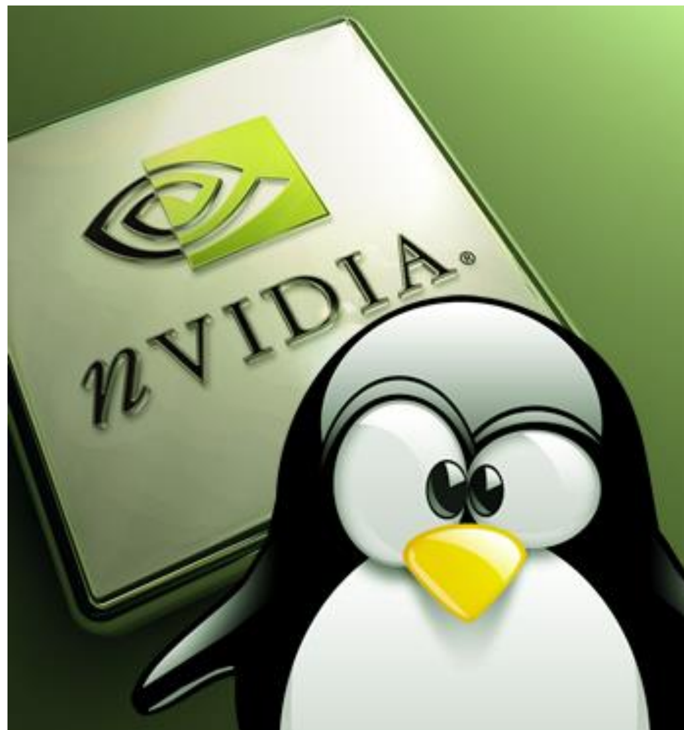
ΑΛΕΞΑΝΔΡΕΙΟ Τ.Ε.Ι. ΘΕΣΣΑΛΟΝΙΚΗΣ ΣΧΟΛΗ ΤΕΧΝΟΛΟΓΙΚΩΝ ΕΦΑΡΜΟΓΩΝ
ΤΜΗΜΑ ΠΛΗΡΟΦΟΡΙΚΗΣ

Τμήμα Μηχανικών
Πληροφορικής ΑΤΕΙΘ

ΠΤΥΧΙΑΚΗ ΕΡΓΑΣΙΑ

ΠΑΡΑΛΛΗΛΟΣ ΠΡΟΓΡΑΜΜΑΤΙΣΜΟΣ ΕΠΙΣΤΗΜΟΝΙΚΩΝ ΕΦΑΡΜΟΓΩΝ ΜΕ ΧΡΗΣΗ GPU / CUDA



Της φοιτήτριας
ΛΑΔΟΠΟΥΛΟΥ ΠΑΡΑΣΚΕΥΗΣ
Αρ. Μητρώου: 06/3042

Επιβλέπων καθηγητής
ΔΙΑΜΑΝΤΑΡΑΣ ΚΩΝΣΤΑΝΤΙΝΟΣ

Θεσσαλονίκη 2014

ΠΡΟΛΟΓΟΣ

Η εκπόνηση της πτυχιακής μου εργασίας αποτέλεσε μια ενδιαφέρουσα αφορμή να ασχοληθώ με τον παράλληλο προγραμματισμό και να εμβαθύνω σε θέματα προσαρμογής του και σε άλλους τομείς όπως είναι η μηχανολογία μιας και ο συνδυασμός των δύο έχει μεγάλο πρακτικό ενδιαφέρον τα τελευταία χρόνια.

Συχνά στον τομέα της μηχανολογίας υπάρχουν προβλήματα τα οποία απαιτούν υπολογιστική δύναμη προκειμένου να επιλυθούν. Έτσι λοιπόν αναπτύσσονται κώδικες οι οποίοι προσομοιώνουν το πρόβλημα αλλά χρειάζονται μεγάλο υπολογιστικό κόστος για να τρέξουν και πολλές φορές και μεγάλα χρονικά διαστήματα για να συγκλίνουν και να δώσουν ένα αποτέλεσμα. Η εφαρμογή του παράλληλου προγραμματισμού σε τέτοια θέματα έχει φέρει την επανάσταση στην κατηγορία των προσομοιώσεων καθώς μειώνει αρκετά και το υπολογιστικό κόστος αλλά και το χρόνο εκτέλεσης.

ΠΕΡΙΛΗΨΗ

Στην παρούσα πτυχιακή εργασία θα γίνει μία αναφορά στην εξέλιξη των προβλημάτων που σχετίζονται με μηχανολογικούς τομείς καθώς επίσης και στο πώς αυτά κατέληξαν να επιδέχονται επεξεργασία και επίλυση μέσω του προγραμματισμού και μάλιστα του παράλληλου. Ένα απλό πρόβλημα, που επιλύει το ρυθμό ροής ενός ρευστού που διέρχεται μέσα από έναν αγωγό απείρου μήκους, θα βοηθήσει στην κατανόηση της εφαρμογής του. Συγκεκριμένα θα αναλυθεί η πλατφόρμα παράλληλου προγραμματισμού CUDA της NVIDIA καθώς και η γλώσσα προγραμματισμού CUDA με σκοπό την προσαρμογή της σε ένα συγκεκριμένο και απλοποιημένο μηχανολογικό πρόβλημα. Επίσης θα παρουσιαστεί ένας αναλυτικός οδηγός εγκατάστασης του πακέτου CUDA μαζί με το Toolkit και το SDK του. Ακόμη θα εξεταστεί το σπουδαίο εργαλείο ανάπτυξης εφαρμογών Nsight Eclipse καθώς και ο Debugger αυτού που είναι απαραίτητος κατά τη διαδικασία συγγραφής κώδικα. Έπειτα θα γίνει αναλυτική επεξήγηση του προβλήματος, της μεταφοράς του σε απλό επαναληπτικό κώδικα γραμμένο σε γλώσσα C++ καθώς επίσης και ο τρόπος σκέψης μετασχηματισμού του κώδικα σε παράλληλο κώδικα με απώτερο σκοπό την υλοποίηση του και την εξαγωγή σωστού αποτελέσματος. Τέλος θα αναφερθούν προβλήματα που ενδεχομένως υπήρξαν σε αυτόν τον μετασχηματισμό, θα συγκριθούν τα αποτελέσματα με τη βιβλιογραφία, θα απεικονιστούν διαγράμματα συγκρίσεων για τους χρόνους εκτέλεσης τόσο για το χρονικό διάστημα που απαιτείται για να τρέξει ο απλός κώδικας σε σχέση με τον παράλληλο, όσο και για τους χρόνους εκτέλεσης με διαφορετικές παραμέτρους στην ίδια κάρτα γραφικών αλλά και σε διαφορετικές με διαφορετικές δυνατότητες.

ΕΥΧΑΡΙΣΤΙΕΣ

Πρώτα απ’ όλα, θέλω να ευχαριστήσω τον επιβλέποντα καθηγητή της πτυχιακής μου εργασίας, κ. Κωνσταντίνο Διαμαντάρα, για την πολύτιμη βοήθεια κατά τη διάρκεια της δουλειάς μου. Ακόμη θα ήθελα να εκφράσω την ευγνωμοσύνη μου στα υπόλοιπα μέλη της εξεταστικής μου επιτροπής για την προσεκτική ανάγνωση της εργασίας μου και για τις πολύτιμες υποδείξεις τους. Επίσης, ευχαριστώ ιδιαίτερα τον κ. Χρήστο Τάντο, αρχικά για την ιδέα αυτού του σπουδαίου θέματος προς εκπόνηση πτυχιακής εργασίας που μου πρότεινε και έπειτα για την πολύτιμη βοήθειά του, την κατανόησή του, την προσωπική του συμμετοχή στην επεξήγηση του μηχανολογικού τμήματος της εργασίας μου και την ακατάπαυστη υπομονή του καθ’ όλη τη διάρκεια της προσπάθειάς μου. Τέλος, είμαι ευγνώμων στους γονείς μου, Αντώνιο και Σοφία Λαδοπούλου καθώς και στην αδερφή μου Γεωργία για την ολόψυχη αγάπη και υποστήριξή τους όλα αυτά τα χρόνια. Αφιερώνω αυτήν την πτυχιακή εργασία στη μνήμη του παππού μου Κούκουνα Νικολάου.

Λαδοπούλου Παρασκευή

ΠΕΡΙΕΧΟΜΕΝΑ

ΠΡΟΛΟΓΟΣ 2
ΠΕΡΙΛΗΨΗ 3
ΕΥΧΑΡΙΣΤΙΕΣ 4
ΠΕΡΙΕΧΟΜΕΝΑ 5
Ευρετήριο Σχημάτων 7
Ευρετήριο Πινάκων 7
Ευρετήριο Εικόνων 7
ΕΙΣΑΓΩΓΗ 9
Κεφάλαιο 1 - Μηχανολογία και παράλληλος προγραμματισμός 10
1.1 Εισαγωγή 11
1.2 Φυσικά προβλήματα και παράλληλος προγραμματισμός 14
1.3 Περιγραφή του προβλήματος 15
Κεφάλαιο 2 - Η GPU και η ιστορική εξέλιξη της 18
2.1 Η CPU και η εξέλιξή της 19
2.2 Η GPU και η εξέλιξή της 19
2.3 Η ιστορική εξέλιξη της GPU 20
2.4 Η αρχιτεκτονική μιας GPU 24
Κεφάλαιο 3 - CUDA (Compute Unified Device Architecture) 25
3.1 Η αρχιτεκτονική CUDA 26
3.2 Παραλληλία δεδομένων 26
3.2.1 Παράλληλες αρχιτεκτονικές κατά Flynn 27
3.3. Η δομή και η εκτέλεση των προγραμμάτων της CUDA 28
3.3.1 Τα νήματα της CUDA 29
3.3.2 Δημιουργία πλέγματος (Εκκίνηση πυρήνα) 31
3.3.3 Μνήμες CUDA 33
Η Καθολική μνήμη (Global Memory) 34
Η Μνήμη σταθερών (Constant Memory) 35
Η μνήμη υφής (Texture Memory) 35
Καταχωρητές (Registers) 35
Τοπική μνήμη (Local Memory) 36
Κοινόχρηστη μνήμη (Shared memory) 36
3.3.4 Δήλωση μεταβλητών CUDA 36
3.4 Το προγραμματιστικό μοντέλο και το API της CUDA 39
3.5 CUDA Occupancy Calculator 42
3.5.1 Καταχωρητές ανά νήμα και κοινόχρηστη μνήμη ανά μπλοκ 44
3.5.2 Οδηγίες χρήσης του CUDA Occupancy Calculator 45
3.5.3 Μερικές σημειώσεις για το CUDA Occupancy Calculator 48
Κεφάλαιο 4 - Εγκατάσταση CUDA 5.5 σε Linux Ubuntu LTS 12.04 και χρήση του Nsight Eclipse 49
4.1 Εισαγωγή 50
4.2 Τα απαραίτητα για την εγκατάσταση 50
4.3 Εκκίνηση της εγκατάστασης 50
4.3.1 Εγκατάσταση του Driver 51
4.3.2 Εγκατάσταση του Toolkit (compiler, libraries) 54
4.3.3 Εγκατάσταση των SDK Samples 57
4.3.4 Τεχνολογία Optimus και πρόγραμμα Bumblebee 61

4.3.5 Εγκατάσταση Bumblebee 62
4.4 Χρήση του Nsight Eclipse 64
4.4.1 Εγκατάσταση Nsight Eclipse 64
4.4.2 Εκκίνηση του Nsight Eclipse 65
4.4.3 Κάνοντας Debugging 69
Κεφάλαιο 5 - Επεξήγηση σειριακού κώδικα και παραλληλοποίηση 72
5.1 Επεξήγηση χαρακτηριστικών του προβλήματος 73
5.2 Επίλυση του προβλήματος με τον σειριακό κώδικα 74
5.3 Παραλληλοποίηση του κώδικα 80
Κεφάλαιο 6 - Συμπεράσματα και μελλοντική εργασία 86
6.1 Αποτελέσματα 87
6.2 Μελλοντική έρευνα 93
ΑΝΑΦΟΡΕΣ 94
ΠΗΓΕΣ INTERNET 95
ΠΑΡΑΡΤΗΜΑ Α 96
Α. Επεξήγηση του κώδικα 101
ΠΑΡΑΡΤΗΜΑ Β 103
Β. Επεξήγηση του κώδικα 110

Ευρετήριο Σχημάτων

Σχήμα 1.1: Γεωμετρικά χαρακτηριστικά του καναλιού 15
Σχήμα 1.2: Διαμόρφωση ροής 16
Σχήμα 3.1: Βασική λειτουργία της CUDA 39
Σχήμα 5.1: Χαρακτηριστικά του αγωγού 73
Σχήμα 5.2: Διάγραμμα ροής παράλληλου κώδικα....15

Ευρετήριο Πινάκων

Πίνακας 3.1: Προσδιοριστικά τύπου για μεταβλητές CUDA....37
Πίνακας 6.1: Αδιάστατες παροχές μάζας G για διάφορες τιμές της παραμέτρου αραιοποίησης δ 87

Ευρετήριο Εικόνων

Εικόνα 2.1: Διαφορά μεταξύ CPU και GPU....20
Εικόνα 2.2: Tesla Αρχιτεκτονική....22
Εικόνα 2.3: Ανανεωμένη αρχιτεκτονική Fermi 23
Εικόνα 2.4: Αρχιτεκτονική Kepler 23
Εικόνα 3.1:Αρχιτεκτονικές κατά Flynn 27
Εικόνα 3.2: Εκτέλεση δύο πλεγμάτων από νήματα....29
Εικόνα 3.3: Οργάνωση νημάτων 30
Εικόνα 3.4: Πιο ρεαλιστική άποψη οργάνωσης των νημάτων 30
Εικόνα 3.5: Παράδειγμα πλέγματος πολλών διαστάσεων στην CUDA....32
Εικόνα 3.6: Νήματα ενός μπλοκ οργανωμένα σε στημόνια 33
Εικόνα 3.7: Γενική άποψη μνημών μέσα σε μια συσκευή GPU 34
Εικόνα 3.8: Έξοδος ετικέτας –ptxas-options=-v 44
Εικόνα 3.9: Το αρχείο CUDA_Occupancy_Calculator στο φάκελο 45
Εικόνα 3.10: Ενεργοποίηση των Macros του Excel 45
Εικόνα 3.11: Επιλογή δυναμικότητας της GPU 46
Εικόνα 3.12: Επιλογή μεγέθους της κοινόχρηστης μνήμης σε bytes 46
Εικόνα 3.13: Εισαγωγή παραμέτρων....46
Εικόνα 3.14: Αποτελέσματα πληρότητας....46
Εικόνα 3 15: Πληρότητα για το μέγεθος του μπλοκ....47
Εικόνα 3.16: Αριθμός των καταχωρητών ανά πυρήνα 47
Εικόνα 3.17: Κοινόχρηστη μνήμη που χρησιμοποίησε ο πυρήνας 48
Εικόνα 4.1: Εικονίδιο που ίσως εμφανιστεί....51
Εικόνα 4.2: Διαθέσιμοι Drivers 52
Εικόνα 4 3: Ενεργοποίηση Universe και Multiverse repositories 63
Εικόνα 4.4: Επιλογή φακέλου εργασίας 65
Εικόνα 4.5: Αρχικό παράθυρο Nsight Eclipse....66
Εικόνα 4.6: Επιλογή σετ εργαλείων....67
Εικόνα 4.7: Εντοπισμός CUDA υλικού τοπικά....68
Εικόνα 4.8: Κατασκευή του κώδικα 68

Εικόνα 4.9: Η κονσόλα 69
Εικόνα 4.10: Εκτέλεση του κώδικα 69
Εικόνα 4.11: Αποτελέσματα στην κονσόλα 69
Εικόνα 4.12: Κουμπί για Debugging 69
Εικόνα 4.13: Μετατροπή της οπτικής των παραθύρων 70
Εικόνα 4.14: Διακοπή στη main 70
Εικόνα 4.15: Τοποθέτηση Breakpoint 70
Εικόνα 4.16: Συνέχεια εφαρμογής 71
Εικόνα 5 1: Υπολογισμός πίνακα Υ με σειριακή επεξεργασία 79
Εικόνα 5 2: Υπολογισμός πίνακα $u(N)$ με σειριακή επεξεργασία 79
Εικόνα 5 3: Υπολογισμός πίνακα $Y(x,\mu)$ με παράλληλη επεξεργασία 81
Εικόνα 6.1: Κατανομή της αδιάστατης μακροσκοπικής ταχύτητας του αερίου κατά μήκος των δύο πλακών με χρήση του σειριακού κώδικα (S) και του παράλληλου (C). 88
Εικόνα 6.2: Τεχνικά χαρακτηριστικά GeForce 8600M GT 89
Εικόνα 6.3: Τεχνικά χαρακτηριστικά GeForce 9600 GT 90
Εικόνα 6.4: Τεχνικά χαρακτηριστικά GeForce GT 640M 91
Εικόνα 6.5: Σύγκριση υπολογιστικών χρόνων μεταξύ διαφορετικών καρτών γραφικών για $\delta=10$. 92
Εικόνα 6.6: Σύγκριση υπολογιστικών χρόνων μεταξύ διαφορετικών καρτών γραφικών για $\delta=100$. 92
Εικόνα 6.7: Σύγκριση υπολογιστικών χρόνων μεταξύ διαφορετικών καρτών γραφικών για $\delta=1000$. 93

ΕΙΣΑΓΩΓΗ

Στόχος της συγκεκριμένης πτυχιακής εργασίας είναι να γίνει αρχικά μια καλή μελέτη όσο αφορά την προγραμματιστική πλατφόρμα παράλληλου προγραμματισμού CUDA της NVIDIA ξεκινώντας από την εγκατάσταση του πλήρους πακέτου μέχρι και την υλοποίηση κώδικα ο οποίος είναι σε θέση να τρέξει και να δώσει αποτελέσματα. Επίσης είναι σκόπιμο να αποδειχθεί πως ο προγραμματισμός και ακόμα περισσότερο ο παράλληλος προγραμματισμός είναι σε θέση να λύσει και να εξηγήσει πολλά προβλήματα που μπορεί κανείς να συναντήσει στον φυσικό κόσμο. Όλα αυτά θα προσπαθήσουμε να τα δείξουμε μέσα από τη χρήση ενός απλοποιημένου μηχανολογικού προβλήματος το οποίο μελετάει το ρυθμό ροής ενός ρευστού δια μέσου ενός αγωγού μεγάλου μήκους.

Στο πρώτο κεφάλαιο θα γίνει μια εισαγωγή σε φυσικές έννοιες καθώς και μια σύντομη ιστορική αναδρομή για το πώς έχουμε καταλήξει σήμερα να αντιμετωπίζουμε πολλά από τα προβλήματα του φυσικού κόσμου σε τομείς όπως είναι αυτός της μηχανολογίας και θα παρουσιάσουμε και το δικό μας ενδεικτικό πρόβλημα. Έπειτα στο δεύτερο κεφάλαιο θα δούμε μια ιστορική αναδρομή για την εξέλιξη της κάρτας γραφικών και το πώς αυτή έφτασε να είναι το δεύτερο αλλά και αποτελεσματικότερο κομμάτι του υπολογιστή όταν πρόκειται για βαριές υπολογιστικές διαδικασίες. Αυτό όμως δεν γίνεται πάντα αλλά σε περιπτώσεις που η κάρτα γραφικών είναι σε θέση να προγραμματιστεί και να της ανατεθεί τέτοια δουλειά. Τέτοιες κάρτες είναι οι κάρτες γραφικών της NVIDIA οι οποίες επιδέχονται εκτέλεση προγραμμάτων μέσω της γλώσσας προγραμματισμού CUDA η οποία είναι μία επέκταση των γλωσσών C/C++. Αυτή η γλώσσα λειτουργεί πάνω σε μια πλατφόρμα η εγκατάσταση της οποίας αναλύεται στο τέταρτο κεφάλαιο μαζί και με ένα σημαντικό εργαλείο ανάπτυξης εφαρμογών το οποίο ονομάζεται Nsight Eclipse. Στο πέμπτο κεφάλαιο αφού έχουμε δει μεμονωμένα πως λειτουργούν όλα αυτά επιλύουμε το μηχανολογικό πρόβλημα με CUDA και επεξηγούμε τις διαφορές του κώδικα επίλυσης που είναι γραμμένος σε CUDA με τον σειριακό κώδικα. Στο έκτο κεφάλαιο παραθέτουμε συμπεράσματα και συγκρίσεις καθώς και μελλοντικούς στόχους και προτάσεις για μελέτη και τέλος στα παραρτήματα μπορεί κανείς να βρει τόσο τον απλό όσο και τον παράλληλο κώδικα που υλοποιήθηκε.

Κεφάλαιο 1

Μηχανολογία και παράλληλος προγραμματισμός

1.1 Εισαγωγή

1.2 Φυσικά προβλήματα και παράλληλος προγραμματισμός

1.3 Περιγραφή του προβλήματος

1.1 Εισαγωγή

Υπάρχουν τρεις βασικές καταστάσεις της ύλης: η στερεά, η υγρή και η αέρια. Στερεά θεωρούνται τα υλικά στα οποία τα μόρια κρατούνται σε σταθερές θέσεις μεταξύ τους στο χώρο. Υγρά θεωρούνται τα υλικά στα οποία τα μόρια είναι κοντά μεταξύ τους αλλά όχι σε σταθερές θέσεις. Αέρια είναι τα υλικά στα οποία τα μόρια βρίσκονται σε σχετικά μεγάλη απόσταση μεταξύ τους και η θέση τους δεν επηρεάζεται από τις δυνάμεις αλληλεπίδρασης των μορίων. Τα σωματίδια (άτομα, μόρια, ιόντα) που συγκροτούν τα αέρια είναι πολύ αραιά κατανεμημένα στο χώρο, σε σχέση με εκείνα των υγρών, πολύ δε περισσότερο των στερεών. Οι μεταξύ τους ελκτικές δυνάμεις είναι πολύ ασθενείς ώστε να συγκρατούν αυτά σε καθορισμένες θέσεις, με αποτέλεσμα να κινούνται αυτά προς διάφορες κατευθύνσεις.

Οι εσωτερικές αραιοποιημένες ροές των μονοατομικών αερίων συναντώνται σε πολλούς τομείς της φυσικής και της μηχανικής και η μελέτη τους είναι ιδιαίτερα ενδιαφέρουσα. Ο βαθμός αραιοποίησης ενός αερίου εκφράζεται με μία παράμετρο δ η οποία ονομάζεται *παράμετρος αραιοποίησης*. Όσο μεγαλύτερο είναι το δ τόσο πιο πυκνό είναι το αέριο. Όταν το αέριο κινείται λοιπόν για να πραγματοποιηθεί ροή, σαφώς τα μόρια συγκρούονται μεταξύ τους. Κάθε μόριο λοιπόν πριν συγκρουστεί με κάποιο άλλο έχει μία απόσταση που μπορεί να διανύσει. Αυτή η απόσταση αναφέρεται ως *η μέση ελεύθερη διαδρομή των μορίων* και συμβολίζεται με λ. Όταν το λ είναι πολύ μικρότερο από την χαρακτηριστική διάσταση της ροής $L\left(\lambda << L\right)$ θεωρούμε ότι για το αέριο ισχύει η ιδιότητα του *συνεχούς μέσου*. Αν υποθέσουμε λοιπόν ότι ένα ρευστό κινείται δια μέσου ενός καναλιού με μήκος L τότε ο λόγος της ελεύθερης διαδρομής των μορίων προς το αντίστοιχο μήκος του καναλιού εκφράζουν έναν αριθμό που ονομάζεται *Knudsen* $\left(K_n\right)$ και είναι αντίστροφος του δ που εξηγήσαμε παραπάνω.

$$Kn = \frac{\lambda}{L} \sim \frac{1}{\delta} \tag{1.1}$$

Με την κίνηση των αερίων ρευστών ασχολήθηκαν οι Claude-Louis Navier και George Gabriel Stokes [1]. Αυτοί μελέτησαν *μακροσκοπικά* την κίνηση των μορίων μέσα στο αέριο και διατύπωσαν κάποιες εξισώσεις οι οποίες ονομάστηκαν εξισώσεις Navier – Stokes. Οι εξισώσεις αυτές είναι χρήσιμες καθώς περιγράφουν τη φυσική σημασία πολλών πραγμάτων που παρουσιάζουν ακαδημαϊκό και οικονομικό ενδιαφέρον. Μπορούν να χρησιμοποιηθούν για να μοντελοποιηθούν οι καιρικές

συνθήκες, τα ωκεάνια ρεύματα, η ροή του νερού σε ένα σωλήνα και η ροή του αέρα γύρω από ένα φτερό. Οι εξισώσεις Navier – Stokes στην πλήρη και απλοποιημένη μορφή τους βοήθησαν με το σχεδιασμό των αεροσκαφών και των αυτοκινήτων, τη μελέτη της ροής του αίματος, το σχεδιασμό των σταθμών παραγωγής ηλεκτρικής ενέργειας, την ανάλυση της ρύπανσης και πολλά άλλα πράγματα.

Αυτές οι εξισώσεις όμως έχουν το μειονέκτημα ότι μπορούν να εφαρμοστούν μόνο σε περιπτώσεις που το αέριο μπορεί να προσεγγίσει την έννοια του συνεχούς μέσου και αυτό συμβαίνει όταν $K_n \leq 0.1$, διαφορετικά τα αποτελέσματα που θα πάρουμε εφαρμόζοντας τες, θα είναι αρκετά μακριά από την πραγματικότητα καθώς η θεωρία του συνεχούς μέσου καταρρέει. Όταν αυτό συμβαίνει βρίσκει εφαρμογή η εξίσωση του Boltzmann.

Ο Ludwig Eduard Boltzmann ήταν ένας Αυστριακός φυσικός και φιλόσοφος, του οποίου το μεγαλύτερο επίτευγμα ήταν η ανάπτυξη της στατιστικής μηχανικής, γεγονός που εξηγεί και προβλέπει πως οι ιδιότητες των ατόμων (όπως η μάζα, το φορτίο, και η δομή) καθορίζουν και τις φυσικές ιδιότητες της ύλης (όπως το ιξώδες, τη θερμική αγωγιμότητα, και τη διάχυση). Αυτός αντίθετα με τους Navier – Stokes μελέτησε την κίνηση των ρευστών *μικροσκοπικά* και χρησιμοποιώντας τη θεωρία των πιθανοτήτων παρουσίασε μια νέα εξίσωση στην προσπάθειά του να εξηγήσει την κίνηση των μορίων μέσα στο ρευστό [2, 3]. Η εξίσωση αυτή δεν είναι εύκολο να επιλυθεί *αναλυτικά* κάνοντας χρήση κλειστών μαθηματικών εκφράσεων. Μπορεί όμως να προσεγγιστεί *υπολογιστικά*, κάτι το οποίο σημαίνει πως χρειάζονται αφ' ενός εξιδανικευμένες περιπτώσεις και αφ' εταίρου μεγάλη υπολογιστική προσπάθεια. Για το σκοπό αυτό λοιπόν κατά διαστήματα έχουν προταθεί διάφορα *κινητικά μοντέλα* τα οποία βασίζονται πάνω στην κινητική θεωρία και στόχο έχουν να απλοποιήσουν την εξίσωση του Boltzmann η οποία δίνεται από την ακόλουθη εξίσωση:

$$\frac{\partial f(\boldsymbol{r},\boldsymbol{\xi},t)}{\partial t} + \xi_i \frac{\partial f(\boldsymbol{r},\boldsymbol{\xi},t)}{\partial r} + F_i \frac{\partial f(\boldsymbol{r},\boldsymbol{\xi},t)}{\partial \xi} = \iiint \left(f' f_*' - f\ f_* \right) g b db d\varepsilon d\boldsymbol{\xi} \tag{1.2}$$

όπου $f(\boldsymbol{r},\boldsymbol{\xi},t)$ είναι η άγνωστη συνάρτηση κατανομής, η οποία περιγράφει την αριθμητική πυκνότητα των σωματιδίων του αερίου με διάνυσμα θέσης $\boldsymbol{r}$ και ταχύτητα $\boldsymbol{\xi}$ τη χρονική στιγμή t. Η ποσότητα g εκφράζει τη σχετική ταχύτητα των μορίων με $g = |\xi - \xi_*|$ ενώ η ποσότητα b ονομάζεται παράμετρος σύγκρουσης και εξαρτάται από το

είδος του ενδομοριακού δυναμικού. Η παράμετρος ε εκφράζει τη στερεά γωνία σύγκρουσης των δύο σωματιδίων. Το αριστερό μέρος της Εξ. (1.2) αντιπροσωπεύει την ελεύθερη κίνηση των σωματιδίων χωρίς την ύπαρξη συγκρούσεων κάτω από την επίδραση του πιθανού πεδίου δυνάμεων $\boldsymbol{F}$. Το δεξί μέλος της Εξ. (1.2) αντιπροσωπεύει τον όρο των συγκρούσεων και αποτελεί μια γραμμική επίλυση της εξίσωσης Boltzmann μη εφικτή έως και στις μέρες μας.

Ένα από τα γνωστά κινητικά μοντέλα είναι το μοντέλο BGK (Bhatnagar-Gross-Krook) [4] το οποίο μέχρι και σήμερα χρησιμοποιείται με αρκετή αξιοπιστία για ροές ενός συστατικού και το οποίο θα μας απασχολήσει στη συνέχεια της πτυχιακής εργασίας. Πάνω στο μοντέλο BGK βασίστηκαν αργότερα και τα μετέπειτα κινητικά μοντέλα όπως τα μοντέλα Shakhov, McCormack, ES κ.ά. Για την επίλυση αυτών των μοντέλων κινητικής θεωρίας χρειάζεται να κάνουμε χρήση αριθμητικών μεθόδων. Μία από τις αριθμητικές μεθόδους που έχει εφαρμοστεί είναι η Μέθοδος Διακριτών Ταχυτήτων (Discrete Velocity Method – DVM) [5]. Εναλλακτικά με τα κινητικά μοντέλα κάποιος μπορεί να εφαρμόσει τη μέθοδο DSMC (Direct Simulation Monte Carlo) η οποία προτάθηκε από τον G. A. Bird [6].

Η μέθοδος DVM είναι η πιο κατάλληλη για να εφαρμοστεί και να αναπτυχθεί για περιπτώσεις όπου η ροή είναι γραμμική σε αγωγούς με άπειρο μήκος. Είναι μία *ντετερμινιστική* προσέγγιση που για να λειτουργήσει χρειάζεται ορισμένες διευθύνσεις και τιμές ταχυτήτων των μορίων, για τις οποίες εάν γνωρίζουμε τη λύση της εξίσωσης Boltzmann, τότε μπορούμε να προσδιορίσουμε τις μακροσκοπικές ιδιότητες του ρευστού. Όσο πιο πολλές διευθύνσεις και ταχύτητες επιλέξουμε τόσο πιο ακριβείς θα είναι και οι μακροσκοπικές ιδιότητες. Για να κάνουμε λοιπόν την καλύτερη επιλογή ταχυτήτων, πράγμα το οποίο σημαίνει ότι θα έχουμε και ακριβέστερο υπολογισμό στις μακροσκοπικές ιδιότητες του ρευστού, θα πάρουμε ταχύτητες οι οποίες ισούνται με τις ρίζες κάποιου ορθογώνιου πολυωνύμου έτσι ώστε να εφαρμοστεί κάποια αριθμητική ολοκλήρωση τύπου Gauss, Legandre κτλ. Το σχήμα είναι επαναληπτικό και στη γενική περίπτωση συγκλίνει. Η σύγκλιση αυτή είναι αργή, ιδίως όταν ο αριθμός Knudsen του αερίου είναι πολύ μικρός.

Η μέθοδος DSMC είναι η δεύτερη μέθοδος επίλυσης η οποία δεν ασχολείται καθόλου με την εξίσωση του Boltzmann αλλά αντιμετωπίζει τις συγκρούσεις των μορίων

με *στοχαστικές* διαδικασίες. Η διαφορά της με την DVM είναι ότι χρησιμοποιείται κυρίως για μη γραμμικές ροές αλλά και σε αγωγούς πεπερασμένου μήκους.

1.2 Φυσικά προβλήματα και παράλληλος προγραμματισμός

Όπως αναφέραμε νωρίτερα για την επίλυση προβλημάτων κινητικής θεωρίας χρησιμοποιούμε διάφορα μοντέλα κινητικής θεωρίας τα οποία και αυτά με τη σειρά τους χρειάζονται κάποιες αριθμητικές μεθόδους. Οι μέθοδοι αυτές που άλλοτε είναι ντετερμινιστικές και άλλοτε στοχαστικές χαρακτηρίζονται από *επαναληπτικές διαδικασίες* μέχρι να δώσουν ένα ικανοποιητικό αποτέλεσμα. Φυσικά οι πράξεις με το χέρι είναι μία διαδικασία η οποία είναι αδύνατο να γίνει.

Η επίλυση λοιπόν μεταφέρεται σχεδόν πάντα σε προγραμματιστικές διαδικασίες αλλά και πάλι απαιτείται πολύ μεγάλος υπολογιστικός χρόνος μέχρι τη σύγκλιση και την απόδοση ενός τελικού αποτελέσματος, ιδίως όταν ο αριθμός Knudsen του αερίου είναι πολύ μικρός. Οι κώδικες αυτοί λοιπόν μπορεί να χρειαστούν από λίγες ώρες έως και αρκετούς μήνες χρονικό περιθώριο για να τρέξουν και να δώσουν αποτέλεσμα κάτι το οποίο σαφέστατα δεν είναι και πάλι τόσο αποδοτικό.

Λύση στην αντιμετώπιση και γρήγορη επίλυση τέτοιων πολύπλοκων και χρονοβόρων προβλημάτων έρχεται να δώσει ο παράλληλος προγραμματισμός με διαφόρων ειδών τεχνολογίες όπως είναι το πρωτόκολλο MPI και η τεχνολογία CUDA της NVIDIA.

Παρόλο που και οι δύο αυτές τεχνικές αποφέρουν τα επιθυμητά αποτελέσματα σε πολύ μικρά χρονικά διαστήματα έχουν μία μεγάλη διαφοροποίηση στο κόστος αν σκεφτεί κανείς πως το πρωτόκολλο MPI χρειάζεται έναν μεγάλο αριθμό από επεξεργαστές (*cluster*) για να μπορέσει να εφαρμοστεί και να γίνει αποδοτικό. Αυτό σημαίνει πως για να το χρησιμοποιήσει κάποιος θα πρέπει να συγκεντρώσει μία ομάδα από επεξεργαστές οι οποίοι δουλεύουν παράλληλα, να διατεθεί ένας μεγάλος χώρος για την φυσική τους τοποθέτηση καθώς επίσης και ένα μεγάλο χρηματικό ποσό για την απόκτηση τους.

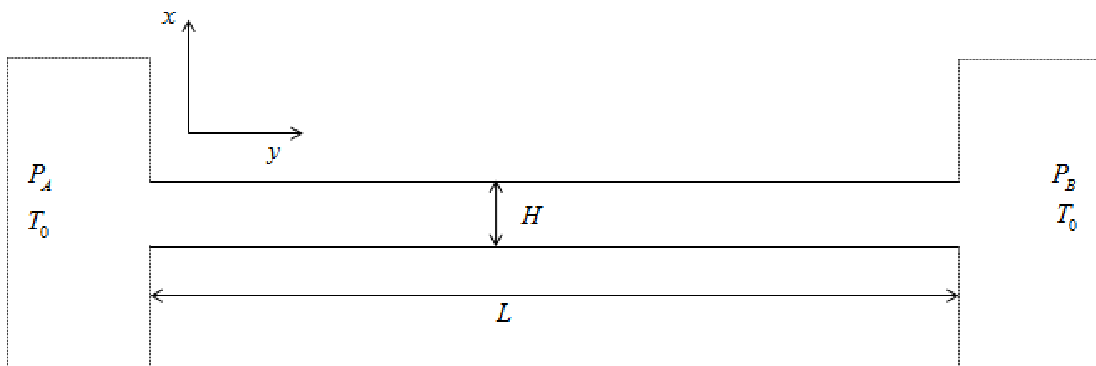
Εν αντιθέσει με το MPI η τεχνολογία CUDA είναι μία νεότατη τεχνολογία η οποία συμφέρει από πάρα πολλές απόψεις να την επιλέξει κανείς. Είναι μία παράλληλη υπολογιστική πλατφόρμα και μοντέλο προγραμματισμού που επιτρέπει εντυπωσιακές εξελίξεις στην υπολογιστική απόδοση εκμεταλλευόμενη την δύναμη της κάρτας

γραφικών (GPU). Η παραλληλοποίηση λοιπόν γίνεται στο εσωτερικό της κάρτας γραφικών με αποτέλεσμα να μην χρειάζεται ούτε μεγάλο χρηματικό κόστος για την απόκτησή της αλλά ούτε και διάθεση φυσικού χώρου για την χρησιμοποίηση της.

Σκοπός της συγκεκριμένης πτυχιακής εργασίας είναι η ανάδειξη της προγραμματιστικής πλατφόρμας CUDA ως ένα ιδιαίτερα χρήσιμο και αποτελεσματικό εργαλείο για την αριθμητική επίλυση προβλημάτων κινητικής θεωρίας. Αυτό θα επιτευχθεί μέσα από ένα πρόβλημα που συναντάται συχνά στην επιστήμη της μηχανικής ρευστών και περιγράφει τη ροή ενός μονοατομικού αερίου από ένα δοχείο σε ένα άλλο δοχείο διαμέσου ενός αγωγού απείρου μήκους εξαιτίας της διαφοράς πίεσης των δοχείων αυτών.

