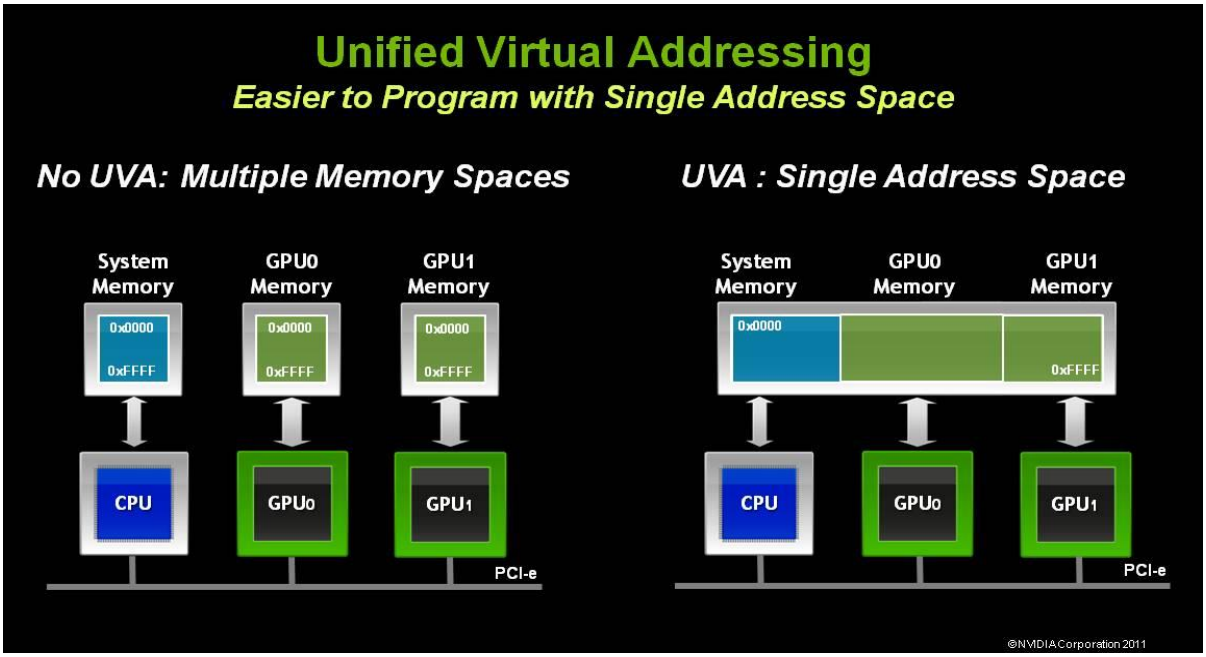
Το ζήτημα αυτό αφορά αφενός την επικοινωνία μεταξύ των διαφόρων νημάτων μέσα στο κεντρικό σύστημα επεξεργασίας και αφετέρου την ανταλλαγή δεδομένων μεταξύ της εκάστοτε κάρτας γραφικών και του αντίστοιχου κεντρικού νήματος. Τα δύο αυτά μέρη της επικοινωνίας είναι ανεξάρτητα μεταξύ τους και κάθε ένα από αυτά μπορεί να βελτιστοποιηθεί με διαφορετικό τρόπο.

Η επικοινωνία μεταξύ κάρτας και κεντρικού νήματος επιτυγχάνεται μέσω της ανταλλαγής δεδομένων από κοινή μνήμη, γι' αυτό και είναι κατά βάση προτιμότερη η δέσμευση και χρήση απευθείας μνήμης, που επιτρέπει την επικάλυψη της επικοινωνίας και της εκτέλεσης πυρήνων την ίδια χρονική στιγμή. Η επικοινωνία μεταξύ των διαφόρων κεντρικών νημάτων εκτελείται με τους συνήθεις τρόπους με τους οποίους επιτυγχάνεται και η ανταλλαγή δεδομένων μεταξύ των πολλαπλών προγραμμάτων που εκτελούνται στους πυρήνες μιας Multi-core CPU.

