Εθνικό Μετσόβιο Πολυτεχνείο
Σχολή Μηχανολόγων Μηχανικών
Τομέας Ρευστών
Εργαστήριο Θερμικών Στροβιλομηχανών
Μονάδα Παράλληλης Υπολογιστικής
Ρευστοδυναμικής & Βελτιστοποίησης

Διπλωματική Εργασία

Γεώργιος Σ. Ελευθερίου

Προγραμματισμός Παράλληλου Επιλύτη Εξισώσεων Euler για 3Δ Ροές σε Δομημένα Πλέγματα σε Συστοιχίες Καρτών Γραφικών

Επιβλέπων
Κ. Χ. Γιαννάκογλου, Καθηγητής ΕΜΠ

Οκτώβριος 2012

Εθνικο Μετσοβιο Πολυτεχνειο
Σχολη Μηχανολογων Μηχανικων
Τομεας Ρευστων
Εργαστηριο Θερμικων Στροβιλομηχανων
Μοναδα Παραλληλης Υπολογιστικης Ρευστοδυναμικης & Βελτιστοποιησης

Προγραμματισμός Παράλληλου Επιλύτη Εξισώσεων Euler για 3Δ Ροές σε Δομημένα Πλέγματα σε Συστοιχίες Καρτών Γραφικών

Διπλωματική Εργασία
Γεώργιος Σ. Ελευθερίου

Επιβλέπων: Κ.Χ. Γιαννάκογλου
Καθηγητής ΕΜΠ

Οκτώβριος 2012

Τα τελευταία τέσσερα έτη, η Μονάδα Παράλληλης Υπολογιστικής Ρευστοδυναμικής & Βελτιστοποίησης του Εργαστηρίου Θερμικών Στροβιλομηχανών (ΕΘΣ) του ΕΜΠ δραστηριοποιείται ερευνητικά στην εκμετάλλευση των δυνατοτήτων των καρτών γραφικών σε κώδικες και εφαρμογές υπολογιστικής ρευστοδυναμικής και βελτιστοποίησης. Σχετικά με τους επιλύτες των εξισώσεων ροής (εξισώσεις Euler, Navier-Stokes), το λογισμικό που ήδη έχει αναπτυχθεί σε CUDA C δίνει, σε κάρτες γραφικών τελευταίας τεχνολογίας της NVIDIA, επιταχύνσεις έως $\times 100$, ανάλογα με το πλέγμα, την εφαρμογή κτλ. Οι επιλύτες που αναπτύχθηκαν χειρίζονται μη-δομημένα αλλά και δομημένα πλέγματα με μεθόδους πεπερασμένων όγκων κεντροκομβικής διατύπωσης των εξισώσεων. Σκοπός της παρούσας διπλωματικής εργασίας είναι η ανάπτυξη/προγραμματισμός (με CUDA C και POSIX threads) ενός παράλληλου επιλύτη εξισώσεων Euler εκτελούμενου σε συστοιχία καρτών γραφικών και η μέτρηση της επιτάχυνσης ως προς την αντίστοιχη υλοποίηση για μία κάρτα γραφικών. Βασίστηκε στον υπάρχοντα επιλύτη για μία κάρτα γραφικών, ο οποίος επιλύει 3Δ εξισώσεις Euler για συμπιεστή ροή σε δομημένα πλέγματα, με διακριτοποίηση πεπερασμένων όγκων κεντροκομβικής διατύπωσης και μέθοδο χρονοπροέλασης (time marching). Ο τελικός κώδικας, εκτελούμενος μέχρι και σε τρεις κάρτες γραφικών NVIDIA TESLA M2050 ενός υπολογιστικού κόμβου, συγκρίθηκε με τον υπάρχοντα για μία κάρτα γραφικών. Η λύση της ροής και η σύγκλιση των δύο επιλυτών είναι ταυτόσημη και ο επιλύτης που αναπτύχθηκε έχει επιτάχυνση έως και $\times 2.7$ ως προς τον αντίστοιχο για μία κάρτα γραφικών.

NATIONAL TECHNICAL UNIVERSITY OF ATHENS
School of Mechanical Engineering
Fluids Section
Laboratory of Thermal Turbomachines
Parallel CFD & Optimization Unit

Programming of a Parallel Flow Solver for 3D Inviscid Flows using Structured Grids on a Many-GPU Cluster

Diploma Thesis
Georgios S. Eleftheriou

Advisor: K. C. Giannakoglou

October 2012

During the past four years, the Parallel CFD & Optimization Unit of the Laboratory of Thermal Turbomachines of the National Technical University of Athens has been focusing on the exploitation of the computational power of new generation Graphics Processing Units (GPU) for CFD and Optimization software and applications. In the field of CFD, in-house solvers, for both the Euler and Navier-Stokes equations, have already been programmed in CUDA C. These codes, handling both structured and unstructured grids and running on modern NVIDIA GPUs, have achieved speed-ups of up to $\times 100$ (compared to a modern CPU), depending on the grid in use, the application etc. The scope of this diploma thesis, is the development (in CUDA C and POSIX threads) of a parallel Euler solver running on a many-GPU cluster, and the measurement of speed-up with respect to the single GPU implementation. The development was based on the existing single GPU solver, which solves the 3D Euler equations for compressible flow , using a vertex-centered finite volume method for structured grids and a time-marching method. The final code was tested on up to 3 GPUs NVIDIA TESLA M2050 of a single computational node. Compared against the single GPU implementation, it delivers identical results and convergence history with a speed-up of $\times 2.7$.

Ευχαριστίες

> The task of the excellent teacher is to stimulate "apparently ordinary" people to unusual effort. The tough problem is not in identifying winners: it is in making winners out of ordinary people.
>
> Patricia Cross

Θα ήθελα να ευχαριστήσω θερμά τον καθηγητή μου, κ. Κυριάκο Γιαννάκογλου διότι μου ανέθεσε ένα τόσο ενδιαφέρον θέμα και με καθοδήγησε εξαιρετικά. Η υπομονή του και οι συμβουλές του υπήρξαν πολυτιμότερες από χιλιάδες ώρες επιμελούς μελέτης.

Δεν βρίσκω λόγια να ευχαριστήσω τους Δρ. Ξενοφώντα Τρομπούκη και Δρ. Βαρβάρα Ασούτη, για την απεριόριστη καθοδήγηση, βοήθεια, έννοια τους και τον χρόνο που μου αφιέρωσαν.

Επίσης, ευχαριστώ όλους τους διδάκτορες ή υποψήφιους διδάκτορες του εργαστηρίου για το κλίμα συνεργασίας και την κουλτούρα δουλειάς και συνέχειας που καθημερινά καλλιεργούσαν (και καλλιεργούν): Ευγενία Κοντολέοντος, Γιάννη Καββαδία, Κώστα Τσιάκα, Δημήτρη Παπαδημητρίου, Βαγγέλη Παπουτσή-Κιαχαγιά.

Τέλος, ευχαριστώ την Ελένη για την αγάπη και την υπομονή της και την οικογένειά μου που με στήριξε και μου συμπαραστάθηκε καθ΄ όλη τη διάρκεια των σπουδών μου.

Περιεχόμενα

Περιεχόμενα viii

1 Εισαγωγή 1
1.1 Η τεχνολογική συγκυρία . . . 1
1.2 Σκοπός της διπλωματικής εργασίας . . . 2
1.3 Δομή της εργασίας . . . 3

2 Διακριτοποίηση των τριδιάστατων εξισώσεων Euler 5
2.1 Οι χρονικά μόνιμες εξισώσεις Euler . . . 5
2.1.1 Διαφορική γραφή των χρονικά μόνιμων εξισώσεων Euler 5
2.2 Διακριτοποίηση του χωρίου ροής . . . 6
2.3 Διακριτοποίηση των εξισώσεων ροής . . . 7
2.3.1 Ορισμός όγκων ελέγχου . . . 7
2.3.2 Ολοκλήρωση στους όγκους ελέγχου . . . 8
2.3.3 Υπολογισμός διανύσματος ροής . . . 11
2.3.4 Αύξηση της ακρίβειας του σχήματος και χρήση περιοριστών 14
2.3.5 Διακριτοποίηση του χρονικού όρου και επιλογή του χρονικού βήματος . . . 16
2.4 Οριακές συνθήκες . . . 16
2.4.1 Στερεά Τοιχώματα . . . 17
2.4.2 Όρια εισόδου και εξόδου της ροής . . . 17
2.4.3 Αξονική συμμετρία . . . 23
2.4.4 Περιοδικά όρια . . . 23
2.5 Επίλυση διακριτοποιημένων εξισώσεων . . . 24
2.5.1 Μέθοδος αριθμητικής επίλυσης . . . 24

3 Η αρχιτεκτονική παράλληλης επεξεργασίας CUDA 27
3.1 Το thread ως η βασική μονάδα επεξεργασίας . . . 28
3.2 Οργάνωση των threads σε μία GPU . . . 28
3.3 Είδη μνήμης της GPU . . . 30
3.4 Η αρχιτεκτονική Fermi . . . 31
3.5 Περιγραφή των ειδών μνήμης της GPU . . . 33

3.5.1 Κεντρική μνήμη (global memory) 35
3.5.2 Constant μνήμη 35
3.5.3 Texture μνήμη 36
3.5.4 Shared μνήμη 38
3.5.5 Τοπική μνήμη (local memory) 38
3.6 Συνεργασία CPU-GPU 38
3.7 Ασύγχρονη επεξεργασία δεδομένων σε μία GPU 39
3.8 Atomic functions 39
3.9 Προγραμματισμός με CUDA 41

4 Προγραμματισμός με νήματα (POSIX threads) 47
4.1 Εισαγωγή 47
4.2 Χρήσιμες έννοιες 48
4.2.1 POSIX 48
4.2.2 Νήμα (thread) 48
4.2.3 POSIX threads 51
4.3 Λειτουργίες ελέγχου του ταυτοχρονισμού 53
4.4 Υλοποίηση 53
4.5 Συγχρονισμός 62
4.5.1 Mutexes 62
4.5.2 Condition Variables (μεταβλητές συνθήκης) 66
4.6 Ορατότητα μνήμης μεταξύ threads 67
4.7 Μοντέλα εργασίας με threads 68
4.7.1 Γραμμή παραγωγής 68
4.7.2 Πελάτης/Εξυπηρετητής (Client-Server) 68
4.7.3 Ομάδα εργασίας 69

5 Συστοιχίες καρτών γραφικών 71
5.1 Χειρισμός πολλών GPUs 72
5.2 Προσπέλαση μνήμης του υπολογιστή 72
5.3 Συνδυασμός pthreads με CUDA 73
5.4 Υλοποίηση της παραλληλίας 77
5.4.1 Τρόπος διαμερισμού υπολογιστικού πλέγματος 78
5.5 Ανάθεση παράλληλων εργασιών στις GPUs και συγχρονισμός . 79

6 Αποτελέσματα και συγκριτικές επιδόσεις 83
6.1 Επίλυση ροής μέσα σε αγωγό σχήματος S 84
6.2 Επίλυση ροής μέσα σε αγωγό γωνίας 90° 84
6.3 Συγκριτικές επιδόσεις ανάμεσα σε πολλές GPU και 1 GPU. . . 84

7 Ανακεφαλαίωση και συμπεράσματα 93

Βιβλιογραφία **95**

Κεφάλαιο 1

Εισαγωγή

1.1 Η τεχνολογική συγκυρία

Σήμερα, οι επεξεργαστές καρτών γραφικών (Graphics Processors Units ή GPUs), χρησιμοποιούνται σε υπολογιστικά απαιτητικά προβλήματα από πολλές διαφορετικές επιστημονικές περιοχές. Πίσω από την ευρεία διάδοσή τους βρίσκεται το γεγονός ότι τα τελευταία χρόνια αναπτύχθηκαν γλώσσες προγραμματισμού για GPUs. Τέτοιες είναι η CUDA και η OpenCL που αποτελούν επέκταση των C/C++. Η CUDA (από το Compute Unified Device Architecture) που αποτελεί ταυτόχρονα την ονομασία της αρχιτεκτονικής αλλά και της γλώσσας προγραμματισμού αναπτύσσεται από την NVIDIA. Η AMD, ονομάζει την αντίστοιχη αρχιτεκτονική AMD APP (Accelerated Parallel Processing). Και οι δύο αρχιτεκτονικές, αν και αναπτύχθηκαν ξεχωριστά, επιτρέπουν τον παράλληλο προγραμματισμό των GPU για οποιαδήποτε εφαρμογή. Αυτό είναι γνωστό και ως General-Purpose Computing on Graphics Processing Units ή GPGPU. Προγράμματα γραμμένα σε OpenCL μπορούν να εκτελεστούν σε GPUs και των δύο εταιριών (διότι η OpenCL αποτελεί βιομηχανικό πρότυπο με το οποίο οφείλουν να συμμορφώνονται και οι δύο εταιρίες) ενώ προγράμματα γραμμένα σε CUDA μπορούν να εκτελεστούν μόνο σε GPUs της NVIDIA.