1.3 Περιγραφή του προβλήματος

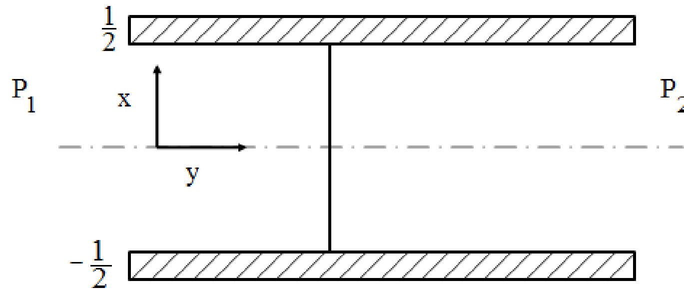
Θεωρείται η ροή ενός μονοατομικού αερίου διαμέσου δύο πλακών με μήκος L. Οι πλάκες αυτές συνδέουν δύο δοχεία τα οποία περιέχουν το αέριο σε θερμοκρασία T_0. Όπως μπορούμε να δούμε και στο Σχ. 1.1 τα δύο δοχεία έχουν διαφορετικές πιέσεις μεταξύ τους P_A και P_B αντίστοιχα με $P_A > P_B$. Η ροή γίνεται από το δοχείο με πίεση P_A προς το δοχείο με πίεση P_B.



Σχήμα 1.1: Γεωμετρικά χαρακτηριστικά του καναλιού

Έχει διαπιστωθεί πως, ενώ η μέθοδος DVM απαιτεί άπειρο μήκος πλακών για να χαρακτηρισθεί η ροή γραμμική, μπορούμε να κάνουμε την ίδια θεώρηση χωρίς σφάλματα και όταν το μήκος L των πλακών είναι απλά πολύ μεγαλύτερο από την μεταξύ τους απόσταση $(H << L)$. Έτσι λοιπόν αναπτύσσεται μία σταθερή μονοδιάστατη

(1D) πλήρως ανεπτυγμένη ροή η οποία περιγράφεται από τη μη γραμμικοποιημένη εξίσωση BGK.



Σχήμα 1.2: Διαμόρφωση ροής

Η εξίσωση BGK περιγράφεται από τη σχέση:

$$\mu \frac{\partial Y(x,\mu)^{(k+1/2)}}{\partial x} + \delta Y(x,\mu)^{(k+1/2)} = \delta u(x)^{(k)} + \frac{1}{2} \tag{1.3}$$

με:

$$u(x)^{(k+1)} = \frac{1}{\sqrt{\pi}} \int_{-\infty}^{\infty} Y(x,\mu)^{(k+1/2)} e^{-\mu^2} d\mu \tag{1.4}$$

Με $Y(x,\mu)$ συμβολίζεται η ζητούμενη συνάρτηση κατανομής των σωματιδίων και με $u(x)$ θεωρούμε τη μακροσκοπική ταχύτητα του ρευστού. Η μεταβλητή $\mu \in (-\infty,\infty)$ εκφράζει την ταχύτητα των μορίων ως προς την συνιστώσα y καθότι η κίνηση των μορίων γίνεται προς τη μία κατεύθυνση και μαζί με τη μεταβλητή $x \in [-1/2, 1/2]$ που εκφράζει τη χωρική μεταβλητή για κάθε ζεύγος (x,μ) παίρνουμε και μία διαφορετική τιμή για τη συνάρτηση $Y(x,\mu)$. Θεωρώντας συνθήκες μη διείσδυσης του αερίου στο τοίχωμα οι ακόλουθες συνθήκες μπορούν να διατυπωθούν:

$$Y(-1/2,\mu) = 0, \mu > 0 \tag{1.5}$$

και

$$Y(1/2,\mu) = 0, \mu < 0 \tag{1.6}$$

Η παράμετρος δ όπως εξηγήσαμε και παραπάνω είναι η παράμετρος αραιοποίησης που χρειάζεται να ορίσουμε για τους υπολογισμούς και δίνεται από τη σχέση:

$$\delta = \frac{PH}{\mu \upsilon_0} \tag{1.7}$$

Και τέλος ο δείκτης k αποτελεί έναν δείκτη επανάληψης.

Το ζητούμενο του προβλήματος είναι να βρεθεί ο ρυθμός ροής G του αερίου από το ένα δοχείο στο άλλο ως συνάρτηση του δ. Ο ρυθμός ροής εκφράζεται από τη σχέση:

$$G = 2\int_{-1/2}^{1/2} u(x)dx \tag{1.8}$$

Περισσότερες πληροφορίες για την εξαγωγή των συγκεκριμένων εξισώσεων μπορούν να βρεθούν στις ακόλουθες αναφορές [7, 8, 9, 10].

Μηχανολογικά προβλήματα τα οποία σχετίζονται με τον κλάδο της στατιστικής μηχανικής όπως αυτά της κινητικής θεωρίας είναι σχεδόν σίγουρο ότι πάντα θα απαιτούν υψηλό υπολογιστικό κόστος. Το πρωτόκολλο MPI και η τεχνολογία CUDA είναι δύο πολύ ικανά εργαλεία για την ικανοποίηση τέτοιων απαιτήσεων. Στο κεφάλαιο 5 θα περιγραφεί λεπτομερώς ο κώδικας που έχει υλοποιηθεί για την επίλυση του αριθμητικού μοντέλου που έχει αναφερθεί παραπάνω ενώ στα παραρτήματα Α και Β μπορεί κανείς να βρει τόσο τον σειριακό όσο και τον παράλληλο κώδικα επίλυσης. Ο συγκεκριμένος σειριακός κώδικας έχει γραφεί σε γλώσσα προγραμματισμού C++ ενώ ο παράλληλος έχει μετασχηματιστεί κάνοντας χρήση του μοντέλου παράλληλου προγραμματισμού CUDA.

Κεφάλαιο 2

Η GPU και η ιστορική εξέλιξη της

2.1 Η CPU και η εξέλιξή της

2.2 Η GPU και η εξέλιξή της

2.3 Η αρχιτεκτονική μιας GPU

2.1 Η CPU και η εξέλιξή της

Η Κεντρική μονάδα επεξεργασίας (Central Processing Unit - CPU) είναι το βασικό εξάρτημα για τη λειτουργία ενός υπολογιστή και συχνά αναφέρεται και ως επεξεργαστής ή μικροεπεξεργαστής. Πρόκειται για ένα ολοκληρωμένο κύκλωμα (IC) γενικού σκοπού που μπορεί να προγραμματιστεί. Είναι υπεύθυνη για τον έλεγχο της λειτουργίας του υπολογιστή καθώς και για την εκτέλεση λειτουργιών επεξεργασίας των δεδομένων. Η επεξεργασία των δεδομένων γίνεται με μία σειρά από εντολές οι οποίες είναι σε γλώσσα μηχανής και είναι αποθηκευμένες στην κύρια μνήμη. Η λειτουργικότητα ενός επεξεργαστή εξαρτάται απόλυτα από το σύνολο των εντολών που είναι ικανός να εκτελέσει. Με το πέρασμα των χρόνων οι επεξεργαστές σχεδιάζονται και κατασκευάζονται με τέτοιο τρόπο ώστε το εσωτερικό ρολόι του επεξεργαστή να αριθμεί ολοένα και περισσότερες εντολές στη μονάδα του χρόνου.

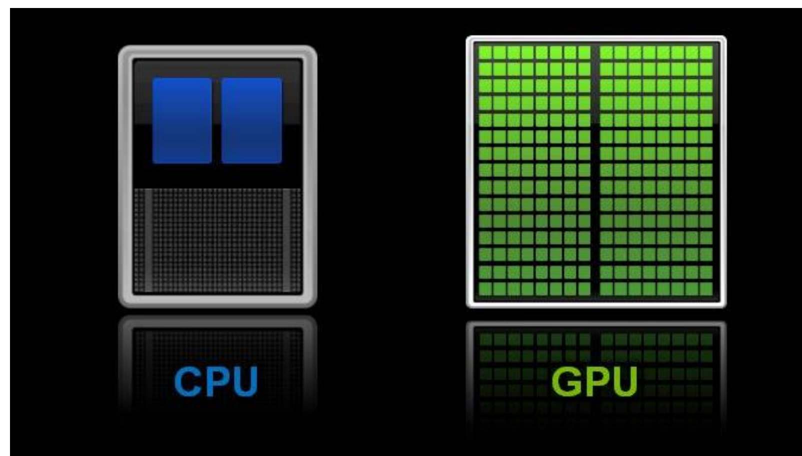
Πριν 30 χρόνια περίπου τα ρολόγια των επεξεργαστών είχαν συχνότητα περίπου 1 Megahertz (MHz). Τα τελευταία χρόνια όμως η συχνότητα του ρολογιού έχει φτάσει να κυμαίνεται από εκατοντάδες Megahertz (MHz) έως αρκετά Gigahertz (GHz). Οι επεξεργαστές τελευταίας γενιάς αγγίζουν μέχρι και τα 5.5 GHz. Παράλληλα, έχει αυξηθεί η πολυπλοκότητα και ο αριθμός των τρανζίστορ που αποτελούν ένα ολοκληρωμένο κύκλωμα. Κυρίαρχες εταιρίες επεξεργαστών είναι η Intel, η AMD και η IBM.

2.2 Η GPU και η εξέλιξή της

Μια κάρτα γραφικών είναι επίσης ένα βασικό τμήμα ενός υπολογιστή, το οποίο λαμβάνει από την κεντρική μονάδα επεξεργασίας (CPU) τα δεδομένα για να τα μετατρέψει σε εικόνα, η οποία θα προβληθεί στην οθόνη. Ουσιαστικά είναι μια πλακέτα κυκλωμάτων, η οποία περιλαμβάνει έναν επεξεργαστή, που ονομάζεται μονάδα επεξεργασίας γραφικών (Graphics Processing Unit, GPU) και είναι παρόμοιος με τον επεξεργαστή ενός υπολογιστή, και τέλος κυκλώματα μνήμης RAM. Διαθέτει επίσης ένα μικροκύκλωμα (chip) εισόδου / εξόδου (BIOS), το οποίο αποθηκεύει τις ρυθμίσεις της κάρτας και εκτελεί την είσοδο και την έξοδο κατά την εκκίνηση του συστήματος. Μια GPU, ωστόσο, είναι ένας παράλληλος επεξεργαστής βελτιστοποιημένος τόσο ώστε να επιταχύνει γραφικούς υπολογισμούς και έχει σχεδιαστεί ειδικά για την εκτέλεση των

πολύπλοκων μαθηματικών, κινητής υποδιαστολής και γεωμετρικών υπολογισμών που είναι απαραίτητοι για την απόδοση 3D γραφικών. Η παράλληλη υπολογιστική ισχύς κινητής υποδιαστολής που έχει βρεθεί σε μια σύγχρονη GPU είναι τάξεις μεγέθους υψηλότερη από ό, τι σε μία CPU.

Οι GPUs μπορεί να βρεθούν σε ένα ευρύ φάσμα συστημάτων, από τους υπολογιστές γραφείου και φορητούς υπολογιστές ως τα κινητά τηλέφωνα και τους σούπερ υπολογιστές. Με την παράλληλη δομή τους, οι GPUs υλοποιούν μια σειρά από 2D και 3D γραφικά που επεξεργάζονται στο υλικό, και τις καθιστά πολύ πιο γρήγορες από ό, τι μια CPU γενικής χρήσης σε αυτές τις λειτουργίες.



Εικόνα 2.1: Διαφορά μεταξύ CPU και GPU

Αρχιτεκτονικά, όπως φαίνεται και στην Εικ. 2.1, η CPU αποτελείται από λίγους μόνο πυρήνες με αρκετή προσωρινή μνήμη που μπορούν να χειριστούν λίγες διεργασίες τη φορά. Αντίθετα μία GPU αποτελείται από εκατοντάδες πυρήνες που μπορούν να χειριστούν χιλιάδες νήματα ταυτόχρονα. Η ικανότητα μίας GPU με περισσότερους από 100 πυρήνες να επεξεργάζονται χιλιάδες νήματα μπορεί να επιταχύνει κάποιο λογισμικό κατά 100 φορές παραπάνω από μία CPU. Πόσο μάλλον, η GPU επιτυγχάνει αυτήν την επιτάχυνση έχοντας μεγαλύτερη ισχύ αλλά όντας και πιο οικονομική από μια CPU.

2.3 Η ιστορική εξέλιξη της GPU

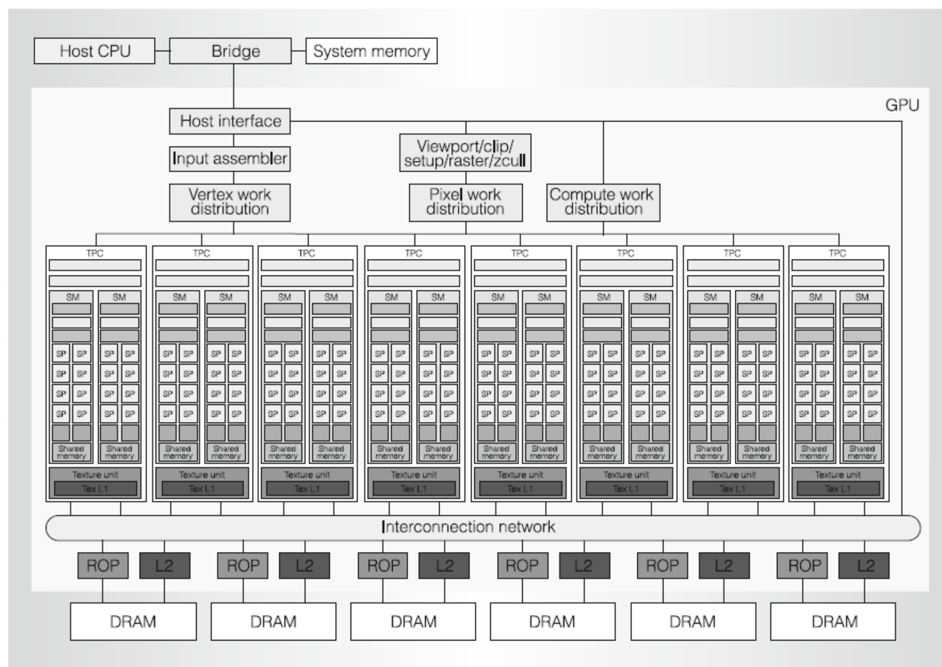
Στις αρχές του 1980 οι GPU της εποχής ήταν απλοί ρυθμιστές πλαισίων. Ήταν πλακίδια από TTL chips στηριζόμενα πάνω στην CPU και μπορούσαν μόνο να σχηματίσουν πλέγματα γραφικών για να αποδώσουν εικόνα. Το 1984 η IBM κυκλοφόρησε μία από τις πρώτες κάρτες 2D/3D βίντεο για PC την Professional Graphics

Controller (PGC). Μέχρι το 1987, περισσότερα χαρακτηριστικά προστέθηκαν στις πρώτες GPU, όπως η σκίαση των στερεών, ο φωτισμός των γωνιών, ο αλγόριθμοι σχεδίασης για το γέμισμα πολυγώνων, η ρύθμιση του βάθους των pixel και η ανάμειξη χρωμάτων. Υπήρχε όμως ακόμη πολλή εξάρτηση από την ανταλλαγή υπολογισμών με την CPU. Στα τέλη της δεκαετίας του 80 η Silicon Graphics Inc. εισήγαγε την OpenGL, μία 2D/3D εφαρμογή ανεξάρτητη πλατφόρμας η οποία μέχρι και σήμερα χρησιμοποιείται για πολύπλοκη αναπαράσταση γραφικών. Στα μέσα της δεκαετίας του 1990 κάρτες SGI βρέθηκαν κυρίως στους σταθμούς εργασίας, ενώ οι κατασκευαστές 3D καρτών γραφικών 3DFX (Voodoo), Nvidia (TNT), ATI (Rage), και Matrox άρχισαν να παρέχουν 3D κάρτες γραφικών στους καταναλωτές. Ένας συνδυασμός "φθηνού" υλικού με παιχνίδια όπως το Quake και το Doom οδήγησε πραγματικά στην εδραίωση της GPU στη βιομηχανία του παιχνιδιού. Το 1996 η 3DFX Voodoo ήταν η πρώτη 3D κάρτα γραφικών για παιχνίδια, η οποία παρείχε μόνο 3D απεικόνιση και χρειαζόταν και έναν 2D επιταχυντή, με 1 εκατομμύριο τρανζίστορ, 4MB 64-bit DRAM και ρολόι επεξεργαστή στα 50 MHz μέχρι που 3 χρόνια μετά η Nvidia GeForce256 που είχε 23 εκατομμύρια τρανζίστορ, 32 MB 128-bit DRAM και ρολόι επεξεργαστή στα 120 MHz απογείωσε τη βιομηχανία των παιχνιδιών. Το επόμενο βήμα ήταν η εισαγωγή του προγραμματισμού με μια γραμμή εντολών της GPU με την κυκλοφορία της Nvidia GeForce 3. Τώρα ο προγραμματιστής μπορούσε όχι μόνο απλά να στέλνει όλα τα δεδομένα περιγραφής των γραφικών στην GPU και να περιμένει την απλή ροή εκτέλεσής τους αλλά να στέλνει τα δεδομένα αυτά μαζί με προγράμματα κορυφών (shaders) που λειτουργούν πάνω στα δεδομένα όταν αυτά βρίσκονται ήδη σε επεξεργασία. Αυτά τα προγράμματα ήταν μικροί «πυρήνες» οι οποίοι ήταν γραμμένοι σε γλώσσα assembly.

Ένα χρόνο αργότερα, το 2002, οι πρώτες πλήρως προγραμματιζόμενες κάρτες γραφικών εμφανίστηκαν στην αγορά όπως η Nvidia GeForce FX με 80 εκατομμύρια τρανζίστορ, 128 MB DDR DRAM και ρολόι επεξεργαστή στα 400 MHz. Το 2003 με την εισαγωγή της DirectX άρχισε να γίνεται εκμετάλλευση της υπολογιστικής ικανότητας πλέον των GPU όχι μόνο στην απόδοση γραφικών αλλά και σε άλλους τομείς όπου χρειάζονταν αρκετά μεγάλη υπολογιστική δύναμη. Κάπως έτσι επιτεύχθηκε η εξολοκλήρου υποστήριξη υπολογισμών κινητής υποδιαστολής αλλά και εξελιγμένη επεξεργασία υφής στις κάρτες. Το 2004 κυκλοφόρησε η GeForce 6 με 146 εκατομμύρια τρανζίστορ, 256 MB 256-bit GDDR3 DRAM και ρολόι επεξεργαστή στα 500 MHz. Από

τη μεριά του λογισμικού επίσης χρησιμοποιήθηκαν γλώσσες όπως η Brook και η Sh οι οποίες προσέφεραν εκφράσεις συνθήκης, βρόχους επανάληψης και δυναμικό έλεγχο ροής.

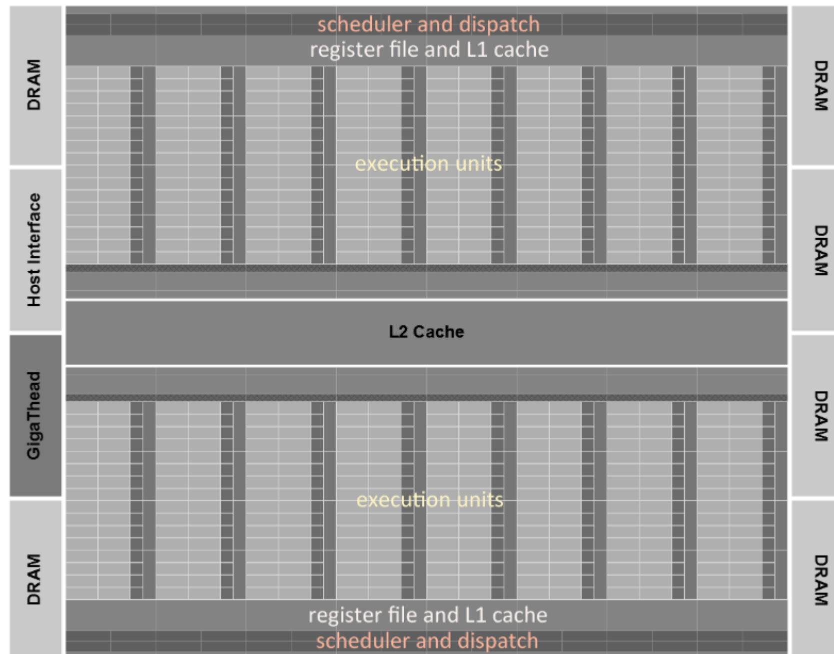
Η εισαγωγή της σειράς NVIDIA GeFroce το 2006 σηματοδότησε το επόμενο βήμα στην εξέλιξη της GPU και στην καθιέρωσή της ως μαζικά παράλληλος επεξεργαστής. Η G80 (GeForce8800) Tesla αρχιτεκτονική ήταν η πρώτη που διέθετε έναν πλήρως προγραμματιζόμενο ενιαίο επεξεργαστή που ονομάζεται Streaming Multiprocessor ή SM. Η Tesla αρχιτεκτονική βασιζόταν σε μία κλιμακούμενη συστοιχία επεξεργαστών. Η Εικ. 2.2 δείχνει ένα σχηματικό διάγραμμα μιας GeForce8800 με 128 Streaming Processors (SPs) οργανωμένοι σε 16 SMs σε 8 ανεξάρτητες επεξεργαστικές μονάδες που ονομάζονται texture/processor clusters. Η πρώτη GeForce 8 είχε 681 εκατομμύρια τρανζίστορ, 768 MB 384 – bit GDDR3 DRAM και ρολόι επεξεργαστή στα 600 MHz. Για την αξιοποίηση όλης αυτής της GPU δύναμης "γενικής χρήσης" αναπτύχθηκε η νέα γλώσσα προγραμματισμού CUDA από την NVIDIA για την NVIDIA και μόνο.



Εικόνα 2.2: Tesla Αρχιτεκτονική

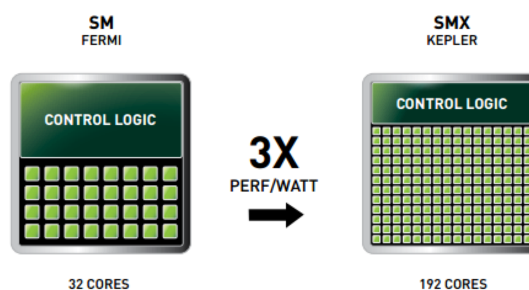
Η τάση εξέλιξης των GPU συνεχίζεται με την εισαγωγή της αρχιτεκτονικής Fermi το 2010. Οι Fermi GPUs με 3 δισεκατομμύρια τρανζίστορ, 1.5 GB 384-bit GDDR5 DRAM και ρολόι επεξεργαστή 700 MHz ήταν οι πρώτες που σχεδιάστηκαν για των υπολογισμό GPGPU (General purpose computing on GPUs). Η GTX480 Fermi είχε συνολικά 480 CUDA πυρήνες στην αρχή της διάδοσής της (15 streaming

multiprocessors*32 πυρήνες CUDA ο καθένας). Πρόσφατα, στα τέλη του 2010, όπως φαίνεται και στην Εικ. 2.3 η NVIDIA ανανεώνει τις Fermi κάρτες προσθέτοντας έναν ακόμη SM (ανεβάζοντας το σύνολο των CUDA πυρήνων στους 512) και προσφέροντας ένα ελαφρώς μεγαλύτερο εύρος ζώνης μνήμης (GTX580).



Εικόνα 2.3: Ανανεωμένη αρχιτεκτονική Fermi

Επόμενη ανανέωση των καρτών γραφικών ήταν οι κάρτες Kepler GK110 με 7.1 δισεκατομμύρια τρανζίστορ και ασύλληπτες ταχύτητες. Αποτελούνται από τους Next Generation Streaming Multiprocessors (SMXs) οι οποίοι συμπεριλαμβάνουν αρκετές αρχιτεκτονικές αναβαθμίσεις που κάνουν τους SM όχι μόνο τους πιο δυνατούς αλλά και τους πιο εύκολα προγραμματίσιμους και αποδοτικούς. Ένας SMX έχει 192 CUDA πυρήνες, 32 Special Function Units (SFU) και 32 Load/Store Units (LD/SD) όπως φαίνεται στην Εικ. 2.4.



Εικόνα 2.4: Αρχιτεκτονική Kepler

2.4 Η αρχιτεκτονική μιας GPU

Η αρχιτεκτονική μιας GPU στην οποία βασίζεται και η πτυχιακή αυτή εργασία οργανώνεται σε μια συστοιχία πολυεπεξεργαστών συνεχούς ροής (streaming multiprocessors - SM) που κάνουν εντατική χρήση νημάτων. Δύο SM σχηματίζουν ένα δομικό στοιχείο όμως ο αριθμός των SM σε ένα δομικό στοιχείο μπορεί να ποικίλει από την μία γενιά GPU CUDA στην άλλη όπως είδαμε και προηγουμένως. Επίσης κάθε SM διαθέτει έναν αριθμό από επεξεργαστές συνεχούς ροής (streaming processors – SP) οι οποίοι μοιράζονται τη λογική ελέγχου και την κρυφή μνήμη εντολών. Προς το παρόν, κάθε GPU διαθέτει μέχρι 4 Gbyte DRAM διπλού ρυθμού δεδομένων γραφικών (Graphics Double Data Rate – GDDR), τα οποία αναφέρονται ως καθολική μνήμη. Αυτές οι DRAM τύπου GDDR διαφέρουν από την DRAM του συστήματος στη μητρική κάρτα της CPU στο ότι αποτελούν ουσιαστικά την προσωρινή μνήμη καρέ η οποία χρησιμοποιείται για γραφικά. Για εφαρμογές γραφικών, φιλοξενούν εικόνες βίντεο και πληροφορίες υφής για τρισδιάστατη (3D) απόδοση, αλλά για υπολογιστικές εργασίες λειτουργούν ως μνήμη «εκτός – τσιπ» (off-chip) πολύ υψηλού εύρους ζώνης, αν και με κάπως μεγαλύτερο λανθάνοντα χρόνο από την τυπική μνήμη συστήματος. Για μαζικά παράλληλες εφαρμογές, το υψηλότερο εύρος ζώνης εξισορροπεί το μεγαλύτερο λανθάνοντα χρόνο.

Το μαζικά παράλληλο τσιπ G80 που εισήγαγε την αρχιτεκτονική CUDA είχε εύρος ζώνης μνήμης 86,4 GB/s, συν ένα εύρος ζώνης επικοινωνίας 8 GB/s με τη CPU. Μια εφαρμογή CUDA μπορεί να μεταφέρει δεδομένα από τη μνήμη συστήματος με 4 GB/s ενώ παράλληλα φορτώνει τα δεδομένα στη μνήμη με 4 GB/s. Συνολικά, υπάρχει ένα συνδυασμένο σύνολο 8 GB/s. Επίσης διαθέτει 128 SP (16 SMs, καθένας με 8 SPs). Κάθε SP διαθέτει μία μονάδα πολλαπλασιασμού και πρόσθεσης (multiply-add – MAD) και μια πρόσθετη μονάδα πολλαπλασιασμού. Με 128 SPs, δημιουργείται ένα σύνολο από περισσότερα των 500 gigaflops με αποτέλεσμα να υποστηρίζει μέχρι 768 νήματα ανά SM, κάτι που παρέχει περίπου 12000 νήματα γι’ αυτό το τσιπ. Το πιο πρόσφατο τσιπ GT200 διαθέτει 240 SPs (30 SMs, καθένας με 8 SPs) υπερβαίνοντας το 1 teraflops με αποτέλεσμα να υποστηρίζει 1024 νήματα ανά SM, δηλαδή μέχρι περίπου 30000 νήματα για το τσιπ.

Κεφάλαιο 3

CUDA (Compute Unified Device Architecture)

3.1 Η αρχιτεκτονική CUDA

3.2 Παραλληλία δεδομένων

3.2.1 Παράλληλες αρχιτεκτονικές κατά Flynn

3.3. Η δομή και η εκτέλεση των προγραμμάτων της CUDA

3.3.1 Τα νήματα της CUDA
3.3.2 Δημιουργία πλέγματος (Εκκίνηση πυρήνα)
3.3.3 Μνήμες CUDA
3.3.4 Δήλωση μεταβλητών CUDA

3.4 Το προγραμματιστικό μοντέλο και το API της CUDA

3.5 CUDA Occupancy Calculator

3.5.1 Καταχωρητές ανά νήμα και κοινόχρηστη μνήμη ανά μπλοκ
3.5.2 Οδηγίες χρήσης του CUDA Occupancy Calculator
3.5.3Μερικές σημειώσεις για το CUDA Occupancy Calculator

3.1 Η αρχιτεκτονική CUDA

Το όνομα CUDA είναι ακρώνυμο για την ονομασία Compute Unified Device Architecture. Αυτή η αρχιτεκτονική μπορεί να βρεθεί σε όλες τις τρεις τελευταίες γενιές καρτών γραφικών της εταιρίας NVIDIA.

Για έναν προγραμματιστή της CUDA, το υπολογιστικό σύστημα αποτελείται από μια *συσκευή υπηρεσίας* (host), η οποία είναι μια παραδοσιακή κεντρική μονάδα επεξεργασίας (CPU) και μία ή περισσότερες *συσκευές* (devices), οι οποίες είναι μαζικά παράλληλοι επεξεργαστές εξοπλισμένοι με ένα μεγάλο αριθμό μονάδων εκτέλεσης αριθμητικής. Επειδή η παραλληλία δεδομένων παίζει σημαντικό ρόλο στην CUDA, θα εξετάσουμε πρώτα την έννοια της παραλληλίας δεδομένων και μετά θα γνωρίσουμε τα βασικά χαρακτηριστικά της CUDA. Επίσης ο αναγνώστης μπορεί να εντοπίσει περαιτέρω πληροφορίες τόσο στο διαδίκτυο [παρατίθεται μία λίστα από πιθανές πηγές στο τέλος] όσο και στη βιβλιογραφία [13, 14, 15].

3.2 Παραλληλία δεδομένων

Στις σύγχρονες εφαρμογές λογισμικού, τα διάφορα τμήματα των προγραμμάτων παρουσιάζουν μεγάλη ποσότητα παραλληλίας δεδομένων (data parallelism), μιας ιδιότητας που επιτρέπει να εκτελούνται παράλληλα πολλές αριθμητικές λειτουργίες στις δομές δεδομένων του προγράμματος, με ταυτόχρονο τρόπο. Έτσι με βάση αυτήν την αρχή μεγάλα προβλήματα διαιρούνται σε μικρότερα, τα οποία με τη σειρά τους επιλύονται και αυτά ταυτόχρονα. Οι συσκευές CUDA επιταχύνουν την εκτέλεση αυτών των εφαρμογών επωφελούμενες σε μεγάλο βαθμό από την παραλληλία δεδομένων.

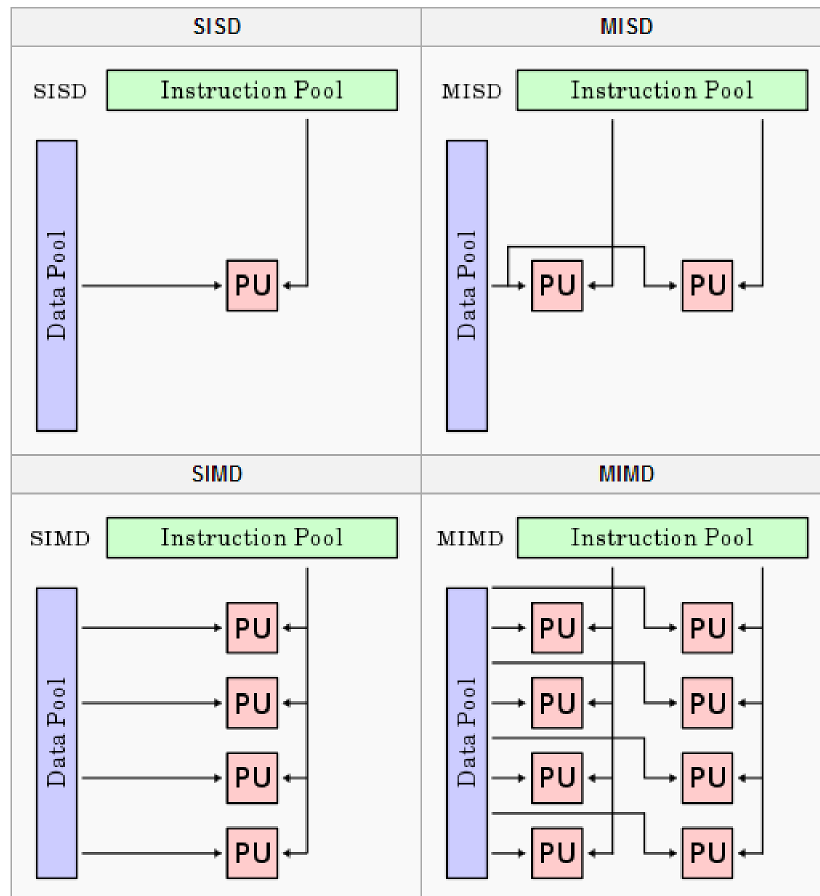
Οι κώδικες παράλληλου προγραμματισμού είναι πιο δύσκολο να συνταχθούν σε σχέση με τους σειριακούς κώδικες καθώς η επικοινωνία και ο συγχρονισμός μεταξύ διαφορετικών υπό-προβλημάτων είναι συχνά το μεγαλύτερο εμπόδιο όσο αφορά την επίτευξη της καλύτερης απόδοσης ενός παράλληλου προγράμματος. Η επιτάχυνση ενός προγράμματος που έχει συνταχθεί με παράλληλο τρόπο σε σχέση με ένα σειριακό δίνεται από το νόμο του Amdahl:

$$S = \frac{1}{(1-P)} \qquad (3.1)$$

όπου S είναι η επιτάχυνση του προγράμματος σε σχέση με το απλό σειριακό και P είναι το ποσοστό κατά το οποίο έχει παραλληλοποιηθεί.

3.2.1 Παράλληλες αρχιτεκτονικές κατά Flynn

Σύμφωνα με τον Michael Flynn το 1966 τα είδη παράλληλων αρχιτεκτονικών υπολογιστικών συστημάτων είναι βασισμένα σε δύο πράγματα: *εντολές και δεδομένα*. Έτσι λοιπόν έχουμε τις παρακάτω αρχιτεκτονικές στην Εικ. 3.1.



Εικόνα 3.1:Αρχιτεκτονικές κατά Flynn

- **Single Instruction, Single Data stream - SISD** είναι το απλό ακολουθιακό (sequential) μοντέλο υπολογισμών όπου γίνεται μία σειρά υπολογισμών ανά ομάδα δεδομένων.
- **Single Instruction, Multiple Data stream - SIMD** είναι το μοντέλο όπου οι επεξεργαστές εκτελούν τις ίδιες πράξεις με διαφορετικά δεδομένα. Αυτή η αρχιτεκτονική είναι κατάλληλη για παραλληλία μικρού όγκου υπολογισμών και δεν μπορεί να εφαρμοστεί σε συστήματα με πολλές επεξεργαστικές μονάδες.

• **Multiple Instruction, Single Data stream - MISD** είναι το αντίστροφο του μοντέλου SIMD το οποίο είναι απλώς μία θεωρητική αρχιτεκτονική που δεν μπορεί να εφαρμοστεί σε πρακτικά προβλήματα.

• **Multiple Instruction, Multiple Data stream - MIMD** είναι το μοντέλο όπου πολλαπλές αυτόνομες επεξεργαστικές μονάδες εκτελούν ταυτόχρονα διαφορετικές οδηγίες για διαφορετικά δεδομένα.

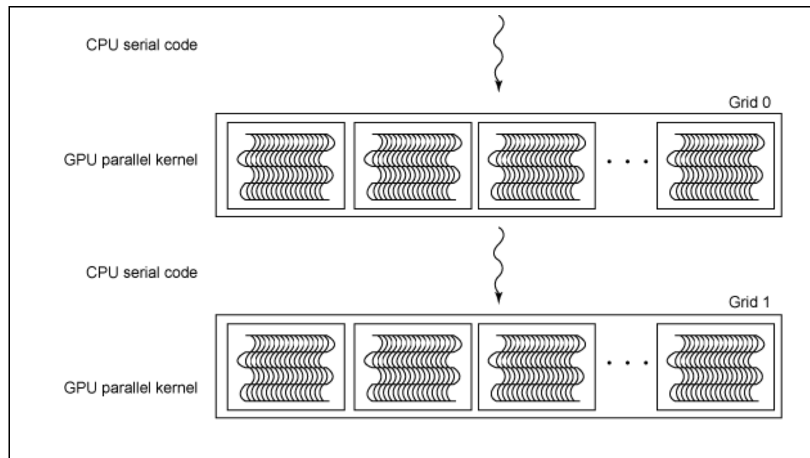
Οι συνηθέστερες αρχιτεκτονικές είναι τύπου SIMD ή τύπου MIMD με την επικρατέστερη να είναι τύπου MIMD όπως και τα προγράμματα που είναι γραμμένα σε CUDA.