Σχετικά με τα ελαφροβαρή νήματα (lightweight threads), η επικοινωνία επιτυγχάνεται μέσω της κοινόχρηστης μνήμης. Δεδομένα από την κάρτα γραφικών αντιγράφονται σε μια κοινή περιοχή μνήμης του κεντρικού συστήματος, ώστε να μπορεί να έχει προσπέλαση σε αυτά οποιοδήποτε κεντρικό νήμα. Μια πολύ σημαντική παρατήρηση σε αυτό το σημείο αφορά τη δέσμευση απευθείας μνήμης στο κεντρικό σύστημα επεξεργασίας, την οποία για να μπορούν να την αντιληφθούν όλα τα νήματα ως απευθείας μνήμη θα πρέπει η δέσμευσή της να γίνει μέσω της συνάρτησης cudaHostAlloc() με ενεργοποιημένο το σηματοφόρο cudaHostAllocPortable.

Η ανταλλαγή δεδομένων για heavyweight νήματα γίνεται μέσω της ανταλλαγής μηνυμάτων, όπως για παράδειγμα την πλατφόρμα MPI. Συνήθως γίνεται η αντιγραφή δεδομένων από την κάρτα σε μια περιοχή μνήμης στο κεντρικό σύστημα και στη συνέχεια καλείται μία συνάρτηση του MPI για ανταλλαγή δεδομένων. Για παράδειγμα, για μια τέτοιου είδους επικοινωνία κοινό πρότυπο είναι τα εξής βήματα:

1. Το κεντρικό νήμα καλεί μια συνάρτηση cudaMemCpy() για αντιγραφή δεδομένων από την κάρτα στη μνήμη που προσπελαύνει το κεντρικό νήμα.
2. Στη συνέχεια το κεντρικό νήμα εκτελεί μια κλήση προς τη συνάρτηση MPI MPI_Sendrecv()
3. Τέλος, το κεντρικό νήμα που χρειάζεται τα δεδομένα σε δεύτερη φάση, καλεί μια συνάρτηση cudaMemCpy() για αντιγραφή δεδομένων από τη μνήμη του κυρίως συστήματος σε μια άλλη κάρτα γραφικών.



Σχήμα 7.5: Πολλαπλές κάρτες γραφικών με υλοποίση κοινής και κατανεμημένης μνήμης ανά κάρτα

Ενότητα 7.5.3

ΜΕΤΑΦΡΑΣΗ ΚΩΔΙΚΑ

Η συνήθης πρακτική στη μετάφραση κώδικα εφαρμογών για εκτέλεση σε πολλαπλές κάρτες γραφικών είναι η μετάφραση του κώδικα CUDA και του κώδικα που αφορά την επικοινωνία των κεντρικών νημάτων (πχ. MPI) σε διαφορετικά αρχεία. Ο κώδικας πυρήνων της CUDA, που εφαρμόζεται για το υπολογιστικό τμήμα της εφαρμογής, χρησιμοποιεί τον μεταφραστή του περιβάλλοντος nvcc (διά του οποίου καλείται είτε ο μεταφραστής gcc των Linux, είτε ο μεταφραστής του Microsoft Visual C++). Ο κώδικας του MPI, που εξασφαλίζει την επικοινωνία, χρησιμοποιεί τον μεταφραστή mpicc.

Υποκεφάλαιο 7.6

ΜΕΤΡΙΚΕΣ ΑΠΟΔΟΣΗΣ

Όταν επιθυμούμε να βελτιώσουμε την απόδοση του κώδικά μας, θα πρέπει να είμαστε σε θέση να γνωρίζουμε με ποιον τρόπο μια διαφορετική υλοποίηση μπορεί να επηρεάσει την ταχύτητα εκτέλεσης ενός αλγορίθμου. Η απόδοση ενός αλγορίθμου μπορεί να μετρηθεί και να δώσει μια γενική ιδέα, σχετικά με το ποια υλοποίηση αποδίδει καλύτερα από την άλλη και κάτω από ποια κριτήρια συμβαίνει αυτό.