Ο συνδυασμός της μεγάλης υπολογιστικής ισχύος, του εφικτού πλέον προγραμματισμού καθώς και του χαμηλού κόστους κτήσης οδήγησε την επιστημονική κοινότητα στην διερεύνηση των νέων δυνατοτήτων που προσφέρουν οι GPU σε όλους τους τομείς της έρευνας και των εφαρμογών. Τα αποτελέσματα είναι εντυπωσιακά, καθώς παρατηρούνται επιταχύνσεις εφαρμογών από 10 εώς και πάνω από 200 φορές, ανάλογα με το είδος της εκάστοτε εφαρμογής και τον βαθμό παραλληλίας που επιδέχεται.

Στη Μονάδα Παράλληλης Υπολογιστικής Ρευστοδυναμικής και Βελτιστοποίησης (ΜΠΥΡ & Β) του Εργαστηρίου Θερμικών Στροβιλομηχανών (ΕΘΣ), όπου εκπονήθηκε η παρούσα διπλωματική εργασία, τα τελευταία χρόνια έχει αποκτηθεί μία σημαντική εμπειρία σχετικά με τις GPUs και τον τρόπο χρήσης τους. Έχουν αναπτυχθεί επιλύτες συνεκτικών και μη-συνεκτικών, χρονικά μόνιμων και μη-μόνιμων, διδιάστατων και τριδιάστατων ροών [1], [2] και παρουσιάζονται επιλύτες ακόμα και 100 φορές πιο γρήγοροι από τους αντίστοιχους σειριακούς επιλύτες της CPU. Ακόμα, έχουν εφαρμοστεί μέθοδοι βελτιστοποίησης, με χρήση GPU [3], [4], [5], [6], [7].

1.2 Σκοπός της διπλωματικής εργασίας

Η διαθέσιμη μνήμη, των σημερινών GPUs δεν επαρκεί για την επίλυση μεγάλης κλίμακας προβλημάτων αεροδυναμικής. Αναπτύχθηκε λοιπόν κώδικας που να εκμεταλλεύεται συστοιχίες καρτών γραφικών, ώστε να επιλύει 3Δ μη-συνεκτικές ροές (αριθμητική επίλυση εξισώσεων Euler) σε δομημένα πλέγματα των οποίων ο όγκος αποθήκευσης υπερβαίνει τη χωρητικότητα μνήμης μιας μόνο GPU. Έτσι, ο κώδικας λαμβάνει ως είσοδο τα υποχωρία ενός «μεγάλου» διαμερισμένου πλέγματος (αναπτύχθηκε επιπλέον ένας κώδικας διαμερισμού σε υποχωρία), αναθέτει κάθε υποχωρίο στην αντίστοιχη κάρτα γραφικών όπου καλείται ο επιλύτης ροής για το συγκεκριμένο υποχωρίο. Προκύπτει, έτσι, η ανάγκη επικοινωνίας με την έννοια της ανταλλαγής δεδομένων ροής μεταξύ των υποχωρίων κατά την επίλυση στους κόμβους επαφής τους. Όταν επιτευχθεί σύγκλιση, γίνεται αναγωγή των αποτελεσμάτων των υποχωρίων στο ενιαίο πλέγμα. Βάση για την ανάπτυξη του κώδικα αποτέλεσε η διπλωματική εργασία [8]. Σ' εκείνη την εργασία, είχε προγραμματιστεί σε CUDA ο επιλύτης για μία κάρτα γραφικών, ο οποίος υπέστη εκτεταμένες τροποποιήσεις για τις ανάγκες της παρούσας.

1.3 Δομή της εργασίας

Η εργασία δομείται ως εξής:

- Στο κεφάλαιο 2 δίνεται η μαθηματική διατύπωση των εξισώσεων Euler για συμπιεστό ρευστό. Επιπλέον αναλύεται η διακριτοποίησή τους με κεντροκομβικό σχήμα πεπερασμένων όγκων.
- Στο κεφάλαιο 3 παρουσιάζεται η αρχιτεκτονική CUDA και ο τρόπος προγραμματισμού σε GPUs με επεξηγηματικό παράδειγμα.
- Στο κεφάλαιο 4 γίνεται αναφορά στον πολυνηματικό προγραμματισμό με POSIX threads που αποτελεί το υπόβαθρο για τον χειρισμό πολλών καρτών γραφικών του ίδιου υπολογιστικού κόμβου.
- Στο κεφάλαιο 5 παρουσιάζονται πτυχές του προγραμματισμού σε πολλές κάρτες γραφικών, προβλήματα που έπρεπε να αντιμετωπιστούν σχετικά με επικοινωνίες μεταξύ των υποχωρίων.
- Το κεφάλαιο 6 περιέχει τα αποτελέσματα του κώδικα όπως προέκυψαν σε διάφορες περιπτώσεις.
- Το κεφάλαιο 7 αποτελεί μια σύνοψη της εργασίας και παρουσιάζονται τα συμπεράσματα που προέκυψαν από αυτή.

Κεφάλαιο 2

Διακριτοποίηση των τριδιάστατων εξισώσεων Euler

Στο κεφάλαιο αυτό παρουσιάζονται οι εξισώσεις ροής, σε συντηρητική γραφή, για χρονικά μόνιμη, τριδιάστατη ροή ατριβούς συμπιεστού ρευστού, δηλαδή οι εξισώσεις Euler. Επιπλέον, παρουσιάζεται ο τρόπος αριθμητικής επίλυσης των εξισώσεων αυτών μέσω μεθόδου χρονοπροέλασης (time marching) πεπερασμένων όγκων με σκοπό να εφαρμοστεί σε δομημένα, τριδιάστατα πλέγματα [9], [10]. Εκτενής κάλυψη αυτού του θέματος γίνεται στο [8].

2.1 Οι χρονικά μόνιμες εξισώσεις Euler

2.1.1 Διαφορική γραφή των χρονικά μόνιμων εξισώσεων Euler

Οι εξισώσεις που παρουσιάζονται είναι οι εξισώσεις Euler, σε συντηρητική γραφή, για χρονικά μόνιμη ροή συμπιεστού ρευστού, απουσία βαρυτικών δυνάμεων, στο καρτεσιανό σύστημα αναφοράς, για τριδιάστατες ροές.

$$\frac{\partial \overrightarrow{U}}{\partial t} + \frac{\partial \overrightarrow{F_r}}{\partial x_r} = 0 \tag{2.1}$$

Με τα διανύσματα των συντηρητικών μεταβλητών $\vec{U}$ και της μη-συνεκτικής ροής ανά κατεύθυνση $\vec{F_r}$ να είναι:

$$\vec{U} = \begin{bmatrix} \varrho \\ \varrho u_1 \\ \varrho u_2 \\ \varrho u_3 \\ \varrho E \end{bmatrix} = \begin{bmatrix} \varrho \\ \varrho u \\ \varrho v \\ \varrho w \\ \varrho E \end{bmatrix} \quad \vec{F_r} = \begin{bmatrix} \varrho u_r \\ \varrho u_1 u_r + p\delta_{1r} \\ \varrho u_2 u_r + p\delta_{2r} \\ \varrho u_3 u_r + p\delta_{3r} \\ u_r(\varrho E + p) \end{bmatrix} \tag{2.2}$$

όπου ϱ είναι η πυκνότητα του ρευστού, u_r $(r = 1, 2, 3)$ η συνιστώσα της ταχύτητας κατά την κατεύθυνση x_r, E η ολική ενέργεια ανά μονάδα μάζας, e η εσωτερική ενέργεια ανά μονάδα μάζας, p η πίεση του ρευστού και δ ο τελεστής Kronecker. Ο χρονικός όρος που υπάρχει στις εξισώσεις αποτελεί έναν ψευδοχρονικό όρο και έχει προστεθεί για την εκμετάλλευση των ιδιοτήτων των υπερβολικών συστημάτων και για την εφαρμογή μίας μεθόδου χρονοπροέλασης. Όπου υπάρχουν διπλά επαναλαμβανόμενοι δείκτες νοείται άθροιση σύμφωνα με τη σύμβαση Einstein, εκτός αν δηλώνεται ρητά το αντίθετο. Η ολική ενέργεια ανά μονάδα μάζας εκφράζεται από τη σχέση:

$$E = e + \frac{1}{2} u_r u_r \tag{2.3}$$

και σχετίζεται με την πίεση ως εξής:

$$E = \frac{p}{\varrho(\gamma - 1)} + \frac{1}{2} u_r u_r \tag{2.4}$$

Επίσης, η ολική ενθαλπία δίνεται από τη σχέση:

$$h_t = \frac{E + p}{\varrho} = \frac{\gamma p}{\varrho(\gamma - 1)} + \frac{1}{2} u_r u_r \tag{2.5}$$

2.2 Διακριτοποίηση του χωρίου ροής

Όπως είναι γνωστό, πριν την αριθμητική επίλυση οποιουδήποτε προβλήματος ροής, απαιτείται η διακριτοποίηση του χωρίου ροής, δηλαδή η κατασκευή του πλέγματος στους κόμβους του οποίου θα επιλυθούν οι εξισώσεις ροής. Τα πλέγματα χωρίζονται σε δύο βασικές κατηγορίες, τα δομημένα και τα μη-δομημένα. Ο επιλύτης ροής της παρούσας εργασίας χειρίζεται δομημένα πλέγματα. Έτσι, όλες οι παρακάτω αναφορές απευθύνονται σε δομημένα πλέγματα. Τα δομημένα πλέγματα αποτελούνται από αυστηρά ταξινομημένους κόμβους και αποκλειστικά απο τετραπλευρικά στοιχεία, για διδιάστατα πλέγματα, ή εξαεδρικά

στοιχεία, για τριδιάστατα πλέγματα. Τα πλεονεκτήματα των δομημένων πλεγμάτων απορρέουν από την αυστηρή ταξινόμηση των κόμβων τους. Βασικό πλεονέκτημα της δομής είναι η εκ των προτέρων γνωστή σχετική θέση κάθε κόμβου, οπότε είναι γνωστοί και οι γείτονες του εκάστοτε κόμβου, δίχως την ανάγκη για αποθήκευση επιπλέον πληροφοριών, όπως συμβαίνει στα μη-δομημένα πλέγματα. Αυτή η βασική διαφορά επιτρέπει στους επιλύτες δομημένων πλεγμάτων να μπορούν να λειτουργήσουν με μικρότερες απαιτήσεις μνήμης, συγκριτικά με αντίστοιχους επιλύτες μη-δομημένων πλεγμάτων. Επίσης, στα δομημένα πλέγματα υπάρχει η δυνατότητα ανάπτυξης επιλυτών, με σχήματα πεπερασμένων διαφορών, μεγαλύτερης ακρίβειας και τα μητρώα των συντελεστών που προκύπτουν είναι διαγώνιας μορφής. Ένα ακόμα σημαντικό πλεονέκτημα είναι ότι λόγω της δομής των κόμβων του πλέγματος, αντίστοιχα είναι ταξινομημένες όλες οι πληροφορίες μέσα στην μνήμη του υπολογιστή. Καλύτερη προσπέλαση μνήμης κατά την εκτέλεση του προγράμματος μεταφράζεται σε καλύτερους χρόνους επίλυσης, ιδιαίτερα στις GPU όπου ο τρόπος προσπέλασης της μνήμης είναι καθοριστικής σημασίας.