3.3. Η δομή και η εκτέλεση των προγραμμάτων της CUDA

Τα προγράμματα που είναι γραμμένα σε CUDA αποτελούνται από μία ή περισσότερες *φάσεις* (phases) οι οποίες είτε εκτελούνται στη λεγόμενη «συσκευή υπηρεσίας» (host) δηλαδή στη CPU είτε σε μια συσκευή GPU (device). Οι φάσεις με μικρή ή καθόλου παραλληλία δεδομένων υλοποιούνται στον κώδικα υπηρεσίας (host code) ενώ οι φάσεις που παρουσιάζουν μεγάλη παραλληλία δεδομένων υλοποιούνται στον κώδικα της συσκευής (device code). Ένα πρόγραμμα CUDA λοιπόν είναι ένας πηγαίος κώδικας ο οποίος περιλαμβάνει τόσο τον κώδικα υπηρεσίας όσο και τον κώδικα συσκευής. Ο μεταγλωττιστής C/C++ της NVIDIA (nvcc) διαχωρίζει τους δύο κώδικες κατά τη μεταγλώττιση. Ο κώδικας υπηρεσίας είναι απλός κώδικας ANSI C/C++ και μεταγλωττίζεται περαιτέρω με τους μεταγλωττιστές της πρότυπης C/C++ της συσκευής υπηρεσίας και εκτελείται ως μια κανονική διεργασία CPU. Ο κώδικας της συσκευής γράφεται επίσης σε ANSI C αλλά με επεκτάσεις, δηλαδή με λέξεις κλειδιά που επισημαίνουν τις συναρτήσεις με παραλληλία δεδομένων, οι οποίες ονομάζονται *πυρήνες* (kernels) καθώς και τις σχετικές με αυτές δομές δεδομένων. Ο κώδικας συσκευής συνήθως μεταγλωττίζεται περαιτέρω από τον nvcc και εκτελείται σε μια συσκευή GPU.

Η εκτέλεση ενός τυπικού προγράμματος CUDA φαίνεται στην Εικ. 3.2. Η εκτέλεση ξεκινάει με την εκτέλεση κώδικα στη συσκευή υπηρεσίας (CPU). Όταν καλείται ή ξεκινάει μία συνάρτηση πυρήνα, η εκτέλεση μεταφέρεται σε μία συσκευή (GPU), όπου παράγεται ένας μεγάλος αριθμός νημάτων ώστε να αξιοποιηθεί η άφθονη παραλληλία των δεδομένων. Όλα τα νήματα που παράγει ένας πυρήνας κατά την κλήση του ονομάζονται συλλογικά *πλέγμα* (grid) και εκτελούν όλα τη συνάρτηση πυρήνα. Η

Εικ.3.2 δείχνει την εκτέλεση δύο πλεγμάτων από νήματα. Θα ασχοληθούμε με τον τρόπο οργάνωσης αυτών των πλεγμάτων αμέσως μετά.

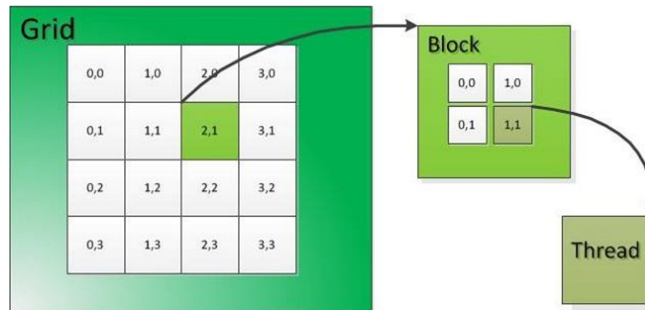


Εικόνα 3.2: Εκτέλεση δύο πλεγμάτων από νήματα

Όταν ολοκληρωθεί η εκτέλεση όλων των νημάτων ενός πυρήνα, το αντίστοιχο πλέγμα τερματίζεται, και η εκτέλεση συνεχίζεται στη συσκευή υπηρεσίας μέχρι να κληθεί κάποιος άλλος πυρήνας.

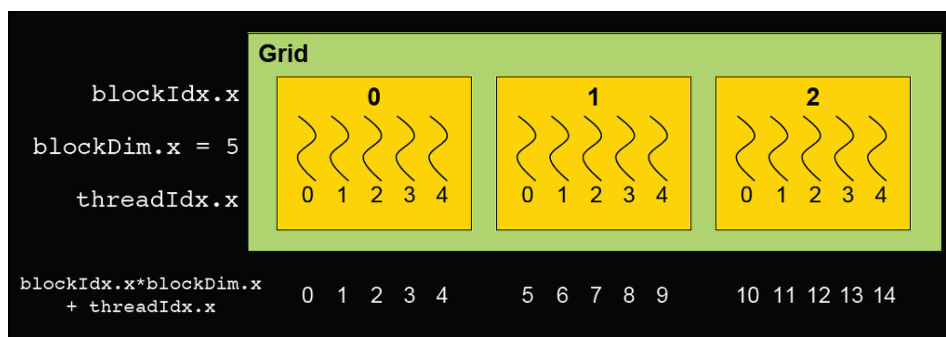
3.3.1 Τα νήματα της CUDA

Επειδή όλα τα νήματα ενός πλέγματος εκτελούν την ίδια συνάρτηση πυρήνα, βασίζονται σε μοναδικές συντεταγμένες για να διακρίνονται μεταξύ τους και να προσδιορίζουν το κατάλληλο τμήμα των δεδομένων που θα επεξεργαστούν. Τα νήματα αυτά είναι οργανωμένα σε δύο επίπεδα με μοναδικές συντεταγμένες για το καθένα οι οποίες είναι η *blockIdx* (από το block index = αριθμοδείκτης μπλοκ) και η *threadIdx* (από το thread index = αριθμοδείκτης νήματος). Αυτές οι δύο μεταβλητές είναι ενσωματωμένες μεταβλητές που έχουν πάρει ήδη αρχική τιμή και είναι προσπελάσιμες στο εσωτερικό των συναρτήσεων πυρήνα. Όταν ένα νήμα εκτελεί τη συνάρτηση πυρήνα οι αναφορές στις μεταβλητές *blockIdx* και *threadIdx* επιστρέφουν τις συντεταγμένες του νήματος. Υπάρχουν άλλες δύο πρόσθετες ενσωματωμένες μεταβλητές, η *blockDim* η οποία επιστρέφει τη διάσταση ενός μπλοκ καθώς επίσης και η μεταβλητή *gridDim* η οποία επιστρέφει το μέγεθος του πλέγματος, δηλαδή πόσα μπλοκ από νήματα έχουν δημιουργηθεί κατά την κλήση του πυρήνα. Στην Εικ. 3.3 μπορούμε να δούμε ακριβώς πως οργανώνονται και ομαδοποιούνται τα νήματα για την καλύτερη κατανόησή τους.



Εικόνα 3.3: Οργάνωση νημάτων

Στην πραγματικότητα όμως στην Εικ. 3.4 φαίνεται μια πιο ρεαλιστική άποψη της οργάνωσης των νημάτων μέσα σε ένα πλέγμα. Το πλέγμα στο παράδειγμα αυτό αποτελείται από 3 μπλοκ νημάτων (από 0 έως 2), καθένα με τιμή *blockIdx.x* και κάθε μπλοκ με τη σειρά του αποτελείται από 5 νήματα το καθένα (από 0 ως 4) με μία τιμή *threadIdx.x*. Στη συγκεκριμένη περίπτωση όλα τα μπλοκ σε επίπεδο πλέγματος είναι οργανωμένα με τη μορφή ενός μονοδιάστατου πίνακα και όλα τα νήματα μέσα σε κάθε μπλοκ είναι επίσης οργανωμένα με τη μορφή ενός μονοδιάστατου πίνακα. Οπότε θα μπορούσαμε να πούμε ότι το πλέγμα διαθέτει συνολικά 15 νήματα. Ωστόσο στη γενική περίπτωση, ένα πλέγμα μπορεί να αποτελείται από ένα τρισδιάστατο πίνακα από μπλοκ και ομοίως κάθε μπλοκ μπορεί να αποτελείται από ένα τρισδιάστατο πίνακα από νήματα.



Εικόνα 3.4: Πιο ρεαλιστική άποψη οργάνωσης των νημάτων

Χρησιμοποιώντας τις ενσωματωμένες μεταβλητές που γνωρίσαμε παραπάνω μπορούμε εύκολα να βρούμε την θέση ενός νήματος μέσα σε ένα πλέγμα η οποία αλλιώς αναφέρεται και ως τιμή νημάτωσης. Το απόσπασμα του κώδικα που υπάρχει σε κάθε πυρήνα χρησιμοποιεί μία ενσωματωμένη μεταβλητή *threadID=blockIdx.x*blockDim.x +threadIdx.x* για να προσδιορίσει το τμήμα των δεδομένων εισόδου από όπου θα διαβάσει το νήμα και το τμήμα των δεδομένων εξόδου όπου θα γράψει το νήμα. Αν υποθέσουμε ότι ένα πλέγμα έχει 128 μπλοκ (N=128) και κάθε μπλοκ έχει 32 νήματα

(M=32) στο πλέγμα συνολικά υπάρχουν M*N = 128*32 = 4096 νήματα. Επιλέγοντας τυχαία το νήμα 5 του μπλοκ 50 η τιμή νημάτωσης του θα είναι 32*50+5=1605.

3.3.2 Δημιουργία πλέγματος (Εκκίνηση πυρήνα)

Γενικά ένα πλέγμα είναι οργανωμένο ως ένας τρισδιάστατος πίνακας μπλοκ και κάθε μπλοκ είναι οργανωμένο ως ένας τρισδιάστατος πίνακας νημάτων. Ακριβώς τις διαστάσεις του πλέγματος ορίζουν οι παράμετροι που τίθενται κατά την εκκίνηση του πυρήνα οι οποίες είναι τρεις. Η πρώτη παράμετρος της εκτέλεσης καθορίζει τις διαστάσεις του πλέγματος ως συνάρτηση του αριθμού των μπλοκ. Η δεύτερη καθορίζει τις διαστάσεις του μπλοκ ως συνάρτηση του αριθμού των νημάτων. Μία τέτοια παράμετρος είναι τύπου Dim3 που βασικά είναι μία δομή της struct της C++ με τρία πεδία απρόσημων ακεραίων: x, y και z. Επειδή τα πλέγματα είναι δισδιάστατοι πίνακες των διαστάσεων των μπλοκ, το τρίτο πεδίο της παραμέτρου διάστασης αγνοείται και τίθεται ίσο με το 1. Έτσι λοιπόν η Εικ. 3.4 θα μπορούσε να είναι αποτέλεσμα κλήσης ενός πυρήνα με τον παρακάτω κώδικα:

```
dim3 dimGrid(3, 1, 1);
dim3 dimBlock(5, 1, 1);
Kernel_Function <<<dimGrid, dimBlock>>> (…);
```

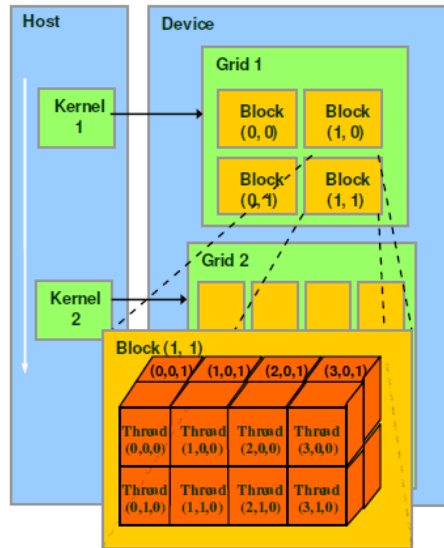
Οι δύο πρώτες εντολές δίνουν αρχικές τιμές στις παραμέτρους εκκίνησης. Επειδή το πλέγμα και τα μπλοκ είναι μονοδιάστατοι πίνακες γι’ αυτό χρησιμοποιούνται μόνο οι δύο πρώτες διαστάσεις των *dimGrid* και *dimBlock* ενώ οι άλλες διαστάσεις παίρνουν την τιμή *1*. Η τρίτη εντολή είναι η πραγματική εκκίνηση του πυρήνα και οι παράμετροι εκτέλεσης τοποθετούνται μέσα στα σύμβολα <<< και >>>. Το ίδιο πλέγμα θα μπορούσε επίσης να ξεκινήσει και με μία μόνο εντολή:

```
KernelFunction <<<3, 5>>> (…);
```

Οι τιμές των *dimGrid.x* και *dimBlock.y* μπορούν να κυμαίνονται από 1 ως 65535 όμως μετά από την εκκίνηση του πυρήνα οι διαστάσεις του πλέγματος δεν μπορούν να αλλάξουν. Όλα τα νήματα ενός μπλοκ μοιράζονται την ίδια τιμή *blockIdx.x* η οποία κυμαίνεται από 0 ως *dimGrid.x-1*, και η τιμή *blockIdx.y* μεταξύ 0 και *dimGrid.y-1*. Εάν υποθέσουμε λοιπόν ότι έχουμε το παρακάτω τμήμα κώδικα

```
dim3 dimGrid(2, 2, 1);
dim3 dimBlock(4, 2, 2);
KernelFunction<<<dimGrid, dimBlock>>> (…);
```

θα έχουμε ένα πλέγμα πολλών διαστάσεων σαν και αυτό που φαίνεται στην Εικ. 3.5.



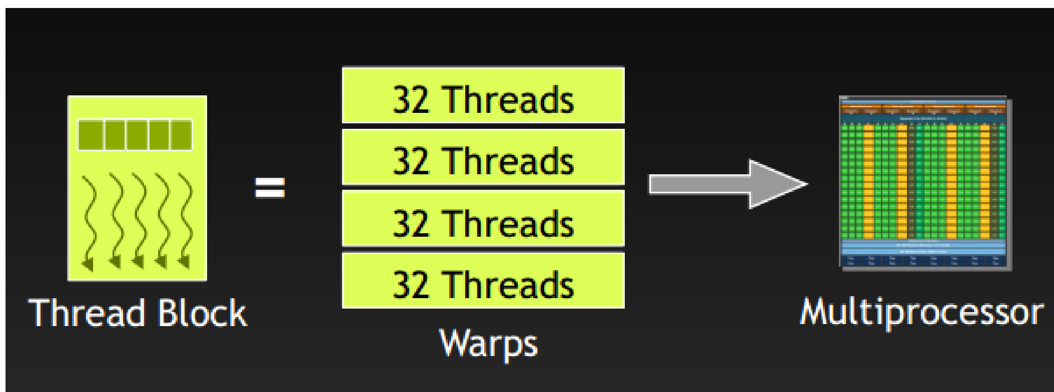
Εικόνα 3.5: Παράδειγμα πλέγματος πολλών διαστάσεων στην CUDA

Το πλέγμα αποτελείται από 4 μπλοκ οργανωμένα σε έναν πίνακα 2x2. Κάθε μπλοκ στην Εικ. 3.5 χαρακτηρίζεται με ένα ζεύγος (*blockIdx.x, blockIdx.y*) όπως για παράδειγμα το μπλοκ (0,1) έχει τιμές *blockIdx.x = 0* και *blockIdx.y = 1*. Τα μπλοκ όπως είδαμε και νωρίτερα είναι οργανωμένα ως τρισδιάστατοι πίνακες νημάτων και όλα τα μπλοκ ενός πλέγματος έχουν τις ίδιες διαστάσεις. Κάθε νήμα αποτελείται από τρεις συνιστώσες: τη συντεταγμένη x *threadIdx.x*, τη συντεταγμένη y *threadIdx.y* και τη συντεταγμένη z *threadIdx.z*. Ο αριθμός των νημάτων σε κάθε διάσταση ενός μπλοκ καθορίζεται από τη δεύτερη παράμετρο εκτέλεσης που δίνεται κατά την εκκίνηση του πυρήνα. Αυτή η παράμετρος μπορεί να προσπελαστεί και ως εκ των προτέρων ορισμένη μεταβλητή *dimGrid* τύπου *struct*. Το συνολικό μέγεθος ενός μπλοκ δεν μπορεί να είναι πάνω από 512 νήματα στις παλιότερες κάρτες γραφικών (1024 και παραπάνω στις νεότερες) γι' αυτό και διαστάσεις όπως (32,32,1) δεν θεωρούνται επιτρεπτές καθ' ότι 32*32=1024 νήματα στο κάθε μπλοκ.

Τέλος τα νήματα στην Εικ. 3.5 είναι οργανωμένα σε πίνακες νημάτων 4x2x2. Επειδή όλα τα μπλοκ μέσα σε ένα πλέγμα είναι ίδιων διαστάσεων θα αναφερθούμε σε ένα από αυτά. Το μπλοκ (1,1) έχει αναπτυχθεί για να φανούν τα 16 νήματα που περιέχει όπως για παράδειγμα το νήμα (2,1,0) έχει *threadIdx.x = 2*, *threadIdx.y = 1* και *threadIdx.z = 0*. Το συγκεκριμένο παράδειγμα είναι ένα πάρα πολύ απλό παράδειγμα για

να διευκολύνει την μελέτη των νημάτων. Στην πραγματικότητα όμως τα τυπικά πλέγματα της CUDA αποτελούνται από χιλιάδες έως εκατομμύρια νήματα.

Η συσκευή λοιπόν για να πετυχαίνει καλύτερη οργάνωση όλων αυτών των εκατομμυρίων νημάτων τα ομαδοποιεί σε ομάδες των 32 κάθε ομάδα από τις οποίες ονομάζεται *στημόνι* (warp). Η GPU λοιπόν χρησιμοποιεί τα στημόνια για να ομαδοποιεί την εκτέλεση πολλαπλών εντολών σε μία μόνο δέσμη νημάτων. Αυτό σημαίνει πως η GPU χρονοπρογραμματίζει τις εντολές σε warps τα οποία αναθέτει στους πολυεπεξεργαστές. Τα στημόνια από τη στιγμή που θα ανατεθούν εκτελούνται μέσα στον κάθε πολυεπεξεργαστή παράλληλα (SIMD) πράγμα το οποίο σημαίνει πως ο κάθε πολυεπεξεργαστής πρέπει να προγραμματιστεί κατάλληλα έτσι ώστε να μην υπάρχει ο κίνδυνος σφάλματος. Η Εικ. 3.6 περιγράφει και οπτικά την έννοια του στημονιού.



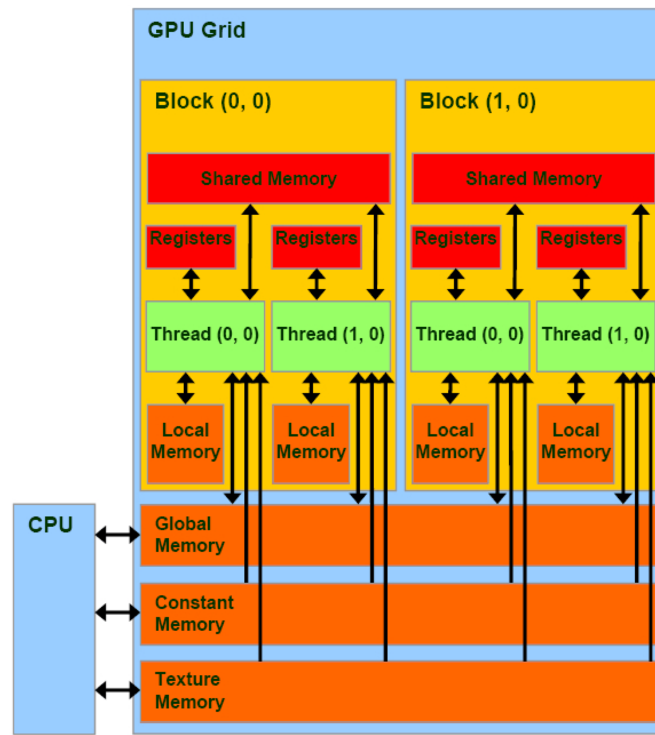
Εικόνα 3.6: Νήματα ενός μπλοκ οργανωμένα σε στημόνια

3.3.3 Μνήμες CUDA

Η CUDA υποστηρίζει πολλούς τύπους μνήμης που μπορούν να χρησιμοποιήσουν οι προγραμματιστές για να επιτύχουν μεγάλες ταχύτητες εκτέλεσης των πυρήνων τους. Στην Εικ. 3.7 μπορούμε να δούμε αυτές τις μνήμες.

Στην CUDA, η CPU υπηρεσίας και οι συσκευές GPU έχουν ξεχωριστούς χώρους μνήμης. Γι' αυτό και για να εκτελέσει ο προγραμματιστής έναν πυρήνα σε μία συσκευή, πρέπει να κάνει κατανομή μνήμης στη συσκευή και να μεταφέρει δεδομένα από τη μνήμη της CPU προς την εκχωρημένη μνήμη στη συσκευή. Με τον ίδιο τρόπο όταν τελειώσει η εκτέλεση στη συσκευή, ο προγραμματιστής πρέπει να μεταφέρει τα δεδομένα του αποτελέσματος από την μνήμη της συσκευής ξανά στη μνήμη της CPU και τελικά να αποδεσμεύσει το χώρο της συσκευής που δεν χρειάζεται πια. Το API (Application Programming Interface) της CUDA διαθέτει πολλές τέτοιες συναρτήσεις

διασύνδεσης προγραμματισμού εφαρμογών που εκτελούν αυτές τις δραστηριότητες για λογαριασμό του χρήστη.



Εικόνα 3.7: Γενική άποψη μνημών μέσα σε μια συσκευή GPU

Στο κάτω μέρος της Εικ. 3.7 βλέπουμε την καθολική μνήμη (global memory), τη μνήμη σταθερών (constant memory) και τη μνήμη υφών (texture memory). Αυτοί οι τύποι μνήμης είναι που επιτρέπουν τη μετακίνηση δεδομένων από και προς τη CPU όπως φαίνεται και από τα βέλη διπλής κατεύθυνσης μεταξύ τους και βρίσκονται έξω από το τσιπ της κάρτας γραφικών.

Η Καθολική μνήμη (Global Memory)

Η καθολική μνήμη είναι η πιο βασική μνήμη σε μια συσκευή GPU και αποτελεί ένα μέρος της μνήμης DRAM της συσκευής. Θα μπορούσε κανείς να την αντιστοιχίσει με τη μνήμη RAM του υπολογιστή. Σε σχέση με τις υπόλοιπες μνήμες της συσκευής είναι η πιο μεγάλη αλλά και η πιο αργή. Αν όμως τη συγκρίνει κανείς με την μνήμη RAM του υπολογιστή θα διαπιστώσει πως είναι σχεδόν δεκαπλάσια σε ταχύτητα. Για να μπορέσει λοιπόν η κάρτα γραφικών μας να δει τα δεδομένα που πρέπει να επεξεργαστεί θα πρέπει αυτά να μεταφερθούν μέσα στην καθολική μνήμη. Σε επόμενη ενότητα θα

γίνει αναφορά στις συναρτήσεις που βοηθούν σε αυτήν την μετακίνηση των δεδομένων από τη CPU προς την καθολική μνήμη και αντίστροφα.

Η Μνήμη σταθερών (Constant Memory)

Η μνήμη σταθερών είναι το αντίστοιχο του ορισμού *const* της μνήμης ενός υπολογιστή, περιέχει δηλαδή κάποια σταθερή μεταβλητή. Η μνήμη σταθερών υποστηρίζει την προσπέλαση μόνο για ανάγνωση με μικρό λανθάνοντα χρόνο και υψηλό εύρος ζώνης από την (GPU), όταν όλα τα νήματα προσπελάζουν την ίδια θέση μνήμης ταυτόχρονα. Θα μπορούσαμε π.χ. να περάσουμε τις διαστάσεις ενός πλέγματος σε αυτήν την μνήμη. Η μνήμη σταθερών είναι και αυτή κομμάτι της DRAM της κάρτας γραφικών. Είναι πολύ γρήγορη καθώς είναι cached, είναι βελτιστοποιημένη για broadcasting (εκπομπή) και το κόστος ανάγνωσης της είναι αμελητέο εφόσον τηρούνται κάποιοι κανόνες. Το μέγεθος της είναι μόλις 8 Kbyte για όλη την κάρτα γραφικών.

Η μνήμη υφής (Texture Memory)

Η μνήμη υφής είναι αρκετά διαφορετική από τις υπόλοιπες μνήμες που γνωρίζουμε. Οι πολυεπεξεργαστές διαθέτουν ειδικά κυκλώματα για την επεξεργασία υφών τα οποία μπορούν να χρησιμοποιηθούν προς όφελος του προγραμματιστή. Πίσω από την μνήμη υφής υπάρχει καθολική μνήμη η οποία είναι «δεμένη» με τα κυκλώματα αυτά και σε αντίθεση με την καθολική μνήμη είναι ορατή σε όλα τα νήματα τα οποία μπορούν να τη διαβάσουν μέσω των κυκλωμάτων αυτών.

Μέσα στο τσιπ υπάρχουν άλλοι τύποι μνημών οι οποίες μπορούν να επικοινωνούν μεταξύ τους αλλά και με τις μνήμες που είδαμε και παραπάνω. Τέτοιες μνήμες είναι οι καταχωρητές (registers), η τοπική μνήμη (local memory) και η κοινόχρηστη μνήμη (shared memory) που θα αναλύσουμε στη συνέχεια. Η προσπέλαση των μεταβλητών που βρίσκονται σε τέτοιους τύπους μνήμης μπορεί να γίνει ταχύτατα με εξαιρετικά παράλληλο τρόπο.

Καταχωρητές (Registers)

Οι καταχωρητές κατανέμονται σε μεμονωμένα νήματα. Κάθε νήμα μπορεί να προσπελάσει μόνο τους δικούς του καταχωρητές. Μια συνάρτηση πυρήνα χρησιμοποιεί

συνήθως καταχωρητές για να αποθηκεύει μεταβλητές που προσπελάζονται συχνά και είναι ιδιωτικές (private) σε κάθε νήμα.

Τοπική μνήμη (Local Memory)

Η τοπική μνήμη είναι η προσωπική μνήμη κάθε νήματος. Κανένα άλλο νήμα δεν έχει πρόσβαση σε αυτή, επομένως οι συναλλαγές είναι πάντα γραμμικές και ευθυγραμμισμένες. Η τοπική μνήμη βρίσκεται στην DRAM της κάρτας γραφικών και έχει, επομένως, την ίδια ταχύτητα και υστέρηση με την καθολική μνήμη. Σπάνια χρησιμοποιούμε αυτή την μνήμη, γιατί δεν χρειάζεται σχεδόν ποτέ. Ουσιαστικά η τοπική μνήμη είναι η τελευταία διέξοδος όταν οι απαιτήσεις σε μνήμη κάθε νήματος είναι πολύ μεγάλες.

Κοινόχρηστη μνήμη (Shared memory)

Η κοινόχρηστη μνήμη κατανέμεται σε μπλοκ νημάτων και έχει τη ζωή του μπλοκ στο οποίο ορίζεται. Κάθε μεταβλητή στις θέσεις της κοινόχρηστης μνήμης ενός μπλοκ μπορεί να προσπελαστεί από όλα τα νήματα ενός μπλοκ αλλά όχι από τα νήματα ενός άλλου μπλοκ. Η κοινόχρηστη μνήμη αποτελεί έναν αποδοτικό τρόπο που επιτρέπει στα νήματα να συνεργάζονται, μέσω κοινής χρήσης των δεδομένων εισόδου τους και των ενδιάμεσων αποτελεσμάτων της εργασίας τους. Είναι η πιο γρήγορη μνήμη (Tbytes/s) αλλά είναι πολύ περιορισμένη σε μέγεθος και η χωρητικότητά της σε κάθε πολυεπεξεργαστή είναι μόλις 16 Kbyte. Ο προγραμματιστής μπορεί να κατανείμει την κοινόχρηστη μνήμη όπως θεωρεί εκείνος απαραίτητο σε κάθε μπλοκ αλλά έτσι κι αλλιώς η κοινόχρηστη μνήμη κατανέμεται ισόποσα και καθολικά επομένως αν ορίσουμε για ένα μπλοκ τη χωρητικότητα της κοινόχρηστης μνήμης το ίδιο θα ισχύει και για όλα.

3.3.4 Δήλωση μεταβλητών CUDA

Οι προγραμματιστές της CUDA, δηλώνοντας μια μεταβλητή CUDA σε έναν από τους τύπους μνήμης της CUDA, καθορίζουν την ορατότητα και την ταχύτητα προσπέλασης της μεταβλητής. Εκτός από αυτό όμως με κάθε τέτοια δήλωση ορίζεται και η εμβέλεια της μεταβλητής καθώς επίσης και ο χρόνος ζωής της. Η εμβέλεια καθορίζει το εύρος των νημάτων που μπορούν να την προσπελάσουν (ένα νήμα, τα νήματα ενός μπλοκ ή και όλα τα νήματα του πλέγματος). Ο χρόνος ζωής καθορίζει το τμήμα χρόνου

εκτέλεσης του προγράμματος κατά το οποίο η μεταβλητή θα είναι διαθέσιμη για χρήση. Αν δηλαδή μια μεταβλητή δηλωθεί μέσα στο σώμα της συνάρτησης πυρήνα τότε ο χρόνος ζωής της είναι μέσα στην διάρκεια κλήσης του πυρήνα και είναι διαθέσιμη για χρήση μόνο από τον πυρήνα. Αν όμως δηλωθεί έξω από οποιοδήποτε σώμα συνάρτησης τότε η διάρκειά της είναι η διάρκεια εκτέλεσης όλης της εφαρμογής και είναι διαθέσιμη σε οποιονδήποτε πυρήνα.

Πίνακας 3.1: Προσδιοριστικά τύπου για μεταβλητές CUDA

Δήλωση μεταβλητής	*Μνήμη*	*Εμβέλεια*	*Χρόνος ζωής*
Αυτόματες μεταβλητές εκτός από πίνακες	Καταχωρητής	Νήμα	Πυρήνας
Αυτόματες μεταβλητές πίνακα	Τοπική	Νήμα	Πυρήνας
`__device__,__shared__, int svar;`	Κοινόχρηστη	Μπλοκ	Πυρήνας
`__device__ int gvar;`	Καθολική	Πλέγμα	Εφαρμογή
`__device__, __constant__ int cvar;`	Σταθερών	Πλέγμα	Εφαρμογή

Οι μεταβλητές που δεν είναι πίνακες ονομάζονται *βαθμωτές μεταβλητές* (scalar variables) και η εμβέλειά τους είναι σε μεμονωμένα νήματα. Όταν καλείται μία συνάρτηση πυρήνα δημιουργείται ένα ιδιωτικό αντίγραφο της μεταβλητής αυτής για κάθε νήμα που εκτελεί τη συνάρτηση πυρήνα. Όταν το νήμα τερματιστεί καταργούνται και οι αυτόματες μεταβλητές του.

Οι *αυτόματες μεταβλητές πίνακα* δεν αποθηκεύονται σε καταχωρητές αλλά στην καθολική μνήμη και προκαλούν μεγάλες καθυστερήσεις. Η εμβέλειά τους είναι σαν και αυτή των βαθμωτών μεταβλητών αλλά η χρήση τους είναι πολύ σπάνια.

Η δήλωση μίας μεταβλητής ως __shared__ αποτελεί μια δήλωση μεταβλητής στην κοινόχρηστη μνήμη ενώ η δεσμευμένη λέξη __device__ μπροστά από την λέξη __shared__ δεν έχει καμία διαφορά. Μια τέτοια δήλωση γίνεται συνήθως μέσα σε μία συνάρτηση πυρήνα ή μία συνάρτηση συσκευής και η εμβέλεια της είναι μέσα σε ένα ολόκληρο μπλοκ νημάτων, δηλαδή όλα τα νήματα βλέπουν αυτή τη μεταβλητή και παράλληλα όσα μπλοκ νημάτων υπάρχουν τόσα αντίγραφά της δημιουργούνται για να εξυπηρετήσουν όλα τα μπλοκ. Ο χρόνος ζωής της είναι η διάρκεια εκτέλεσης του πυρήνα και όταν αυτός τελειώσει τα περιεχόμενα της κοινόχρηστης μνήμης παύουν να υπάρχουν. Η προσπέλασή της είναι ταχύτατη και εξαιρετικά παράλληλη και αυτός είναι και ο λόγος

που οι προγραμματιστές πολύ συχνά μεταφέρουν τα πιο συχνά χρησιμοποιούμενα δεδομένα από την καθολική μνήμη στην κοινόχρηστη μνήμη.

Η δήλωση μιας μεταβλητής ως __constant__ χρησιμοποιώντας και πάλι προαιρετικά τη δεσμευμένη λέξη __device__ από μπροστά έχει ως αποτέλεσμα τη δήλωση μιας σταθερής μεταβλητής. Οι δηλώσεις σταθερών μεταβλητών πρέπει να βρίσκονται έξω από κάθε σώμα συνάρτησης και η εμβέλειά της είναι όλα τα νήματα σε όλα τα πλέγματα, πράγμα το οποίο σημαίνει πως όλα τα νήματα βλέπουν ακριβώς την ίδια έκδοση της ίδιας μεταβλητής. Έτσι λοιπόν είναι λογικό ο χρόνος ζωής της να είναι όλη η διάρκεια εκτέλεσης του προγράμματος. Είναι επίσης εξαιρετικά ταχύτατη και παράλληλη και το μέγεθός της σε μία εφαρμογή περιορίζεται στα 65536 byte.

Αν προηγείται μόνο η δεσμευμένη λέξη __device__ για τη δήλωση μιας μεταβλητής τότε η μεταβλητή είναι δηλωμένη στην καθολική μνήμη όπου οι προσπελάσεις μπορεί να είναι πολύ αργές αλλά με πλεονέκτημα ότι όλα τα νήματα όλων των πυρήνων έχουν πρόσβαση σε αυτή. Τα περιεχόμενα της διατηρούνται σε όλη τη διάρκεια εκτέλεσης δίνοντας έτσι την ευκαιρία σε νήματα διαφορετικών μπλοκ να μπορούν να συνεργάζονται μεταξύ τους. Πρέπει να τονιστεί πως προς το παρόν δεν υπάρχει άλλος τρόπος επικοινωνίας και συγχρονισμού των νημάτων διαφορετικών μπλοκ εκτός από την καθολική μνήμη, εκτός από τον τερματισμό της εκτέλεσης του τρέχοντος πυρήνα.

Τέλος οι μεταβλητές υφών που δεν αναφέρονται στον Πίν. 3.1 έχουν μια εντελώς διαφορετική φιλοσοφία στη δήλωσή τους. Δηλώνονται μόνο εκτός συνάρτησης αλλά και μόνο στην επικεφαλίδα του αρχείου που περιέχει τον kernel που θα τις χρησιμοποιήσει. Η εμβέλεια τους είναι τόση όση και η διάρκεια ζωής του kernel. Ο ορισμός ενός texture πριν τον kernel. Έχει την μορφή:

```
Texture < Type, Dim, ReadMode > texName;
```

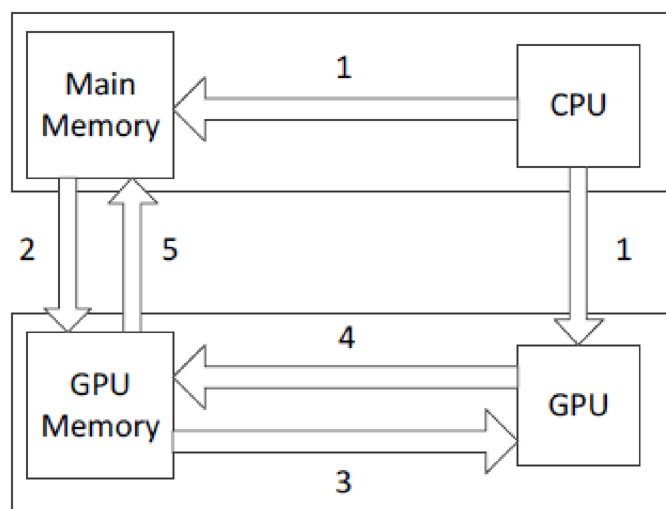
όπου *Type* είναι ο τύπος της μεταβλητής *texName* (int, float κτλ), στο πεδίο *Dim* ορίζεται ο αριθμός των επιτρεπτών διαστάσεων του texture (1-3) και το πεδίο *ReadMode* ορίζει πως θα δίνονται οι συντεταγμένες αυτής της μεταβλητής υφής (*cudaReadMode, NormalizedFloat ή cudaReadModeElementType*).

3.4 Το προγραμματιστικό μοντέλο και το API της CUDA

Ο κώδικας που πρόκειται να εκτελεστεί στην GPU γράφεται σε ένα αρχείο με κατάληξη .cu. και μεταγλωττίζεται με τον nvcc (NVIDIA CUDA Compiler). Ο κώδικας του αρχείου αυτού περιλαμβάνει τμήματα που θα εκτελεστούν στην CPU και τμήματα που θα εκτελεστούν στην GPU. Τα τμήματα που θα εκτελεστούν στην GPU είναι υποχρεωμένος να τα αναγνωρίσει ο μεταγλωττιστής της CUDA. Μπροστά από τον ορισμό μιας συνάρτησης υπάρχει ειδική σήμανση με δεσμευμένες λέξεις που δηλώνει από πού καλείται η συνάρτηση και πού εκτελείται:

- Αν η συνάρτηση είναι __global__ καλείται από την CPU και εκτελείται στην GPU (αυτός είναι και ο λεγόμενος kernel)
- Αν η συνάρτηση είναι __device__ καλείται μόνο από την GPU και εκτελείται επίσης στην GPU
- Αν η συνάρτηση δεν έχει καμία δήλωση ή την δήλωση __host__ τότε καλείται από τη CPU και εκτελείται και στη CPU.

Ο βασικός τρόπος λειτουργίας των προγραμμάτων είναι η μεταφορά των δεδομένων από την κύρια μνήμη του συστήματος στη μνήμη της GPU, η παράλληλη εκτέλεση τους στην GPU και η επιστροφή τους πίσω στην κύρια μνήμη. Στο Σχ. 3.1 φαίνεται αυτή η διαδικασία.