Κριτήρια καθορισμού της απόδοσης μιας συγκεκριμένης υλοποίησης αλγορίθμου αποτελούν οι εξής παράγοντες :

- Ο **χρόνος** που απαιτεί η εκτέλεση μιας εφαρμογής, δηλαδή τον χρόνο που μεσολαβεί μεταξύ των κλήσεων των συναρτήσεων της πλατφόρμας CUDA και της επιστροφής του ελέγχου στο κεντρικό νήμα εκτέλεσης. Το χρονικό διάστημα αυτό μπορεί να μετρηθεί με τη βοήθεια ειδικών χρονιστών, που υλοποιούνται είτε από τη μεριά του κεντρικού συστήματος (**CPU χρονιστές)** είτε από τη μεριά της κάρτας γραφικών (**GPU χρονιστές**).
- Το **εύρος ζώνης** που αποτελεί τον ρυθμό με τον οποίο μεταφέρονται τα δεδομένα. Το εύρος ζώνης επηρεάζεται άμεσα από το είδος της μνήμης που χρησιμοποιείται για την αποθήκευση δεδομένων, το πώς αυτά οργανώνονται μέσα στη μνήμη αποθήκευσης, καθώς επίσης και τη σειρά προσπέλασής τους.

Ενότητα 7.6.1

ΧΡΟΝΙΣΤΕΣ

Οι κλήσεις στο περιβάλλον CUDA και οι πυρήνες μπορούν να χρονομετρηθούν με τη χρήση είτε χρονιστών στο κεντρικό σύστημα, είτε στην κάρτα γραφικών.

- **Μετρητές CPU:** Όταν χρησιμοποιούνται οι χρονιστές του κεντρικού συστήματος θα πρέπει να ελέγχουμε, ότι όλα τα νήματα της κάρτας γραφικών έχουν τελειώσει την εκτέλεσή τους πριν επιστραφεί ο έλεγχος στο κεντρικό σύστημα, ώστε να γίνει η μέτρηση του χρόνου εκτέλεσης. Τα νήματα του κεντρικού συστήματος και της

κάρτας γραφικών πρέπει να συγχρονιστούν μέσω της συνάρτησης cudaThreadSynchronize() πριν από την εκκίνηση ή τον τερματισμό του μετρητή.

- **Μετρητές GPU:** Το περιβάλλον CUDA παρέχει κατάλληλες συναρτήσεις με τις οποίες μπορεί να γίνει η δημιουργία ενός γεγονότος, ο τερματισμός του και η καταγραφή του χρονικού διαστήματος μεταξύ της δημιουργίας και του τερματισμού του.

```
cudaEvent_t start, stop;
float time;

cudaEventCreate(&start);
cudaEventCreate(&stop);

cudaEventRecord( start, 0 );
kernel<<<grid,threads>>> ( d_odata, d_idata, size_x, size_y,
NUM_REPS);
cudaEventRecord( stop, 0 );
cudaEventSynchronize( stop );

cudaEventElapsedTime( &time, start, stop );
cudaEventDestroy( start );
cudaEventDestroy( stop );
```

Κώδικας 7.6: Εντολές για καταγραφή και χρονομέτρηση εκτέλεσης εντολών

Ενότητα 7.6.2

ΕΥΡΟΣ ΖΩΝΗΣ

Το εύρος ζώνης ως έννοια μπορεί να αναφέρεται είτε στο

- **θεωρητικό εύρος ζώνης** είτε στο
- **πραγματικό εύρος ζώνης.**

1. Το θεωρητικό εύρος ζώνης μπορεί να υπολογιστεί με βάση τα χαρακτηριστικά του υλικού της κάρτας γραφικών όπως δηλώνονται από την κατασκευάστρια εταιρεία. Για παράδειγμα, αν πάρουμε την κάρτα γραφικών Nvidia GeForce GTX 280 της οποίας τα χαρακτηριστικά είναι:

   - DDR Ram στα 1,107 MHz

   - 512 bit για εύρος ζώνης μνήμης

   το θεωρητικό εύρος ζώνης βρίσκεται ως εξής: $(1107 \times 10^6 \times (512 / 8) \times 2) / 10^9 = 141.6$ GB/s.