2.3 Διακριτοποίηση των εξισώσεων ροής

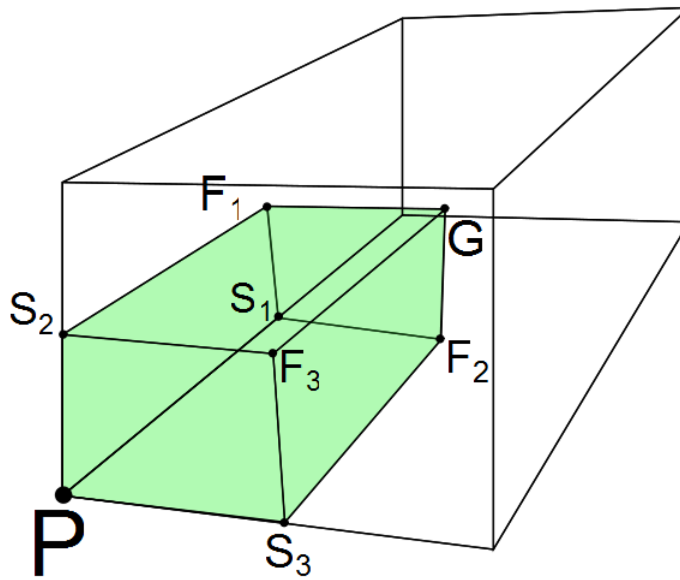
2.3.1 Ορισμός όγκων ελέγχου

Για την διακριτοποίηση των εξισώσεων 2.1 χρησιμοποιείται ένα σχήμα πεπερασμένων όγκων, με κεντροκομβική προσέγγιση. Για την εφαρμογή αυτού χρειάζεται ο ορισμός των όγκων ελέγχου γύρω από τους κόμβους του πλέγματος, στους οποίους και αποθηκεύονται όλες οι μεταβλητές της ροής. Μία σωστή επίλυση του προβλήματος ροής επιβάλλει οι όγκοι ελέγχου να καλύπτουν ολόκληρο το χωρίο ροής, αλλά και να μην αλληλοκαλύπτονται.

Η μέθοδος επίλυσης αναφέρεται σε δομημένα, τριδιάστατα πλέγματα, οπότε όλα τα πλεγματικά στοιχεία που περιβάλλουν έναν τυχαίο κόμβο P είναι εξαεδρικά. Ακόμα, είναι γνωστό εκ των προτέρων πως ένας εσωτερικός κόμβος P περιβάλλεται από 8 εξάεδρα και 6 γειτονικούς κόμβους πρώτου βαθμού (δηλαδή συνδέονται με ακμή). Για έναν οριακό κόμβο P τα περιβάλλοντα εξάεδρα μπορεί να είναι 4, 2 ή 1 και οι γειτονικοί κόμβοι 5, 4 ή 3 ανάλογα με το αν ο κόμβος είναι οριακός σε 1, 2 ή και 3 από τους άξονες i, j, k του πλέγματος, αντίστοιχα. Κάθε γειτονικό εξάεδρο προσφέρει στον όγκο ελέγχου ενός τυχαίου κόμβου P έναν όγκο, όπως απεικονίζεται στο σχήμα 2.1, που ορίζεται από το σημείο P, τους μεσόκομβους κάθε ακμής που συντρέχει στον κόμβο P, που συμβολίζονται με S_1, S_2 και S_3 στο σχήμα, τα κέντρα βάρους των πλευρών στις οποίες ανήκουν οι προαναφερθείσες ακμές, τα σημεία F_1, F_2 και F_3, καθώς και

το κέντρο βάρους του εκάστοτε εξαέδρου, το σημείο G. Για λόγους ευκρίνειας παρουσιάζεται μόνο ένα από τα 8 γειτονικά εξάεδρα, αντί για ολόκληρο τον όγκο ελέγχου.

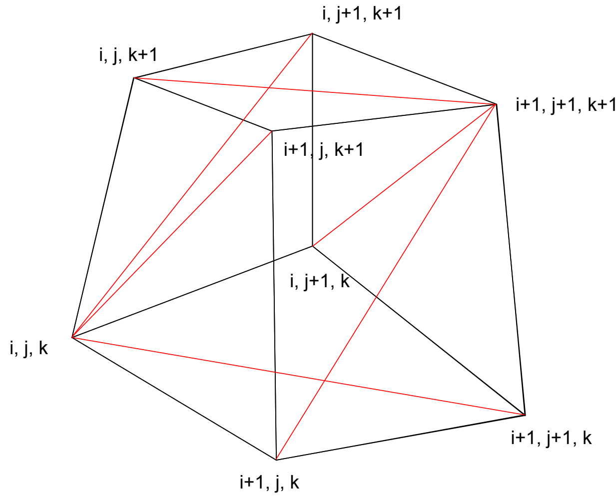
Είναι σημαντικό να αναφερθεί πως τα 4 σημεία που ορίζουν κάθε επιφάνεια δεν είναι απαραίτητα συνεπίπεδα, οπότε και πρέπει να βρεθεί μονοσήμαντος τρόπος διαχείρισης των επιφανειών αυτών ώστε να τηρείται η συνθήκη της μη αλληλοκάλυψης των όγκων ελέγχου. Η μέθοδος που χρησιμοποιήθηκε στην παρούσα εργασία είναι η διάσπαση κάθε επιφάνειας σε 2 τρίγωνα όπως φαίνεται στο σχήμα 2.2. Κατά την διάσπαση κάθε επιφάνειας χρειάζεται προσοχή, καθώς η διάσπαση πρέπει να ορίζεται μονοσήμαντα και πρέπει να γίνει με τέτοιο τρόπο ώστε 2 γειτονικά εξάεδρα να μην αλληλοκαλύπτονται, αλλά και να μην δημιουργείται κενό ανάμεσά τους.



Σχήμα 2.1: Προσφορά κάθε γειτονικού εξάεδρου στον όγκο ελέγχου.

2.3.2 Ολοκλήρωση στους όγκους ελέγχου

Στη συνέχεια, γίνεται ολοκλήρωση των εξισώσεων 2.1 στους όγκους ελέγχου. Η ολοκλήρωση του όγκου ελέγχου ενός τυχαίου κόμβου P δίνει:



Σχήμα 2.2: Διάσπαση κάθε επιφάνειας ενός εξάεδρου στοιχείου σε 2 τρίγωνα.

$$\iiint\limits_{V^P} \frac{\partial \overrightarrow{U}}{\partial t} dV + \iiint\limits_{V^P} \frac{\partial \overrightarrow{F_r}}{\partial x_r} dV = 0 \tag{2.6}$$

Εφαρμόζοντας το θεώρημα Green-Gauss και θεωρώντας ότι $\iiint\limits_{V^P} \frac{\partial \overrightarrow{U}}{\partial t} dV = \left(\frac{\partial \overrightarrow{U}}{\partial t}\right)_P V^P$, ισχύει:

$$\left(\frac{\partial \overrightarrow{U}}{\partial t}\right)_P V^P + \iint\limits_{\partial V^P} \overrightarrow{F}_r \hat{n}_r d(\partial V) = 0 \tag{2.7}$$

όπου V^P είναι ο όγκος του όγκου ελέγχου, ∂V^P η οριακή επιφάνεια αυτού και $\overrightarrow{n} = (\hat{n}_x, \hat{n}_y, \hat{n}_z)$ το κάθετο μοναδιαίο διάνυσμα στην οριακή επιφάνεια με φορά προς το εξωτερικό του. Θέτοντας :

$$\begin{aligned} \overrightarrow{\hat{H}} &= \overrightarrow{F}_r \hat{n}_r \\ \overrightarrow{H} &= \overrightarrow{F}_r n_r \end{aligned} \tag{2.8}$$

η έκφραση 2.7 γίνεται:

$$\left(\frac{\partial \overrightarrow{U}}{\partial t}\right)_P V^P + \iint\limits_{\partial V^P} \overrightarrow{\hat{H}} d(\partial V) = 0$$

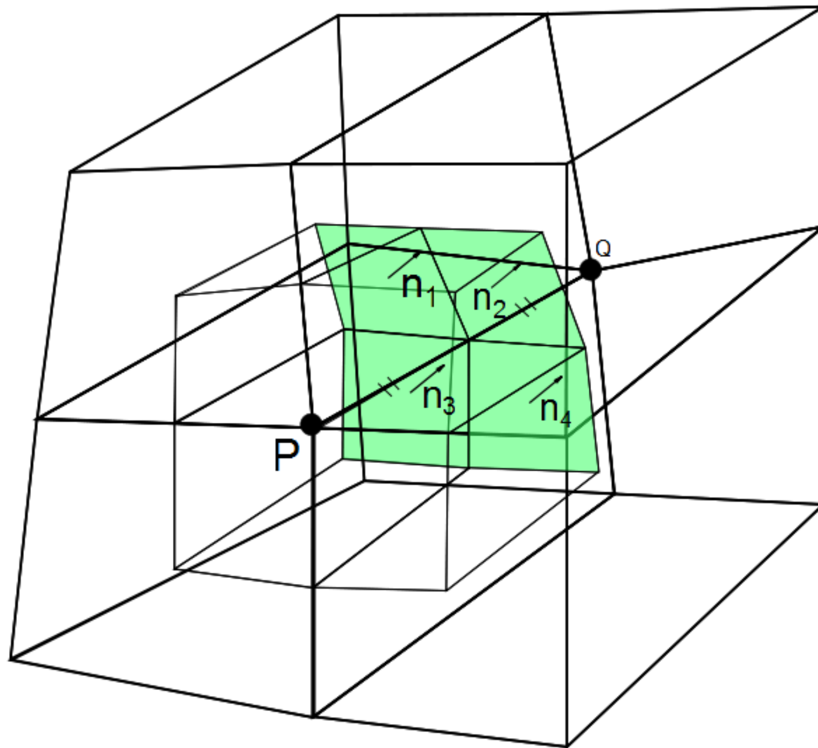
ή

$$\left(\frac{\partial \overrightarrow{U}}{\partial t}\right)_P V^P + \sum_{Q\in K(P)} \overrightarrow{\Phi}_{PQ} = 0 \tag{2.9}$$

όπου Q οι γειτονικοί κόμβοι του P και το διάνυσμα ροής $\overrightarrow{\Phi}_{PQ}$ ορίζεται:

$$\overrightarrow{\Phi}_{PQ} = \iint\limits_{\partial V^P} \overrightarrow{\tilde{H}} d(\partial V) \tag{2.10}$$

Η επιφάνεια ολοκλήρωσης για τον υπολογισμό του διανύσματος ροής $\overrightarrow{\Phi}_{PQ}$ αντιστοιχεί στο κοινό όριο των όγκων ελέγχου, του κόμβου P και του εκάστοτε γειτονικού κόμβου Q, όπως φαίνεται στο σχήμα 2.3.