Σχήμα 3.1: Βασική λειτουργία της CUDA

Αρχικά η CPU οργανώνει τα δεδομένα της και τα μεταβιβάζει στην κύρια μνήμη (host). Παράλληλα η CPU δίνει εντολή για μεταφορά δεδομένων από την κύρια μνήμη

στην GPU (device) όπου εδώ πρέπει να σημειωθεί πως αυτή η διαδικασία καθυστερεί λίγο τη συνολική ταχύτητα. Έπειτα προσδιορίζονται κάποιες ρυθμίσεις για την εκτέλεση του kernel όπως είναι η οργάνωση των δεδομένων εισόδου και η δέσμευση των block. Πολλές φορές ιδιαίτερα βοηθητική διαδικασία είναι και η μεταφορά δεδομένων από την καθολική στην κοινόχρηστη μνήμη αλλά δεν είναι απαραίτητο πάντα. Στη συνέχεια η GPU εκτελεί παράλληλα τον κώδικα που της ανατέθηκε με τα δεδομένα που της μεταβιβάστηκαν και τέλος τα αποτελέσματα της εκτέλεσης του kernel βρίσκονται στη μνήμη της GPU και μεταβιβάζονται μετά το τέλος της εκτέλεσης πίσω στην κύρια μνήμη του συστήματος.

Για να μεταφερθούν τα δεδομένα από την κύρια μνήμη στην καθολική μνήμη της GPU υπάρχουν συναρτήσεις API που υποστηρίζονται από το μοντέλο της CUDA και αναλαμβάνουν αυτές τις διαδικασίες. Η μεταφορά των δεδομένων αποτελείται από τρία βασικά βήματα για καθένα από τα οποία υπάρχει και η κατάλληλη συνάρτηση.

Το πρώτο βήμα ορίζει πως προτού γίνει η μεταφορά θα πρέπει να δεσμευτεί εκ των προτέρων συγκεκριμένος χώρος που θα λάβει τα δεδομένα. Αυτό επιτυγχάνεται με τη συνάρτηση:

- ❖ `cudaMalloc((void**)&xx, size);`

Η πρώτη παράμετρος της συνάρτησης cudaMalloc() είναι η διεύθυνση μιας μεταβλητής τύπου δείκτη που πρέπει να δείχνει στο κατανεμημένο αντικείμενο μετά από την κατανομή. Η διεύθυνση της μεταβλητής δείκτη πρέπει να μετατραπεί σε τύπο (void**) επειδή η συνάρτηση αναμένει μια τιμή γενικού δείκτη. Η συνάρτηση cudaMalloc() είναι μια γενική συνάρτηση που δεν περιορίζεται σε κάποιο συγκεκριμένο τύπο αντικειμένων. Η δεύτερη παράμετρος δίνει το μέγεθος του αντικειμένου προς κατανομή σε byte. Παράδειγμα:

```
float *a_d;
int size = b*c*sizeof(float);
cudaMalloc((void**) &a_d, size);
…
```

Μετά από τον υπολογισμό η μνήμη που έχει δεσμευθεί στην καθολική μνήμη πρέπει να αποδεσμευθεί. Αυτό γίνεται με τη συνάρτηση:

- `cudaFree(a_d);`

η οποία δέχεται μόνο ένα όρισμα και αυτό είναι το όνομα του αντικειμένου που πρέπει να αποδεσμεύσει τη μνήμη. Εδώ πρέπει να σημειώσουμε πως όταν αναφερόμαστε σε

μεταβλητές συσκευής είναι καλό να τερματίζομε το όνομα της μεταβλητής με το γράμμα d (ως ένδειξη του device).

Ενδιάμεσα όμως από τη δέσμευση και την αποδέσμευση μνήμης τα δεδομένα πρέπει να μεταφερθούν από τη συσκευή στην GPU. Αυτό είναι δουλειά της συνάρτησης:

- `cudaMemcpy()`

η οποία δέχεται τέσσερεις παραμέτρους. Η πρώτη παράμετρος είναι ένας δείκτης προς τη θέση προορισμού της λειτουργίας αντιγραφής. Η δεύτερη παράμετρος δείχνει στο αντικείμενο προέλευσης των δεδομένων που θα αντιγραφεί. Η τρίτη παράμετρος καθορίζει τον αριθμό των bytes που θα αντιγραφούν και η τέταρτη παράμετρος δείχνει τους τύπους μνήμης που εμπλέκονται στην αντιγραφή καθώς και την κατεύθυνση της αντιγραφής

- `cudaMemcpyHostToHost` (από την CPU στην CPU)
- `cudaMemcpyHostToDevice` (από την CPU στην GPU)
- `cudaMemcpyDeviceToHost` (από την GPU στην CPU)
- `cudaMemcpyDeviceToDevice` (από την GPU στην GPU)

Οι δύο από αυτές είναι οι συνηθέστερες και είναι αυτές που μεταφέρουν δεδομένα μεταξύ των διαφορετικών μνημών.

- `cudaMemcpy(a_d, a, size, cudaMemcpyHostToDevice)`
- `cudaMemcpy(b, b_a, size, cudaMemcpyDeviceToHost)`

Όταν ο κώδικας υπηρεσίας καλεί έναν πυρήνα, ορίζει τις διατάσεις του πλέγματος και των μπλοκ νημάτων με τη βοήθεια των μεταβλητών *struct* τύπου *dim3* όπως είδαμε και νωρίτερα.

Η δήλωση ενός kernel γίνεται ως εξής:

- `__global__ void KernelFunc(…);`

ενώ καλείται με

- `dim3 dimBlock(…, …, …);`
- `dim3 dimGrid(…, …);`
- `KernelFunc<<<dimGrid, dimBlock>>> (…);`

Η τελευταία γραμμή καλεί τον πυρήνα που θα εκτελεστεί από την GPU.

Επίσης είναι σημαντικό να συγχρονίζονται τα νήματα όταν τελειώνουν από τη μία φάση εκτέλεσης πριν περάσουν στην επόμενη. Αυτό επιτυγχάνεται με την κλήση της συνάρτησης

- `__synchthreads();`

η οποία μπορεί να γίνει μόνο σε επίπεδο μπλοκ και όχι σε επίπεδο πλέγματος.

Οι συσκευές CUDA πολλές φορές μπορεί να είναι παραπάνω από μία μέσα σε έναν υπολογιστή. Η CPU μπορεί να αναζητήσει αυτές τις συσκευές και να επιλέξει ποια θέλει να χρησιμοποιηθεί. Αρχικά μπορεί να ρωτήσει και να της επιστραφεί ο αριθμός των διαθέσιμων GPU

- `cudaGetDeviceCount (int *count);`

όπου στη μεταβλητή count επιστρέφεται ο αριθμός των devices. Έπειτα μπορεί να επιλέξει μία από όλες τις GPU που διαθέτει ώστε σε αυτήν να εκτελούνται τα νήματα που καλεί η host συσκευή

- `cudaSetDevice (int device);`

Όπου η μεταβλητή device περιέχει τον αριθμό της κάρτας που έχει επιλεγεί. Αν υπάρχει μόνο μία τότε προεπιλογή είναι το μηδέν (0). Ακόμη μπορεί να ρωτήσει και να ενημερωθεί για το ποια συσκευή GPU χρησιμοποιείται αυτή τη στιγμή

- `cudaGetDevice (int *current_device);`

όπου στη μεταβλητή current_device επιστρέφεται ο αριθμός της κάρτας που είναι τώρα σε χρήση. Μπορεί επίσης να ερωτηθεί για τις ιδιότητες της τρέχουσας κάρτας

- `cudaGetDeviceProperties (cudaDeviceProp* prop, int device);`

όπου prop είναι οι ιδιότητες της συγκεκριμένης συσκευής και device ο αριθμός της συσκευής για την οποία θέλουμε τις ιδιότητες και τέλος μπορεί να ρωτήσει και να λάβει σαν απάντηση τον αριθμό της κάρτας που είναι πιο κατάλληλη για χρήση στο συγκεκριμένο κομμάτι κώδικα

- `cudaChooseDevice (int *device, cudaDeviceProp* prop);`

όπου device είναι και πάλι ο αριθμός της πιο κατάλληλης κάρτας και prop οι επιθυμητές ιδιότητες της κάρτας.

Υπάρχουν πάρα πολλές συναρτήσεις μέσα στο API της CUDA οι οποίες είναι προφανές πως δεν μπορούν να αναφερθούν μέσα στα πλαίσια μιας πτυχιακής εργασίας. Αναφέρθηκαν παραπάνω λίγες από τις πιο σημαντικές και συνηθισμένες όπως επίσης και από αυτές που χρησιμοποιήθηκαν στην ανάπτυξη του κώδικα στα παραρτήματα. Περισσότερες συναρτήσεις μπορεί κανείς να βρει στο πλήρες API της CUDA.

3.5 CUDA Occupancy Calculator

Κατά την εγκατάσταση του Toolkit στη διαδικασία εγκατάστασης της CUDA μέσα στον φάκελο

```
~/NVIDIA_CUDA-5.5_Samples/NVIDIA_CUDA-5.5_Samples/tools
```

βρίσκεται ένα αρχείο .xls το οποίο ονομάζεται CUDA_Occupancy_Calculator.xls.

Ο μετρητής πληρότητας της CUDA (CUDA Occupancy Calculator) όπως δηλώνεται και από το όνομά του επιτρέπει στον χρήστη να υπολογίσει την πληρότητα του πολυεπεξεργαστή (SM) μίας GPU μέσω ενός συγκεκριμένου πυρήνα. Η πληρότητα των πολυεπεξεργαστών είναι η αναλογία των ενεργών στημονιών προς το μέγιστο αριθμό στημονιών που μπορεί να υποστηρίξει ένας πολυεπεξεργαστής GPU. Κάθε πολυεπεξεργαστής στη συσκευή έχει ένα σετ από N καταχωρητές διαθέσιμους για χρήση από τα νήματα των CUDA προγραμμάτων. Αυτοί οι καταχωρητές είναι ένας κοινόχρηστος πόρος που κατανέμεται μεταξύ των μπλοκ των νημάτων που εκτελούνται στους πολυεπεξεργαστές. Ο μεταγλωττιστής της CUDA προσπαθεί να ελαχιστοποιήσει τη χρήση αυτών των καταχωρητών για να μεγιστοποιήσει τον αριθμό των ενεργών νημάτων μέσα σε ένα μηχάνημα που λειτουργούν ταυτόχρονα. Εάν ένα πρόγραμμα προσπαθήσει να ξεκινήσει έναν πυρήνα για τον οποίο οι καταχωρητές που χρησιμοποιούνται ανά νήμα σε όλο το μπλοκ των νημάτων είναι μεγαλύτεροι από N τότε ο πυρήνας θα αποτύχει.

Το μέγεθος N σε GPUs με υπολογιστική δυναμικότητα 1.0-1.1 είναι 8192 32-bit καταχωρητές ανά πολυεπεξεργαστή. Στις GPU με υπολογιστική δυναμικότητα 1.2-1.3 το N είναι 16384. Στις GPU με υπολογιστική δυναμικότητα 2.0-2.1 το N είναι 32768 και τέλος σε αυτές με υπολογιστική δυναμικότητα 3.0 και πάνω το N είναι 65536.

Η μεγιστοποίηση της πληρότητας μπορεί να βοηθήσει να καλυφθεί κάποιος λανθάνοντας χρόνος κατά τη διάρκεια φόρτωσης της καθολικής μνήμης που ακολουθείται από __syncthreads(). Η πληρότητα καθορίζεται από πολλούς παράγοντες όπως:

1) Τους καταχωρητές που χρησιμοποιεί το κάθε νήμα. (Οι καταχωρητές του κάθε SM διαμοιράζονται στα νήματα που φιλοξενεί ο SM).
2) Την κοινόχρηστη μνήμη ανά μπλοκ νημάτων. (Η κοινόχρηστη μνήμη ενός SM διαμοιράζεται στα μπλοκ που φιλοξενεί ο SM).
3) Τα νήματα ανά μπλοκ. (Τα νήματα κατανέμονται σε διακριτά μπλοκ νημάτων).

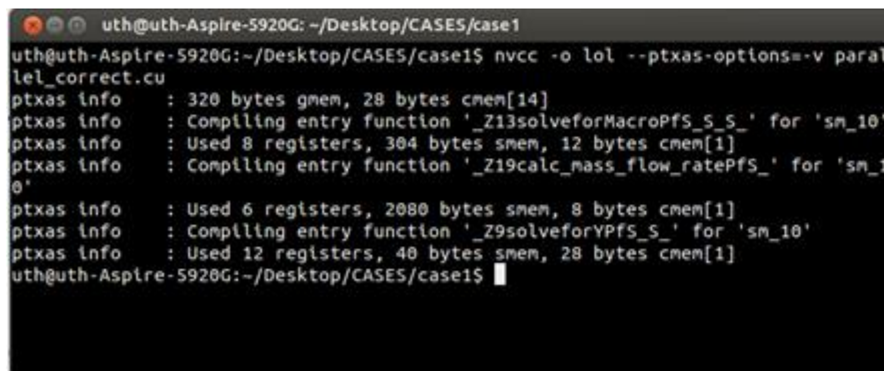
Εξαιτίας αυτού οι προγραμματιστές πρέπει να επιλέξουν το μέγεθος του μπλοκ των νημάτων έτσι ώστε να μεγιστοποιήσουν την πληρότητα. Αυτός λοιπόν ο μετρητής πληρότητας της GPU μπορεί να βοηθήσει στην επιλογή του μεγέθους του μπλοκ των

νημάτων βασιζόμενος στην απαίτηση της κοινόχρηστης μνήμης και των καταχωρητών ανά πυρήνα.

Υψηλότερη πληρότητα δεν συνεπάγεται πάντα και υψηλότερη απόδοση. Παρόλα αυτά ένας πυρήνας ο οποίος έχει πολύ χαμηλή πληρότητα είναι δύσκολο να καλύψει λανθάνοντες χρόνους. Γι' αυτό το λόγο λοιπόν ο σημαντικότερος παράγοντας για τη μέγιστη πληρότητα ενός πυρήνα είναι η εύρεση του βέλτιστου αριθμού νημάτων ανά μπλοκ ανάλογα πάντα με τη δυναμικότητα της εκάστοτε GPU.

3.5.1 Καταχωρητές ανά νήμα και κοινόχρηστη μνήμη ανά μπλοκ

Για να δούμε τον αριθμό των καταχωρητών που χρησιμοποιούνται σε κάθε νήμα μέσα στον πυρήνα, απλά μεταγλωττίζουμε τον πυρήνα χρησιμοποιώντας την ετικέτα –ptxas-options=-v στον nvcc. Αυτό θα έχει ως εξαγόμενο πληροφορίες όσο αφορά τους καταχωρητές, την τοπική μνήμη, την κοινόχρηστη μνήμη και τη μνήμη σταθερών που χρησιμοποιήθηκαν για κάθε πυρήνα στο αρχείο .cu. Όμως αν ο πυρήνας δηλώνει κάποια εξωτερική κοινόχρηστη μνήμη η οποία δεσμεύεται δυναμικά, θα χρειαστεί να προσθέσουμε την (στατικά κατανεμημένη) κοινόχρηστη μνήμη που αναφέρθηκε από το ptxas στο ποσό που δυναμικά έχουμε κατανείμει κατά τη διάρκεια εκτέλεσης για να πάρουμε τη σωστή χρήση της κοινόχρηστης μνήμης. Ένα απλό παράδειγμα του ptxas εξαγόμενου είναι το ακόλουθο και τα αποτελέσματα του λειτουργούν σαν είσοδος δεδομένων για το Occupancy Calculator:



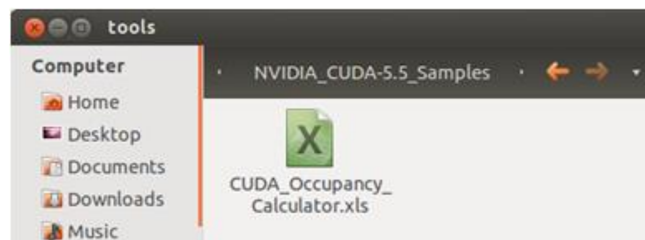
Εικόνα 3.8: Έξοδος ετικέτας –ptxas-options=-v

Η εφαρμογή χρησιμοποίησε συνολικά 320 bytes από την καθολική μνήμη και 28 bytes από τη μνήμη σταθερών για τις πράξεις που έκανε εκτός πυρήνα. Ο πρώτος πυρήνας που χρησιμοποιεί κοινόχρηστη μνήμη ονομάζεται solveforMacro και έχει χρησιμοποιήσει συνολικά 8 καταχωρητές ανά νήμα, 304 bytes κοινόχρηστη μνήμη ανά

μπλοκ και 12 bytes από τη μνήμη σταθερών. Ο δεύτερος πυρήνας που χρησιμοποίησε κοινόχρηστη μνήμη ονομάζεται calc_mass_flow_rate και έχει χρησιμοποιήσει 6 καταχωρητές ανά νήμα, 2080 bytes κοινόχρηστη μνήμη ανά μπλοκ και 8 bytes από τη μνήμη σταθερών. Τέλος ο πυρήνας με όνομα solveforY κάνει ελάχιστη χρήση της κοινόχρηστης μνήμης καθώς έχει δημιουργηθεί μόνο ένα μπλοκ νημάτων και χρησιμοποιεί 40 bytes μόνο από αυτήν, 12 καταχωρητές ανά νήμα και 28 bytes από την μνήμη σταθερών.

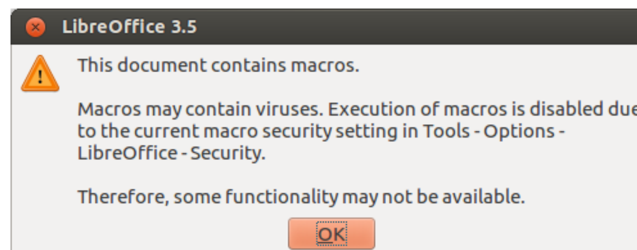
3.5.2 Οδηγίες χρήσης του CUDA Occupancy Calculator

Η χρήση του Calculator είναι μία πολύ απλή διαδικασία τριών βημάτων. Αρκεί να ανοίξει κανείς το αρχείο xls από το φάκελο που προαναφέραμε.



Εικόνα 3.9: Το αρχείο CUDA_Occupancy_Calculator στο φάκελο

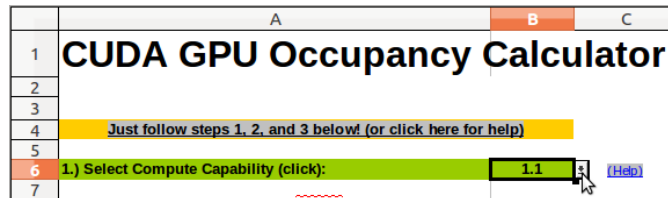
Για να ανοίξει και να λειτουργήσει σωστά ο Calculator πρέπει να ενεργοποιηθούν τα Macros του Excel οπότε στο αντίστοιχο παράθυρο που θα εμφανιστεί πριν ανοίξει το αρχείο πατάμε ΟΚ.



Εικόνα 3.10: Ενεργοποίηση των Macros του Excel

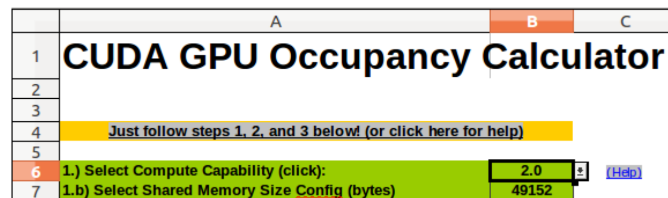
Με το άνοιγμα του Excel βρισκόμαστε στο φύλλο Calculator όπου και θα δουλέψουμε.

1) Αρχικά επιλέγουμε την υπολογιστική δυναμικότητα της συσκευής που διαθέτουμε από το πράσινο κελί.



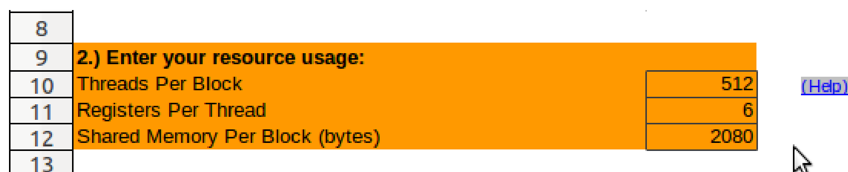
Εικόνα 3.11: Επιλογή δυναμικότητας της GPU

Εάν η δυναμικότητα της κάρτας μας το επιτρέπει θα δούμε ακόμα ένα πράσινο κελί από κάτω μέσα στο οποίο μπορούμε να επιλέξουμε το μέγεθος σε bytes της κοινόχρηστης μνήμης (το οποίο ρυθμίζεται κατά τη διάρκεια εκτέλεσης)



Εικόνα 3.12: Επιλογή μεγέθους της κοινόχρηστης μνήμης σε bytes

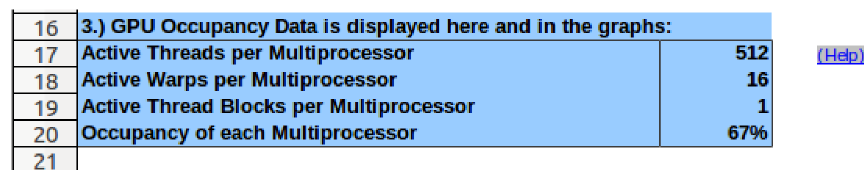
2) Έπειτα για τον πυρήνα τον οποίο θέλουμε να διερευνήσουμε εισάγουμε τον αριθμό των νημάτων ανά μπλοκ, τον αριθμό των καταχωρητών που χρησιμοποιούνται ανά νήμα και τη συνολική κοινόχρηστη μνήμη που χρησιμοποιείται από κάθε μπλοκ σε bytes μέσα στα πορτοκαλί κελιά.



8			
9	2.) Enter your resource usage:		
10	Threads Per Block	512	(Help)
11	Registers Per Thread	6	
12	Shared Memory Per Block (bytes)	2080	
13			

Εικόνα 3.13: Εισαγωγή παραμέτρων

3) Τέλος τα αποτελέσματα των υπολογισμών φαίνονται στο μπλε κελί.

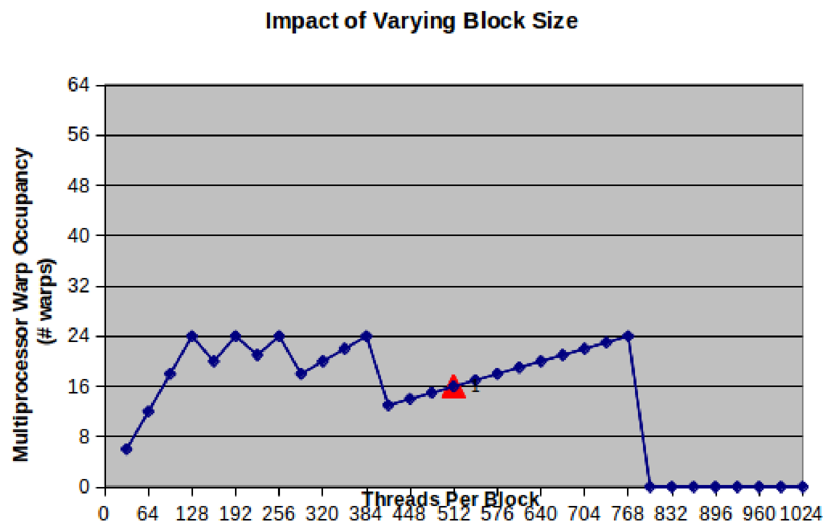


16	3.) GPU Occupancy Data is displayed here and in the graphs:		
17	Active Threads per Multiprocessor	512	(Help)
18	Active Warps per Multiprocessor	16	
19	Active Thread Blocks per Multiprocessor	1	
20	Occupancy of each Multiprocessor	67%	
21			

Εικόνα 3.14: Αποτελέσματα πληρότητας

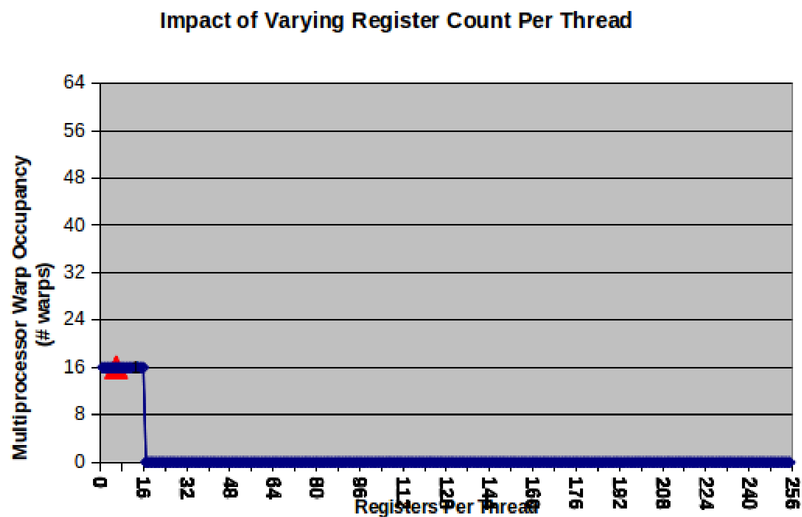
Αυτό θα μας πει την πληρότητα όπως επίσης και τον αριθμό των ενεργών νημάτων, των στημονιών και τον αριθμό των νημάτων ανά πολυεπεξεργαστή καθώς και τον μέγιστο αριθμό των ενεργών μπλοκ στην GPU.

Τα διαγράμματα από τη στιγμή που έχουμε επιλέξει εμείς την υπολογιστική δυναμικότητα που διαθέτουμε θα μας δείξουν τα εξής:



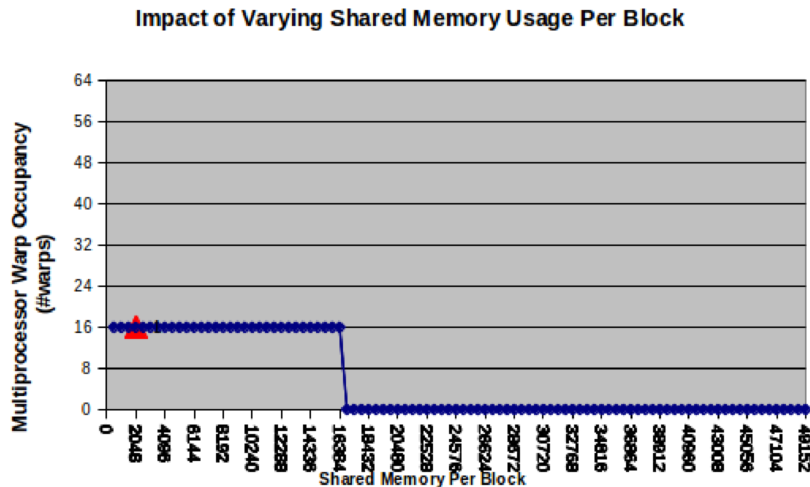
Εικόνα 3.15: Πληρότητα για το μέγεθος του μπλοκ

Αυτό που αλλάζει σύμφωνα με τις παραμέτρους του κάθε πυρήνα που έχουμε δώσει στα πορτοκαλί κελιά είναι ένα κόκκινο τρίγωνο, το οποίο στο πρώτο διάγραμμα θα μας δώσει την πληρότητα για το μέγεθος του μπλοκ που επιλέξαμε



Εικόνα 3.16: Αριθμός των καταχωρητών ανά πυρήνα

Στο δεύτερο διάγραμμα θα δούμε τον αριθμό των καταχωρητών ανά νήμα που χρησιμοποίησε ο πυρήνας ανάλογα με το τι δυνατότητα υπήρχε με βάση τη δυναμικότητα της κάρτας γραφικών και



Εικόνα 3.17: Κοινόχρηστη μνήμη που χρησιμοποίησε ο πυρήνας

Στο τρίτο διάγραμμα θα δούμε την κοινόχρηστη μνήμη που χρησιμοποίησε ο πυρήνας σε σχέση με το πόση διαθέσιμη είχε με βάση τη δυναμικότητα της κάρτας γραφικών.

3.5.3 Μερικές σημειώσεις για το CUDA Occupancy Calculator

Όπως αναφέραμε και παραπάνω υψηλότερη πληρότητα δεν σημαίνει απαραίτητα και υψηλότερη απόδοση. Εάν ένας πυρήνας δεν είναι περιορισμένος όσο αφορά το εύρος ζώνης ή τους λανθάνοντες χρόνους, τότε η αύξηση της πληρότητας δεν θα αποφέρει και καλύτερη απόδοση. Εάν ένα πλέγμα μέσα σε έναν πυρήνα τρέχει το λιγότερο ένα μπλοκ νημάτων ανά πολυεπεξεργαστή σε μία GPU, και είναι συμφορημένο από υπολογισμούς και όχι από πρόσβαση στην καθολική μνήμη, τότε η αύξηση της πληρότητας μπορεί να μην έχει και κανένα απολύτως αποτέλεσμα. Στην πραγματικότητα το να γίνονται αλλαγές μόνο και μόνο με σκοπό την αύξηση της πληρότητας μπορεί να επιφέρει άλλα αποτελέσματα, όπως επιπρόσθετες οδηγίες, περισσότερη σπατάλη καταχωρητών στην τοπική μνήμη (η οποία είναι off-chip), πιο αποκλίνουσες διακλαδώσεις κτλ. Όπως και με κάθε βελτιστοποίηση, θα πρέπει κανείς να πειραματιστεί έτσι ώστε να παρατηρήσει τις αλλαγές που προκύπτουν στους χρόνους εκτέλεσης του πυρήνα. Για ευρυζωνικές εφαρμογές από την άλλη πλευρά, η αύξηση της πληρότητας μπορεί να βοηθήσει στην καλύτερη απόκρυψη λανθανόντων χρόνων πρόσβασης στην μνήμη, και ως εκ τούτου τη βελτίωση των επιδόσεων.

Κεφάλαιο 4

Εγκατάσταση CUDA 5.5 σε Linux Ubuntu LTS 12.04 και χρήση του Nsight Eclipse

4.1 Εισαγωγή

4.2 Τα απαραίτητα για την εγκατάσταση

4.3 Εκκίνηση της εγκατάστασης

4.3.1 Εγκατάσταση του Driver
4.3.2 Εγκατάσταση του Toolkit (compiler, libraries)
4.3.3 Εγκατάσταση των SDK Samples
4.3.4 Τεχνολογία Optimus και πρόγραμμα Bumblebee
4.3.5 Εγκατάσταση Bumblebee

4.4 Χρήση του Nsight Eclipse

4.4.1 Εγκατάσταση Nsight Eclipse
4.4.2 Εκκίνηση του Nsight Eclipse
4.4.3 Κάνοντας Debugging

4.1 Εισαγωγή

Η εγκατάσταση της CUDA πάνω σε λειτουργικό Linux Ubuntu είναι μια χρονοβόρα διαδικασία για κάποιον που θα το επιχειρήσει για πρώτη φορά και πολλές φορές και αρκετά επίπονη. Σημαντικό ρόλο παίζει αρχικά να γνωρίζει ο χρήστης τι ακριβώς πρόκειται να κάνει με την κάθε εντολή εγκατάστασης και σε κάθε περίπτωση επίσης να έχει ήδη δημιουργήσει ένα αντίγραφο των αρχείων του και να έχει εξασφαλισμένο το DVD του λειτουργικού συστήματος που χρησιμοποιεί κοντά του διότι η πιθανότητα του format είναι σχεδόν σίγουρη. Πρέπει να αναφέρουμε επίσης πως η εγκατάσταση έχει πάντα διαφορές από έκδοση σε έκδοση της CUDA όπως επίσης και από έκδοση λειτουργικού Linux σε άλλη έκδοση λειτουργικού. Τέλος θα μπορούσαμε να πούμε ότι η εγκατάσταση παρουσιάζει και διαφορές και στον κάθε προσωπικό υπολογιστή ξεχωριστά άρα τα προβλήματα που μπορούν να καταγραφούν είναι πάντα προσωπικά και δεν σημαίνει πως θα συμβούν και σε όλους τους χρήστες.

4.2 Τα απαραίτητα για την εγκατάσταση

Για να εγκατασταθεί η CUDA χρειάζονται να γίνουν με επιτυχία τρεις εγκαταστάσεις τριών διαφορετικών εκτελέσιμων αρχείων και με μια συγκεκριμένη σειρά. Πρώτα πρέπει να εγκατασταθεί ο Driver της CUDA, στη συνέχεια το Toolkit το οποίο περιλαμβάνει τον compiler και τις βιβλιοθήκες και τέλος το SDK το οποίο περιλαμβάνει παραδείγματα κώδικα. Στις παλιότερες εκδόσεις της CUDA όπως στην CUDA 4.2 και τις πιο παλιές αυτά τα εκτελέσιμα ήταν ξεχωριστά. Τώρα πλέον στις τελευταίες εκδόσεις της CUDA οι οποίες είναι η CUDA 5 και η CUDA 5.5 περιλαμβάνονται μέσα σε ένα μόνο εκτελέσιμο και ο Driver και το Toolkit αλλά και το SDK. Μπορεί κάποιος να βρει τα εκτελέσιμα στο επίσημο site της NVIDIA https://developer.nvidia.com/cuda-toolkit.

4.3 Εκκίνηση της εγκατάστασης

Σε αυτήν την πτυχιακή θα γίνει εγκατάσταση της CUDA 5.5 πάνω στο λειτουργικό Linux Ubuntu 12.04 LTS 64-bit. Αφού λοιπόν έχει δοκιμαστεί αρκετά η συγκεκριμένη διαδικασία πάνω σε μηχάνημα με Ubuntu 12.04 χωρίς επιπρόσθετα προγράμματα εκτός

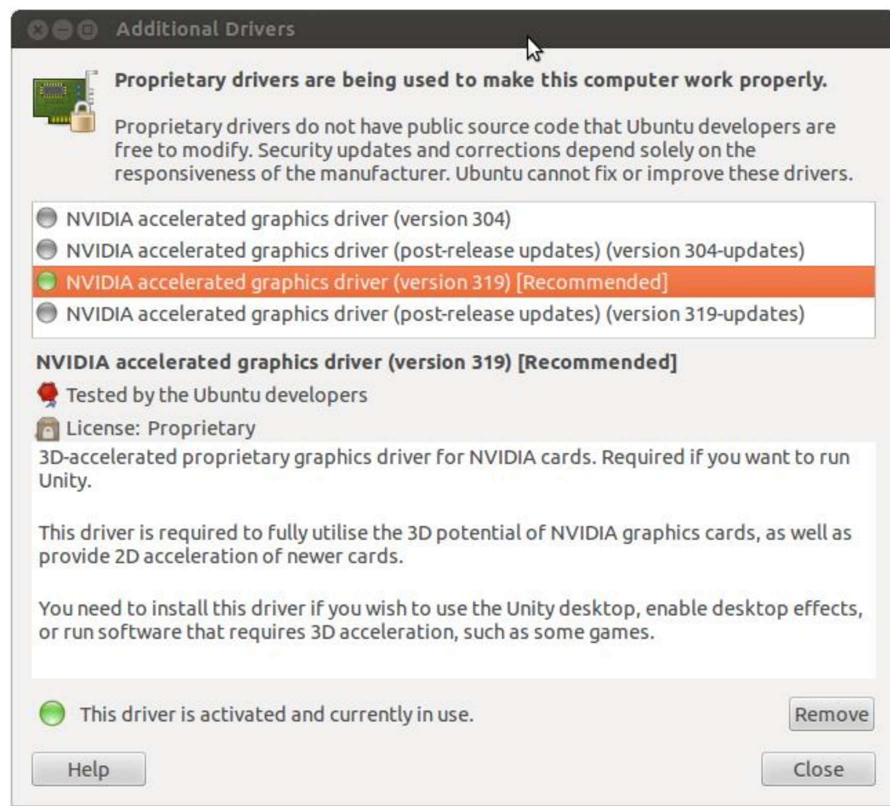
από κάποιες ενημερώσεις μπορεί κάποιος να θεωρήσει ότι ο συγκεκριμένος οδηγός έχει πολλές πιθανότητες να έχει επιτυχία.

4.3.1 Εγκατάσταση του Driver

Αφού η κάρτα γραφικών της NVIDIA έχει εγκατασταθεί και το Ubuntu την έχει αναγνωρίσει υπάρχουν δύο τρόποι για να γίνει η εγκατάσταση του driver στην CUDA. Ο πρώτος τρόπος είναι εύκολος και αφήνει το ίδιο το λειτουργικό να κάνει την δύσκολη δουλειά. Το αποτέλεσμα αυτού είναι η εγκατάσταση της v. 319 NVIDIA κάρτας γραφικών ή νεότερης. Για κάποιες κάρτες υπάρχει επίσης η επιλογή της v. 304 αλλά καλό θα ήταν να μην την επιλέξει κανείς. Μετά από κάποια εγκατάσταση ίσως εμφανιστεί το εικονίδιο που φαίνεται στην Εικ. 4.1.

Εικόνα 4.1: Εικονίδιο που ίσως εμφανιστεί

Έτσι λοιπόν υπάρχουν πλέον εγκατεστημένοι κάποιοι drivers τους οποίους μπορούμε να δούμε στην Εικ. 4.2 αν προχωρήσουμε στις **System Settings > Additional drivers**. Όπως βλέπουμε μπορεί να υπάρχουν και άλλοι drivers οι οποίοι δεν είναι ενεργοποιημένοι. Αυτός που προτιμάται είναι ο v. 319 ο οποίος φαίνεται από το πράσινο σήμα ότι είναι ενεργοποιημένος. Αν δεν είναι μπορούμε να τον επιλέξουμε και να πατήσουμε το κουμπί Activate που θα εμφανιστεί στη θέση του Remove που φαίνεται στην εικόνα.