2. Το πραγματικό εύρος ζώνης υπολογίζεται από τη μέτρηση της διάρκειας εκτέλεσης συγκεκριμένων λειτουργιών του αλγορίθμου και την εκτίμηση του τρόπου προσπέλασης των δεδομένων στη μνήμη και υπολογίζεται με βάση τον τύπο που δίνεται παρακάτω:

   Πραγματικό εύρος ζώνης = $(B_r + B_w) / 10^9) /$ time,

   όπου:

   - B_r είναι ο αριθμός των bytes δεδομένων που προσπελαύνει κάθε πυρήνας εκτέλεσης για ανάγνωση

   - B_w είναι ο αριθμός των bytes δεδομένων που προσπελαύνει κάθε πυρήνας για εγγραφή.

   - το όρισμα time δίνεται μετρημένο σε δευτερόλεπτα.

Κεφάλαιο 8

Συμπεράσματα

Υποκεφάλαιο 8.1

ΓΕΝΙΚΟ ΣΥΜΠΕΡΑΣΜΑ

Είναι γεγονός, ότι ο παράλληλος προγραμματισμός αποτελεί γενικά έναν τομέα συνεχώς εξελισσόμενο, και το μοντέλο CUDA ειδικότερα περιλαμβάνει ένα πολύ μεγάλο αριθμό από συναρτήσεις, οι οποίες σε καμία περίπτωση δεν θα μπορούσαν να αναλυθούν πλήρως στο πλαίσιο μιας πτυχιακής εργασίας. Στόχος της παρούσας πτυχιακής ήταν η περιγραφή της πλατφόρμας CUDA και του κεντρικού ρόλου που διαδραματίζει στον προγραμματισμό των καρτών γραφικών. Παρουσιάστηκαν με λεπτομέρεια οι βασικές έννοιες του μοντέλου αυτού, η αρχιτεκτονική των συστημάτων που παρέχουν υποστήριξη για CUDA, καθώς και αρκετές από τις συναρτήσεις προγραμματισμού, δεδομένου ότι το μοντέλο CUDA διαθέτει ένα τεράστιο αριθμό διαθέσιμων συναρτήσεων. Μέσα από την ενασχόλησή μου με την πτυχιακή αυτή, κατάφερα να αποσαφηνίσω τις βασικές έννοιες της παράλληλης υπολογιστικής, καθώς και να εντοπίσω τη θέση που κατέχει το μοντέλο προγραμματισμού CUDA έναντι άλλων μοντέλων παρόμοιας λειτουργικότητας. Η κεντρική ιδέα που αποκόμισα από την εκπόνηση αυτής της πτυχιακής, αφορά το ότι παρά την πολυπλοκότητα, που αρχικά φαίνεται να παρουσιάζει η συγκεκριμένη πλατφόρμα, εντούτοις είναι πολύ εύκολα εφαρμόσιμη. Αυτό γίνεται εφικτό αν ξεκινήσει κανείς από την κατανόηση των βασικών εννοιών και στη συνέχεια υλοποιήσει βήμα-βήμα τις συναρτήσεις κλιμακούμενης δυσκολίας, σύμφωνα πάντα με τις απαιτήσεις και τις ανάγκες της εκάστοτε εφαρμογής.

Υποκεφάλαιο 8.2

ΕΦΑΡΜΟΓΕΣ ΤΗΣ CUDA

- ✓ **Ιατρικές εφαρμογές:** Η πλατφόρμα CUDA αποτελεί σημαντικό βοήθημα στην έγκαιρη διάγνωση του καρκίνου του μαστού, μέσω εξελιγμένων τεχνικών απεικόνισης με υπερήχους. Μια από τις τεχνικές αυτές χρησιμοποιεί τους υπερήχους για να κατασκευάσει ένα αρχείο με την εικόνα που παρήχθηκε από τους υπερηχους. Η CUDA χρησιμοποιείται για τη διαχείριση των αρχείων αυτών, τα οποία αποθηκεύουν δεδομένα μεγέθους 35GB και αφού εκτελέσει τους απαραίτητους αλγορίθμους να παράγει μια πλήρη εικόνα σχετικά με την κατάσταση του ασθενή, ώστε να γίνει η διάγνωση και η θεραπεία του προβλήματος.