Σχήμα 2.3: Κοινό όριο των όγκων ελέγχου ανάμεσα στους κόμβους P και Q.

2.3.3 Υπολογισμός διανύσματος ροής

Το διάνυσμα ροής $\overrightarrow{\Phi}_{PQ}$ υπολογίζεται σε κάθε ακμή και αφαιρείται ή προστίθεται ανάλογα, στον ισολογισμό της κυψέλης στην οποία αναφέρεται. Ο υπολογισμός του γίνεται σύμφωνα με τις τιμές των συντηρητικών μεταβλητών εκατέρωθεν του μέσου της ακμής PQ, οι οποίες υπολογίζονται συναρτήσει των αντίστοιχων τιμών στους κόμβους P και Q, με προεκβολή ακρίβειας δεύτερης τάξης. Επίσης συνυπολογίζεται το κάθετο διάνυσμα $\overrightarrow{n}_{PQ}$ που είναι το διανυσματικό άθροισμα των $\overrightarrow{n}_1$, $\overrightarrow{n}_2$, $\overrightarrow{n}_3$ και $\overrightarrow{n}_4$, όπως αυτά φαίνονται στο σχήμα 2.3. Όμως, όπως έχει αναφερθεί και παραπάνω, τα 4 σημεία που ορίζουν κάθε επιφάνεια σε κάθε στοιχειώδες εξάεδρο δεν είναι κατ΄ ανάγκη συνεπίπεδα. Έτσι, για τον υπολογισμό καθενός από τα διανύσματα αυτά, χρειάζεται η διαίρεση της επιφάνειας σε 2 τριγωνικές, επίσης όπως έχει αναφερθεί παραπάνω, ο υπολογισμός των κάθετων προς αυτές διανυσμάτων και η άθροισή τους, ανά δύο, για να προκύψει το αποτέλεσμα του σχήματος 2.3. Έτσι προκύπτει:

$$\overrightarrow{n}_{PQ} = \overrightarrow{n}_1 + \overrightarrow{n}_2 + \overrightarrow{n}_3 + \overrightarrow{n}_4 \tag{2.11}$$

$$\overrightarrow{\Phi}_{PQ} = \overrightarrow{f}\left(\overrightarrow{U}^L_{PQ}, \overrightarrow{U}^R_{PQ}, \overrightarrow{n}_{PQ}\right) \tag{2.12}$$

Πριν τη διατύπωση της έκφρασης του διανύσματος ροής, χρειάζεται να οριστεί το ιακωβιανό μητρώο του διανύσματος $\overrightarrow{F}_r$ ως προς τις συντηρητικές μεταβλητές $\overrightarrow{U}$.

$$A_r \hat{=} \frac{\partial \overrightarrow{F}_r}{\partial \overrightarrow{U}} \tag{2.13}$$

Ακόμα οριζεται:

$$\underline{A} \hat{=} A_r n_r \Rightarrow \underline{A} = \frac{\partial \overrightarrow{F}_r}{\partial \overrightarrow{U}} n_r = \frac{\partial (\overrightarrow{F}_r n_r)}{\partial \overrightarrow{U}} \Rightarrow \underline{A} = \frac{\partial \overrightarrow{H}}{\partial \overrightarrow{U}} \tag{2.14}$$

Από την καταστατική εξίσωση των τελείων αερίων εύκολα μπορεί να αποδειχθεί ότι ικανοποιείται η έκφραση $p = \varrho f(e)$. Σε συνδυασμό με τον τρόπο ορισμού του ιακωβιανού μητρώου A_r, το διάνυσμα της ροής $\overrightarrow{F}_r$ είναι ομογενής συνάρτηση πρώτου βαθμού, κύρια ιδιότητα των οποίων είναι:

$$\overrightarrow{F}_r = A_r \overrightarrow{U} \tag{2.15}$$

Ακολουθώντας την πορεία της εξαγωγής της σχέσης 2.14 εύκολα προκύπτει:

$$\overrightarrow{H} = \underline{A}\overrightarrow{U} \tag{2.16}$$

Ιακωβιανό μητρώο $\underline{A}$

$$\underline{A}(:,1) = \begin{bmatrix} 0 \\ -u\left(u_r n_r\right) + \frac{\gamma-1}{2}\left(u_r u_r\right) n_x \\ -v\left(u_r n_r\right) + \frac{\gamma-1}{2}\left(u_r u_r\right) n_y \\ -w\left(u_r n_r\right) + \frac{\gamma-1}{2}\left(u_r u_r\right) n_z \\ \left[-\gamma E + (\gamma-1)\left(u_r u_r\right)\right]\left(u_r n_r\right) \end{bmatrix}$$

$$\underline{A}(:,2) = \begin{bmatrix} n_x \\ u_r n_r + (2-\gamma) u n_x \\ v n_x - (\gamma-1) u n_y \\ w n_x - (\gamma-1) u n_z \\ \left[\gamma E - \frac{\gamma-1}{2}\left(u_r u_r\right)\right] n_x - (\gamma-1) u \left(u_r n_r\right) \end{bmatrix}$$

$$\underline{A}(:,3) = \begin{bmatrix} n_y \\ u n_y - (\gamma-1) v n_x \\ u_r n_r + (2-\gamma) v n_y \\ w n_y - (\gamma-1) v n_z \\ \left[\gamma E - \frac{\gamma-1}{2}\left(u_r u_r\right)\right] n_y - (\gamma-1) v \left(u_r n_r\right) \end{bmatrix} \tag{2.17}$$

$$\underline{A}(:,4) = \begin{bmatrix} n_z \\ u n_z - (\gamma-1) w n_x \\ v n_z - (\gamma-1) w n_y \\ u_r n_r + (2-\gamma) w n_z \\ \left[\gamma E - \frac{\gamma-1}{2}\left(u_r u_r\right)\right] n_z - (\gamma-1) w \left(u_r n_r\right) \end{bmatrix}$$

$$\underline{A}(:,5) = \begin{bmatrix} 0 \\ (\gamma-1) n_x \\ (\gamma-1) n_y \\ (\gamma-1) n_z \\ \gamma\left(u_r n_r\right) \end{bmatrix}$$

οι ιδιοτιμές του μητρώου $\underline{A}$ έχουν υπολογιστεί και είναι:

$$\begin{aligned} \lambda_1 &= \overrightarrow{u} \cdot \overrightarrow{n} \\ \lambda_2 &= \overrightarrow{u} \cdot \overrightarrow{n} \\ \lambda_3 &= \overrightarrow{u} \cdot \overrightarrow{n} \end{aligned} \tag{2.18}$$

$$\lambda_4 = \left(\overrightarrow{u} \cdot \overrightarrow{\hat{n}} + c\right)|\overrightarrow{n}|$$

$$\lambda_5 = \left(\overrightarrow{u} \cdot \overrightarrow{\hat{n}} - c\right)|\overrightarrow{n}|$$

Ακόμα, υπολογίζονται τα αριστερά και δεξιά ιδιοδιανύσματα του μητρώου $\underline{A}$. Τα δεξιά ιδιοδιανύσματα ικανοποιούν την σχέση $(\underline{A} - \lambda_k I_k)r_k = 0$, ενώ τα αριστερά ιδιοδιανύσματα ικανοποιούν την σχέση $l_k(\underline{A} - \lambda_k I) = 0$ $(k = 1, 2, 3, 4, 5)$. Με I συμβολίζεται ο μοναδιαίος πίνακας. Πλέον, το μητρώο $\underline{A}$ μπορεί εύκολα να γραφεί στην μορφή

$$\underline{A} = P \Lambda P^{-1} \tag{2.19}$$

όπου Λ διαγώνιος πίνακας με τις ιδιοτιμές του $\underline{A}$ και

$$P = \begin{bmatrix} r_1 & r_2 & r_3 & r_4 & r_5 \end{bmatrix}, P^{-1} = \begin{bmatrix} l_1 \\ l_2 \\ l_3 \\ l_4 \\ l_5 \end{bmatrix} \tag{2.20}$$

Ακόμα, ορίζονται τα μητρώα:

$$\underline{A}^+ = P \Lambda^+ P^{-1}, \underline{A}^- = P \Lambda^- P^{-1} \tag{2.21}$$

$$|\underline{A}| = \underline{A}^+ - \underline{A}^- \tag{2.22}$$

όπου το μητρώο Λ^+ περιέχει τις θετικές ιδιοτιμές, ενώ το μητρώο Λ^- τις αρνητικές ιδιοτιμές.

Έχοντας ορίσει τα ιακωβιανά μητρώα, ακολουθεί ο ορισμός του διανύσματος της ροής Φ_{PQ}, σύμφωνα με το σχήμα του Roe [11]. Έτσι, ισχύει:

$$\begin{aligned}
&\overrightarrow{\Phi}_{PQ} = \frac{1}{2}\left[\overrightarrow{H}\left(\overrightarrow{U}^R_{PQ}, \overrightarrow{n}_{PQ}\right) + \overrightarrow{H}\left(\overrightarrow{U}^L_{PQ}, \overrightarrow{n}_{PQ}\right)\right] - \frac{1}{2}\left|\tilde{\underline{A}}_{PQ}\right|\left(\overrightarrow{U}^R_{PQ} - \overrightarrow{U}^L_{PQ}\right) \Rightarrow \\
&\overrightarrow{\Phi}_{PQ} = \frac{1}{2}\left[\underline{A}_R \overrightarrow{U}^R_{PQ} + \underline{A}_L \overrightarrow{U}^L_{PQ}\right] - \frac{1}{2}\left|\tilde{\underline{A}}_{PQ}\right|\left(\overrightarrow{U}^R_{PQ} - \overrightarrow{U}^L_{PQ}\right) \Rightarrow \\
&\overrightarrow{\Phi}_{PQ} = \frac{1}{2}\left(\underline{A}_R - \left|\tilde{\underline{A}}_{PQ}\right|\right)\overrightarrow{U}^R_{PQ} + \frac{1}{2}\left(\underline{A}_L - \left|\tilde{\underline{A}}_{PQ}\right|\right)\overrightarrow{U}^L_{PQ} \Rightarrow \\
&\overrightarrow{\Phi}_{PQ} = \mathcal{A}_R \overrightarrow{U}^R_{PQ} + \mathcal{A}_L \overrightarrow{U}^L_{PQ}
\end{aligned} \tag{2.23}$$

όπου $|\underline{\tilde{A}}_{PQ}|$ είναι το μητρώο που προκύπτει από τις απόλυτες τιμές των ιδιοτιμών του $\underline{A}$, σχέση 2.22, υπολογισμένο με βάση τις κατά Roe μέσες τιμές των πρωτευουσών μεταβλητών. Αυτές δίνονται:

$$\overrightarrow{\tilde{U}}_{PQ} = \begin{bmatrix} \tilde{\varrho} & \tilde{u} & \tilde{v} & \tilde{w} & \tilde{p} \end{bmatrix}^T \tag{2.24}$$