Εικόνα 4.2: Διαθέσιμοι Drivers

Γενικότερα όμως αυτή η διαδικασία είναι πολύ πιθανό να μην χρειαστεί καν καθ' ότι κατά την εγκατάσταση του Linux Ubuntu και με τις ενημερώσεις που γίνονται μπορεί να υπάρχει ήδη ο Driver ενεργοποιημένος και να περάσει ο χρήστης κατ' ευθείαν στην εγκατάσταση του Toolkit.

Ο δεύτερος τρόπος θεωρείται ο δύσκολος τρόπος και λειτουργεί σαν εναλλακτική λύση στην περίπτωση που ο πρώτος δεν λειτουργήσει. Ανοίγουμε λοιπόν την κονσόλα και ξεκινάμε να γράφουμε εντολές. Αρχικά πρέπει να βεβαιωθούμε ότι υπάρχουν τα απαιτούμενα εργαλεία και βιβλιοθήκες με την εντολή

```
sudo apt-get install freeglut3-dev build-essential libx11-dev libxmu-dev
libxi-dev libgl1-mesa-glx libglu1-mesa libglu1-mesa-dev
```

Στη συνέχεια, πρέπει να μπουν σε *blacklist* κάποια στοιχεία (έτσι ώστε να μην συμπίπτουν με την εγκατάσταση του driver). Έτσι πρέπει να ανοίξουμε το αρχείο blacklist.conf

```
sudo gedit /etc/modprobe.d/blacklist.conf
```

και να προσθέσουμε στο τέλος

```
blacklist amd76x_edac

blacklist vga16fb

blacklist nouveau

blacklist rivafb

blacklist nvidiafb

blacklist rivatv
```

Πριν το κλείσουμε πρέπει να γίνει απαραίτητα αποθήκευση. Έπειτα από αυτό είναι αναγκαίο να πετάξουμε ότι κατάλοιπα της NVIDIA που υπάρχουν με την εντολή

```
sudo apt-get remove --purge nvidia*
```

Αυτό μπορεί να διαρκέσει λίγη ώρα αλλά μόλις τελειώσει μία επανεκκίνηση απαιτείται ξανά. Μόλις όμως ανοίξει και πάλι ο υπολογιστής και όντας ακόμη στην οθόνη εισόδου δεν πρέπει να κάνουμε login αλλά να πατήσουμε Ctrl+Alt+F1 για να κάνουμε login στην μαύρη οθόνη. Από εκεί μεταβαίνουμε στον φάκελο που περιέχει τον κατεβασμένο Driver από το site της NVIDIA και τρέχουμε τις επόμενες εντολές.

```
sudo service lightdm stop

chmod +x NVIDIA*.run
```

όπου με την πρώτη σταματάμε τη λειτουργία του GUI και στη δεύτερη το όνομα NVIDIA*.run είναι το πλήρες όνομα του Driver. Έπειτα ξεκινάει η εγκατάσταση με την εντολή

```
sudo ./NVIDIA*.run
```

Ακολουθούμε τις οδηγίες που εμφανίζονται στην οθόνη αλλά αν κατά την εγκατάσταση υπάρξει σφάλμα τύπου "*nouveau still running*", το αφήνουμε να δημιουργήσει μια blacklist για το nouveau, σταματάμε την εγκατάσταση και κάνουμε επανεκκίνηση. Σε αυτήν την περίπτωση τρέχουμε και πάλι τις εντολές

```
sudo service lightdm stop

sudo ./NVIDIA*.run
```

Τώρα η εγκατάσταση θα πρέπει να έχει γίνει σωστά. Όταν μας ρωτηθεί αν θέλουμε τις βιβλιοθήκες 32-bit και αν θέλουμε το xorg.conf να χρησιμοποιεί αυτούς τους Driver από προεπιλογή, τα επιτρέπουμε και τα δύο.

Άλλη μια επανεκκίνηση και η εγκατάσταση του Driver έχει τελειώσει. Σειρά έχει η εγκατάσταση του Toolkit.

4.3.2 Εγκατάσταση του Toolkit (compiler, libraries)

Πριν ξεκινήσει η εγκατάσταση του Toolkit καλό είναι να γίνουν κάποιες ενημερώσεις, κάποιες αναβαθμίσεις και κάποιες εγκαταστάσεις βιβλιοθηκών και πάλι. Έτσι λοιπόν σαν root εκτελούμε τις εντολές:

```
user@host:~/$ sudo apt-get update

user@host:~/$ sudo apt-get upgrade

user@host:~/$ sudo apt-get install freeglut3-dev build-essential libx11-dev libxmu-dev libxi-dev libgl1-mesa-glx-lts-raring libglu1-mesa libglu1-mesa-dev libglu-dev
```

από όπου θα πάρουμε το ακόλουθο σφάλμα σε μία πρώτη εγκατάσταση σε 12.04 LTS:

```
Reading package lists... Done
Building dependency tree
Reading state information... Done
libglu1-mesa is already the newest version.
libglu1-mesa set to manually installed.
Some packages could not be installed. This may mean that you have
requested an impossible situation or if you are using the unstable
distribution that some required packages have not yet been created
or been moved out of Incoming.
The following information may help to resolve the situation:

The following packages have unmet dependencies:
libgl1-mesa-glx : Depends: libglapi-mesa (= 8.0.4-0ubuntu0.6)
Recommends: libgl1-mesa-dri (>= 7.2)
E: Unable to correct problems, you have held broken packages.
```

Αυτό που συμβαίνει είναι ότι κάποιες βιβλιοθήκες ή κάποια μέρη βιβλιοθηκών δεν έχουν εγκατασταθεί ή δεν έχουν μπει σωστά. Αυτό όμως δεν αποτελεί πρόβλημα προς το παρόν γιατί έχει σαν αποτέλεσμα αργότερα να μην λειτουργούν κάποια Samples από το SDK που συμπεριλαμβάνουν την υλοποίηση γραφικών. Έπειτα από τη σελίδα developer.NVIDIA.com/cuda-downloads επιλέγουμε για το λειτουργικό σύστημα Linux Ubuntu 12.04 και για 64-bit το αρχείο .run το οποίο κατεβάζει το εκτελέσιμο cuda_5.5.22_linux_64.run. Έτσι λοιπόν μέσα στο φάκελο Downloads εκτελούμε τις παρακάτω εντολές για να ξεκινήσει η εγκατάσταση

```
user@host:~/$ cd Downloads
user@host:~/Downloads$ chmod +x cuda_5.5.22_linux_64.run
user@host:~/Downloads$ sudo ./cuda_5.5.22_linux_64.run
```

οι οποίες θα παράγουν

```
Logging to /tmp/cuda_install_14755.log
Using more to view the EULA.
End User License Agreement
---------
...
and cannot be linked to any personally identifiable
information. Personally identifiable information such as your
username or hostname is not collected.

----------------------
```

και μετά από αρκετή κύλιση προς τα κάτω θα πρέπει να απαντήσουμε με τους εξής τρόπους στις ερωτήσεις της κονσόλας.

```
Do you accept the previously read EULA? (accept/decline/quit): accept
Install NVIDIA Accelerated Graphics Driver for Linux-x86_64 319.37?
((y)es/(n)o/(q)uit): n
Install the CUDA 5.5 Toolkit? ((y)es/(n)o/(q)uit): y
Enter Toolkit Location [ default is /usr/local/cuda-5.5 ]:
Install the CUDA 5.5 Samples? ((y)es/(n)o/(q)uit): y
Enter CUDA Samples Location [ default is /home/user/NVIDIA_CUDA-
5.5_Samples ]:
```

Όπως είπαμε και προηγουμένως υπάρχει πολύ μεγάλη πιθανότητα στα νέα λειτουργικά Linux Ubuntu να υπάρχει ήδη ο Driver μέσα ή τον έχουμε ήδη βάλει μέσω των δύο τρόπων που αναλύσαμε παραπάνω. Έτσι λοιπόν θα πούμε «Όχι» στην εγκατάσταση του driver και θα προχωρήσουμε λέγοντας «Ναι» στην εγκατάσταση του Toolkit και των CUDA 5.5 Samples. Αν θέλουμε κρατάμε την προεπιλεγμένη διαδρομή για εγκατάσταση. Αν όχι μπορούμε να δώσουμε εμείς αυτήν που θέλουμε αρκεί να την θυμόμαστε και μετέπειτα. Αυτή η διαδικασία θα έχει το εξής αποτέλεσμα στην οθόνη

```
Installing the CUDA Toolkit in /usr/local/cuda-5.5 ...
Installing the CUDA Samples in /home/user/NVIDIA_CUDA-5.5_Samples ...
Copying samples to /home/user/NVIDIA_CUDA-5.5_Samples/NVIDIA_CUDA-5.5_Samples now...
Finished copying samples.

===========
= Summary =
===========

Driver: Not Selected
Toolkit: Installed in /usr/local/cuda-5.5
Samples: Installed in /home/user/NVIDIA_CUDA-5.5_Samples

* Please make sure your PATH includes /usr/local/cuda-5.5/bin

* Please make sure your LD_LIBRARY_PATH
* for 32-bit Linux distributions includes /usr/local/cuda-5.5/lib
* for 64-bit Linux distributions includes /usr/local/cuda-5.5/lib64:/lib
* OR
* for 32-bit Linux distributions add /usr/local/cuda-5.5/lib
* for 64-bit Linux distributions add /usr/local/cuda-5.5/lib64 and /lib
* to /etc/ld.so.conf and run ldconfig as root

* To uninstall CUDA, remove the CUDA files in /usr/local/cuda-5.5
* Installation Complete

Please see CUDA_Getting_Started_Linux.pdf in /usr/local/cuda-5.5/doc/pdf for detailed information on setting up
CUDA.

***WARNING: Incomplete installation! This installation did not install the CUDA Driver. A driver of version at least
319.00 is required for CUDA 5.5 functionality to work.
To install the driver using this installer, run the following command, replacing with the name of this run file:
sudo .run -silent -driver

Logfile is /tmp/cuda_install_14755.log
```

Όσο αφορά την ειδοποίηση στο τέλος μπορούμε να την αγνοήσουμε αλλά όπως βλέπουμε από το μήνυμα που προκύπτει στην οθόνη θα πρέπει να προστεθούν στο αρχείο .bashrc οι ακόλουθες γραμμές. Βγαίνουμε λοιπόν σε root και ανοίγουμε το αρχείο .bashrc

```
user@host:~/Downloads$ cd
user@host:~/$ gedit .bashrc
```

και προσθέτουμε στο τέλος τις δυο παρακάτω γραμμές

```
PATH=$PATH:/usr/local/cuda-5.5/bin
LD_LIBRARY_PATH=$LD_LIBRARY_PATH:/usr/local/cuda-5.5/lib64:/lib
```

και έπειτα

```
user@host:~/$ source .bashrc
```

Η εγκατάσταση του Toolkit έχει ολοκληρωθεί επιτυχώς.

4.3.3 Εγκατάσταση των SDK Samples

Τα δείγματα κώδικα (Samples) στην εγκατάσταση της CUDA δεν αποτελούν βασική προϋπόθεση για να δουλεύει σωστά η CUDA. Η εγκατάσταση τους γίνεται για να επιβεβαιωθεί πως η CUDA έχει εγκατασταθεί σωστά και για να μπορέσει ο προγραμματιστής να είναι σίγουρος ότι προβλήματα πάνω σε δικά του προγράμματα σίγουρα δεν θα προέρχονται από την διαδικασία της εγκατάστασης.

Η εγκατάσταση των Samples ξεκινάει με την εντολή make αφού πρώτα μεταβούμε στους υποφακέλους των samples

```
user@host:~/$ cd NVIDIA_CUDA-5.5_Samples/NVIDIA_CUDA-5.5_Samples
user@host:~/NVIDIA_CUDA-5.5_Samples/NVIDIA_CUDA-5.5_Samples$ make
```

Αυτό θα παράγει το ακόλουθο αποτέλεσμα

```
make[1]: Entering directory `/home/user/NVIDIA_CUDA-5.5_Samples/NVIDIA_CUDA-5.5_Samples/0_Simple/asyncAPI'
"/usr/local/cuda-5.5"/bin/nvcc -ccbin g++ -I../../common/inc -m64 -gencode arch=compute_10,code=sm_10 -gencode arch=compute_20,code=sm_20 -gencode arch=compute_30,code=sm_30 -gencode arch=compute_35,code=\"sm_35,compute_35\" -o asyncAPI.o -c asyncAPI.cu
...
"/usr/local/cuda-5.5"/bin/nvcc -ccbin g++ -I../../common/inc -m64 -o vectorAddDrv.o -c vectorAddDrv.cpp
"/usr/local/cuda-5.5"/bin/nvcc -ccbin g++ -m64 -o vectorAddDrv vectorAddDrv.o -L/usr/lib/NVIDIA-current -lcuda
/usr/bin/ld: cannot find -lcuda
collect2: ld returned 1 exit status
make[1]: *** [vectorAddDrv] Error 1
make[1]: Leaving directory `/home/user/NVIDIA_CUDA-5.5_Samples/NVIDIA_CUDA-5.5_Samples/0_Simple/vectorAddDrv'
make: *** [0_Simple/vectorAddDrv/Makefile.ph_build] Error 2
```

Αυτό λύνεται κάνοντας ένα συμβολικό σύνδεσμο για την libcuda.so από το φάκελο /usr/lib/NVIDIA-319/ στο φάκελο /usr/lib/.

```
user@host:~/$ sudo ln -s /usr/lib/NVIDIA-319/libcuda.so /usr/lib/libcuda.so
```

Εάν εργαζόμαστε σε διαδικασία build για να διορθώσουμε το σφάλμα, εκτελούμε ένα "make clean" πριν ξαναεκτελέσουμε.

Η δεύτερη προσπάθεια build

```
user@host:~/NVIDIA_CUDA-5.5_Samples/NVIDIA_CUDA-5.5_Samples$ make
```

έχει το παρακάτω αποτέλεσμα

```
make[1]: Entering directory `/home/user/NVIDIA_CUDA-5.5_Samples/NVIDIA_CUDA-5.5_Samples/0_Simple/asyncAPI'
"/usr/local/cuda-5.5"/bin/nvcc -ccbin g++ -I../../common/inc -m64 -gencode arch=compute_10,code=sm_10 -gencode arch=compute_20,code=sm_20 -gencode arch=compute_30,code=sm_30 -gencode arch=compute_35,code=\"sm_35,compute_35\" -o asyncAPI.o -c asyncAPI.cu
"/usr/local/cuda-5.5"/bin/nvcc -ccbin g++ -m64 -o asyncAPI asyncAPI.o
. . .
cp simpleCubemapTexture ../../bin/x86_64/linux/release
make[1]: Leaving directory `/home/user/NVIDIA_CUDA-5.5_Samples/NVIDIA_CUDA-5.5_Samples/0_Simple/simpleCubemapTexture'
---------------------------------
WARNING - No MPI compiler found.
---------------------------------
CUDA Sample "simpleMPI" cannot be built without an MPI Compiler.
This will be a dry-run of the Makefile.
For more information on how to set up your environment to build and run this
sample, please refer the CUDA Samples documentation and release notes
---------------------------------
make[1]: Entering directory `/home/user/NVIDIA_CUDA-5.5_Samples/NVIDIA_CUDA-5.5_Samples/0_Simple/simpleMPI'
[@] mpicxx -I../../common/inc -o simpleMPI.o -c simpleMPI.cpp
. . .
mkdir -p ../../bin/x86_64/linux/release
cp histEqualizationNPP ../../bin/x86_64/linux/release
make[1]: Leaving directory `/home/user/NVIDIA_CUDA-5.5_Samples/NVIDIA_CUDA-5.5_Samples/7_CUDALibraries/histEqualizationNPP'
Finished building CUDA samples
```

Αλλά τελειώνει με επιτυχία.

Για να λύσουμε αυτό το πρόβλημα πρέπει να εγκαταστήσουμε το OpenMPI γιατί η συγκεκριμένη προειδοποίηση είναι η ανάγκη του mpicc (mpi compiler) ο οποίος είναι μέσα στο libopenmpi-dev. Έτσι λοιπόν τρέχουμε το mpicc και παίρνουμε αυτές τις επιλογές.

```
user@host:~/NVIDIA_CUDA-5.5_Samples/NVIDIA_CUDA-5.5_Samples$ mpicc
The program 'mpicc' can be found in the following packages:
* lam4-dev
* libmpich-mpd1.0-dev
* libmpich-shmem1.0-dev
* libmpich1.0-dev
* libmpich2-dev
* libopenmpi-dev
* libopenmpi1.5-dev
Try: sudo apt-get install
```

Επιλέγουμε μόνο τρεις από αυτές

```
user@host:~/$ sudo apt-get install openmpi-bin openmpi-common
libopenmpi-dev
```

και στην συνέχεια παίρνουμε το ακόλουθο σφάλμα

```
user@host:~/NVIDIA_CUDA-5.5_Samples/NVIDIA_CUDA-5.5_Samples$ mpicc
gcc: fatal error: no input files
compilation terminated.
```

το οποίο σημαίνει πως ο compiler έχει μπει. Άρα στην πορεία αν κάνουμε μία δεύτερη φορά make δεν θα υπάρχει πρόβλημα.

Και θα δούμε το ακόλουθο αποτέλεσμα

```
make[1]: Entering directory `/home/user/NVIDIA_CUDA-5.5_Samples/NVIDIA_CUDA-5.5_Samples/0_Simple/asyncAPI'
"/usr/local/cuda-5.5"/bin/nvcc -ccbin g++ -I../../common/inc -m64 -gencode arch=compute_10,code=sm_10 -gencode arch=compute_20,code=sm_20 -gencode arch=compute_30,code=sm_30 -gencode arch=compute_35,code=\"sm_35,compute_35\" -o asyncAPI.o -c asyncAPI.cu
"/usr/local/cuda-5.5"/bin/nvcc -ccbin g++ -m64 -o asyncAPI asyncAPI.o
mkdir -p ../../bin/x86_64/linux/release
. . .
mkdir -p ../../bin/x86_64/linux/release
cp histEqualizationNPP ../../bin/x86_64/linux/release
make[1]: Leaving directory `/home/user/NVIDIA_CUDA-5.5_Samples/NVIDIA_CUDA-5.5_Samples/7_CUDALibraries/histEqualizationNPP'
Finished building CUDA samples
```

Κατά την εγκατάσταση του SDK κανείς θα αντιμετωπίσει αρκετά προβλήματα με πολλές βιβλιοθήκες που θα έπρεπε να υπάρχουν για να τρέχουν όλα τα δείγματα κώδικα. Οπότε συνήθως τα σφάλματα που εμφανίζονται στην οθόνη είναι καθοδηγητικά όσο αφορά το ποια βιβλιοθήκη λείπει ή το γενικότερο πρόβλημα που εντοπίζει η διαδικασία εκτέλεσης. Συνήθως με την εγκατάσταση της απαιτούμενης βιβλιοθήκης το πρόβλημα λύνεται. Παρακάτω υπάρχει ένα παράδειγμα προβλήματος στην εκτέλεση ενός δείγματος κώδικα μέσα από το SDK της CUDA.

Ας πούμε πως θέλουμε να τρέξουμε το randomFog που βρίσκεται μέσα στο φάκελο

```
NVIDIA_CUDA-5.5_Samples/NVIDIA_CUDA-5.5_Samples/bin/x86_64/linux/release
```

```
user@host:~/NVIDIA_CUDA-5.5_Samples/NVIDIA_CUDA-5.5_Samples/$ cd bin/x86_64/linux/release
user@host:~/NVIDIA_CUDA-5.5_Samples/NVIDIA_CUDA-5.5_Samples/bin/x86_64/linux/release$ ls
alignedTypes HSOpticalFlow simpleCUBLAS
asyncAPI imageDenoising simpleCUDA2GL
bandwidthTest imageSegmentationNPP simpleCUFFT
batchCUBLAS inlinePTX simpleDevLibCUBLAS
bicubicTexture interval simpleGL
bilateralFilter jpegNPP simpleHyperQ
bindlessTexture lineOfSight simpleIPC
binomialOptions Mandelbrot simpleLayeredTexture
BlackScholes marchingCubes simpleMPI
boxFilter matrixMul simpleMultiCopy
boxFilterNPP matrixMulCUBLAS simpleMultiGPU
cdpAdvancedQuicksort matrixMulDrv simpleP2P
```

Όπου και εμφανίζονται όλα τα διαθέσιμα samples. Αν προσπαθήσουμε να τρέξουμε το randomFog

```
user@host:~/NVIDIA_CUDA-5.5_Samples/NVIDIA_CUDA-5.5_Samples/bin/x86_64/linux/release$ ./randomFog
```

θα ακολουθήσει το εξής σφάλμα

```
user@host:~/NVIDIA_CUDA-5.5_Samples/NVIDIA_CUDA-5.5_Samples/bin/x86_64/linux/release$ ./randomFog
./randomFog: error while loading shared libraries: libcurand.so.5.5: cannot open shared object file: No such file or directory
```

Προσθέτουμε συμβολικό σύνδεσμο από τον φάκελο /usr/lib/x86_64-linux-gnu στο /usr/lib

```
user@host:~$ sudo ln -s /usr/lib/x86_64-linux-gnu/libglut.so.3 /usr/lib/libglut.so
```

θα δώσει

```
user@host:~/NVIDIA_CUDA-5.5_Samples/NVIDIA_CUDA-5.5_Samples/bin/x86_64/linux/release$ ./randomFog
./randomFog: error while loading shared libraries: libcurand.so.5.5: cannot open shared object file: No such file or directory
```

Για να βρούμε τη θέση ή την παρουσία της libcurand τρέχουμε ldconfig –v

```
user@host:~/$ ldconfig -v
```

Το αποτέλεσμα είναι κάπως έτσι

```
user@host:~/NVIDIA_CUDA-5.5_Samples/NVIDIA_CUDA-5.5_Samples/bin/x86_64/linux/release$ ldconfig -v
/sbin/ldconfig.real: Path `/lib/x86_64-linux-gnu' given more than once
/sbin/ldconfig.real: Path `/usr/lib/x86_64-linux-gnu' given more than once
/usr/local/cuda-5.5/lib64:
libcuinj64.so.5.5 -> libcuinj64.so.5.5.22
libcufft.so.5.5 -> libcufft.so.5.5.22
libcurand.so.5.5 -> libcurand.so.5.5.22
libcusparse.so.5.5 -> libcusparse.so.5.5.22
. . .
libnvToolsExt.so.1 -> libnvToolsExt.so.1.0.0
/usr/local/cuda-5.5/lib:
libcufft.so.5.5 -> libcufft.so.5.5.22
libcurand.so.5.5 -> libcurand.so.5.5.22
libcusparse.so.5.5 -> libcusparse.so.5.5.22
. . .
/usr/lib/NVIDIA-319/tls: (hwcap: 0x8000000000000000)
libNVIDIA-tls.so.319.32 -> libNVIDIA-tls.so.319.32
/usr/lib32/NVIDIA-319/tls: (hwcap: 0x8000000000000000)
libNVIDIA-tls.so.319.32 -> libNVIDIA-tls.so.319.32
/sbin/ldconfig.real: Can't create temporary cache file /etc/ld.so.cache~: Permission denied
```

Που σημαίνει ότι έχει βρεθεί διπλά οπότε μπορούμε να ανοίξουμε το αρχείο NVIDIA.conf

```
user@host:~/$ sudo gedit /etc/ld.so.conf.d/NVIDIA.conf
```

και να προσθέσουμε

```
/usr/local/cuda-5.5/lib64
/usr/local/cuda-5.5/lib
```

Τρέχουμε ξανά το ldconfig

```
user@host:~/$ sudo ldconfig
```

και κάπως έτσι τρέχει πλέον το randomFog

```
user@host:~/NVIDIA_CUDA-5.5_Samples/NVIDIA_CUDA-
5.5_Samples/bin/x86_64/linux/ release$ ./randomFog
```

και δίνει το παρακάτω

```
Random Fog
==========

CURAND initialized

Random number visualization
```

Όταν ένα πρόβλημα λύνεται για μία μεμονωμένη περίπτωση είναι σχεδόν βέβαιο πως θα είναι ευνοϊκό και για άλλες περιπτώσεις.

4.3.4 Τεχνολογία Optimus και πρόγραμμα Bumblebee

Σε μηχανήματα νέας τεχνολογίας κανείς θα συναντήσει την νέα τεχνολογία Optimus η οποία βασίζεται πάνω στην ύπαρξη δύο καρτών γραφικών μέσα σε ένα σύστημα, μία της NVIDIA και μία άλλη απλή η οποία είναι ενσωματωμένη πάνω στη μητρική κάρτα. Το Nvidia Optimus έχει δημιουργηθεί για τη βελτιστοποίηση ισχύος/ απόδοσης από την Nvidia η οποία, ανάλογα με το φορτίο των πόρων που δημιουργούνται από εφαρμογές λογισμικού πελάτη, με διαφανή τρόπο και χωρίς να γίνεται αντιληπτό, κάνει την εναλλαγή μεταξύ των δύο καρτών γραφικών μέσα στο σύστημα έτσι ώστε να παρέχεται μέγιστη απόδοση ή ελάχιστη κατανάλωση ενέργειας από τα γραφικά του υλικού. Πιο συγκεκριμένα είναι μία ιδιαίτερη υλοποίηση αυτού που είναι γνωστό ως «εναλλασσόμενη κάρτα γραφικών» ή αλλιώς «εναλλαγή GPU».

Το Optimus σώζει τη ζωή της μπαταρίας ενεργοποιώντας αυτόματα την ισχύ της διακριτής μονάδας επεξεργασίας γραφικών (GPU) όταν δεν είναι απαραίτητη και ενεργοποιείται όταν χρειάζεται και πάλι. Η τεχνολογία απευθύνεται κυρίως σε φορητούς υπολογιστές, όπως laptops. Όταν η δύναμη της GPU είναι απενεργοποιημένη, ο οδηγός ανακατευθύνει εντολές γραφικών στο ενσωματωμένο chip γραφικών. Η αλλαγή έχει σχεδιαστεί για να είναι εντελώς απρόσκοπτη και να συμβαίνει "πίσω από τα παρασκήνια". Λειτουργικά όπως είναι τα Windows 8 υποστηρίζουν αυτήν την τεχνολογία αλλά ακόμη δεν έχει εφαρμοστεί σε λειτουργικά όπως είναι τα Linux. Ένα πρόγραμμα που ονομάζεται Bumblebee φέρνει την υποστήριξη της τεχνολογίας Optimus σε GNU/LINUX.

Το Bumblebee είναι ένα πρόγραμμα ανοιχτού κώδικα το οποίο προσπαθεί να παρέχει την λεγόμενη εναλλαγή καρτών γραφικών σε περιβάλλον Linux. Αυτή η διαδικασία σε περιβάλλον Windows μπορεί να γίνεται αυτόματα αλλά ακόμα σε περιβάλλοντα Linux χρειάζεται ο καθορισμός της κάρτας γραφικών που θέλει ο χρήστης να χρησιμοποιηθεί. Αυτό επιτυγχάνεται με διάφορες εντολές από τη γραμμή εντολών ή από συντομεύσεις εικονιδίων.

Πολλές φορές ο χρήστης δεν γνωρίζει την τεχνολογία Optimus και ενώ μπορεί να έχει εγκαταστήσει σωστά την CUDA ή γενικότερα κάποια άλλη εφαρμογή που χρειάζεται την κάρτα γραφικών της NVIDIA, να παίρνει μηνύματα σφάλματος ή μη εύρεσης της κάρτας. Έτσι το Bumblebee διευκολύνει τον χρήστη να επιλέξει εκείνος σε ποια κάρτα γραφικών θέλει να τρέξει την εφαρμογή του.

4.3.5 Εγκατάσταση Bumblebee

Αρχικά θα πρέπει α αναφερθεί και πάλι πως δουλεύουμε σε Linux 12.04 LTS και ανοίγοντας το λειτουργικό για πρώτη φορά κάνουμε όλες τις απαραίτητες ενημερώσεις με την εντολή

```
sudo apt-get update
```

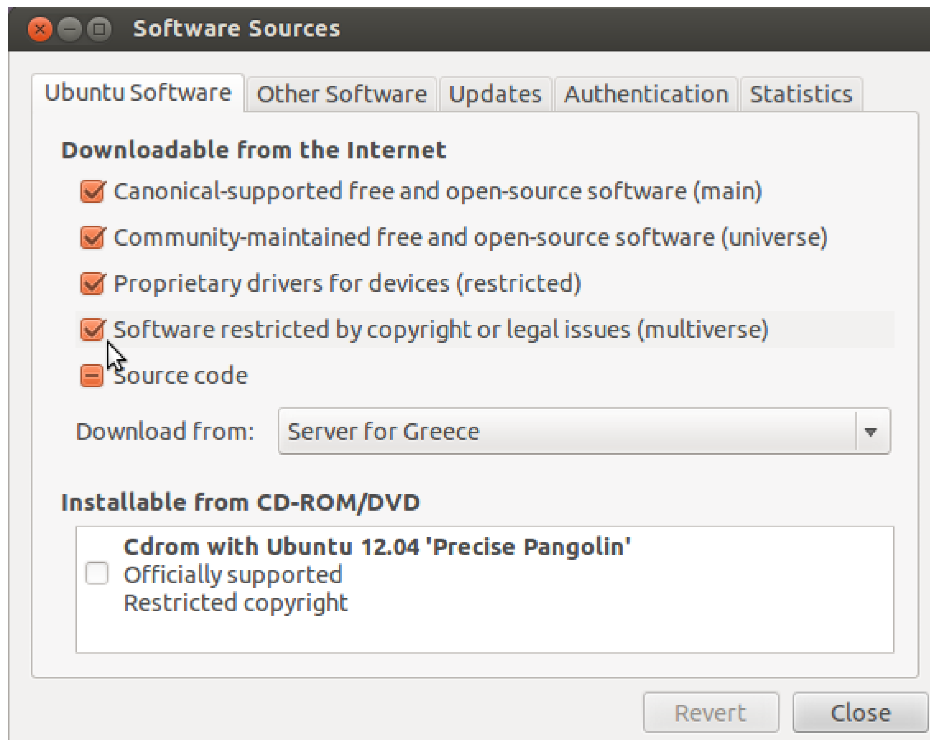
Έπειτα προχωράμε σε μία επανεκκίνηση.

Στη συνέχεια στο μενού **System Settings > Additional Drivers** ενεργοποιούμε τον νεότερο Driver ο οποίος πρέπει να είναι ο v. 331 πλέον και κάνουμε μια ακόμα επανεκκίνηση. Τώρα πλέον ο υπολογιστής είναι έτοιμος να φορτώσει και να εγκαταστήσει το πρόγραμμα Bumblebee ως εξής.

Αρχικά πρέπει να ανοίξουμε το Terminal και να εισάγουμε τις εντολές που βλέπουμε παρακάτω. Αν το λειτουργικό στο οποίο δουλεύουμε είναι το 12.04.3 τότε πρέπει να αντικαταστήσουμε το linux-headers-generic με το linux-headers-generic-lts-raring και μετά τρέχουμε

```
sudo add-apt-repository ppa:bumblebee/stable
```

Έπειτα ενεργοποιούμε τα Universe και Multiverse repositories από Software Center > Edit > Software Sources και καρτέλα Ubuntu Software όπως φαίνεται παρακάτω



Εικόνα 4.3: Ενεργοποίηση Universe και Multiverse repositories

και έπειτα κάνουμε

```
sudo apt-get update
```

Στη συνέχεια μπορεί να γίνει η εγκατάσταση του Bubblebee με την ακόλουθη εντολή για 12.04

```
sudo apt-get install bumblebee bumblebee-nvidia virtualgl linux-headers-generic
```

και για 13.10

```
sudo apt-get install bumblebee bumblebee-nvidia primus linux-headers-generic
```

και άλλη μία επανεκκίνηση.

Για να τρέξει λοιπόν κανείς την εφαρμογή του με την κάρτα της NVIDIA τότε θα εκτελέσει στο Terminal

```
$optirun [options] <application> [application parameters]
```

Επίσης θα πρέπει να γίνει εγκατάσταση μιας νέας βιβλιοθήκης μαζί με τις υπόλοιπες που είδαμε παραπάνω η libgl1-mesa-glx-lts-raring και τρέχουμε για όλες μαζί

```
sudo apt-get install freeglut3-dev build-essential libx11-dev libxmu-dev
libxi-dev libgl1-mesa-glx-lts-raring libglu1-mesa libglu1-mesa-dev
```

Ακόμα μία επανεκκίνηση είναι απαραίτητη και μετά από αυτό τρέχουμε το πακέτο της CUDA εγκαθιστώντας μόνο το Toolkit και τα Samples όπως δείξαμε παραπάνω.

Το αρχείο ~/.bashrc πρέπει να έχει τα παρακάτω για σύστημα 32-bit

```
export PATH=$PATH:/usr/local/cuda-5.5/bin

export LD_LIBRARY_PATH=/usr/local/cuda-5.5/lib
```

και για 64-bit

```
export PATH=$PATH:/usr/local/cuda-5.5/bin

export LD_LIBRARY_PATH=/usr/local/cuda-5.5/lib64:/lib
```

Άλλη μία επανεκκίνηση και είναι έτοιμο για χρήση.

Τα Samples ή η εφαρμογή του χρήστη τρέχουν με την εντολή

```
optirun ./onoma_ektelesimou
```

4.4 Χρήση του Nsight Eclipse

Το Eclipse είναι ένα ολοκληρωμένο περιβάλλον ανάπτυξης (IDE – Integrated Development Environment) για προγραμματιστές CUDA. Έχει σχεδιαστεί για να παρέχει στους προγραμματιστές βοήθεια σε όλα τα στάδια της διαδικασίας ανάπτυξης εφαρμογών. Περιέχει έναν βασικό χώρο εργασίας και ένα επεκτάσιμο σύστημα μέσω διαφόρων plug-ins για την προσαρμογή του περιβάλλοντος. Είναι γραμμένο κυρίως σε Java και χρησιμοποιείται κυρίως για ανάπτυξη εφαρμογών πάλι σε Java. Μέσω ομαδοποίησης των plug-ins μπορεί επίσης να χρησιμοποιηθεί για την ανάπτυξη εφαρμογών και σε άλλες γλώσσες προγραμματισμού όπως C, C++, COBOL, Fortran, JavaScript, Perl, PHP, Python και πολλές ακόμα. Μπορεί επίσης να χρησιμοποιηθεί και για την ανάπτυξη πακέτων για το λογισμικό Mathematica. Τα περιβάλλοντα ανάπτυξης περιλαμβάνουν τα εργαλεία ανάπτυξης Eclipse Java (για Java), Eclipse CDT (για C/C++) και Eclipse PDT για PHP, μεταξύ άλλων. Ο αρχικός του κώδικας γράφτηκε από την IBM όπως επίσης και το SDK (Software Development Kit).

4.4.1 Εγκατάσταση Nsight Eclipse

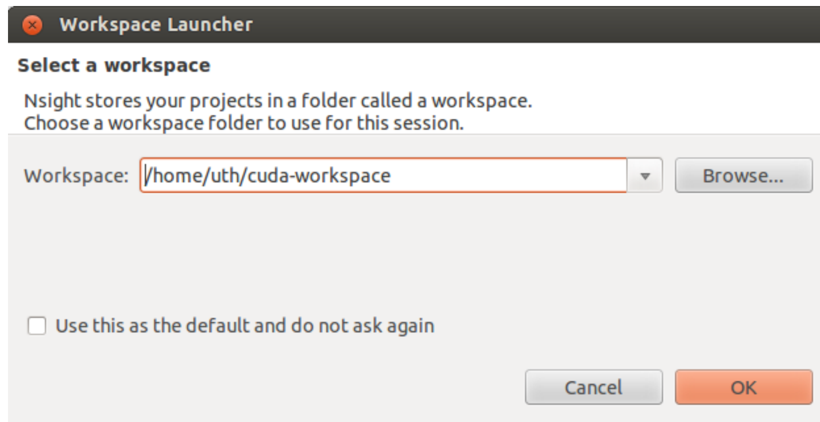
Από τη στιγμή που ο χρήστης έχει εγκαταστήσει το Toolkit της CUDA το Eclipse εγκαθίσταται αυτόματα διότι βρίσκεται ήδη μέσα στο πακέτο εγκατάστασης της CUDA. Για τα περιβάλλοντα προγραμματισμού Linux και Mac OS υπάρχει η έκδοση που

ονομάζεται Nsight Eclipse. Ανάλογα με το πακέτο της CUDA που θα εγκαταστήσει κάποιος θα έχει και την αντίστοιχη έκδοση του Nsight Eclipse.