- ✓ **Δυναμική Ρευστών:** Τα μαθηματικά μοντέλα που περιγράφουν τη ροή του αέρα και την κίνηση των ρευστών είναι εξαιρετικά πολύπλοκα. Οι υπολογισμοί αυτοί μέχρι πρότινος εκτελούνταν από υπερυπολογιστές στους οποίους όμως η πρόσβαση είναι περιορισμένη και δαπανηρή. Η αξιοποίηση της υπολογιστικής δυνατότητας πολλαπλών καρτών με τη χρήση του μοντέλου CUDA έδωσε τη δυνατότητα για την εκτέλεση υπολογισμών (που άλλοτε ήταν απαγορευτικά πολύπλοκοι) καθώς και την πειραματική υλοποίηση άλλων αλγορίθμων, δημιουργώντας ομάδες (clusters) από υπολογιστές χαμηλού κόστους, αλλά με πολύ μεγάλες υπολογιστικές δυνατότητες.

- ✓ **Βιομηχανία και Περιβαλλοντικές Επιστήμες:** Τα παντός είδους καθαριστικά είναι προϊόντα ιδιαίτερα επιβλαβή προϊόντα για το περιβάλλον. Οι καθαριστικοί παράγοντες (επιφανειοδραστικά μόρια) αποτελούν το κύριο συστατικό των καθαριστικών ουσιών, αλλά δυστυχώς είναι ιδιαίτερα επιβλαβείς για το περιβάλλον. Γίνεται προσπάθεια εξισορρόπησης της αποτελεσματικότητας των επιφανειοδραστικών ουσιών και του αντίκτυπου που έχουν στα οικοσυστήματα. Προς αυτήν την κατεύθυνση, το πανεπιστήμιο Temple σε συνεργασία με την εταιρεία Procter & Gamble χρησιμοποιούν 2 Nvidia Tesla GPUs με υλοποίηση CUDA για να προσομοιώσουν την αντίδραση των επιφανειοδραστικών ουσιών με υλικά, κάτω από διαφορετικές συνθήκες. Η απόδοση που κατάφεραν να επιτύχουν είναι αντίστοιχη με αυτήν των 128 πυρήνων ενός Cray XT3 και των 1024 επεξεργαστών μιας IBM Blue Gene/L μηχανής.

Υποκεφάλαιο 8.3

ΜΕΛΛΟΝΤΙΚΗ ΕΡΓΑΣΙΑ

Θεωρώ, ότι η συγκεκριμένη πτυχιακή εργασία μπορεί να αποτελέσει ένα χρήσιμο εφόδιο για περαιτέρω ενασχόληση με θέματα προγραμματισμού σε κάρτες γραφικών υποστήριξης CUDA. Οι κάρτες γραφικών της Nvidia από το 2006 και έπειτα είναι συμβατές με την πλατφόρμα CUDA (CUDA-enabled) και το υλικό τους παρέχει υποστήριξη για την εκτέλεση του CUDA API, είτε σε βασικό επίπεδο για τις παλαιότερες κάρτες, είτε σε ανώτατο επίπεδο για τις πιο πρόσφατες κάρτες γραφικών. Θα ήταν πολύ χρήσιμο -κατά την προσωπική μου άποψη- να γίνει και μια περαιτέρω προσπάθεια συγγραφής και μελέτης αλγορίθμων με την υποστήριξη καρτών γραφικών σε περιβάλλον CUDA.

Λίστα Βιβλιογραφικών Αναφορών

Διαμαντάρας, Κ. (2012), Προηγμένες Αρχιτεκτονικές Η/Υ και Παράλληλα Συστήματα.

Παπαδάκης Στ. και Διαμαντάρας Κ., Προγραμματισμός και Αρχιτεκτονική Συστημάτων Παράλληλης Επεξεργασίας.

Farber, Rob (2011), CUDA Application Design and Development.

Kirk, David B. & Hwu Wen-Mei W. (2010), Προγραμματισμός Μαζικά Παράλληλων Επεξεργαστών.