Για τον υπολογισμό των συνιστωσών, πέραν της πίεσης, χρησιμοποιείται η σχέση 2.25, ενώ για τον υπολογισμό της μέσης, κατά Roe πίεσης, υπολογίζεται η μέση τιμή της ολικής ενθαλπίας σύμφωνα με την σχέση 2.26 και στη συνέχεια υπολογίζεται η πίεση.

$$\overrightarrow{\tilde{U}}_{PQ} = \frac{\sqrt{\varrho_L}\overrightarrow{U}_L + \sqrt{\varrho_R}\overrightarrow{U}_R}{\sqrt{\varrho_L} + \sqrt{\varrho_R}} \tag{2.25}$$

$$h_t = \frac{\gamma p}{(\gamma - 1)\varrho} + \frac{1}{2}(u_r u_r) \tag{2.26}$$

2.3.4 Αύξηση της ακρίβειας του σχήματος και χρήση περιοριστών

Στην μέχρι τώρα ανάλυση, έχει γίνει αναφορά στις τιμές εκατέρωθεν του μέσου της ακμής PQ, αλλά δεν έχει αναφερθεί ο τρόπος υπολογισμού των τιμών αυτών. Ο τρόπος υπολογισμού των τιμών αυτών σχετίζεται με την ακρίβεια της χωρικής διακριτοποίησης των εξισώσεων ροής. Η πιο απλή λύση είναι η θεώρηση

$$\begin{aligned} \overrightarrow{U}^L_{PQ} &= \overrightarrow{U}_P \\ \overrightarrow{U}^R_{PQ} &= \overrightarrow{U}_Q \end{aligned} \tag{2.27}$$

η οποία και αντιστοιχεί σε χωρική διακριτοποίηση πρώτης τάξης. Εναλλακτικά, με χρήση του θεωρήματος του Taylor μπορεί να αυξηθεί η τάξη της χωρικής διακριτοποίησης. Για χωρική διακριτοποίηση δεύτερης τάξης ακρίβειας, ο τύπος υπολογισμού των τιμών εκατέρωθεν του μέσου της ακμής PQ είναι:

$$\begin{aligned} \overrightarrow{U}^L_{PQ} &= \overrightarrow{U}_P + \frac{1}{2}\overrightarrow{PQ} \cdot \left(\nabla \overrightarrow{U}\right)_P \\ \overrightarrow{U}^R_{PQ} &= \overrightarrow{U}_Q - \frac{1}{2}\overrightarrow{PQ} \cdot \left(\nabla \overrightarrow{U}\right)_Q \end{aligned} \tag{2.28}$$

Βασικό μειονέκτημα της αύξησης της ακρίβειας της χωρικής διακριτοποίησης με αυτόν τον τρόπο, είναι η ανάγκη για υπολογισμό της πρώτης παραγώγου των μεταβλητών της ροής.

Περιοριστής Van Leer-Van Albada [12]

Ο περιοριστής αυτός, αποτελεί τροποποίηση του αρχικού περιοριστη του Van Leer, ώστε σε περιοχές μικρής κλίσης να παίρνει τιμές πλησιέστερες στη μονάδα, δηλαδή να επεμβαίνει λιγότερο στην υπολογισμένη κατά Taylor προεκβολή σε περιοχές που δεν υπάρχει λόγος περιορισμού της κλίσης. Είναι ένας από τους πρώτους περιοριστές που αναπτύχθηκαν, είναι απλός στην εφαρμογή και δεν επηρεάζει ιδιαίτερα την σύγκλιση των εξισώσεων ροής. Τα βασικότερα μειονεκτήματά του είναι ο μονοδιάστατος χαρακτήρας του, δηλαδή η εξάλειψη των ταλαντώσεων γίνεται μόνο κατά τη διεύθυνση της προεκβολής, δηλαδή αυτήν της ακμής στην οποία υπολογίζεται το διάνυσμα ροής, και η έλλειψη μαθηματικού μηχανισμού απενεργοποίησής του σε περιοχές που δεν είναι απαραίτητος, όπως σε περιοχές ελεύθερης ροής, ενώ μπορεί να μειώσει την τάξη ακρίβειας της λύσης σε περιοχές του πεδίου όπου υπάρχουν ακρότατα στη λύση. Με χρήση του περιοριστη Van Leer-Van Albada, ο τύπος υπολογισμού των τιμών εκατέρωθεν του μέσου της ακμής PQ είναι:

$$\begin{aligned} \overrightarrow{U}^L_{PQ} &= \overrightarrow{U}_P + \frac{1}{2} LIM \left[\left(2 \left(\frac{\partial \overrightarrow{U}}{\partial x_r} \right)^P x_r^{PQ} - \Delta \overrightarrow{U}^{PQ} \right), \Delta \overrightarrow{U}^{PQ} \right] \\ \overrightarrow{U}^R_{PQ} &= \overrightarrow{U}_Q - \frac{1}{2} LIM \left[\left(2 \left(\frac{\partial \overrightarrow{U}}{\partial x_r} \right)^Q x_r^{PQ} - \Delta \overrightarrow{U}^{PQ} \right), \Delta \overrightarrow{U}^{PQ} \right] \end{aligned} \tag{2.29}$$

όπου

$$\Delta \overrightarrow{U}^{PQ} = \overrightarrow{U}^Q - \overrightarrow{U}^P$$

και

$$LIM(a,b) = \begin{cases} \frac{(a^2+\eta)b+(b^2+\eta)a}{a^2+b^2+2\eta}, & ab > 0 \\ 0, & ab <= 0 \end{cases}$$

2.3.5 Διακριτοποίηση του χρονικού όρου και επιλογή του χρονικού βήματος

Στη παρούσα εργασία αναλύεται η μέθοδος επίλυσης χρονικά μόνιμων ροών, όπως παρατηρείται η ύπαρξη του χρονικού όρου $\left(\frac{\partial \overrightarrow{U}}{\partial t}\right)_P V^P$. Ο χρονικός όρος αποτελεί ψευδοχρονικό όρο και έχει προστεθεί για διευκόλυνση της σύγκλισης των εξισώσεων, σύμφωνα με την τεχνική της χρονοπροέλασης. Η διακριτοποίησή του γίνεται μέσω πρώτης τάξης σχήματος ανάντι διαφόρισης του Euler

$$\left(\frac{\partial \overrightarrow{U}}{\partial t}\right)_P V^P = \frac{V^P}{\Delta t^P} \Delta \overrightarrow{U}^P \tag{2.30}$$

όπου $\Delta \overrightarrow{U}^P = \left(\overrightarrow{U}^P\right)^{n+1} - \left(\overrightarrow{U}^P\right)^n$ (ο εκθέτης n αντιστοιχεί στο τρέχον χρονικό βήμα). Για επιτάχυνση της σύγκλισης εφαρμόζεται η τεχνική του τοπικού χρονικού βήματος [13]. Ο τύπος από τον οποίο προκύπτει το ψευδοχρονικό βήμα σε κάθε κόμβο είναι:

$$\Delta t^P = CFL \frac{V^P}{C} \tag{2.31}$$

όπου CFL είναι ο αριθμός Courant-Friedrichs-Lewy [14] και ο όρος C υπολογίζεται από την σχέση

$$C = \left(|u_r^P| + c^P\right) S_r^P \tag{2.32}$$

όπου S_i^P είναι η προβολή των τμημάτων που απαρτίζουν τα όρια του όγκου ελέγχου του κόμβου P κατά την κατεύθυνση i, δηλαδή

$$S_r^P = \frac{1}{2} \sum_{Q \in K(P)} |n_r^{PQ}| \tag{2.33}$$

2.4 Οριακές συνθήκες

Ολόκληρη η ανάλυση που προηγήθηκε θεωρούσε έναν τυχαίο κόμβο P, ο οποίος ήταν εσωτερικός του πλέγματος. Όμως, σε περίπτωση οριακού κόμβου πρέπει να συμπεριληφθεί ένας ακόμα όρος, το διάνυσμα της ροής που εξέρχεται προς το περιβάλλον από τον όγκο ελέγχου. Έτσι, η ολοκλήρωση των εξισώσεων 2.1 καταλήγει στη μορφή:

$$\left(\frac{\partial \overrightarrow{U}}{\partial t}\right)_P V^P + \sum_{Q \in K(P)} \overrightarrow{\Phi}_{PQ} + \overrightarrow{\Phi}_{\text{Όριο χωρίου ροής}} = 0 \tag{2.34}$$

Το οριακό διάνυσμα ροής υπολογίζεται ανάλογα με το είδος του ορίου και στη συνέχεια θα παρουσιαστούν οι τρόποι υπολογισμού για οριακούς κόμβους τοιχωμάτων, ειδόδου-εξόδου, επιφάνειας συμμετρίας ή και περιοδικότητας.

2.4.1 Στερεά Τοιχώματα

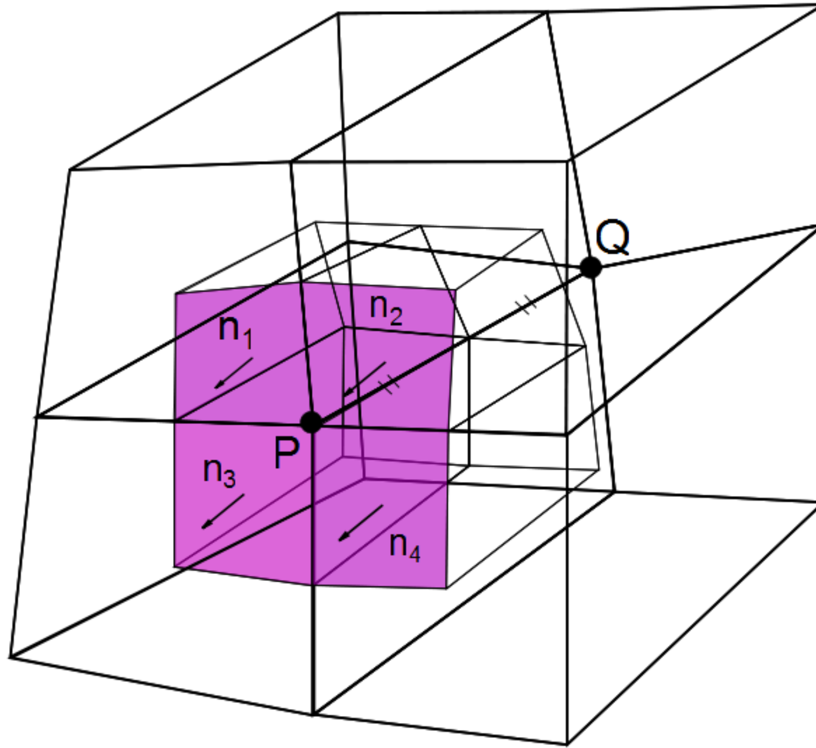
Στα στερεά τοιχώματα, για ατριβείς ροές, επιβάλλεται η συνθήκη μη-εισχώρησης ($\overrightarrow{u} \cdot \overrightarrow{n} = 0$). Η επιβολή γίνεται με ασθενή διατύπωση, δηλαδή με εισαγωγή της συνθήκης στο διάνυσμα της ροής, το οποίο, κατά μήκος των στερεών τοιχωμάτων, λαμβάνει τη μορφή:

$$\overrightarrow{\Phi}_{\text{όριο}} = \overrightarrow{F}_r^P n_r^P = \begin{bmatrix} \varrho(u_r n_r) \\ \varrho u(u_r n_r) + p n_x \\ \varrho v(u_r n_r) + p n_y \\ \varrho w(u_r n_r) + p n_z \\ (\varrho E + p)(u_r n_r) \end{bmatrix}_P = \begin{bmatrix} 0 \\ p n_x \\ p n_y \\ p n_z \\ 0 \end{bmatrix}_P \tag{2.35}$$

με $\overrightarrow{n} = \overrightarrow{n_1} + \overrightarrow{n_2} + \overrightarrow{n_3} + \overrightarrow{n_4}$, όπου τα διανύσματα $\overrightarrow{n_1}$, $\overrightarrow{n_2}$, $\overrightarrow{n_3}$ και $\overrightarrow{n_4}$ αντιστοιχούν στα διανύσματα από το όριο του όγκου ελέγχου προς το τοίχωμα, όπως φαίνεται στο σχήμα 2.4.