4.4.2 Εκκίνηση του Nsight Eclipse

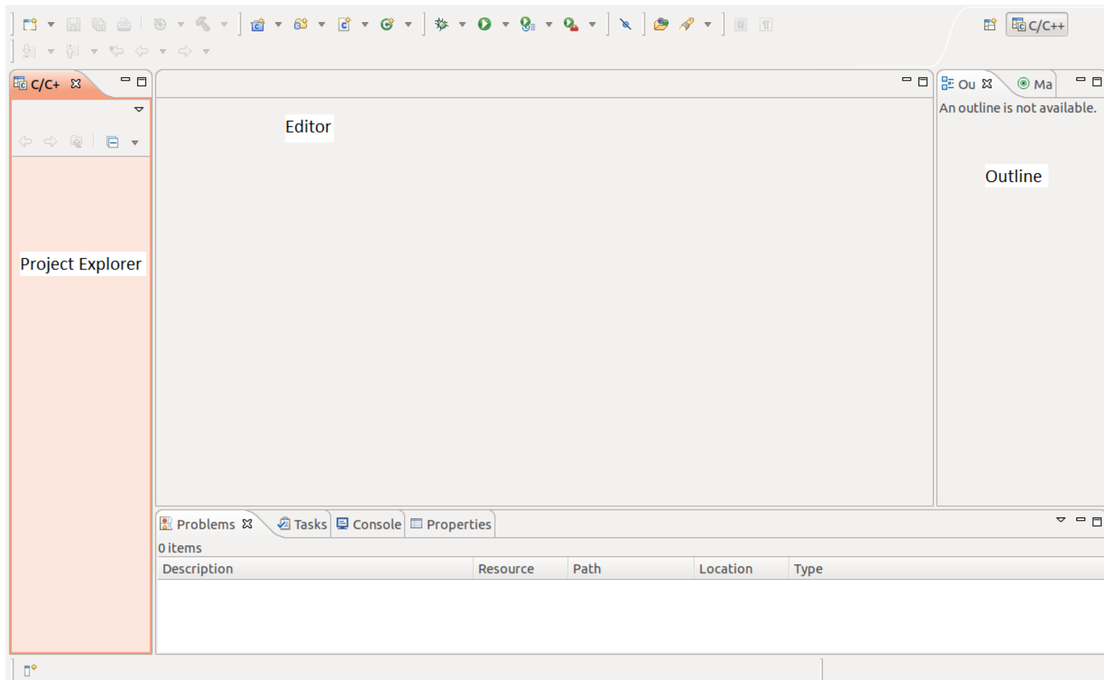
Πληκτρολογώντας `nsight` στο terminal θα ανοίξει το περιβάλλον του Nsight Eclipse. Ανοίγοντας την πρώτη φορά εμφανίζεται το παράθυρο όπου θα πρέπει να επιλέξει ο χρήστης μία τοποθεσία εργασίας (Workspace Launcher).

Η τοποθεσία αυτή είναι ένας φάκελος όπου το Nsight θα αποθηκεύσει τις ρυθμίσεις του, τα τοπικά αρχεία, το ιστορικό και την κρυφή του μνήμη. Προκειμένου όμως να αποφευχθεί η αντικατάσταση ήδη υπάρχοντων αρχείων καλό είναι να επιλέξει κανείς ένα κενό φάκελο.



Εικόνα 4.4: Επιλογή φακέλου εργασίας

Το αρχικό παράθυρο θα ανοίξει αφότου έχει επιλεγεί η τοποθεσία εργασίας το οποίο χωρίζεται στις ακόλουθες περιοχές:



Εικόνα 4.5: Αρχικό παράθυρο Nsight Eclipse

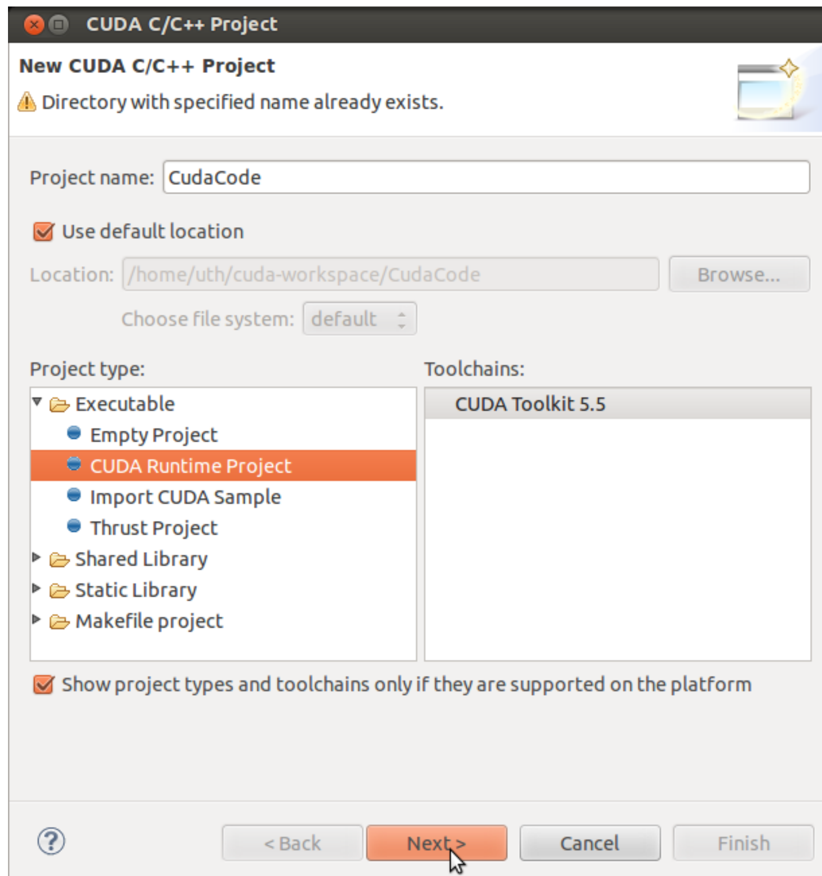
- *Editor* – Προβάλει αρχεία πηγαίου κώδικα όπου είναι ανοιχτά προς επεξεργασία
- *Project Explorer* – Εμφανίζει τα αρχεία του project
- *Outline* – Εμφανίζει τη δομή του πηγαίου αρχείου στον υπάρχοντα editor.
- *Problems* – Εμφανίζει σφάλματα και προειδοποιήσεις που εντοπίστηκαν από τον στατικό κώδικα κατά τη διάρκεια ανάλυσης στο IDE ή από τον compiler κατά τη διάρκεια της κατασκευής.
- *Console* – Εμφανίζει το αποτέλεσμα που εξήχθη από τη διαδικασία κατασκευής ή εκτέλεσης του κώδικα.

Για τη δημιουργία ενός νέου Project επιλέγουμε - **File > New.. > CUDA C/C++ project.**

Ο οδηγός του C++ Project ανοίγει.

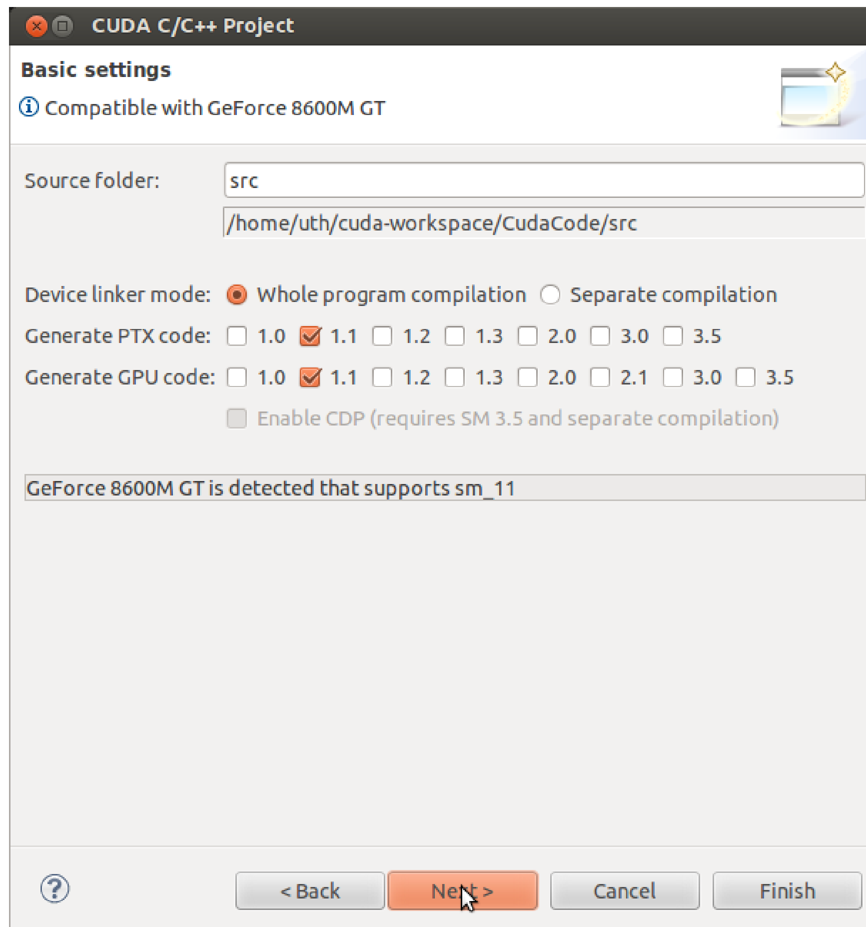
1. Στο πεδίο **Project Name** δίνουμε ένα όνομα για το Project (όπως *CudaCode* για παράδειγμα).
2. Στο πεδίο **Project type** επιλέγουμε τον τύπο του Project που θα δημιουργήσουμε όπως ένα νέο κενό Project (*Empty Project*) ή μία εφαρμογή χρόνου εκτέλεσης (*CUDA runtime application*) και δίνουμε και την επιθυμητή τοποθεσία ακριβώς από κάτω στο πεδίο **Location.**

3. Στο πεδίο **Toolchains** πρέπει να διαλέξουμε ένα σετ από εργαλεία που θέλουμε να χρησιμοποιήσουμε. Στην δική μας περίπτωση έχουμε μόνο το Toolkit της CUDA. Μπορεί όμως ανάλογα με το σύστημα του κάθε χρήστη να υπάρχουν κι άλλα.



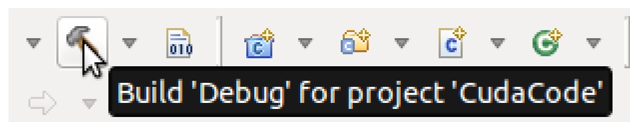
Εικόνα 4.6: Επιλογή σετ εργαλείων

4. Πατώντας το κουμπί **Next** επιλέγουμε κάποιες βασικές παραμέτρους. Ανάλογα με την κάρτα γραφικών του κάθε χρήστη το Nsight αυτόματα θα εντοπίσει το υλικό της CUDA που είναι διαθέσιμο τοπικά. Αν όμως δεν εντοπίσει καθόλου CUDA υλικό τότε αυτόματα θα επιλέξει το SM 1.0.



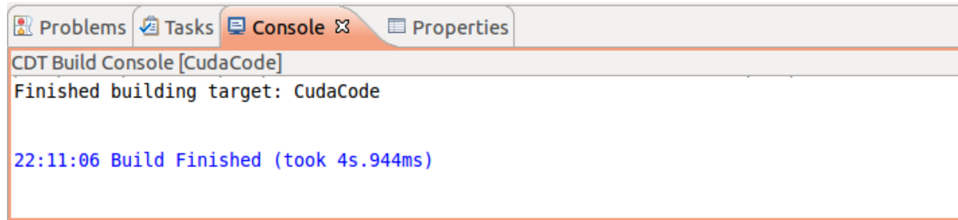
Εικόνα 4.7: Εντοπισμός CUDA υλικού τοπικά

5. Πατώντας **Finish** έχουμε ξεκινήσει τη δημιουργία του Project το οποίο θα εμφανιστεί στο πεδίο **Project Explorer** και θα ανοίξει ο επεξεργαστής του πηγαίου κώδικα στο κέντρο του Eclipse.
6. Αφού γράψουμε τον πηγαίο κώδικα πρέπει να βεβαιωθούμε πως όλα είναι σωστά. Κάτω από τον πηγαίο κώδικα και στην καρτέλα **Problems** γίνεται περιγραφή του προβλήματος εάν υπάρχει.
7. Με το εικονίδιο που έχει ένα σφυρί πάνω γίνεται κατασκευή του κώδικα



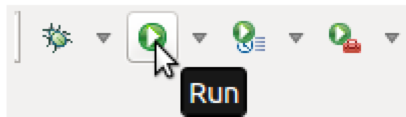
Εικόνα 4.8: Κατασκευή του κώδικα

8. Στην καρτέλα **Console** φαίνεται η παρακάτω εικόνα.



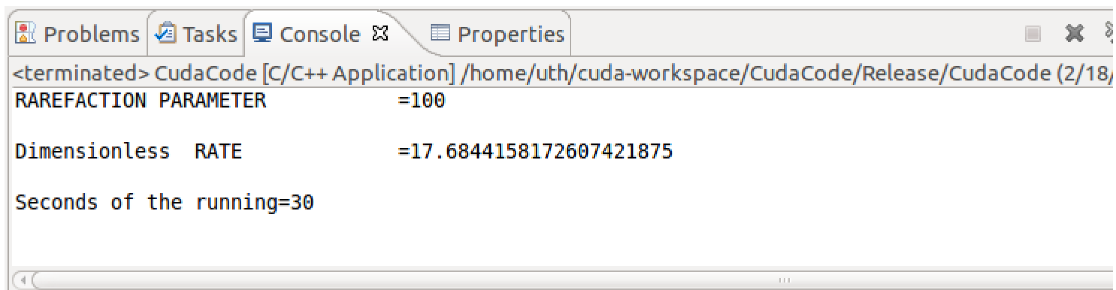
Εικόνα 4.9: Η κονσόλα

9. Για να τρέξει η εφαρμογή πατάμε δεξί κλικ πάνω στο project και στη συνέχεια επιλέγουμε **Run As > 1 Local C/C++ Application** ή απλά με το κουμπί Run πάνω από το παράθυρο του πηγαίου κώδικα.



Εικόνα 4.10: Εκτέλεση του κώδικα

10. Τα αποτελέσματα είναι ορατά και πάλι στην καρτέλα **Console**.

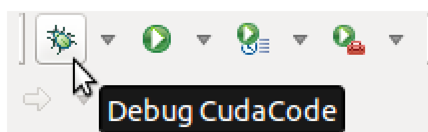


Εικόνα 4.11: Αποτελέσματα στην κονσόλα

4.4.3 Κάνοντας Debugging

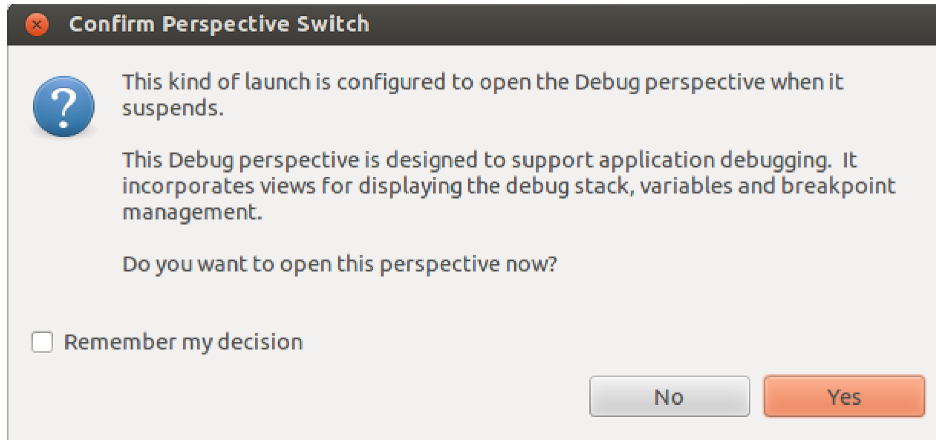
1. Στο παράθυρο του **Project Explorer** επιλέγουμε το project στο οποίο θέλουμε να κάνουμε debug. Πρέπει να βεβαιωθούμε πρώτα πως το εκτελέσιμο έχει μεταγλωττιστεί και κανένα σφάλμα δεν εμφανίζεται.

2. Στην κύρια γραμμή εργαλείων πατάμε το κουμπί Debug (αυτό με το πράσινο έντομο)



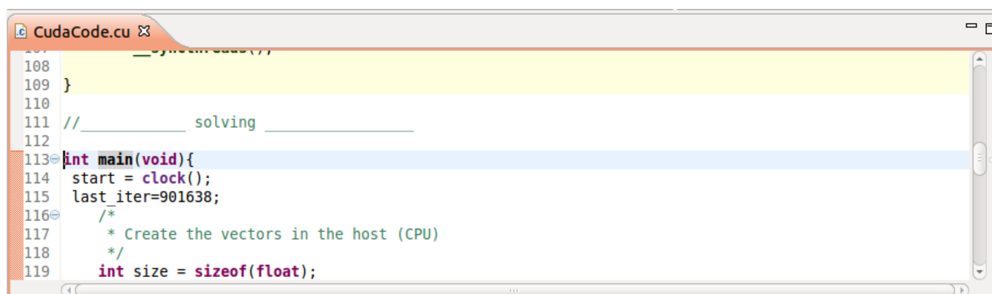
Εικόνα 4.12: Κουμπί για Debugging

3. Εάν τρέχουμε το Debug για πρώτη φορά θα μας προσφερθεί να κάνουμε αλλαγή της οπτικής το οποίο και αποδεχόμαστε. Η οπτική είναι μία προκαθορισμένη διάταξη παραθύρων που είναι σχεδιασμένη για συγκεκριμένες εργασίες.



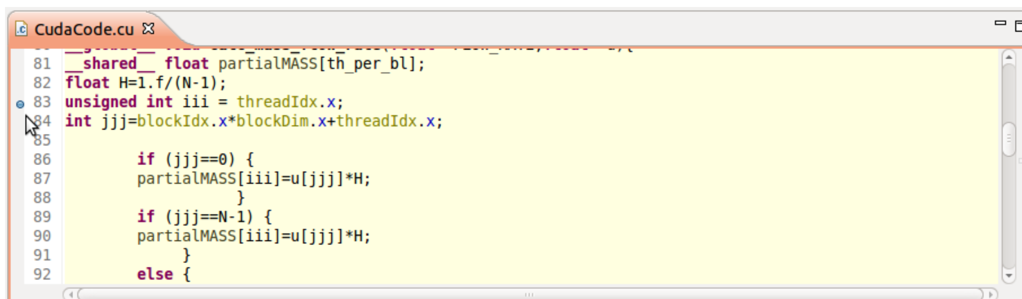
Εικόνα 4.13: Μετατροπή της οπτικής των παραθύρων

4. Η εφαρμογή θα διακοπεί στη συνάρτηση *main*. Σε αυτό το σημείο δεν υπάρχει κώδικας GPU που να εκτελείται.



Εικόνα 4.14: Διακοπή στη main

5. Προσθέτουμε το Breakpoint που θέλουμε στον κώδικα συσκευής και συνεχίζουμε την εφαρμογή.



Εικόνα 4.15: Τοποθέτηση Breakpoint

Ο Debugger θα σταματήσει όταν φτάσει στο breakpoint. Έτσι τώρα μπορούμε να εξερευνήσουμε την κατάσταση της συσκευής CUDA, να προχωρήσουμε μέσα στον κώδικα της GPU ή να συνεχίσουμε την εφαρμογή με αυτό το κουμπί

Εικόνα 4.16: Συνέχεια εφαρμογής

Κεφάλαιο 5

Επεξήγηση σειριακού κώδικα και παραλληλοποίηση

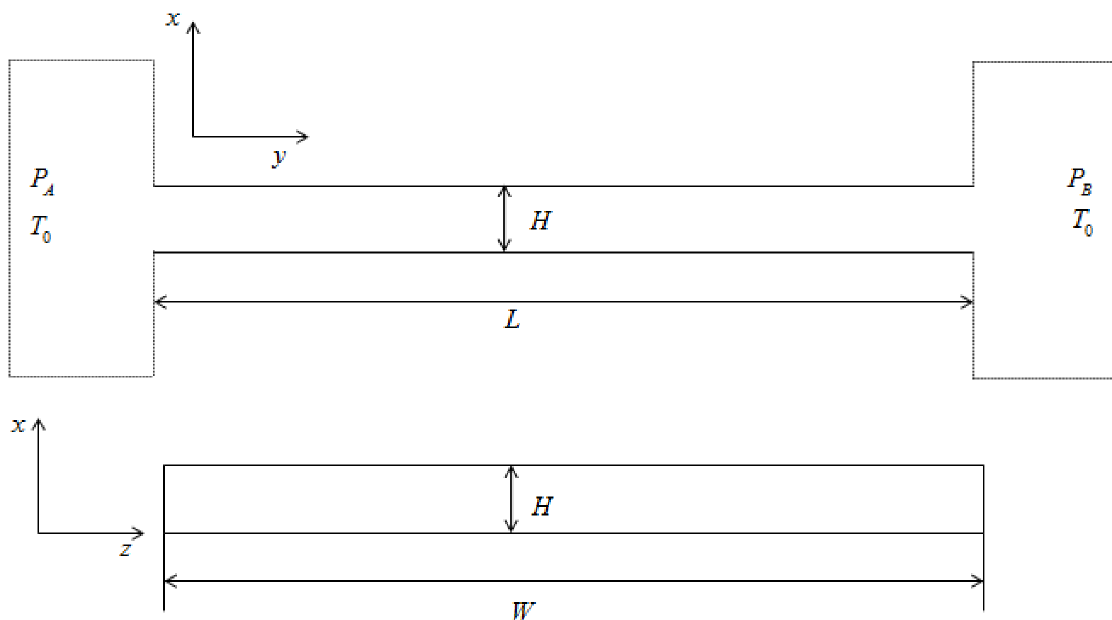
5.1 Επεξήγηση χαρακτηριστικών του προβλήματος

5.2 Επίλυση του προβλήματος με τον σειριακό κώδικα

5.3 Παραλληλοποίηση του κώδικα

5.1 Επεξήγηση χαρακτηριστικών του προβλήματος

Όπως αναλύθηκε και στο πρώτο κεφάλαιο το πρόβλημα που θα επιλυθεί σε αυτήν την πτυχιακή εργασία, με τη βοήθεια της αρχιτεκτονικής CUDA, έχει σαν ζητούμενο την εύρεση του ρυθμού ροής (flow rate) ενός αραιοποιημένου ρευστού διαμέσου ενός αγωγού από ένα δοχείο σε ένα άλλο λόγω της διαφοράς πίεσης που υπάρχει μεταξύ αυτών των δοχείων. Η διατομή αυτού του αγωγού είναι ορθογωνική και για λόγους διευκόλυνσης θεωρούμε ότι το πλάτος και το μήκος του L θεωρούνται πολύ μεγαλύτερα σε σχέση με το ύψος του H $(H << D)$. Όπως φαίνεται και στην Εικ 5.1 αν «κόψουμε» τον αγωγό (πάνω) κάθετα σε ένα τυχαίο σημείο τότε θα δούμε το σχήμα της Εικ 5.1 (κάτω).



Σχήμα 5.1: Χαρακτηριστικά του αγωγού

Αυτές οι παραδοχές καθιστούν το πρόβλημα να είναι μονοδιάστατο στον φυσικό χώρο αγνοώντας τα end effects που υπάρχουν τόσο στην κατεύθυνση y όσο και στην κατεύθυνση z με την κατανομή της αδιάστατης ταχύτητας κατά μήκος των δύο πλακών. Το πρόβλημα θεωρείται ισοθερμοκρασιακό όπως φαίνεται και στο Σχ. 5.1 με θερμοκρασία T_0.

5.2 Επίλυση του προβλήματος με τον σειριακό κώδικα

Σε αυτό το υποκεφάλαιο θα αναλυθεί ο τρόπος προγραμματισμού της μεθόδου DVM μέσα από την επίλυση του προβλήματος που έχει αναφερθεί στο Σχ. 5.1. Ο κώδικας [11] και για λόγους προσαρμογής σε γλώσσα προγραμματισμού CUDA έχει μετατραπεί πρώτα σε C++.

Η λύση λαμβάνεται από μια διαδικασία επανάληψης που υποδεικνύεται από τον δείκτη k της γραμμικοποιημένης BGK εξίσωσης (1.3). Θεωρούμε το $u(x)$ στη δεξιά πλευρά της εξίσωσης (1.3) και βρίσκουμε την τιμή $Y(x,\mu)$. Η πρώτη τιμή του $u(x)$ μπορεί να είναι μηδέν. Έπειτα αντικαθιστούμε την $Y(x,\mu)$ στη δεξιά πλευρά της εξίσωσης (1.3) και βρίσκουμε τη νέα τιμή $u(x)$ την οποία συγκρίνουμε με αυτήν της προηγούμενης επανάληψης. Η επανάληψη συνεχίζεται μέχρι να επιτευχθεί σύγκλιση.

Η σύγκλιση επιτυγχάνεται όταν η διαφορά της τιμής του $u(x)$ σε δύο διαδοχικές επαναλήψεις είναι μικρότερη από μία τυχαία πολύ μικρή τιμή την οποία επιλέγει να θέσει ο χρήστης ανάλογα με την ακρίβεια που επιθυμεί. Η επαναληπτική λοιπόν αυτή διαδικασία εφαρμόζεται στις διακριτοποιημένες εξισώσεις (1.3) και (1.4) και ορίζεται μέσα στον κώδικα κάπως έτσι:

```
Iter=0;
    while (relmax>ERROR) {
     …
}
```

Ξεκινάμε με τον ορισμό ενός συνόλου μοριακών ταχυτήτων $p_m, m=1,2,...,M$. Μπορούμε όπως αναφέραμε να το κάνουμε αυτό με πολλούς τρόπους. Εδώ επιλέγουμε τις ρίζες του πολυωνύμου Legendre. Το αρχείο Legendre.dat μπορεί να περιέχει έναν αριθμό ριζών που τον επιλέγει ο χρήστης. Η επιλογή βασίζεται στην επιθυμητή ακρίβεια η οποία σχετίζεται με την τιμή της παραμέτρου αραιοποίησης δ. Για μικρές τιμές δ απαιτείται ένας μεγάλος αριθμός ριζών σε αντίθεση με τις μεγάλες τιμές της παραμέτρου αραιοποίησης δ όπου εκεί απαιτείται ένας μεγάλος αριθμός κόμβων. Διαβάζουμε τις ρίζες και τα αντίστοιχα βάρη από το αρχείο Legendre.dat. Επισημαίνεται ότι στο Legendre.dat βρίσκονται μόνο τα μισά ζεύγη τιμών, δεδομένου ότι οι άλλες μισές ρίζες είναι οι ακριβώς αντίθετες από αυτές που ήδη υπάρχουν στο πρώτο μισό ενώ τα αντίστοιχα βάρη είναι ακριβώς τα ίδια με αυτά που ήδη υπάρχουν στο πρώτο μισό. Αυτό

συμβαίνει γιατί η σάρωση του πλέγματος θα γίνει δύο φορές. Μία προς τα πάνω με τις θετικές τιμές και μία προς τα κάτω με τις αρνητικές τιμές.

```
ifstream openfile1;  // read the roots
openfile1.open("Legendre_64.dat",ios::in);
   for(p=0; p<M/2; p++){
         openfile1>>CC[p]>>WW[p];
   }
   openfile1.close();

   for (p =0; p<M/2; p++)  // make the negatives
   {
     CC[M/2+p] = -CC[p];
     WW[M/2+p] = WW[p];
   }
```

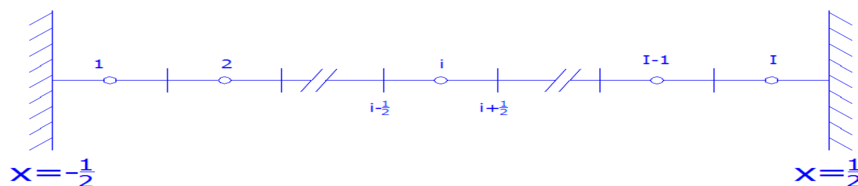
Το αρχικό σετ $p_m, m=1,2,...,M$ των μοριακών ταχυτήτων που έχουν επιλεγεί είναι απαραίτητο να μετατραπεί από $[-1,1]$ σε $[0,+\infty)$ για τις ανάγκες της ολοκλήρωσης Legendre. Αυτό επιτυγχάνεται με τη νέα μεταβλητή

$$\mu = \frac{1+p}{1-p} \text{ με } d\mu = \frac{2dp}{(1-p)^2}$$

η οποία ορίζει ένα σύνολο διακριτών ταχυτήτων $\mu_m, m=1,2,...,M$ στο $[0,+\infty)$

```
for (p=0; p<M; p++)      { //metatropi twn diakritwn taxititwn
      C[p]=(1+CC[p])/(1-CC[p]);
      W[p]=2.*WW[p]/pow((1-CC[p]),2);
      }
```

Στη συνέχεια, το χωρικό πεδίο $x \in [-1/2, 1/2]$ χωρίζεται σε I ίσες αποστάσεις με μήκος $H = 1/I$ και επίσης ορίζεται το μέσον i από κάθε διάστημα.



Έχουμε $Y(x,\mu)\big|_{i,m} = Y(x_i, \mu_m) = Y_{i,m}$ και διακριτοποιούμε την κινητική εξίσωση στους κόμβους (i,m), $i = 1,2,\ldots,I$, $m = 1,2,\ldots,M$ για να πάρουμε

$$\mu_m \left.\frac{\partial Y(x,\mu)^{(k+1/2)}}{\partial x}\right|_{i,m} + \delta Y(x,\mu)^{(k+1/2)}\Big|_{i,m} = \delta u(x)^{(k)}\Big|_i - \frac{1}{2} \quad (5.1)$$

Με προσέγγιση

$$\left.\frac{\partial Y}{\partial x}\right|_{i,m} = \frac{1}{h}\left(Y_{i+\frac{1}{2},m} - Y_{i-\frac{1}{2},m}\right) + O\left[h^2\right] \quad (5.2)$$

Και

$$Y_{i,m} = \frac{1}{2}\left(Y_{i+\frac{1}{2},m} + Y_{i-\frac{1}{2},m}\right) + O\left[h^2\right] \quad (5.3)$$

Συμπεραίνουμε

$$\mu_m \frac{Y^{(k+1/2)}_{i+\frac{1}{2},m} - Y^{(k+1/2)}_{i-\frac{1}{2},m}}{h} + \frac{\delta}{2}\left(Y^{(k+1/2)}_{i+\frac{1}{2},m} + Y^{(k+1/2)}_{i-\frac{1}{2},m}\right) = \frac{\delta}{2}\left(u_{i+\frac{1}{2}} + u_{i-\frac{1}{2}}\right)^{(k)} + \frac{1}{2} \quad (5.4)$$

Το παραπάνω σύστημα της Εξ. (5.4) έχει λυθεί ακολουθώντας τις πορείες των σωματιδίων. Αν αφήσουμε το $T_0 = \dfrac{h\delta}{2\mu_m}$ έχουμε:

Για $(\mu_m > 0)$:

$$Y^{(k+1/2)}_{i+\frac{1}{2},m} = \left[1+T_0\right]^{-1}\left\{\left(1-T_0\right)Y^{(k+1/2)}_{i+\frac{1}{2},m} + T_0\left(u_{i+\frac{1}{2}} + u_{i-\frac{1}{2}}\right)^{(k)} + \frac{h}{2\mu_m}\right\} \quad (5.5)$$

$m = 1, 2, \ldots, M \qquad i = 1, 2, \ldots, I$

Με $Y_{\frac{1}{2},m} = 0$.

```
//Oriakes sinthikes
Y[0][j]=0.; //Left wall
   for (i=1; i<N; i++)
   Y[i][j]=(((1-T0)*Y[i-1][j]+T0*(u[i-1]+u[i])+H/(2*C[j])))/(1+T0);
```

Για $(\mu_m < 0)$

$$Y_{i-\frac{1}{2},m}^{(k+1/2)} = \left[1 - T_0\right]^{-1} \left\{ \left(1 + T_0\right) Y_{i+\frac{1}{2},m}^{(k+1/2)} + T_0 \left(u_{i+\frac{1}{2}} + u_{i-\frac{1}{2}} \right)^{(k)} + \frac{1}{2\mu_m} \right\} \tag{5.6}$$

$m = 1, 2, \ldots, M \qquad i = 1, 2, \ldots, I$

Με $Y_{i+\frac{1}{2},m} = 0$

Στην Εξ. 5.6 ($\mu_m < 0$) εάν ορίσουμε $T_0 = \frac{h\delta}{2|\mu_m|}$ τότε ξαναγράφεται ως εξής

$$Y_{i-\frac{1}{2},m}^{(k+1/2)} = \left[1 + T_0\right]^{-1} \left\{ \left(1 - T_0\right) Y_{i+\frac{1}{2},m}^{(k+1/2)} + T_0 \left(u_{i+\frac{1}{2}} + u_{i-\frac{1}{2}} \right)^{(k)} + \frac{1}{2|\mu_m|} \right\} \tag{5.7}$$

Έτσι οι διακριτές Εξ. (5.5) και (5.7) για $\mu_m > 0$ και $\mu_m < 0$ αντίστοιχα έχουν την ίδια μορφή και το ίδιο σετ των διακριτών μοριακών ταχυτήτων $\mu \in [0, +\infty)$ μπορεί να χρησιμοποιηθεί. Αυτή είναι μία πρώτη εξήγηση γιατί το αρχικό σύνολο των διακριτών ταχυτήτων έχει μετατραπεί από $[-1,1]$ σε $[0, +\infty)$. Η μόνη διαφορά στις Εξ. (5.5) και (5.7) είναι ότι στην πρώτη σαρώνουμε το πλέγμα από κάτω προς τα πάνω (ή από τα αριστερά προς τα δεξιά) και στη δεύτερη από πάνω προς τα κάτω (ή από δεξιά προς τα αριστερά).

```
//Oriakes sinthikes
Y[N-1][j+M]=0.;   //Right wall
   for (i=N-2; i>=0; i--)
   Y[i][j+M]=(((1-T0)*Y[i+1][j+M]+T0*(u[i]+u[i+1])+H/(2*C[j])))/(1+T0);
```

Οι παραπάνω δύο βρόχοι αναφορικά με τη χωρική μεταβλητή συμπεριλαμβάνονται σε ένα βρόχο σχετικό με τις μοριακές ταχύτητες:

```
//Vroxos sxetikos me tis moriakes taxitites
for (j=0; j<M; j++)
      { …
}
```

Έτσι για κάθε διακριτή ταχύτητα περνάμε μέσα από το χωρικό πλέγμα.

Τέλος έχοντας τις τιμές του Y είναι αναγκαίο να ενσωματωθεί πάνω στο μ για να πάρουμε το u:

$$u(x)^{(k+1)} = \frac{1}{\sqrt{\pi}} \int_{-\infty}^{\infty} Y(x,\mu)^{(k+1/2)} e^{-\mu^2} d\mu =$$

$$= \frac{1}{\sqrt{\pi}} \int_{0}^{\infty} Y(x,\mu) e^{-\mu^2} d\mu + \frac{1}{\sqrt{\pi}} \int_{0}^{\infty} Y(x,-\mu) e^{-\mu^2} d\mu = \tag{5.8}$$

$$= 2\left[\sum_{m=1}^{M} \Upsilon(x,\mu_m) w_m e^{-\mu_m^2} + \sum_{m=M}^{2M} \Upsilon(x,\mu_m) w_m e^{-\mu_m^2}\right]$$

Όπως φαίνεται η ενσωμάτωση είναι πάνω στο $\mu \in [0,+\infty)$ και αυτός είναι ο δεύτερος λόγος της αλλαγής των μ_m από $[-1,1]$ σε $[0,+\infty)$.

```
for (i=0; i<N; i++)      {      //ipologismos tou neou u
      u[i] = 0.;
      for (j=0; j<M; j++)
            u[i] = u[i]+(Y[i][j]+Y[i][j+M]);
      u[i]=u[i]/RPi;
      }
```

Καταλήγουμε με την ενσωμάτωση της μακροσκοπικής ταχύτητας πάνω στο x για να βρούμε το ρυθμό ροής G χρησιμοποιώντας τον τραπεζοειδή κανόνα:

$$G = 2\int_{-1/2}^{1/2} u(x)dx = h\left[u_1 + 2\sum_{i=2}^{N-1} u_i + u_N\right] \tag{5.9}$$

Σε γενικές γραμμές η υλοποίηση του κώδικα έχει ως εξής:

Για κάθε ταχύτητα (M) του αρχείου Legendre.dat που διαβάζεται γεμίζει σειριακά ο πίνακας $Y(x,\mu)$ με τον εξής τρόπο: Για κάθε μοριακή ταχύτητα (M) υπολογίζεται η τιμή της συνάρτησης $Y(x,\mu)$ σε όλους τους κόμβους (N). Πριν να διαβάσει την επόμενη μοριακή ταχύτητα παίρνει από το αρχείο Legendre.dat την αντίστοιχη αντίθετη ταχύτητα και γεμίζει ανάποδα (από κάτω προς τα πάνω) την πρώτη στήλη του δεύτερου μισού του πίνακα. Αφού τελειώσει ο υπολογισμός σε όλους τους κόμβους διαβάζεται η επόμενη μοριακή ταχύτητα και επαναλαμβάνεται η ίδια διαδικασία μέχρι να γεμίσει ο πίνακας $Y(x,\mu)$.

	0 ... M-1	M ... 2*M-1
0	0 0 0 ... 0 0 0	. . . .
	X X X ... X X X	. . . .
	. . . .	. . . .
	. . . .	. . . .
	. . . .	. . . .
	. . . .	. . . .
	. . . .	. . . .
...	. . . .	. . . .
	. . . .	. . . .
	. . . .	. . . .
	. . . .	. . . .
	. . . .	X X X ... X X X
N-1	X X X ... X X X	0 0 0 ... 0 0 0

Εικόνα 5.1: Υπολογισμός πίνακα Y με σειριακή επεξεργασία

Αυτό γίνεται για κάθε επανάληψη. Και έπειτα από αυτό υπολογίζεται ο νέος πίνακας $u(N)$ ο οποίος είναι η συνολική ταχύτητα του ρευστού σε όλους τους κόμβους και υπολογίζεται από το άθροισμα των τιμών του πίνακα $Y(x,\mu)$ ανά γραμμή συν το αντίστοιχο $u(N)$ της προηγούμενης επανάληψης διαιρεμένο με τη ρίζα του π.