Sanders Jason and Kandrot Edward, Foreword by Dongarra J., CUDA BY EXAMPLE: An introduction to General-Purpose GPU Programming

NVIDIA_CUDA_Programming_Guide_3.0.

NVIDIA_CUDA_C_Best_Practices Guide_3.1.

NVIDIA_openCL_Jumpstart_Guide_Technical_Brief.

CUDA_Toolkit_Reference_Manual_Version 4.1, (2012).

NVIDIA_Compute_PTX_ISA Version 1.4, (2009).

CUDA Ανασύρθηκε από : http://en.wikipedia.org/wiki/CUDA.

DirectCompute Lecture Series 101 (2010), Introduction to DirectCompute Ανασύρθηκε από: http://channel9.msdn.com/tags/DirectCompute-Lecture-Series/.

OpenCL Ανασύρθηκε από: http://en.wikipedia.org/wiki/OpenCL.

Trent R., (2009), University of Utah Computer Engineering, Cache Organization and Memory Management of the Intel Nehalem Computer Architecture, Ανασύρθηκε από: http://rolfed.com/nehalem/nehalemPaper.pdf

Schauer B., (n. d.), Multicore Processors - A necessity, Ανασύρθηκε από: http://www.csa.com/discoveryguides/multicore/review.pdf

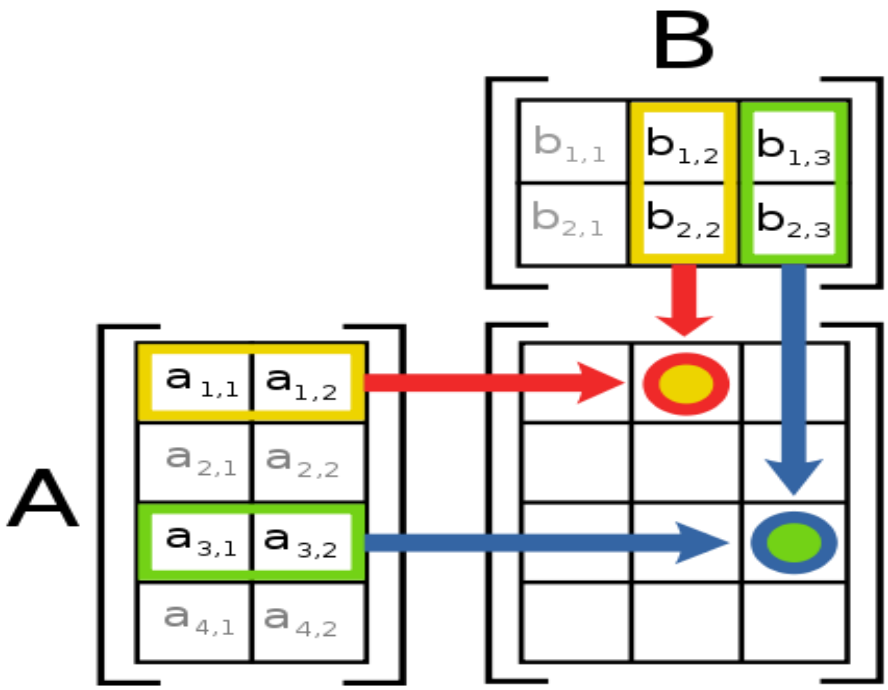
Παράρτημα

1. Κώδικας πολλαπλασιασμού πινάκων χωρίς τη χρήση κοινόχρηστης μνήμης

Στον παρακάτω κώδικα γίνεται χρήση της καθολικής μνήμης, καθώς κάθε νήμα διαβάζει μια γραμμή του πίνακα A και μια στήλη του πίνακα B, εκτελεί τον μεταξύ τους πολλαπλασιασμό και με αυτόν τον τρόπο υπολογίζει το αντίστοιχο στοιχείο του πίνακα C.