2.4.2 Όρια εισόδου και εξόδου της ροής

Έχει αποδειχθεί ότι το πρόσημο των ιδιοτιμών του μητρώου $\underline{A}$ καθορίζει την κατεύθυνση μεταφοράς της «πληροφορίας» μέσα στη ροή. Από τις ιδιοτιμές του μητρώου $\underline{A}$, όπως έχουν υπολογιστεί στην σχέση 2.18, είναι θετικές οι λ_1, λ_2, λ_3 και λ_4 ενώ η λ_5 εξαρτάται από το αν η ροή είναι υποηχητική ή υπερηχητική. Για τις θετικές ιδιοτιμές, η αντίστοιχη «πληροφορία» μεταφέρεται μαζί με την ροή, ενώ όταν η ιδιοτιμή είναι αρνητική, η «πληροφορία» ταξιδεύει αντίθετα από την τοπική ταχύτητα της ροής. Έτσι, για το κλείσιμο των εξισώσεων ροής, απαιτούνται 4 μεγέθη στην είσοδο της ροής και 1 στην έξοδο σε υποηχητικές ροές, ενώ σε υπερηχητικές ροές απαιτούνται 5 μεγέθη στην είδοδο της ροής. Σε εφαρμογές εξωτερικής αεροδυναμικής, τα μεγέθη αυτά είναι η πυκνότητα (ϱ_∞) και το μέτρο ($|\overrightarrow{u}_\infty|$) και οι γωνίες θ_1 και θ_2 της επ΄ άπειρον ταχύτητας, αλλά και ο αριθμός Mach της επ΄ άπειρον ροής. Αντιθετα, σε εφαρμογές εσωτερικής



Σχήμα 2.4: Όγκος ελέγχου οριακού κόμβου.

αεροδυναμικής δίνονται οι τιμές της ολικής πίεσης (p_t) και θερμοκρασίας (T_t) στην είσοδο της ροής, τις γωνίες θ_1 και θ_2 της ταχύτητας στην είσοδο της ροής και την τιμή της στατικής πίεσης στην έξοδο της ροής, για υποηχητική ροή, ή στην είσοδο, για υπερηχητική ροή. Συχνά, στην πράξη, αντί για την στατική πίεση, δίνεται ο αριθμός Mach, ισεντροπικός εξόδου σε υποηχητικές ροές και εισόδου σε υπερηχητικές, από τον οποίο και υπολογίζεται η τιμή της στατικής πίεσης.

Το διάνυσμα ροής στους οριακούς κόμβους εισόδου ή εξόδου υπολογίζεται σύμφωνα με το ανάντι σχήμα πρώτης τάξης των Steger-Warming [15], το οποίο για έναν οριακό κόμβο P γράφεται:

$$\overrightarrow{\Phi}^{P}_{out} = \underline{A}^{+}_{P}\overrightarrow{U}_{P} + \underline{A}^{-}_{P}\overrightarrow{U}_{out} \tag{2.36}$$

όπου με «out» συμβολίζεται ένας υποθετικός κόμβος, εξωτερικά του πεδίου ροής, στον οποίο και επιβάλλονται οι οριακές συνθήκες.

Εξωτερική Αεροδυναμική

Όπως έχει αναφερθεί νωρίτερα, σε εφαρμογές εξωτερικής αεροδυναμικής, δηλαδή σε εφαρμογές που η ροή γύρω από ένα αεροδυναμικό σώμα επηρεάζεται μόνο από την παρουσία του ιδίου, τα μεγέθη που συνήθως δίνονται στο επ΄ άπειρο όριο για το κλείσιμο των εξισώσεων ροής είναι η πυκνότητα, το διάνυσμα της ταχύτητας και ο αριθμός $Mach$ της επ΄ άπειρον ροής (ϱ_∞, $|\overrightarrow{u}_\infty|$, $\theta_{1\infty}$, $\theta_{2\infty}$, M_∞). Επομένως:

$$
\begin{aligned}
\varrho_{out} &= \varrho_\infty \\
(\varrho u)_{out} &= \varrho_\infty |\overrightarrow{u}_\infty| \cos\theta_{1\infty} \cos\theta_{2\infty} \\
(\varrho v)_{out} &= \varrho_\infty |\overrightarrow{u}_\infty| \sin\theta_{1\infty} \\
(\varrho w)_{out} &= \varrho_\infty |\overrightarrow{u}_\infty| \cos\theta_{1\infty} \sin\theta_{2\infty}
\end{aligned}
$$

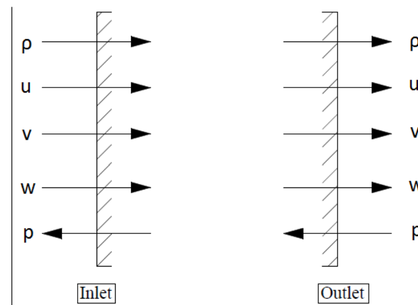
$$
\left.
\begin{aligned}
M_\infty = \frac{|\overrightarrow{u}_\infty|}{c} = \frac{|\overrightarrow{u}_\infty|}{\sqrt{\gamma(\gamma-1)T_\infty}} \\
\frac{p_{out}}{\varrho_\infty} = (\gamma-1)T_\infty
\end{aligned}
\right\} \Rightarrow M_\infty = |\overrightarrow{u}_\infty| \sqrt{\tfrac{\varrho_\infty}{p_{out}}} \Rightarrow p_{out} = \varrho_\infty \left(\tfrac{|\overrightarrow{u}_\infty|}{M_\infty}\right)^2
$$

$$
\varrho E_{out} = \tfrac{p_{out}}{\gamma-1} + \tfrac{1}{2}\varrho_{far} |\overrightarrow{u}_\infty|^2
$$

Εσωτερική Αεροδυναμική

Προηγουμένως έγινε αναφορά για μεταφορά «πληροφορίας» προς κατεύθυνση ανάλογη του προσήμου της αντίστοιχης ιδιοτιμής. Ας θεωρηθεί, εδώ, ως «πληροφορία» οι συντηρητικές μεταβλητές της ροής. Έχει αποδειχθεί ότι αν γράψουμε τις εξισώσεις Euler συναρτήσει των πρωτευουσών μεταβλητών με ανάλογη διαδικασία προκύπτει ένα αντίστοιχο μητρώο $\underline{A}$ με ιδιοτιμές ίδιες με εκείνες που παρουσιάστηκαν νωρίτερα. Δηλαδή το πρόσημο των ιδιοτιμών μπορεί να δώσει τη κατεύθυνση που ακολουθούν οι πρωτεύουσες μεταβλητές της ροής στο εσωτερικό του πεδίου ροής.

Στην περίπτωση υποηχητικής ροής χρειάζεται να ορισθούν τέσσερα μεγέθη στην είσοδο $(p_t, T_t, \theta_1, \theta_2)$ και ένα στην έξοδο (M_{is}). Τα υπόλοιπα στοιχεία των θεωρητικών κόμβων θα προκύψουν από το εσωτερικό του πεδίου ροής. Δηλαδή:



Είσοδος

$$
\begin{aligned}
p_{out} &= p_P \\
T_{out} &= T_t \left(\frac{p_{out}}{p_t}\right)^{\frac{\gamma-1}{\gamma}} \\
T_t &= T + \frac{1}{2\gamma}\,|\overrightarrow{u}_{out}|^2 \Rightarrow |\overrightarrow{u}_{out}| = \sqrt{2\gamma\,(T_t - T)} \\
u_{out} &= |\overrightarrow{u}_{out}| \cos\theta_1 \cos\theta_2 \\
v_{out} &= |\overrightarrow{u}_{out}| \sin\theta_1 \\
w_{out} &= |\overrightarrow{u}_{out}| \cos\theta_1 sin\theta_2
\end{aligned}
$$

Επομένως

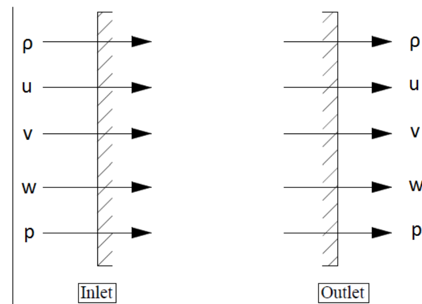
$$\begin{aligned}
\varrho_{out} &= \frac{p_{out}}{(\gamma-1)T_{out})} \\
(\varrho u)_{out} &= \varrho_{out} u_{out} \\
(\varrho v)_{out} &= \varrho_{out} v_{out} \\
(\varrho w)_{out} &= \varrho_{out} w_{out} \\
\varrho E_{out} &= \frac{p_{out}}{\gamma-1} + \frac{1}{2}\varrho_{out} \left|\overrightarrow{u}_{out}\right|^2
\end{aligned}$$

Έξοδος

$$p_{out} = p_t \left(1 + \frac{\gamma-1}{2} M_{is}^2\right)^{-\frac{\gamma}{\gamma-1}}$$

$$\begin{aligned}
\varrho_{out} &= \varrho_P \\
(\varrho u)_{out} &= (\varrho u)_P \\
(\varrho v)_{out} &= (\varrho v)_P \\
(\varrho w)_{out} &= (\varrho w)_P \\
\varrho E_{out} &= \frac{p_{out}}{\gamma-1} + \frac{1}{2}\varrho_P \left(u_P^2 + v_P^2\right)
\end{aligned}$$

Αντίθετα σε υπερηχητικές ροές όπου και οι πέντε ιδιοτιμές είναι θετικές ουσιαστικά θα δοθούν όλα τα στοιχεία που απαιτούνται για τον υπολογισμό των συντηρητικών μεταβλητών στον ψευδο-κόμβο εισόδου, ενώ οι αντίστοιχες τιμές του ψευδο-κόμβου εξόδου ταυτίζονται με εκείνες του αντίστοιχου οριακού κόμβου εφόσον η πληροφορία μεταφέρεται εξ ολοκλήρου προς τη κατεύθυνση της ροής . Τα μεγέθη που δίνονται σε τέτοιου είδους εφαρμογές είναι όπως και πριν η ολική πίεση (p_t), η θερμοκρασία (T_t) και οι γωνίες θ_1 και θ_2. Αντίθετα ο αριθμός $Mach$ αντιστοιχεί στην είσοδο.