Y(x,μ)	0 ... M-1	M ... 2*M-1	u(x)	
0	0 0 0 ... 0 0 0	. . . .	X	0
	X X X ... X X X	. . . .	X	
	. . . .	. . . .	.	
	. . . .	. . . .	.	
	. . . .	. . . .	.	
	. . . .	. . . .	.	
	. . . .	. . . .	.	
...	. . . .	. . . .	.	
	. . . .	. . . .	.	
	. . . .	. . . .	.	
	. . . .	. . . .	.	
	. . . .	X X X ... X X X	.	
N-1	X X X ... X X X	0 0 0 ... 0 0 0	X	N-1

Εικόνα 5.2: Υπολογισμός πίνακα $u(N)$ με σειριακή επεξεργασία

Αν παρατηρήσει κανείς τη συνάρτηση $Y(x,\mu)$ θα καταλάβει πως για τον νέο πίνακα $Y(x,\mu)$ που θα προκύψει στην επόμενη επανάληψη χρειάζεται η τιμή της ταχύτητας $u(N)$ της προηγούμενης επανάληψης. Έτσι λοιπόν καταλαβαίνουμε πως για να ξεκινήσει μια επανάληψη πρέπει να έχει τελειώσει η προηγούμενη, πράγμα που καθιστά τον κώδικα σειριακό. Στη συνέχεια κρατάμε τις τιμές του $u(N)$ της τρέχουσας επανάληψης σε έναν πίνακα $uPR(N)$ έτσι ώστε να είμαστε σε θέση να κάνουμε τον έλεγχο με τις τιμές του πίνακα $u(N)$ της επόμενης επανάληψης. Ο έλεγχος αυτός έχει σκοπό να μας δείξει αν οι τιμές του πίνακα $u(N)$ σε δύο διαδοχικές επαναλήψεις είναι μικρότερες από την τυχαία τιμή που έχει θέσει ο χρήστης ανάλογα με την ακρίβεια που επιθυμεί.

```
relmax=0.f;
for (i=0; i<N; i++)
      relmax =relmax+fabsf(u[i]-uPR[i]);
```

Αν όχι ξεκινάει ξανά η παραπάνω διαδικασία από την αρχή. Αν όμως φτάσουμε στην τελευταία επανάληψη όπου το ERROR πλέον είναι όντως μικρότερο από την τυχαία τιμή, προχωρούμε στον υπολογισμό του αδιάστατου ρυθμού ροής (dimensionless flow rate) ακολουθώντας τον τραπεζοειδή κανόνα που δείξαμε και παραπάνω στην Εξ. (5.9) και φαίνεται και προγραμματιστικά στο παρακάτω κομμάτι κώδικα.

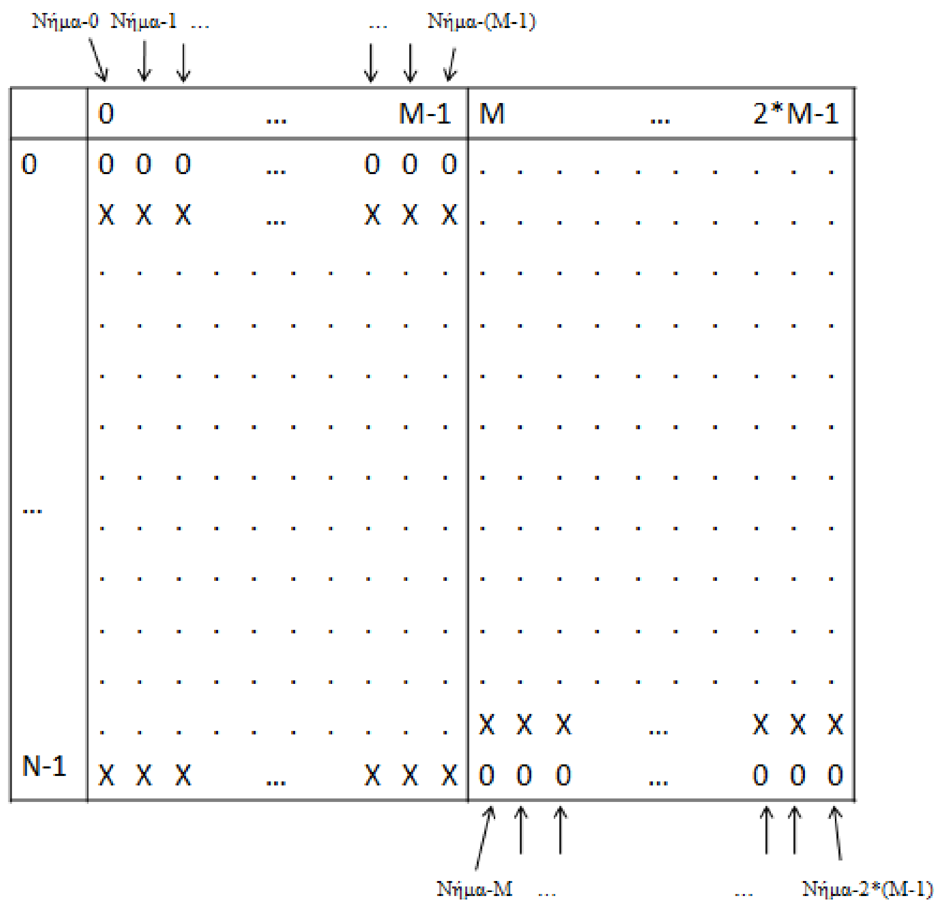
```
for (i=1; i<N-1; i++)
            S=S+u[i];
FLOW_RATE=(u[0]+(2.0*S)+u[N-1])*H;
```

Ο κανόνας αυτός λέει πως ο αδιάστατος ρυθμός ροής (flow rate) υπολογίζεται από το άθροισμα των ακριανών ταχυτήτων του πίνακα $u(N)$ με το διπλάσιο όλων των ενδιάμεσων ταχυτήτων του. Μπορούμε να πούμε λοιπόν πως μετά από όλη αυτή τη διαδικασία και από έναν αρκετά μεγάλο αριθμό επαναλήψεων, εξαρτώμενο πάντα από τις ανάγκες του προβλήματος και την επιθυμητή ακρίβεια που θέλει να επιτύχει ο χρήστης, έχουμε φτάσει στο σημείο να έχουμε λύση για το εξεταζόμενο πρόβλημα.

5.3 Παραλληλοποίηση του κώδικα

Στον απλό κώδικα είδαμε πως για τον υπολογισμό της τιμής της συνάρτησης $Y(x,\mu)$ διαβάζοντας μία-μία τις τιμές του αρχείου Legendre.dat και υπολογίζοντας την

τιμή της συνάρτησης σε όλους τους κόμβους χρειαζόμαστε την τιμή της συνάρτησης $Y(x,\mu)$ στον προηγούμενο κόμβο καθώς και την τιμή της ταχύτητας $u(N)$ στον προηγούμενο κόμβο. Για το λόγο αυτό η παραλληλοποίηση ως προς N είναι πάρα πολύ δύσκολη και απαιτεί εφαρμογή άλλων τεχνικών, αφού το κάθε νήμα για να εκτελεστεί θα χρειάζεται την τιμή του προηγούμενου νήματος για να δώσει αποτέλεσμα. Γι' αυτό το λόγο λοιπόν η παραλληλοποίηση προτιμάται να γίνει ως προς Μ, δηλαδή για κάθε κόμβο θα υπολογίζεται ταυτόχρονα η τιμή της συνάρτησης $Y(x,\mu)$ για όλες τις ταχύτητες του αρχείου Legendre.dat αφού μεταξύ αυτών των τιμών δεν υπάρχει αλληλεξάρτηση για τον υπολογισμό τους. Έτσι λοιπόν σε κάθε επανάληψη ο πίνακας $Y(x,\mu)$ θα υπολογίζεται κάπως έτσι:



Εικόνα 5.3: Υπολογισμός πίνακα $Y(x,\mu)$ με παράλληλη επεξεργασία

Όπου θα γεμίζει τώρα πια ανά γραμμή, ο πρώτος μισός πίνακας από πάνω προς τα κάτω και ο δεύτερος μισός ταυτόχρονα από κάτω προς τα πάνω. Αυτή η παράλληλη διαδικασία έχει συνταχθεί στον πρώτο πυρήνα ο οποίος καλείται σε κάθε επανάληψη. Εδώ πρέπει να σημειώσουμε πως επειδή η μνήμη της CUDA είναι σειριακή, ο πίνακας

$Y(x,\mu)$ από δισδιάστατος έχει μετατραπεί σε μονοδιάστατο και απλά για να υπάρξει αντιστοιχία με τον $Y(x,\mu)$ του σειριακού κώδικα χρησιμοποιούνται οι ενσωματωμένες μεταβλητές *blockDim.x*, και *threadIdx.x.*

```
__global__ void solveforY(float *Y,float *C, float *u){
int i;
float H=1.f/(N-1);
float TO=(0.5f*H*DELTA);
float TO2=H/2.f;
unsigned int j = threadIdx.x;
      if (j < M){
      Y[j]=0.f;               //left wall
      __syncthreads();
      for (i=1; i<N; i++)   {
            Y[i*blockDim.x+j]=(((1.f-(TO/C[j]))*(Y[(i*blockDim.x+j)-
(blockDim.x)])+((TO/C[j])*(u[i-1]+u[i]))+TO2/(C[j])))/(1.f+(TO/C[j]));
                    __syncthreads();
                                } //i
      }else       {
      Y[(N-1)*blockDim.x+j]=0.f;    //right wall
                    __syncthreads();
      for (i=N-2; i>=0; i--){
            Y[i*blockDim.x+j]=(((1.f-(TO/C[j-
M]))*(Y[((i+1)*blockDim.x+j)])+((TO/C[j-M])*(u[i]+u[i+1]))+TO2/(C[j-
M])))/(1.f+(TO/C[j-M]));
                    __syncthreads();

                                          } //i
```

Στη συνέχεια αφού έχει υπολογιστεί ο πίνακας $Y(x,\mu)$ για μία επανάληψη αυτό που ακολουθεί είναι ο υπολογισμός του πίνακα $u(N)$ ο οποίος γίνεται ξεχωριστά σε νέο πυρήνα. Ο πίνακας $u(N)$ είναι μονοδιάστατος και παίρνει σαν δεδομένο τον πίνακα $Y(x,\mu)$ ο οποίος είναι έτοιμος από τον προηγούμενο πυρήνα. Ο πίνακας $u(N)$ για να υπολογιστεί χρησιμοποιεί περισσότερα από ένα μπλοκ οπότε και κάνει χρήση της κοινόχρηστης μνήμης των μπλοκ. Για την ακρίβεια χρησιμοποιεί N μπλοκ κάθε ένα από τα οποία είναι υπεύθυνο για την άθροιση της κάθε γραμμής του πίνακα $Y(x,\mu)$. Έτσι υπολογίζεται ο πίνακας $u(N)$.

```
__global__ void solveforMacro(float *u,float *C,float *W,float *Y){
__shared__ float partialU[M];
unsigned int jj = threadIdx.x;
int ii=blockIdx.x*(2*blockDim.x)+threadIdx.x;
 partialU[jj]=(Y[ii]+Y[ii+(M)])*W[jj]*(__expf(-(__powf(C[jj],2))));
            __syncthreads();

for (unsigned int counter=blockDim.x/2 ; counter>0; counter >>= 1) {
      if (jj<counter)
            partialU[jj]+=partialU[jj+counter];
            __syncthreads();
            }
  if (jj==0)
      u[blockIdx.x] = partialU[0]/(RPi);
            __syncthreads();
}
```

Στον τρίτο πυρήνα έχουμε προσαρμόσει τον υπολογισμό ενός πίνακα ο οποίος περιλαμβάνει επιμέρους αθροίσματα του τελικού αδιάστατου ρυθμού ροής (flow rate). Εδώ είναι απαραίτητη η μετατροπή του κώδικα έτσι ώστε ανάλογα με τη δυναμικότητα της κάρτας γραφικών που χρησιμοποιείται να είναι σε θέση πάντα να δώσει αποτέλεσμα.

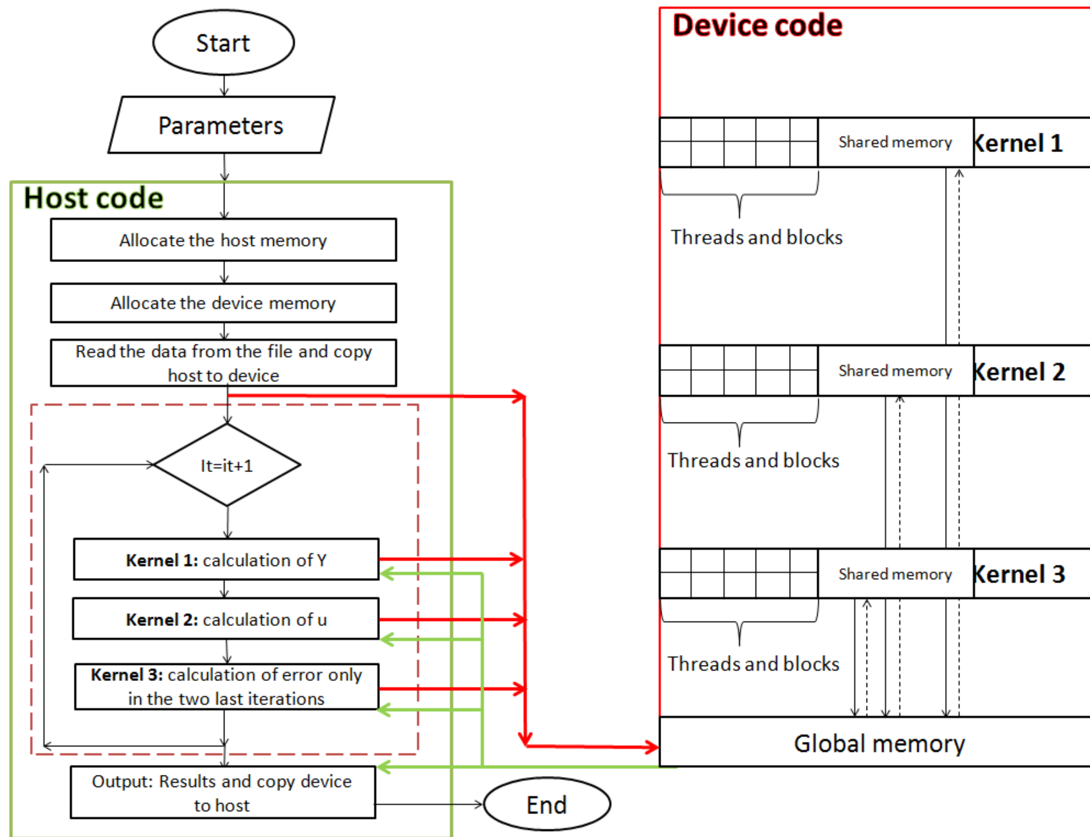
Ο συγκεκριμένος πυρήνας ουσιαστικά δημιουργεί ένα πλέγμα από μπλοκ που το καθένα κάθε φορά έχει 512 νήματα. Αυτά τα νήματα είναι ο αριθμός των κόμβων (N) που χρησιμοποιούνται. Εμείς όμως χρειαζόμαστε σε κάποιες περιπτώσεις μεγαλύτερη ακρίβεια και θέλουμε να έχουμε περισσότερους από 512 κόμβους. Ο αριθμός των κόμβων είναι μία σταθερά που την καθορίζει ο χρήστης ανάλογα με τις ανάγκες του προβλήματος και της ακρίβειας που θέλει να έχει οπότε όσο πιο πολλοί κόμβοι υπάρχουν τόσο πιο μεγάλη ακρίβεια θα έχει το αποτέλεσμα. Έτσι λοιπόν κάθε φορά που ο χρήστης επιλέγει έναν αριθμό κόμβων μεγαλύτερο από 512 ο πυρήνας κάνει αυτόματα το διαχωρισμό των νημάτων σε μπλοκ. Για την CUDA καλό είναι πάντα να χρησιμοποιούνται ισόποσα τα *threads per bloc*k δηλαδή όσα νήματα χρησιμοποιεί το ένα μπλοκ τόσα να χρησιμοποιούν όλα. Επίσης καλό είναι ο αριθμός των κόμβων που θέλουμε να χρησιμοποιήσουμε να είναι πολλαπλάσιος του 512 όπως είναι το 1024, το 2048, το 4096 κτλ γιατί αν το κάνω αυτό η κάρτα γραφικών μου χρησιμοποιεί καλά τη δυναμικότητά της. Τέλος εφ' όσον καλούμαστε να δημιουργήσουμε και πάλι

περισσότερα του ενός μπλοκ η χρήση της κοινόχρηστης μνήμης και σε αυτόν τον πυρήνα είναι απαραίτητη.

```
__global__ void calc_mass_flow_rate(float *FLOW_RATE,float *u){
__shared__ float partialMASS[th_per_bl];
float H=1.f/(N-1);
unsigned int iii = threadIdx.x;
int jjj=blockIdx.x*blockDim.x+threadIdx.x;
            if (jjj==0)
        partialMASS[iii]=u[jjj]*H;
            if (jjj==N-1)
          partialMASS[iii]=u[jjj]*H;
            else
        partialMASS[iii]=2.f*u[jjj]*H;
            __syncthreads();
for (unsigned int counter=blockDim.x/2 ; counter>0; counter >>= 1) {
            if (iii<counter)
                  partialMASS[iii]+=partialMASS[iii+counter];
            __syncthreads();
            }
         FLOW_RATE[blockIdx.x] = partialMASS[0];
            __syncthreads();
}
```

Έτσι λοιπόν αυτός ο πυρήνας ανάλογα με το πόσους κόμβους θα του δώσει ο χρήστης θα κάνει διαχωρισμό των νημάτων σε τόσα μπλοκ όσα αντιστοιχούν σε N συνολικά νήματα και θα δώσει και πάλι ένα τελικό άθροισμα από το κάθε μπλοκ το οποίο θα είναι ένα μεμονωμένο στοιχείο του πίνακα flow rate. Βγαίνοντας λοιπόν από τον πυρήνα και επιστρέφοντας τον πίνακα flow rate στην CPU γίνεται μια τελική άθροιση των στοιχείων του όπου και υπολογίζεται το ζητούμενο του προβλήματος.

Το Σχ. 5.2 παρουσιάζει ένα απλουστευμένο διάγραμμα ροής για την παράλληλη έκδοση του κώδικα όπως αυτός αναλύθηκε παραπάνω.



Σχήμα 5.2: Διάγραμμα ροής παράλληλου κώδικα

Όπως φαίνεται και στο διάγραμμα ροής η μεταφορά από την GPU στη CPU γίνεται μόνο μία φορά εφ' όσον έχει επιτευχθεί η σύγκλιση του κώδικα. Αυτό γίνεται γιατί η μεταφορά από και προς την CPU είναι μία χρονοβόρα διαδικασία και γενικότερα πρέπει όσο μπορεί να αποφεύγεται.

Θα πρέπει να σημειωθεί ότι ο παράλληλος κώδικας εκτελείται για έναν αριθμό επαναλήψεων που δίνεται από το χρήστη. Η μεγάλη εμπειρία που υπάρχει στο χώρο της κινητικής θεωρίας τα τελευταία χρόνια καθιστά δυνατή την επιλογή του αριθμού των επαναλήψεων ανάλογο με την παράμετρο αραιοποίησης δ ώστε να επιτευχθεί η επιθυμητή ακρίβεια.

Τόσο ο σειριακός όσο και ο παράλληλος κώδικας παρατίθενται με επεξήγηση στα Παραρτήματα Α και Β.

Κεφάλαιο 6

Συμπεράσματα και μελλοντική εργασία

6.1 Αποτελέσματα

6.2 Μελλοντική έρευνα

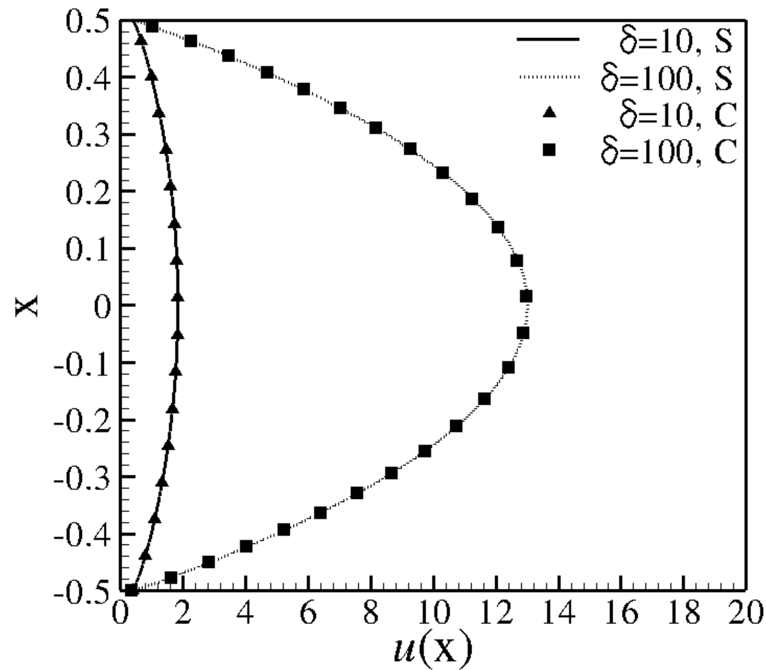
6.1 Αποτελέσματα

Στο παρόν κεφάλαιο θα παρουσιαστούν τα αποτελέσματα που αφορούν τους χρόνους εκτέλεσης του παράλληλου και του σειριακού κώδικα. Επιπρόσθετα θα δοθούν και κάποια αποτελέσματα για τον έλεγχο της ορθότητας των αποτελεσμάτων του παράλληλου κώδικα. Σε όλες τις περιπτώσεις το αρχείο Legendre περιέχει 64 ρίζες. Οι αριθμοί των κόμβων που μελετήθηκαν είναι 512, 1024, 2048 και 4096. Επίσης για κάθε έναν από τους προτεινόμενους αριθμούς κόμβων εξετάστηκαν τρεις τιμές της παραμέτρου αραιοποίησης δ ($\delta=10$, $\delta=100$ και $\delta=1000$). Τιμές για $\delta<10$ δεν εξετάστηκαν εξαιτίας της γρήγορης εκτέλεσης του κώδικα για τις συγκεκριμένες, τόσο για τις περιπτώσεις του σειριακού όσο και του παράλληλου κώδικα. Σε όλες τις περιπτώσεις το σφάλμα ERROR ήταν μικρότερο από 10^{-6}. Στον Πίν. 6.1 παρουσιάζονται οι τιμές του αδιάστατου ρυθμού ροής που έχουν προκύψει τόσο από την εκτέλεση του παράλληλου κώδικα, όσο και από την εκτέλεση του σειριακού για ένα αντιπροσωπευτικό εύρος της παραμέτρου αραιοποίησης.

Πίνακας 6.1: Αδιάστατες παροχές μάζας G για διάφορες τιμές της παραμέτρου αραιοποίησης δ.

δ	[12]	*Σειριακός κώδικας*	*Παράλληλος κώδικας*
0.1	2.03	2.03	2.03
1	1.54	1.54	1.54
10	2.76	2.77	2.77
100	17.7	17.7	17.7

Σε όλες τις περιπτώσεις τα αποτελέσματα της παράλληλης επεξεργασίας ταυτίζονται με τα αποτελέσματα της σειριακής επεξεργασίας. Παράλληλα τα αποτελέσματα συγκρίνονται με τα αντίστοιχα αποτελέσματα από την βιβλιογραφία και παρατηρείται η πολύ καλή ταύτιση αυτών. Στην Εικ. 6.1 δίνεται η κατανομή και η αδιάστατη μακροσκοπική ταχύτητα του ρευστού κατά μήκος των δύο πλακών όπως προκύπτει από την σειριακή και την παράλληλη επεξεργασία για δύο τιμές της παραμέτρου αραιοποίησης ($\delta=10$ και $\delta=100$).



Εικόνα 6.1: Κατανομή της αδιάστατης μακροσκοπικής ταχύτητας του αερίου κατά μήκος των δύο πλακών με χρήση του σειριακού κώδικα (S) και του παράλληλου (C).

Σε όλες τις περιπτώσεις βλέπουμε ότι τα αποτελέσματα που προκύπτουν από την παράλληλη επεξεργασία βρίσκονται σε άριστη συμφωνία με τα αποτελέσματα που προκύπτουν από τη σειριακή.

Έχοντας πιστοποιήσει την αξιοπιστία του παράλληλου κώδικα θα ακολουθήσει η περιγραφή που σχετίζεται με τους υπολογιστικούς χρόνους με χρήση και του σειριακού και του παράλληλου κώδικα. Σημειώνεται ότι ο κώδικας ο οποίος χρησιμοποιεί την τεχνολογία CUDA έχει εκτελεστεί σε τρεις διαφορετικές κάρτες γραφικών οι οποίες είναι η GeForce 8600M GT, η GeForce 9600 GT και η GeForce GT 640M. Η δυναμικότητα και τα τεχνικά χαρακτηριστικά των καρτών αυτών λαμβάνονται από την εκτέλεση της εντολής ./deviceQuery μέσα από τα Samples η εγκατάσταση των οποίων αναλύθηκε στο κεφάλαιο 4. Στις Εικ. 6.2, 6.3 και 6.4 φαίνονται τα τεχνικά χαρακτηριστικά της κάθε κάρτας γραφικών.

```
uth@uth-Aspire-5920G: ~/NVIDIA_CUDA-5.5_Samples/NVIDIA_CUDA-5.5_Samples/bin/x
Detected 1 CUDA Capable device(s)

Device 0: "GeForce 8600M GT"
  CUDA Driver Version / Runtime Version          6.0 / 5.5
  CUDA Capability Major/Minor version number:    1.1
  Total amount of global memory:                 255 MBytes (267714560 bytes)
  ( 4) Multiprocessors, (  8) CUDA Cores/MP:     32 CUDA Cores
  GPU Clock rate:                                950 MHz (0.95 GHz)
  Memory Clock rate:                             400 Mhz
  Memory Bus Width:                              128-bit
  Maximum Texture Dimension Size (x,y,z)         1D=(8192), 2D=(65536, 32768), 3
D=(2048, 2048, 2048)
  Maximum Layered 1D Texture Size, (num) layers  1D=(8192), 512 layers
  Maximum Layered 2D Texture Size, (num) layers  2D=(8192, 8192), 512 layers
  Total amount of constant memory:               65536 bytes
  Total amount of shared memory per block:       16384 bytes
  Total number of registers available per block: 8192
  Warp size:                                     32
  Maximum number of threads per multiprocessor:  768
  Maximum number of threads per block:           512
  Max dimension size of a thread block (x,y,z): (512, 512, 64)
  Max dimension size of a grid size    (x,y,z): (65535, 65535, 1)
  Maximum memory pitch:                          2147483647 bytes
  Texture alignment:                             256 bytes
  Concurrent copy and kernel execution:          Yes with 1 copy engine(s)
  Run time limit on kernels:                     Yes
  Integrated GPU sharing Host Memory:            No
  Support host page-locked memory mapping:       Yes
  Alignment requirement for Surfaces:            Yes
  Device has ECC support:                        Disabled
  Device supports Unified Addressing (UVA):      No
  Device PCI Bus ID / PCI location ID:           1 / 0
  Compute Mode:
     < Default (multiple host threads can use ::cudaSetDevice() with device simu
ltaneously) >

deviceQuery, CUDA Driver = CUDART, CUDA Driver Version = 6.0, CUDA Runtime Versi
on = 5.5, NumDevs = 1, Device0 = GeForce 8600M GT
Result = PASS
uth@uth-Aspire-5920G:~/NVIDIA_CUDA-5.5_Samples/NVIDIA_CUDA-5.5_Samples/bin/x86_64/linux/release$
```

Εικόνα 6.2: Τεχνικά χαρακτηριστικά GeForce 8600M GT

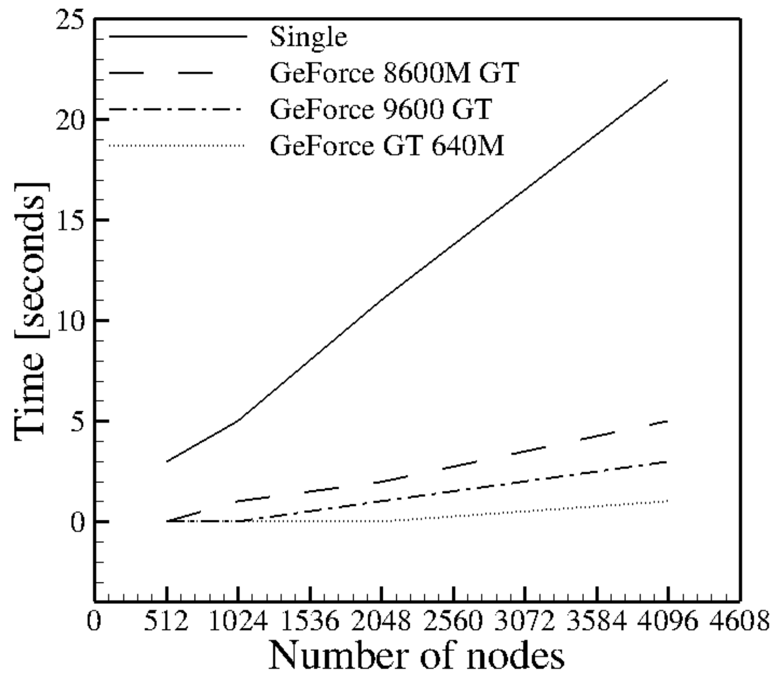
```
evi@evi-System-Product-Name:~/NVIDIA_CUDA-5.0_Samples/1_Utilities/deviceQuery$ .
/deviceQuery
./deviceQuery Starting...

 CUDA Device Query (Runtime API) version (CUDART static linking)

Detected 1 CUDA Capable device(s)

Device 0: "GeForce 9600 GT"
  CUDA Driver Version / Runtime Version          5.0 / 5.0
  CUDA Capability Major/Minor version number:    1.1
  Total amount of global memory:                 1024 MBytes (1073414144 bytes)
  ( 8) Multiprocessors x (  8) CUDA Cores/MP:    64 CUDA Cores
  GPU Clock rate:                                1625 MHz (1.62 GHz)
  Memory Clock rate:                             900 Mhz
  Memory Bus Width:                              256-bit
  Max Texture Dimension Size (x,y,z)             1D=(8192), 2D=(65536,32768), 3D
=(2048,2048,2048)
  Max Layered Texture Size (dim) x layers        1D=(8192) x 512, 2D=(8192,8192)
 x 512
  Total amount of constant memory:               65536 bytes
  Total amount of shared memory per block:       16384 bytes
  Total number of registers available per block: 8192
  Warp size:                                     32
  Maximum number of threads per multiprocessor:  768
  Maximum number of threads per block:           512
  Maximum sizes of each dimension of a block:    512 x 512 x 64
  Maximum sizes of each dimension of a grid:     65535 x 65535 x 1
  Maximum memory pitch:                          2147483647 bytes
  Texture alignment:                             256 bytes
  Concurrent copy and kernel execution:          Yes with 1 copy engine(s)
  Run time limit on kernels:                     Yes
  Integrated GPU sharing Host Memory:            No
  Support host page-locked memory mapping:       Yes
  Alignment requirement for Surfaces:            Yes
  Device has ECC support:                        Disabled
  Device supports Unified Addressing (UVA):      No
  Device PCI Bus ID / PCI location ID:           1 / 0
  Compute Mode:
     < Default (multiple host threads can use ::cudaSetDevice() with device simu
ltaneously) >

deviceQuery, CUDA Driver = CUDART, CUDA Driver Version = 5.0, CUDA Runtime Versi
on = 5.0, NumDevs = 1, Device0 = GeForce 9600 GT
evi@evi-System-Product-Name:~/NVIDIA_CUDA-5.0_Samples/1_Utilities/deviceQuery$
```

Εικόνα 6.3: Τεχνικά χαρακτηριστικά GeForce 9600 GT

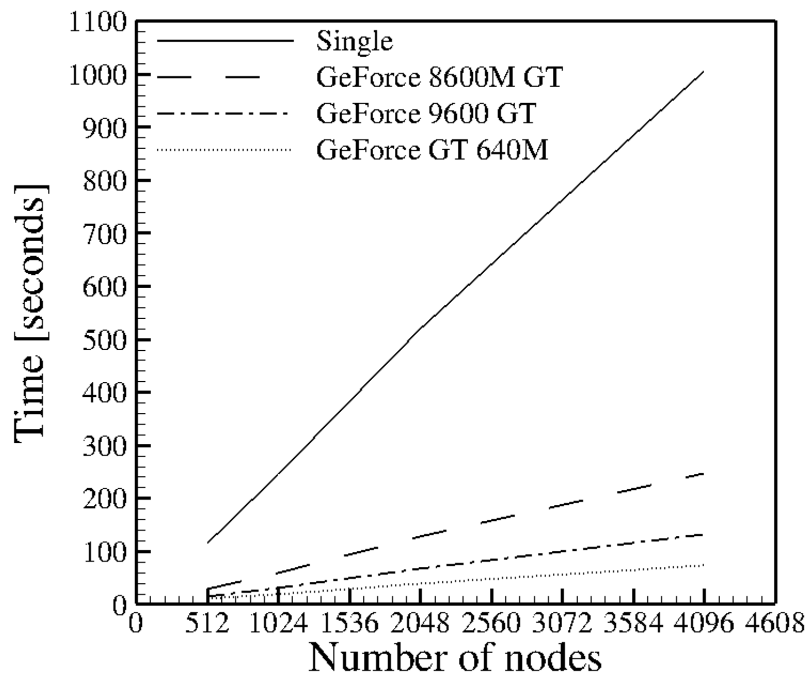
```
chris@chris-SATELLITE-P855:~/NVIDIA_CUDA-5.5_Samples/NVIDIA_CUDA-5.5_Samples/1_U
tilities/deviceQuery$ optirun ./deviceQuery
./deviceQuery Starting...

 CUDA Device Query (Runtime API) version (CUDART static linking)

Detected 1 CUDA Capable device(s)

Device 0: "GeForce GT 640M"
  CUDA Driver Version / Runtime Version          6.0 / 5.5
  CUDA Capability Major/Minor version number:    3.0
  Total amount of global memory:                 2048 MBytes (2147287040 bytes)
  ( 2) Multiprocessors, (192) CUDA Cores/MP:     384 CUDA Cores
  GPU Clock rate:                                709 MHz (0.71 GHz)
  Memory Clock rate:                             900 Mhz
  Memory Bus Width:                              128-bit
  L2 Cache Size:                                 262144 bytes
  Maximum Texture Dimension Size (x,y,z)         1D=(65536), 2D=(65536, 65536),
3D=(4096, 4096, 4096)
  Maximum Layered 1D Texture Size, (num) layers  1D=(16384), 2048 layers
  Maximum Layered 2D Texture Size, (num) layers  2D=(16384, 16384), 2048 layers
  Total amount of constant memory:               65536 bytes
  Total amount of shared memory per block:       49152 bytes
  Total number of registers available per block: 65536
  Warp size:                                     32
  Maximum number of threads per multiprocessor:  2048
  Maximum number of threads per block:           1024
  Max dimension size of a thread block (x,y,z): (1024, 1024, 64)
  Max dimension size of a grid size    (x,y,z): (2147483647, 65535, 65535)
  Maximum memory pitch:                          2147483647 bytes
  Texture alignment:                             512 bytes
  Concurrent copy and kernel execution:          Yes with 1 copy engine(s)
  Run time limit on kernels:                     Yes
  Integrated GPU sharing Host Memory:            No
  Support host page-locked memory mapping:       Yes
  Alignment requirement for Surfaces:            Yes
  Device has ECC support:                        Disabled
  Device supports Unified Addressing (UVA):      Yes
  Device PCI Bus ID / PCI location ID:           1 / 0
  Compute Mode:
     < Default (multiple host threads can use ::cudaSetDevice() with device simu
ltaneously) >

deviceQuery, CUDA Driver = CUDART, CUDA Driver Version = 6.0, CUDA Runtime Versi
on = 5.5, NumDevs = 1, Device0 = GeForce GT 640M
Result = PASS
chris@chris-SATELLITE-P855:~/NVIDIA_CUDA-5.5_Samples/NVIDIA_CUDA-5.5_Samples/1_Utilities/deviceQuery$
```

Εικόνα 6.4: Τεχνικά χαρακτηριστικά GeForce GT 640M

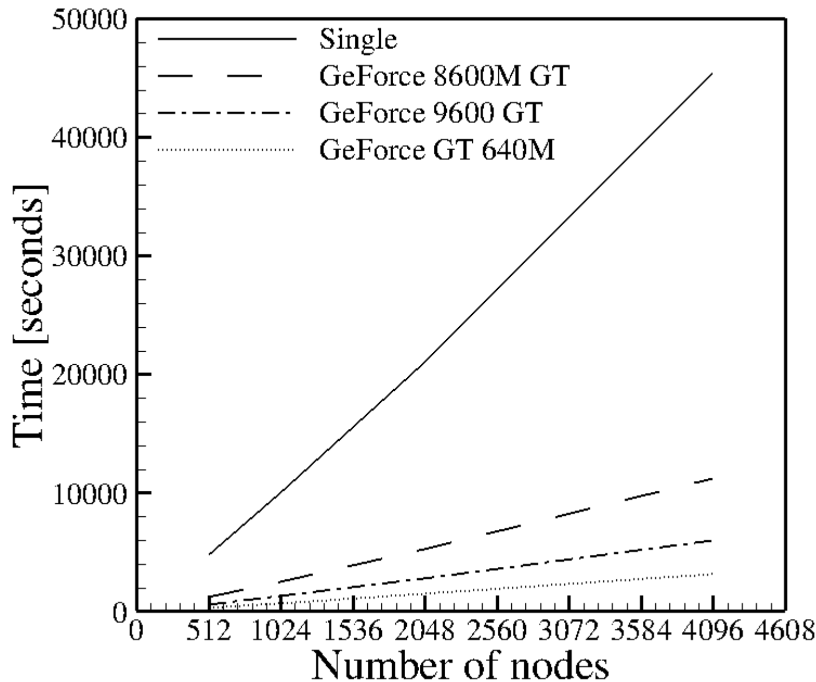
Στις Εικ. 6.5, 6.6 και 6.7 δίνονται οι χρόνοι εκτέλεσης σε δευτερόλεπτα (sec) τόσο του σειριακού όσο και του παράλληλου κώδικα σε σχέση με τον αριθμό των κόμβων (N).