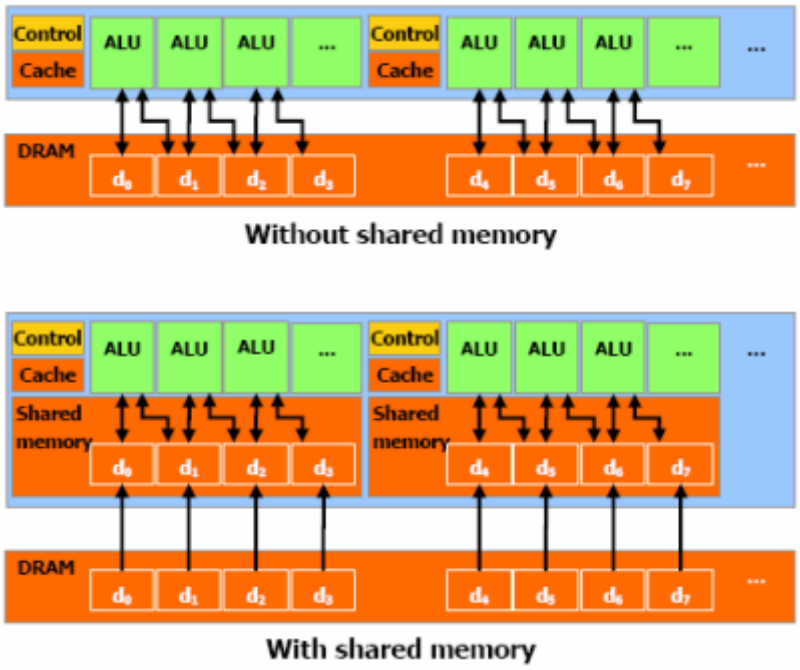
```
1.     typedef struct {
2.     int width;
3.     int height;
4.     float* elements;

5. } Matrix;
6.     #define BLOCK_SIZE 16

7.     __global__void MatMulKernel (const Matrix, const Matrix, Matrix);

8.     void MatMul (const Matrix A, const Matrix B, Matrix C) {

9.     Matrix d_A;
10.    d_A.width = A.width; d_A.height = A.height;
11.    size_t size = A.width * A.height * sizeof (float);
12.    cudaMalloc((void**)&d_A.elements, size);
13.    cudaMemcpy (d_A.elements, A.elements, size, cudaMemcpyHostToDevice);

14.    Matrix d_B;
15.    d_B.width = B.width; d_B.height = B.height;
16.    size = B.width * B.height * sizeof (float);
```

```
17.    cudaMalloc((void**)&d_B.elements, size);
18.    cudaMemcpy (d_B.elements, A.elements, size, cudaMemcpyHostToDevice);
19.    Matrix d_C;
20.    d_C.width = C.width; d_C.height= C.height;
21.    size = C.width * C.height * sizeof (float);
22.    cudaMalloc((void**)&d_C.elements, size);

23.    dim3 dimBlock (BLOCK_SIZE, BLOCK_SIZE);
24.    dim3 dimGrid ( B.width / dimBlock.x, A.height / dimBlock.y );
25     MatMulKernel<<<dimGrid,dimBlock>>> (d_A, d_B, d_C);

26.    cudaMemcpy(C.elements, d_C.elements, size, cudaMemcpyDeviceToHost);

27.    cudaFree(d_A.elements);
28.    cudaFree(d_B.elements);
29.    cudaFree(d_C.elements);

30.    }

31.    __global__void MatMulKernel (Matrix A, Matrix B, Matrix C) {

32.    float Cvalue = 0;
33.    int row = blockIdx.y * blockDim.y + threadIdx.y;
34.    int col = blockIdx.x * blockDim.x + threadIdx.x;
35.    for (int e = 0; e< A.width; ++e)
36.           Cvalue += A.elements[row * A.width + e]
37.                          * B.elements[e * B.width + col];
38.    C.elements[row * C.width + col] = Cvalue;
39.    }
```



Παράρτημα – Σχήμα 1: Πολλαπλασιασμός πινάκων

3. Κώδικας με τη χρήση κοινόχρηστης μνήμης



Παράρτημα – Σχήμα 2: Πολλαπλασιασμός πινάκων με χρήση καθολικής και κοινόχρηστης μνήμης