Είσοδος

$$
\begin{aligned}
p_{out} &= p_t \left(1 + \frac{\gamma - 1}{2} M^2\right)^{-\frac{\gamma}{\gamma-1}} \\
T_{out} &= T_t \left(\frac{p_{out}}{p_t}\right)^{\frac{\gamma-1}{\gamma}} \\
|\overrightarrow{u}_{out}| &= \sqrt{2\gamma(T_t - T_{out})} \\
u_{out} &= |\overrightarrow{u}_{out}| \cos\theta_1 \cos\theta_2 \\
v_{out} &= |\overrightarrow{u}_{out}| \sin\theta_1 \\
w_{out} &= |\overrightarrow{u}_{out}| \cos\theta_1 \sin\theta_2
\end{aligned}
$$

$$
\begin{aligned}
\varrho_{out} &= \frac{p_{out}}{(\gamma - 1) T_{out}} \\
(\varrho u)_{out} &= \varrho_{out} u_{out} \\
(\varrho v)_{out} &= \varrho_{out} v_{out} \\
(\varrho w)_{out} &= \varrho_{out} w_{out} \\
\varrho E_{out} &= \frac{p_{out}}{\gamma - 1} + \frac{1}{2} \varrho_{out} \left|\overrightarrow{u}_{out}\right|^2
\end{aligned}
$$

Έξοδος

$$
\begin{aligned}
\varrho_{out} &= \varrho_P \\
(\varrho u)_{out} &= (\varrho u)_P \\
(\varrho v)_{out} &= (\varrho v)_P \\
(\varrho w)_{out} &= (\varrho w)_P \\
\varrho E_{out} &= \frac{p_P}{\gamma - 1} + \frac{1}{2} \varrho_P (u_r u_r)_P
\end{aligned}
$$

2.4.3 Αξονική συμμετρία

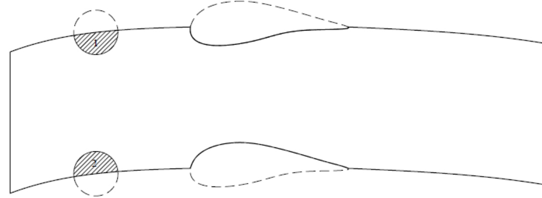
Αρκετά συχνά συναντώνται χωρία ροής συμμετρικά ως προς κάποιο επίπεδο. Στις περιπτώσεις αυτές για προφανείς λόγους συμφέρει να χωρίζεται το χωρίο στα δύο συμμετρικά όμοια κομμάτια του και να γίνεται αριθμητική επίλυση της ροής σε ένα από αυτά. Ο υπολογισμός λοιπόν του διανύσματος ροής $\left(\overrightarrow{\Phi}\right)$ που εξέρχεται από όρια συμμετρίας είναι ίδιος με εκείνον στα στερεά τοιχώματα. Αυτό συμβαίνει επειδή η ύπαρξη της συμμετρίας υπαγορεύει το διάνυσμα της ταχύτητας να είναι εφαπτομενικό στο όριο, δηλαδή να ισχύει $\overrightarrow{u} \cdot \overrightarrow{n} = 0$, κάτι το οποίο θυμίζει έντονα τη συνθήκη μη εισχώρησης για τα ατριβή ρευστά κατά μήκος των στερεών τοιχωμάτων.

2.4.4 Περιοδικά όρια

Υπολογιστικά χωρία όπως λ.χ. οι πτερυγώσεις στροβιλομηχανών είναι χαρακτηριστικά παραδείγματα χωρίων με περιοδικά όρια. Στα χωρία αυτά οι υπολογιστικές κυψέλες του ενός περιοδικού ορίου συμπληρώνουν τις κυψέλες των αντίστοιχων κόμβων του άλλου περιοδικού ορίου. Οπότε στις περιπτώσεις αυτές δεν είναι απαραίτητος ο υπολογισμός των εξερχόμενων διανυσμάτων ροής από τα περιοδικά όρια του χωρίου, καθώς στα διανύσματα ροής που έχουν υπολογιστεί για μία κυψέλη προστίθενται τα διανύσματα που έχουν υπολογιστεί για την κυψέλη του αντίστοιχου κόμβου του άλλου περιοδικού ορίου. Τα παραπάνω γίνονται εύκολα κατανοητά από το ακόλουθο σχήμα όπου παρίσταται μία επιφάνεια από πτερύγιο σε πτερύγιο. Το πάνω και κάτω όριο του χωρίου (εκτός των τοιχωμάτων των αεροτομών) αποτελούν περιοδικά όρια εφόσον το εν λόγω χωρίο επαναλαμβάνεται και προς τα πάνω και προς τα κάτω ώστε να συμπληρωθεί μία επιφάνεια S_1 μίας στροβιλομηχανής. Από το σχήμα φαίνεται ότι ουσιαστικά αυτό που «λείπει» από την κυψέλη 1 είναι η κυψέλη 2 και αντιστρόφως. Επιπλέον αφού το χωρίο επαναλαμβάνεται τα χαρακτηριστικά της ροής πάνω στο κάτω όριο του χωρίου ροής πρέπει να είναι τα ίδια με εκείνα εξωτερικά του χωρίου, πάνω από το πάνω όριο. Δηλαδή, η ένωση των δύο κυψελών δομεί την κυψέλη που θα σχηματίζονταν στους αντίστοιχους κόμβους αν ως υπολογιστικό χωρίο χρησιμοποιούσαμε ολόκληρη την αλληλουχία αεροτομών πτερυγίων κατά την κατεύθυνση του βήματος (αντί μόνο μίας πτερύγωσης). Για τον λόγο αυτό, το εξερχόμενο από το όριο του χωρίου διάνυσμα ροής στη περιοχή της κυψέλης 1 ισούται με το άθροισμα των υπολογισμένων διανυσμάτων ροής γύρω από την κυψέλη 2, και αντιστρόφως.

Τα παραπάνω ισχύουν σε εφαρμογές γραμμικών πτερυγώσεων. Σε εφαρμογές περιφερειακών πτερυγώσεων επαναλαμβάνονται τα παραπάνω, με μόνη

διαφορά ότι η περιοδικότητα εμφανίζεται στις πολικές συντεταγμένες και, ως εκ τούτου, χρειάζεται περιστροφή των χωρίων πριν την άθροισή τους, κατά τον άξονα της ροής.



Σχήμα 2.5: Περιοδικό χωρίο ροής.

2.5 Επίλυση διακριτοποιημένων εξισώσεων

2.5.1 Μέθοδος αριθμητικής επίλυσης

Οι διακριτοποιημένες εξισώσεις μπορούν να ξαναγραφούν με χρήση του τελεστή υπολοίπου $\overrightarrow{R}$ στην παρακάτω δέλτα-μορφή:

$$\frac{V^P}{\Delta t^P}\Delta\overrightarrow{U}^P + \overrightarrow{R}^{P,n+1} = 0 \tag{2.37}$$

όπου n το παρόν ψευδοχρονικό βημα.

Η ανανέωση των μεταβλητών σε κάθε ψευδοχρονικό βήμα γίνεται με χρήση σημειακά πεπλεγμένου σχήματος και εσωτερικών επαναλήψεων του σημειακά πεπλεγμένου επιλυτή Jacobi. Με γραμμικοποίηση του τελεστή υπολοίπου $\overrightarrow{R}$ προκύπτει το παρακάτω σχήμα για την νέα επανάληψη του ψευδοχρονικού βήματος

$$\left[\frac{V^P}{\Delta t^P}I + \frac{\partial\overrightarrow{R}^P}{\partial\overrightarrow{U}^P}\right]\Delta\overrightarrow{U}^P + \sum_{Q\in K_P}\left(\frac{\partial\overrightarrow{R}^P}{\partial\overrightarrow{U}^Q}\right)\Delta\overrightarrow{U}^Q = -\overrightarrow{R}^P \tag{2.38}$$

ή εναλλακτικά, σε τανυστική γραφή

$$\left[\frac{V^P}{\Delta t^P}\delta_{nk}\delta_{PK} + \frac{\partial R_n^P}{\partial U_k^K}\right]\Delta U_k^K = -R_n^P \tag{2.39}$$

όπου K είναι όλοι οι κόμβοι που συμμετέχουν στην εξίσωση του κόμβου P συμπεριλαμβανόμενου και του ίδιου. Το παραπάνω γραμμικοποιημένο σύστημα επιλύεται με την επαναληπτική μέθοδο Jacobi. Η επιλογή αυτή δικαιολογείται από το γεγονός ότι η μέθοδος επωφελείται από τη διαγώνια κυριαρχία που παρέχει το ανάντι σχήμα διακριτοποίησης [16] και από το γεγονός ότι προσφέρεται για παραλληλοποίηση, στοιχείο πολύ σημαντικό καθώς ο επιλύτης προορίζεται για GPU η οποία και λειτουργεί εντόνως παράλληλα. Η γραμμικοποίηση που χρησιμοποιείται αποτελεί προσέγγιση της ακριβούς και επιλέγεται έτσι ώστε να ενισχύεται η διαγώνια κυριαρχία, προκειμένου να διευκολυνθεί η σύγκλιση της μεθόδου και να μη αυξηθούν υπερβολικά οι απαιτήσεις σε μνήμη υπολογιστή. Τα παραπάνω επιτυγχάνονται αν δεν ληφθούν οι μη-διαγώνιες συνεισφορές όλων των κόμβων πέρα από τους πρώτους γείτονες.

Κεφάλαιο 3

Η αρχιτεκτονική παράλληλης επεξεργασίας CUDA

Η αρχιτεκτονική CUDA (Compute Unified Device Architecture) αναπτύσσεται από την NVIDIA και αποτελεί την αρχή κατασκευής των προγραμματιζόμενων καρτών γραφικών της εταιρίας. Οι G80, GT200 και Fermi αποτελούν μερικές ενδεικτικές εκδόσεις αρχιτεκτονικών CUDA τη χρονική περίοδο 2006-2012.

Κάθε προγραμματιζόμενη GPU της NVIDIA φέρει έναν αριθμό που χαρακτηρίζει την υπολογιστική δυνατότητα της κάρτας. Για παράδειγμα, οι GPUs υπολογιστικής δυνατότητας 1.0, ή 1.1 δεν υποστηρίζουν την αριθμητική διπλής ακρίβειας. Οι Tesla M2050 που χρησιμοποιήθηκαν στην παρούσα εργασία, έχουν υπολογιστική δυνατότητα 2.0 και είναι δομημένες σύμφωνα με την αρχιτεκτονική Fermi. Στη συνέχεια του κειμένου, χάριν ευκολίας και συντόμευσης, μπορεί να παραλείπεται ο όρος «υπολογιστική δυνατότητα», δηλαδή ο όρος «κάρτα γραφικών 2.x» σημαίνει «κάρτα γραφικών υπολογιστικής δυνατότητας 2.x».

Στο κεφάλαιο αυτό περιγράφεται η δομή της αρχιτεκτονικής Fermi και παρουσιάζεται το περιβάλλον προγραμματισμού της CUDA. Για περισσότερες λεπτομέρειες σχετικά με το περιβάλλον προγραμματισμού της CUDA ή τη δομή παλαιότερων αρχιτεκτονικών, ο αναγνώστης μπορεί να ανατρέξει στο αντίστοιχο εγχειρίδιο [17], ή στα βιβλία [18, 19] και στη διατριβή [7].

3.1 Το thread ως η βασική μονάδα επεξεργασίας

Το thread ορίζεται ως η βασική μονάδα επεξεργασίας. Κάθε GPU μπορεί να χειρίζεται τόσα threads όσα επιτρέπει η υπολογιστική της δυνατότητα και η αρχιτεκτονική με την οποία αυτή είναι δομημένη. Δηλαδή, οι σημερινές GPUs έχουν τη δυνατότητα δημιουργίας threads, στα οποία αναθέτουν την εκτέλεση ενός προγράμματος (kernel).