Εικόνα 6.5: Σύγκριση υπολογιστικών χρόνων μεταξύ διαφορετικών καρτών γραφικών για δ=10.



Εικόνα 6.6: Σύγκριση υπολογιστικών χρόνων μεταξύ διαφορετικών καρτών γραφικών για δ=100.



Εικόνα 6.7: Σύγκριση υπολογιστικών χρόνων μεταξύ διαφορετικών καρτών γραφικών για $\delta = 1000$.

Σημειώνεται ότι σε όλες τις περιπτώσεις ο παράλληλος κώδικας είναι ταχύτερος από τον αντίστοιχο σειριακό. Επιπλέον παρατηρείται ότι όσο αυξάνονται οι αριθμοί των κόμβων τόσο πιο αποδοτική γίνεται η εκτέλεση του παράλληλου κώδικα δεδομένου ότι σε αυτήν την περίπτωση χρησιμοποιούνται περισσότερα μπλοκ και νήματα. Με τα αποτελέσματα αυτά αναδεικνύεται η σημαντικότητα της εφαρμογής της τεχνολογίας CUDA σε προβλήματα όπως είναι αυτά της κινητικής θεωρίας.

6.2 Μελλοντική έρευνα

Με την εργασία αυτή γίνεται μια προσπάθεια εφαρμογής της τεχνολογίας CUDA με σκοπό την επίλυση ενός αρκετά διαδεδομένου προβλήματος της μηχανικής ρευστών σε όλο το εύρος της παραμέτρου αραιοποίησης δ. Τα αποτελέσματα της Ενότητας 6.1 ανέδειξαν τα σημαντικά οφέλη σε υπολογιστικό χρόνο που προκύπτουν από την εφαρμογή της συγκεκριμένης τεχνολογίας. Η συγκεκριμένη εργασία θα μπορούσε να επεκταθεί με σκοπό την επίλυση δισδιάστατων ροών καθώς και πιο πολύπλοκων υπολογιστικών πεδίων. Παράλληλα μια ακόμα σημαντική κίνηση θα ήταν η παραλληλοποίηση στον φυσικό χώρο κάνοντας χρήση προηγμένων υπολογιστικών μεθόδων με χρήση της τεχνολογίας CUDA.

ΑΝΑΦΟΡΕΣ

[1] F M White. Viscous Fluid Flows. McGraw-Hill, New York, 1974.

[2] S Harris. An Introduction to the Theory of the Boltzmann Equation. Dover Publications, New York, 1971.

[3] J.H. Ferziger, H.G. Kaper, Mathematical Theory of Transport Processes in Gases, North-Holland, Amsterdam, 1972.

[4] P L Bhatnagar, E P Gross, and M A Krook. A model for collision processes in gases. Phys. Rev., 94:511-525, 1954.

[5] E J Broadwell. Study of rarefied flow by the discrete velocity method. J. Fluid Mech., 19:401-414, 1964.

[6] G A Bird. Molecular Gas Dynamics and the Direct Simulation of Gas Flows. Oxford, University Press, Oxford, 1994.

[7] F. Sharipov, “Rarefied gas flow through a long rectangular channel”, J. Vac. Sci. Tech. A 17(5), 1999.

[8] S. Varoutis, V. Hauer, C. Day and D. Valougeorgis, “Experimental and computational study of gas flows through long channels of various cross sections in the whole range of the Knudsen number”, J. Vac. Sci. Tech. A, 27 (1), 89-100, 2009

[9] S. Naris, D. Valougeorgis, The driven cavity flow over the whole range of the Knudsen number, Physics of Fluids, 17(9), 907106, 2005.

[10] F. Sharipov, Application of the Cercignani-Lampis scattering kernel to calculations of rarefied gas flows. I. Plane flow between two parallel plates, European Journal of Mechanics - B/Fluids, 21, 113-123, 2002.

[11] http://vacutecschool2012.uth.gr/

[12] F. Sharipov, V. Seleznev, Data on internal rarefied gas flows, Journal of Physical and Chemical Reference Data, 27(3), 657-706, 1998.

[13] Kirk, David B. and Hwu Wen-Mei W, Προγραμματισμός Μαζικά Παράλληλων Επεξεργαστών, κλειδάριθμος, 2010.

[14] J. Sanders and E. Kandrot, Cuda by example: An introduction to General-Purpose GPU Programming, 2010.

[15] Σ. Παπαδάκης και Κ. Διαμαντάρας, Προγραμματισμός και Αρχιτεκτονική Συστημάτων Παράλληλης Επεξεργασίας, εκδόσεις Κλειδάριθμος, 2011.

ΠΗΓΕΣ INTERNET:

http://docs.nvidia.com/cuda/cuda-c-best-practices-guide/
http://mcclanahoochie.com/blog/wp-content/uploads/2011/03/gpu-hist-paper.pdf
http://www.astro.auth.gr/documents/diplomas_MSc/Antoniadis-presentation-CUDA-2.pdf
http://www.uni-graz.at/~liebma/CUDA/NVISION08-Getting_Started_with_CUDA.pdf
http://www.cs.cmu.edu/afs/cs/academic/class/15869-f11/www/readings/lindholm08_tesla.pdf
https://www.google.gr/url?sa=t&rct=j&q=&esrc=s&source=web&cd=1&cad=rja&uact=8&ved=0CC0QFjAA&url=http%3A%2F%2Fwww.physics.ntua.gr%2F~konstant%2Fhomepage%2Fdiplomatikes%2F10.Karkoulis.pdf&ei=5FgjU5frMaKR0AXP5oG4Dg&usg=AFQjCNHvhyHFURqOqEcozG0elAPF-8OhUw&sig2=_cjOECKSRHN36mbd9D6DIw
http://www.ogf.org/OGF25/materials/1605/CUDA_Programming.pdf
http://www.cs.unc.edu/~prins/Classes/633/Readings/CUDA_C_Programming_Guide_4.2.pdf
http://vivliothmmy.ee.auth.gr/713/1/Diplomati_Nalpmantis_Georgios_Koufos_Dimitrios.pdf
http://vivliothmmy.ee.auth.gr/1724/1/Diplomatiki_Karafyllias-Manolis.pdf
https://dspace.lib.uom.gr/bitstream/2159/15100/11/SalonikidisDionysiosMsc2012.pdf
http://sn0v.wordpress.com/2012/12/07/installing-cuda-5-on-ubuntu-12-04/
http://www.somewhereville.com/?p=1896
http://docs.nvidia.com/cuda/cuda-getting-started-guide-for-linux/index.html
http://www.it.uom.gr/teaching/llnl-gr/Introduction%20to%20Parallel%20Computing.htm
http://www.cs.rutgers.edu/~venugopa/parallel_summer2012/cuda.html
http://docs.nvidia.com/cuda/cuda-c-programming-guide/
http://cs.txstate.edu/labs/tutorials/tut_docs/EclipseTutorial.pdf
http://en.wikipedia.org/wiki/Nvidia_Optimus
https://wiki.ubuntu.com/Bumblebee
http://docs.nvidia.com/cuda/cuda-runtime-api/
https://devtalk.nvidia.com/default/topic/368105/cuda-programming-and-performance/cuda-occupancy-calculator-helps-pick-optimal-thread-block-size/
Farber, Rob (2011), CUDA Application Design and Development.

ΠΑΡΑΡΤΗΜΑ Α

Σειριακός (single) κώδικας για την επίλυση της ροής Poiseuille

```
1.   #include <stdio.h>
2.   #include <stdlib.h>
3.   #include <iostream>
4.   #include <math.h>
5.   #include <istream>
6.   #include <fstream>
7.   #include <iomanip>
8.   #include <time.h>

9.   using namespace std;

10.  #define M 64
11.  #define N 2048
12.  #define DELTA 100
13.  #define ERROR 0.000001
14.  #define Pi 3.141592654f
15.  #define RPi 1.772453851f

16.  float CC[M], WW[M], C[M], W[M];
17.  float Y[N][2*M];
18.  float  u[N], uPR[N];
19.  float FLOW_RATE, S, H;
20.  int i, j;
21.  float relmax=1000., T0;
22.  int ITER;
23.  time_t start, end;
24.  double cpuTime;
```

```
25.   int main(void){

26.   H=1.f/(N-1.);
27.   start = clock();

28.   ifstream openfile;
29.      openfile.open("Legendre64.dat",ios::in);
30.           for(i=0; i<M/2; i++)
31.                 openfile>>CC[i]>>WW[i];
32.        openfile.close();

33.    for (i =0; i<M/2; i++){
34.           CC[M/2+i] = -CC[i];
35.           WW[M/2+i] = WW[i];
36.   }

37.    for (i=0; i<M; i++)      {
38.           C[i]=(1+CC[i])/(1-CC[i]);
39.           W[i]=2.*WW[i]/pow((1-CC[i]),2);
40.   }

41.    for (i=0; i<N; i++)
42.           u[i]=0.;

43.    ITER = 0;
44.    while (relmax>ERROR)     {
45.           for (j=0; j<M; j++)           {
46.                 T0=(0.5f*H*DELTA)/C[j];
47.                 Y[0][j]=0;
48.           for (i=1; i<N; i++)
49.    Y[i][j]=(((1.f-T0)*Y[i-1][j]+T0*(u[i-1]+
      u[i])+H/(2*C[j])))/(1+T0);
```

```
50.             Y[N-1][j+M]=0.;
51.         for (i=N-2; i>=0; i--)
52.     Y[i][j+M]=(((1.f-
        T0)*Y[i+1][j+M]+T0*(u[i]+u[i+1])+H/(2*C[j])))/(1+T0);
53.         }

54.     for (i=0; i<N; i++)
55.         for (j=0; j<M; j++){
56.         Y[i][j]=Y[i][j]*W[j]*(exp(-pow(C[j],2)));
57.         Y[i][j+M]=Y[i][j+M]*W[j]*(exp(-pow(C[j],2)));
58.         }
59.
60.         for (i=0; i<N; i++)
61.         uPR[i]=u[i];

62.         for (i=0; i<N; i++)     {
63.             u[i] = 0.;
64.             for (j=0; j<M; j++)
65.                 u[i] = u[i]+(Y[i][j]+Y[i][j+M]);
66.             u[i]=u[i]/RPi;
67.         }

68.     relmax=0.f;
69.     for (i=0; i<N; i++){
70.         relmax =relmax+fabsf(u[i]-uPR[i]);}

71.     cout <<  "ITER=" << ITER << "\t" <<"Relmax=" << relmax << endl;
72.      ITER++;
73.     }

74.     ofstream resultsfile;
75.     resultsfile.open("Results.txt");
```

```
76.   resultsfile  <<  "\n";

77.   resultsfile << "################ MAIN PARAMETERS ################"
     <<  "\n";

78.  resultsfile << "GAUSS LEGENDRE QUADRATURE POINTS=" << M << "\n";

79.  resultsfile << "MESH POINTS                    =" << N << "\n";

80.

81.  //*******************************************

82.  //**    Calculate the Flow Rate            **

83.  //*******************************************

84.

85.  //Integrate velocity over x [-0.5,0.5] using Trapezoidal rule

86.  //(Eq. 14)

87.

88.   for (i=1; i<N-1; i++)

89.         S=S+u[i];

90.   FLOW_RATE=(u[0]+(2.f*S)+u[N-1])*H;

91.   resultsfile << "RAREFACTION PARAMETER            =" << DELTA <<
     "\n";

92.   cout << "RAREFACTION PARAMETER           =" << DELTA << "\n"<<
     endl;

93.   resultsfile << "NUMBER OF ITERATIONS        =" << ITER << "\n";

94.   resultsfile << "################ MAIN PARAMETERS ################"
     <<  "\n";

95.   resultsfile << "\n";

96.   resultsfile << "\t" << "x" << "\t\t" << "VELOCITY OF GAS  " <<
     "\n";

97.   for (i=0; i<N; i++)

98.         resultsfile << "\t" << -0.5f+i*H << "\t\t" << u[i] << "\n";

99.

100.  resultsfile << "\n";
```

```
101.   resultsfile << "Dimensionless FLOW RATE=" << FLOW_RATE << "\n";
102.   cout << "Dimensionless  RATE          =" << FLOW_RATE << "\n"<<
     endl;
103.   resultsfile << "\n";
104.   resultsfile <<
     "*************************************************************";
105.
106.         /*resultsfile << " pinakas Y"<< "\n\n\n";
107.   for (i=0; i<N; i++){
108.   resultsfile << "i="<< i <<"\n";
109.         for(j=0; j<2*M; j++)
110.         resultsfile << "j=" << j << "\t" << Y[i][j] << "\t";
111.   resultsfile << "\n";}*/
112.   resultsfile.close();
113.  end = clock();
114.
115.
116.  cpuTime= (end-start)/ (CLOCKS_PER_SEC);
117.  cout  << "Seconds of the running="<<cpuTime<< endl;
118.  system("PAUSE");
119.  return 0;
120.  }
```

A. Επεξήγηση του κώδικα

Γραμμές 1-8: Δηλώνονται οι απαραίτητες βιβλιοθήκες.

Γραμμή 9: Ορίζεται ο χώρος των μεταβλητών και των συναρτήσεων

Γραμμές 10-15: Δηλώνονται οι σταθερές

Γραμμές 16-24: Δηλώνονται οι μεταβλητές

Γραμμή 25: Ξεκινάει η main

Γραμμή 26: Το χωρικό πεδίο χωρίζεται σε ίσα κομμάτια

Γραμμή 27: Η μεταβλητή start θα μετρήσει τα δευτερόλεπτα που θα κάνει ο κώδικας για να εκτελεστεί με τη βοήθεια του ρολογιού του επεξεργαστή.

Γραμμές 28-32: Δημιουργείται ένα ρεύμα το οποίο θα διαβάσει τις δύο στήλες του αρχείου Legendre64.dat. Η πρώτη στήλη περιέχει 32 τιμές μοριακής ταχύτητας οι οποίες θα αποθηκευθούν στο μισό πίνακα CC ενώ η δεύτερη στήλη περιέχει τα αντίστοιχα βάρη των τιμών τους τα οποία θα αποθηκευθούν στο μισό πίνακα WW.

Γραμμές 33-36: Ο υπόλοιπος μισός πίνακας CC γεμίζει με τις αντίθετες τιμές της πρώτης στήλης του αρχείου Legendre64.dat. Ενώ ο υπόλοιπος μισός πίνακας WW γεμίζει με τα ίδια βάρη.

Γραμμές 37-40: Στη συνέχεια ο πίνακας CC και ο πίνακας WW χρειάζονται να μετασχηματιστούν για τις ανάγκες της ολοκλήρωσης. Τις νέες τιμές παίρνουν οι πίνακες C και W αντίστοιχα

Γραμμές 41-42: Δίνονται αρχικές τιμές στον πίνακα u ο οποίος περιέχει τις τιμές της ταχύτητας του ρευστού σε κάθε κόμβο.

Γραμμές 43-73: Είναι η κύρια επανάληψη για τη λύση της διαφορικής εξίσωσης BGK.

Γραμμή 43: Η μεταβλητή ITER κρατάει τον αριθμό των επαναλήψεων ο οποίος χρησιμοποιείται μόνο στην εμφάνιση.

Γραμμή 44: Η σταθερά ERROR έχει δοθεί εμπειρικά και υποδηλώνει ακριβώς πόσο μικρό πρέπει να είναι το σφάλμα έτσι ώστε να θεωρείται αμελητέο.

Γραμμή 45: Θα γίνουν Μ επαναλήψεις, μία για κάθε μοριακή ταχύτητα του αρχείου Legendre64.dat.

Γραμμή 46: Για όλες τις μοριακές ταχύτητες υπολογίζεται μία θερμοκρασία σύμφωνα με κάποιες σταθερές και με το αντίστοιχο βάρος της κάθε μοριακής ταχύτητας.

Γραμμή 47: Η τιμή της ταχύτητας στο κάτω τοίχωμα είναι μηδέν.

Γραμμή 48: Υπολογισμός της ταχύτητας του ρευστού σε όλους τους κόμβους

Γραμμή 49: Ο πίνακας Y είναι δύο διαστάσεων [N,2*M]. Η κάθε γραμμή περιλαμβάνει για κάθε κόμβο ένα σύνολο ταχυτήτων. Σε αυτό το κομμάτι [N,M] υπολογίζεται το πρώτο μισό του πίνακα καθ’ ότι η σάρωση των κόμβων πρέπει να γίνει δύο φορές, μία από κάτω προς τα πάνω και μία από πάνω προς τα κάτω όπως επιβάλει και η τεχνική εύρεσης του αποτελέσματος.

Γραμμή 50: Η τιμή της ταχύτητας στο πάνω τοίχωμα είναι μηδέν.

Γραμμή 51: Το γέμισμα τώρα γίνεται ανάποδα γιατί πρέπει να παρθούν οι σωστές τιμές του αρχείου Legendre.dat στον κάθε κόμβο.

Γραμμή 52: Γεμίζει ο υπόλοιπος μισός πίνακας

Γραμμή 53: Ο πίνακας Y έχει γεμίσει.

Γραμμές 54-58: Τα στοιχεία του πίνακα Y χρειάζονται απλά κάποιες επιπλέον πράξεις

Γραμμές 60-61: Ο πίνακας uPR[i] κρατάει την προηγούμενη τιμή από τον πίνακα u[i]

Γραμμές 62-67: Υπολογίζεται ο νέος πίνακας u με βάση την προηγούμενη επανάληψη. Ο πίνακας u είναι μονοδιάστατος με N στοιχεία τα οποία είναι το άθροισμα της κάθε γραμμής του πίνακα Y, δηλαδή η συνολική ταχύτητα του ρευστού σε κάθε κόμβο.

Γραμμή 68: Η τιμή της relmax γίνεται 0

Γραμμή 69-70: Υπολογίζεται το relmax για όλους τους κόμβους το οποίο σχετίζει το uPR[i] με το u[i]

Γραμμή 71: Εμφανίζεται το relmax στην αντίστοιχη επανάληψη

Γραμμή 72: Αυξάνεται ο αριθμός της επανάληψης

Γραμμή 74: Δημιουργείται ένα ρεύμα για την εγγραφή των αποτελεσμάτων στο αρχείο Results.txt

Γραμμές 88-89: Γίνεται μία πρόσθεση των στοιχείων του πίνακα u στη μεταβλητή S εκτός από τα δύο ακριανά, u[0] και u[N-1] τα οποία πρέπει πρώτα να διπλασιαστούν πριν προστεθούν στο τελικό αποτέλεσμα

Γραμμή 90: Υπολογίζεται τελικά ο ρυθμός ροής του ρευστού.

Γραμμή 116: Υπολογίζεται ο χρόνος που έκανε η cpu για να τρέξει τον κώδικα.

ΠΑΡΑΡΤΗΜΑ Β

Παράλληλος (parallel) κώδικας για την επίλυση της ροής Poiseuille

```
1.   #include <stdio.h>
2.   #include <stdlib.h>
3.   #include <iostream>
4.   #include <math.h>
5.   #include <istream>
6.   #include <fstream>
7.   #include <iomanip>
8.   #include <time.h>
9.   #include <cuda.h>
10.  #include <cuda_runtime_api.h>

11.  using namespace std;

12.  #define M 64
13.  #define N 512
14.  #define th_per_bl 512
15.  #define DELTA 10
16.  #define Pi  3.141592654f
17.  #define RPi 1.772453851f

18.  float CC[M], WW[M];
19.  float mass;
20.  int p,Iter,k,last_iter;
21.  clock_t start, end;
22.  double cpuTime;

23.  __global__ void solveforY(float *Y,float *C, float *u){
24.  int i;
```

```
25.  float H=1.f/(N-1);

26.  float TO=(0.5f*H*DELTA);

27.  float TO2=H/2.f;

28.  unsigned int j = threadIdx.x;

29.  if (j < M)  {

30.        Y[j]=0.f;               //left wall

31.        __syncthreads();

32.        for (i=1; i<N; i++)  {

33.  Y[i*blockDim.x+j]=(((1.f-(TO/C[j]))*(Y[(i*blockDim.x+j)-
     (blockDim.x)])+((TO/C[j])*(u[i-1]
     +u[i]))+TO2/(C[j])))/(1.f+(TO/C[j]));

34.        __syncthreads();

35.           } //i

36.  }else {

37.        Y[(N-1)*blockDim.x+j]=0.f;          //right wall

38.        __syncthreads();

39.        for (i=N-2; i>=0; i--)  {

40.  Y[i*blockDim.x+j]=(((1.f-(TO/C[j-M]))*(Y[((i+1)*blockDim.x+j)])
     +((TO/C[j-M])*(u[i]+u[i+1]))+TO2/(C[j-M])))/(1.f+(TO/C[j-M]));

41.        __syncthreads();

42.           } //i

43.        }

44.  }


45.  __global__ void solveforMacro(float *u,float *C,float *W,float
     *Y){

46.  __shared__ float partialU[M];

47.  unsigned int jj = threadIdx.x;

48.  int ii=blockIdx.x*(2*blockDim.x)+threadIdx.x;

49.  partialU[jj]=(Y[ii]+Y[ii+(M)])*W[jj]*(__expf(-(__powf(C[jj],2))));

50.        __syncthreads();
```

```
51.  for (unsigned int counter=blockDim.x/2 ; counter>0; counter>>=1) {
52.        if (jj<counter)
53.              partialU[jj]+=partialU[jj+counter];
54.        __syncthreads();
55.              }
56.      if (jj==0)
57.            u[blockIdx.x] = partialU[0]/(RPi);
58.        __syncthreads();
59.  }
60.  __global__ void calc_mass_flow_rate(float *FLOW_RATE,float *u){
61.  __shared__ float partialMASS[th_per_bl];
62.  float H=1.f/(N-1);
63.  unsigned int iii = threadIdx.x;
64.  int jjj=blockIdx.x*blockDim.x+threadIdx.x;
65.        if (jjj==0)
66.          partialMASS[iii]=u[jjj]*H;
67.        if (jjj==N-1)
68.      partialMASS[iii]=u[jjj]*H;
69.        else
70.          partialMASS[iii]=2.f*u[jjj]*H;
71.        __syncthreads();
72.  for (unsigned int counter=blockDim.x/2 ; counter>0; counter >>=
     1){
73.        if (iii<counter)
74.              partialMASS[iii]+=partialMASS[iii+counter];
75.        __syncthreads();
76.        }
77.        FLOW_RATE[blockIdx.x] = partialMASS[0];
```

```
78.         __syncthreads();
79.    }

80.    int main(void){
81.    start = clock();
82.    last_iter=471;

83.    int size = sizeof(float);
84.    float *u = (float *)malloc(N*size);
85.    float *Y = (float *)malloc(2*N*M*size);
86.    float *C = (float *)malloc(M*size);
87.    float *W = (float *)malloc(M*size);
88.    float *FLOW_RATE = (float *)malloc((N/th_per_bl)*size);

89.    ifstream openfile1;
90.    openfile1.open("Legendre_64.dat",ios::in);
91.    for(p=0; p<M/2; p++)
92.          openfile1>>CC[p]>>WW[p];
93.    openfile1.close();

94.    for (p =0; p<M/2; p++){
95.    CC[M/2+p] = -CC[p];
96.    WW[M/2+p] = WW[p];
97.    }

98.    for (p=0; p<M; p++)      {
99.    C[p]=(1+CC[p])/(1-CC[p]);
100.   W[p]=2.*WW[p]/pow((1-CC[p]),2);
101.   }

102.   for (p=0; p<N; p++)
```

```
103.    u[p] = 0.f;

104. float *u_dev, *C_dev, *Y_dev,*W_dev,*FLOW_RATE_dev;
105. cudaMalloc((void **)&u_dev, N*sizeof(float));
106. cudaMalloc((void **)&Y_dev, 2*M*N*sizeof(float));
107. cudaMalloc((void **)&C_dev, M*sizeof(float));
108. cudaMalloc((void **)&W_dev, M*sizeof(float));
109. cudaMalloc((void **)&FLOW_RATE_dev,(N/th_per_bl)*sizeof(float));

110. cudaMemcpy(u_dev, u, N*sizeof(float), cudaMemcpyHostToDevice);
111. cudaMemcpy(C_dev, C, M*sizeof(float), cudaMemcpyHostToDevice);
112. cudaMemcpy(W_dev, W, M*sizeof(float), cudaMemcpyHostToDevice);

113. Iter=0;
114. while (Iter<last_iter)  {           //ITER
115. Iter=Iter+1;
116. solveforY<<< 1, 2*M>>>(Y_dev,C_dev,u_dev);
117. solveforMacro<<< N, M>>>(u_dev,C_dev,W_dev,Y_dev);
118. if (Iter==last_iter)
119. calc_mass_flow_rate<<<N/th_per_bl,
     th_per_bl>>>(FLOW_RATE_dev,u_dev);
120.    }


121. cudaMemcpy(FLOW_RATE,FLOW_RATE_dev,(N/th_per_bl)*sizeof(float),cud
     aMemcpyDeviceToHost);
122. cudaMemcpy(u, u_dev, N*sizeof(float),cudaMemcpyDeviceToHost);
123. cudaFree(u_dev);
124. cudaFree(W_dev);
125. cudaFree(C_dev);
126. cudaFree(Y_dev);
127. cudaFree(FLOW_RATE_dev);
```

```
128. ofstream resultsfile;
129. resultsfile.open("Results.txt");
130. resultsfile  <<  "\n";
131. resultsfile << "################ MAIN PARAMETERS ################"
     << "\n";
132. resultsfile << "GAUSS LEGENDRE QUADRATURE POINTS=" << M <<"\n";
133. resultsfile << "MESH POINTS                    =" << N <<"\n";
134. resultsfile << "RAREFACTION PARAMETER       =" << DELTA <<"\n";
135. cout << setprecision (28) << "RAREFACTION PARAMETER =" << DELTA <<
     "\n"<< endl;
136. resultsfile << "################ MAIN PARAMETERS ################"
     <<"\n";
137. resultsfile << "\n";
138. resultsfile << "\t" << "x" << "\t\t" << "VELOCITY OF GAS  "
     <<"\n";
139. for (p=0; p<N; p++)
140.       resultsfile << "\t" << -0.5+p*(1.f/(N-1)) << "\t\t" << u[p]
     << "\n";

141. resultsfile << "\n";
142. mass=0.f;
143. for (p=0; p<N/th_per_bl; p++)
144.       mass+=FLOW_RATE[p];

145. resultsfile << "Dimensionless FLOW RATE=" << mass << "\n";
146. cout  << "Dimensionless  RATE         =" << mass << "\n"<< endl;
147. resultsfile << "\n";
148. resultsfile <<"***********************************************";
149. resultsfile.close();

150. end = clock();
151. cpuTime= (end-start)/ (CLOCKS_PER_SEC);
```

```
152. cout  << "Seconds of the running="<<cpuTime<< endl;
153.   return 0;
154. }
155. //_________________END__________________
```

B. Επεξήγηση του κώδικα

Γραμμές 1-10: Δηλώνονται οι απαραίτητες βιβλιοθήκες.

Γραμμή 11: Ορίζεται ο χώρος των μεταβλητών και των συναρτήσεων.

Γραμμές 12-17: Δηλώνονται οι σταθερές.

Γραμμές 18-22: Δηλώνονται οι μεταβλητές.

Γραμμή 23: Δηλώνεται η πρώτη συνάρτηση πυρήνα που θα εκτελεστεί στην κάρτα γραφικών με παραμέτρους στον πίνακα Y, τον πίνακα C και τον πίνακα u.

Γραμμές 24-27: Δηλώνονται κάποιες ακόμα τοπικές μεταβλητές.

Γραμμή 28: Η μεταβλητή j περιέχει τον αριθμοδείκτη του κάθε νήματος.

Γραμμή 29: Εάν το j είναι μικρότερο από M.

Γραμμή 30: Γέμισε τα πρώτα M στοιχεία του πίνακα με 0 λόγω του ότι η ταχύτητα στα τοιχώματα είναι μηδέν.

Γραμμή 31: Συγχρονίζονται τα threads για να είναι έτοιμα για την επόμενη χρήση τους.

Γραμμή 32: Για όλους τους κόμβους.

Γραμμή 33: Υπολογίζει το Y για όλες τις μοριακές ταχύτητες ανά κόμβο μόνο για τα θετικά M.

Γραμμή 34: Συγχρονίζονται τα threads για να είναι έτοιμα για την επόμενη χρήση τους.

Γραμμή 36: Αλλιώς εάν το M είναι μεγαλύτερο από 64.

Γραμμή 37: Οι τελευταίες 64 τιμές του πίνακα Y θα είναι και πάλι μηδέν λόγω του ότι η τιμή της ταχύτητας στα τοιχώματα είναι μηδέν.

Γραμμή 38: Συγχρονίζονται τα threads για να είναι έτοιμα για την επόμενη χρήση τους.

Γραμμή 39-40: Υπολογίζει το Y για όλες τις μοριακές ταχύτητες ανά κόμβο και για τα αρνητικά M.

Γραμμή 41: Συγχρονίζονται τα threads για να είναι έτοιμα για την επόμενη χρήση τους.

Γραμμή 45: Δεύτερος kernel με παραμέτρους τον πίνακα u, τον πίνακα C, τον πίνακα W και τον πίνακα Y.

Γραμμή 46: Ο πίνακας partialU δηλώνεται πίνακας της shared memory.

Γραμμή 47: Η μεταβλητή jj είναι ο αριθμοδείκτης του κάθε νήματος.

Γραμμή 48: Η μεταβλητή ii είναι ο δείκτης για την προσπέλαση του πίνακα Y.

Γραμμή 49: Ο πίνακας partialU γεμίζει σύμφωνα με κάποιες πράξεις πάνω στα στοιχεία του Y

Γραμμή 50: Συγχρονίζονται τα threads για να είναι έτοιμα για την επόμενη χρήση τους.

Γραμμές 51-53: Γίνεται μία άθροιση των στοιχείων του πίνακα partialU του κάθε block στο στοιχείο partialU[0].

Γραμμές 54: Συγχρονίζονται τα threads για να είναι έτοιμα για την επόμενη χρήση τους.

Γραμμές 56-57: Όταν τελειώσει η άθροιση τότε το κάθε στοιχείο του πίνακα u αποτελεί το κάθε επιμέρους άθροισμα του κάθε partialU ανά block διαιρεμένο με το RPi.

Γραμμή 58: Συγχρονίζονται τα threads για να είναι έτοιμα για την επόμενη χρήση τους.

Γραμμή 60: Ορίζεται ο τρίτος kernel με παραμέτρους τον πίνακα FLOW RATE και τον πίνακα u

Γραμμή 61: Δηλώνεται ο πίνακας partialMASS στην κοινόχρηστη μνήμη.

Γραμμή 62: Δηλώνεται και πάλι το χωρικό πεδίο που θα χωριστεί από τους κόμβους.

Γραμμή 63: Η μεταβλητή iii είναι ο αριθμοδείκτης του κάθε νήματος.

Γραμμή 64: Η μεταβλητή jjj είναι ο αριθμοδείκτης του κάθε στοιχείου του πίνακα u.

Γραμμές 65-66: Το πρώτο στοιχείο του πίνακα partialMASS είναι το πρώτο στοιχείο του πίνακα u πολλαπλασιασμένο με το H.

Γραμμές 67-68: Το τελευταίο στοιχείο του πίνακα partialMASS είναι το τελευταίο στοιχείο του πίνακα u πολλαπλασιασμένο με το H.

Γραμμές 69-70: Όλα τα υπόλοιπα είναι διπλασιασμένα.

Γραμμή 71: Συγχρονίζονται τα threads για να είναι έτοιμα για την επόμενη χρήση τους.

Γραμμές 72-74: Γίνεται μία άθροιση των στοιχείων του πίνακα partialMASS του κάθε block στο στοιχείο partialMASS[0].

Γραμμή 75: Συγχρονίζονται τα threads για να είναι έτοιμα για την επόμενη χρήση τους.

Γραμμή 77: Όταν τελειώσει η άθροιση τότε το κάθε στοιχείο του πίνακα FLOW RATE αποτελεί το κάθε επιμέρους άθροισμα του κάθε partialMASS ανά block.

Γραμμή 80: Ξεκινάει η main().

Γραμμή 81: Η μεταβλητή start ξεκινάει να μετράει το χρόνο εκτέλεσης του προγράμματος.

Γραμμή 82: Η μεταβλητή last_iter δείχνει τον αριθμό των επαναλήψεων που θέλουμε να κάνει ο κώδικας ο οποίος είναι αποτέλεσμα του σειριακού κώδικα και μπαίνει σαν δεδομένο στον παράλληλο.

Γραμμές 83-88: Δεσμεύονται οι θέσεις μνήμης στη CPU.

Γραμμή 89: Δημιουργείται το ρεύμα openfile1.

Γραμμή 90: Ανοίγει το αρχείο Legendre_64.dat.

Γραμμές 91-92: Διαβάζονται οι τιμές του αρχείου και τοποθετούνται μέσα σε δύο πίνακες, τον CC και τον WW.

Γραμμή 93: Κλείνει το αρχείο Legendre_64.dat.

Γραμμές 94-97: Ο υπόλοιπος μισός πίνακας CC γεμίζει με τις αντίθετες τιμές του πρώτου μισού ενώ ο WW γεμίζει ξανά με τις ίδιες.

Γραμμές 98-101: Γίνεται ο μετασχηματισμός των τιμών από [-1,1] σε [0,+∞]

Γραμμές 102-103: Αρχικοποιείται ο πίνακας u.

Γραμμές 104-109: Δεσμεύονται οι θέσεις μνήμης και στην GPU.

Γραμμές 110-112: Γίνεται αντιγραφή των πινάκων που θα χρησιμοποιηθούν από την CPU στην GPU.

Γραμμή 113: Αρχικοποιείται ο μετρητής των επαναλήψεων

Γραμμή 114: Όσο ο αριθμός των επαναλήψεων δεν είναι ίδιος με τον αριθμό των επαναλήψεων που έκανε ο απλός κώδικας τότε

Γραμμή 115: Αύξανε κατά ένα τον αριθμό τον επαναλήψεων

Γραμμή 116: Και κάλεσε τον πρώτο πυρήνα με 1 μπλοκ και 2*M νήματα με παραμέτρους τους πίνακες Y_dev, C_dev και u_dev οι οποίοι είναι οι αντίστοιχοι Y, C, u που έχουν αντιγραφεί από την CPU στην GPU.

Γραμμή 117: Έπειτα κάλεσε τον δεύτερο πυρήνα με N μπλοκ και M νήματα το κάθε μπλοκ με παραμέτρους τους πίνακες u_dev, C,dev, W_dev και Y_dev.

Γραμμή 118: Αν είναι στην τελευταία επανάληψη

Γραμμές 119-120: Κάλεσε τον τρίτο πυρήνα με μπλοκ N/th_per_block και νήματα th_per_block για να δημιουργηθούν σύμφωνα με την κάρτα γραφικών τόσα μπλοκ με όλα τα νήματά τους να χρησιμοποιούνται και παραμέτρους τους πίνακες FLOW_RATE_dev και u_dev.

Γραμμές 121-122: Αντιγράφονται τα αποτελέσματα από τους πίνακες FLOW_RATE_dev και u_dev στους αντίστοιχους της CPU

Γραμμές 123-127: Απελευθερώνονται οι θέσεις μνήμης στην GPU

Γραμμές 128-129: Δημιουργείται ένα ρεύμα προς ένα αρχείου εξόδου για να τυπωθούν τα αποτελέσματα. Το αρχείο Results.txt.

Γραμμές 130-141: Τυπώνονται κάποια δεδομένα στο αρχείο και κάποια αποτελέσματα από την κάθε επανάληψη

Γραμμές 142-144: Στην μεταβλητή mass γίνεται μία τελευταία άθροιση του πίνακα FLOW_RATE για να πάρουμε το τελικό FLOW RATE που είναι ένας αδιάστατος αριθμός.

Γραμμές 145-149: Τυπώνεται το αποτέλεσμα της μεταβλητής mass και στο αρχείο αλλά και στην οθόνη και το αρχείο κλείνει αφού έχουν καταγραφεί όλα τα αποτελέσματα

Γραμμή 150: Η μεταβλητή end μετράει το χρόνο που έχει εκείνη τη στιγμή η CPU

Γραμμή 151: Η μεταβλητή cpuTime θα μας δώσει το συνολικό χρόνο που έκανε το πρόγραμμα για να συγκλίνει σε δευτερόλεπτα.

Γραμμή 152: Τυπώνονται και τα δευτερόλεπτα στην οθόνη

Γραμμές 153-155: Τέλος του προγράμματος