Στο παρακάτω κώδικα ο πολλαπλασιασμός των πινάκων A και B γίνεται με τη χρήση της κοινόχρηστης μνήμης, που είναι σαφώς πιο γρήγορη από την καθολική μνήμη. Κάθε μπλοκ νημάτων υπολογίζει έναν υποπίνακα του C και κάθε νήμα μέσα στο μπλοκ υπολογίζει από ένα στοιχείο αυτού του υποπίνακα.

```
1.     typedef struct {
2.             int width;
3.             int height;
4.             int stride;
```

```
5.          float* elements;
6.    } Matrix;

7.    __device__float GetElement (const Matrix A, int row, int col) {
8.    return A.elements [row + A.stride + col];
9.    }

10.   __device__float SetElement (Matrix A, int row, int col, float value) {
11.         A.elements[row * A.stride + col ] = value;
12.   }

13.   __device__Matrix GetSubMatrix(Matrix A, int row, int col){
14.         Matrix Asub;
15.         Asub.width = BLOCK_SIZE;
16.         Asub.height = BLOCK_SIZE;
17.         Asub.stride = A.stride;
18.         Asub.elements = &A.elements [ A.stride * BLOCK_SIZE * row + BLOCK_SIZE
      * col];

19.   return Asub;
20.   }

21.   #define BLOCK_SIZE 16

22.   __global__void MatMulKernel(const Matrix, const Matrix, Matrix);

23.   void MatMul(const Matrix A, const Matrix B, Matrix C) {

24.         Matrix d_A;
25.         d_A.width = d_A.stride = A.width;
26.         d_A.height = A.height;
27.         size_t size = A.width * A.height * sizeof (float);
28.         cudaMalloc((void**) &d_A.elements, size);
```

```
29.          cudaMemcpy(d_A.elements, A.elements, size, cudaMemcpyHostToDevice);


30.          Matrix d_B;
31.          d_B.width = d_B.stride = B.width;
32.          d_B.height = B.height;
33.          size = B.width * B.height * sizeof (float);
34.          cudaMalloc((void**) &d_B.elements, size);
35.          cudaMemcpy(d_B.elements, b.elements, size, cudaMemcpyHostToDevice);

36.          Matrix d_C;
37.          d_C.width = d_C.stride = C.width;
38.          d_C.height = C.height;
39.          size = C.width * C.height * sizeof (float);
40.          cudaMalloc((void**) &d_C.elements, size);


41.          dim3 dimBlock (BLOCK_SIZE, BLOCK_SIZE);
42.          dim3 dimGrid (B.width / dimBlock.x, A.height / dimBlock.y );
43.          MatMulKernel<<<dimGrid, dimBlock>>>(d_A, d_B, d_C);

44.          cudaMemcpy(C.elements, d_C.elements, size, cudaMemcpyDeviceToHost);
45.          cudaFree(d_A.elements);
46.          cudaFree(d_B.elements);
47.          cudaFree(d_C.elements);
48.    }

50.    __global__void MatMulKernel (Matrix A, Matrix B, Matrix C) {

51.          int blockRow = blockIdx.y;
52.          int blockCol = blockIdx.x;
```

```
53.          Matrix Csub = GetSubMAtrix (C, blockRow, blockCol);

54.          float Cvalue = 0;

55.          int row = threadIdx.y;
56.          int col = threadIdx.x;

57.    for(int m = 0; m <(A.width / BLOCK_SIZE); ++m) {

58.          Matrix Asub = GetSubMatrix(A, blockRow, m);
59.          Matrix Bsub = GetSubMatrix (B, m, blockCol);

60.          __shared__float As[BLOCK_SIZE][BLOCK_SIZE];
61.          __shared__float Bs[BLOCK_SIZE][BLOCK_SIZE];

62.          As[row][col] = GetElement(Asub, row, col);
63.          Bs[row][col] = GetElement(Bsub, row, col);

64.          __syncthreads();

65.          for(int e = 0; e < BLOCK_SIZE; e++)
66.                Cvalue += As[row][e] * Bs[e][col];

67.          __syncthreads();


68.    }

69.    SetElement(Csub, row, col, Cvalue);

70. }
```