3.2 Οργάνωση των threads σε μία GPU

Οι σημερινές GPUs μπορούν να χειρίζονται ταυτόχρονα χιλιάδες threads. Για τη διαχείριση αυτών εφαρμόζουν τη λογική SIMT (Single Instruction Multiple Thread). Έτσι το ίδιο τμήμα κώδικα (kernel) εκτελείται από μία ομάδα από threads, με κάθε thread να διαχειρίζεται διαφορετικό όγκο δεδομένων.

Τα threads οργανώνονται σε blocks και εκτελούνται στους διαθέσιμους πολυεπεξεργαστές της GPU. Κάθε πολυεπεξεργαστής μπορεί να αναλάβει την εκτέλεση έως και 8 blocks. Στην περίπτωση που δεν επαρκούν οι διαθέσιμοι πολυεπεξεργαστές της GPU, κάποια blocks μένουν ανενεργά, αναμένοντας την ολοκλήρωση της εκτέλεσης ορισμένων εκ των ενεργών. Το γεγονός αυτό αποτελεί ένα μεγάλο πλεονέκτημα της CUDA, καθώς ο ίδιος κώδικας εκτελείται σε κάρτες γραφικών διαφορετικού αριθμού πολυεπεξεργαστών, χωρίς να απαιτείται η τροποποίηση ή προσαρμογή αυτού. Πρακτικά, όσους περισσότερους πολυεπεξεργαστές έχει μία GPU, τόσα περισσότερα blocks από threads εκτελούνται ταυτόχρονα. Ένα block μπορεί να αποτελείται το πολύ από 512 ή 1024 threads, για GPUs 1.x ή 2.x αντίστοιχα.

Τα blocks που εκτελούν το ίδιο kernel, σχηματίζουν ένα πλέγμα από blocks. Στο υπόλοιπο του κειμένου της εργασίας, προκειμένου να αποφευχθεί η σύγχυση ανάμεσα στο πλέγμα των blocks και στο πλέγμα που χρησιμοποιείται για τη χωρική διακριτοποίηση του υπολογιστικού χωρίου, το πρώτο θα αποκαλείται «grid» ενώ το δεύτερο «πλέγμα».

Η εκτέλεση των threads όπως αναφέρθηκε προηγουμένως γίνεται στους διαθέσιμους πολυεπεξεργαστές της GPU. Για την ακρίβεια, κάθε πολυεπεξεργαστής ομαδοποιεί τα threads σε ομάδες των 32, που ονομάζονται warps. Ο αριθμός των warps που μπορεί να χειρίζεται ταυτόχρονα ο πολυεπεξεργαστής εξαρτάται από την υπολογιστική δυνατότητα της GPU. Half-warp ορίζεται το πρώτο, ή το δεύτερο μισό ενός warp. Τονίζεται ότι τα threads του ίδιου warp

εκτελούν την ίδια σειρά εντολών ταυτόχρονα. Επομένως στη περίπτωση που η δομή ενός kernel είναι αντίστοιχη του ψευδοκώδικα 1, ο προγραμματιστής πρέπει να εγγυηθεί ότι όλα τα threads του ίδιου warp θα ακολουθήσουν την ίδια σειρά εντολών (Α-Β-Δ), ή (Α-Γ-Δ). Στην περίπτωση που τα threads ενός warp διακρίνονται σε ομάδες O_B, O_Γ ανάλογα με ποια λογική έκφραση ($\Lambda\mathrm{E}_B$ ή $\Lambda\mathrm{E}_\Gamma$) είναι αληθής, τότε όταν τα threads της O_B εκτελούν τη σειρά εντολών Β, εκείνα της O_Γ παραμένουν ανενεργά. Αντίστοιχα, όταν τα threads της O_Γ εκτελούν τη «δική τους» σειρά εντολών (Γ), τα threads της O_B παραμένουν ανενεργά. Γίνεται αντιληπτό ότι όσο περισσότερες ομάδες από threads σχηματίζονται στο ίδιο warp, τόσο περισσότερο ζημιώνεται η απόδοση του kernel. Threads διαφορετικών warps επιτρέπεται να ακολουθούν διαφορετική πορεία εκτέλεσης.

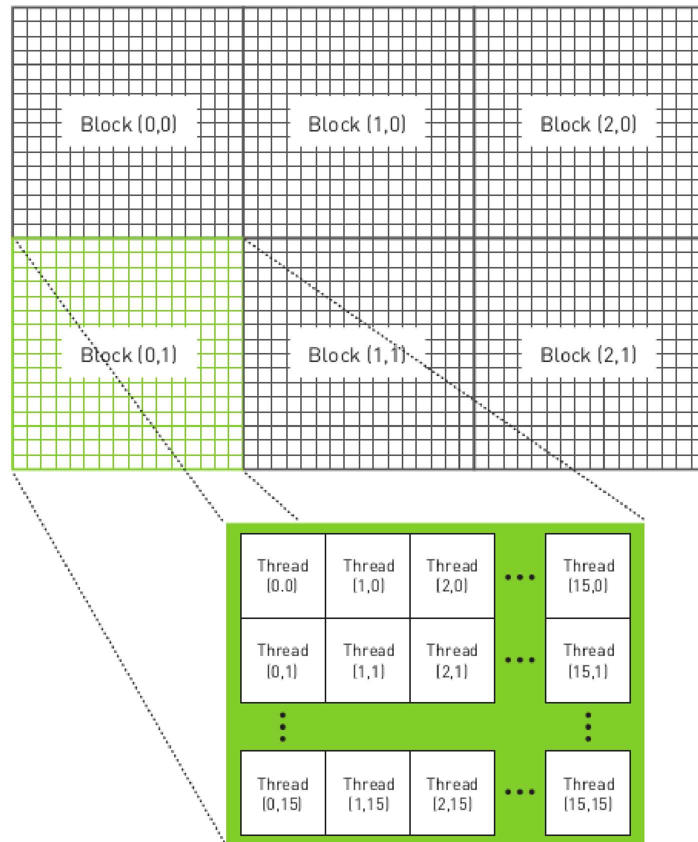
Ψευδοκώδικας 1

```
1: Σειρά εντολών Α
2: if Λογική έκφραση ΛΕ_B then
3:     Σειρά εντολών Β
4: else if Λογική έκφραση ΛΕ_Γ then
5:     Σειρά εντολών Γ
6: end if
7: Σειρά εντολών Δ
```

Κάθε thread χαρακτηρίζεται από έναν τοπικό αύξοντα αριθμό στο block που αυτό ανήκει και, αντίστοιχα, κάθε block από έναν αύξοντα αριθμό του στο grid. Έτσι, όπως θα δειχθεί στο παράδειγμα της παραγράφου 3.9, κάθε thread μπορεί να συσχετιστεί με συγκεκριμένες θέσεις μνήμης με βάση τον αύξοντα αριθμό του στο block και τον αύξοντα αριθμό του block (στο οποίο ανήκει) στο grid. Έτσι, κάθε thread επεξεργάζεται διαφορετικά δεδομένα.

Ο προγραμματιστής επιλέγει ανάμεσα σε 1Δ, 2Δ, ή 3Δ διάταξη των threads στα blocks και των blocks στο grid (Σχήμα 3.1). Σημειώνεται ότι μόνο οι κάρτες γραφικών αρχιτεκτονικής Fermi υποστηρίζουν την 3Δ διάταξη των blocks στο grid. Η επιλογή της διαμέρισης του grid σε blocks και των blocks σε threads, εξαρτάται άμεσα με το προς επίλυση πρόβλημα.

Η διάταξη και η οργάνωση των threads σε blocks και των blocks σε grid φαίνεται στο σχήμα 3.2. Στο σχήμα αυτό φαίνονται επιπλέον οι μνήμες της GPU στις οποίες έχει πρόσβαση κάθε thread. Οι μνήμες αυτές παρουσιάζονται επιγραμματικά στην παράγραφο 3.3 και αναλυτικότερα στην παράγραφο 3.5.



Σχήμα 3.1: Διάταξη από blocks και threads σε 2 διαστάσεις [18]

3.3 Είδη μνήμης της GPU

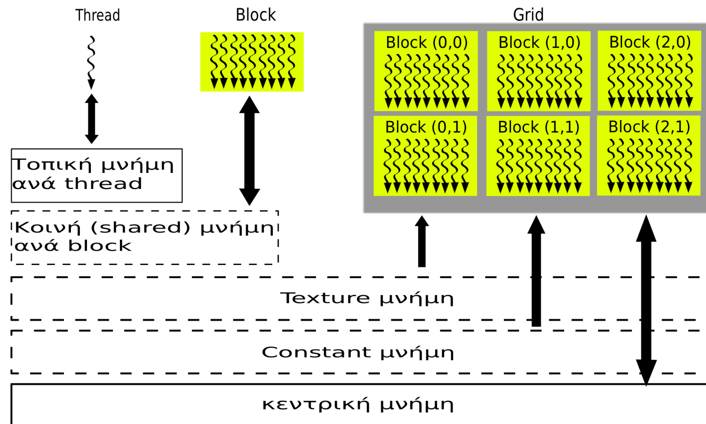
Μία GPU έχει ενσωματωμένες τις ακόλουθες μνήμες:

1. Τοπική (Local) μνήμη ανά thread.
2. Κοινή μνήμη ανάμεσα στα threads του ίδιου block (shared μνήμη).
3. Τις texture και constant μνήμες που είναι προσβάσιμες από όλα τα threads.
4. Την κεντρική μνήμη της GPU που είναι προσβάσιμη από όλα τα threads.

Έτσι, κάθε thread έχει τη δική του τοπική μνήμη. Τα threads του ίδιου block έχουν τη δυνατότητα συγχρονισμού και μοιράζονται δεδομένα μέσω μίας κοινής, γρήγορα προσπελάσιμης μνήμης (shared μνήμη). Όλα τα threads του grid

έχουν πρόσβαση στην κεντρική μνήμη της GPU και στις texture και constant, οι οποίες επιτρέπουν μόνο την ανάγνωση δεδομένων από αυτές.

Η οργάνωση των threads όπως παρουσιάστηκε στην παράγραφο 3.2 φαίνεται στο σχήμα 3.2. Στο ίδιο σχήμα φαίνονται και οι διαθέσιμες μνήμες μίας GPU.



Σχήμα 3.2: Οργάνωση των threads σε blocks, και των blocks σε grid. Εδώ, ο προγραμματιστής έχει επιλέξει 1Δ κατανομή των threads σε blocks, και 2Δ κατανομή των blocks στο grid. Κάθε thread έχει τη δική του τοπική μνήμη. Τα threads του ίδιου block μοιράζονται δεδομένα μέσω της κοινής γρήγορα προσπελάσιμης shared μνήμης. Όλα τα threads του grid έχουν πρόσβαση στην κεντρική μνήμη της GPU, την texture και την constant. Οι τελευταίες επιτρέπουν μόνο την ανάγνωση δεδομένων από αυτές.

3.4 Η αρχιτεκτονική Fermi

Οι πολυεπεξεργαστές των καρτών γραφικών που βασίζονται στην αρχιτεκτονική Fermi (σχήμα 3.3, [20]) συγκροτούνται σε GPCs (Graphics Processor Clusters). Αναλυτικά, κάθε GPC περιέχει:

- 4 πολυεπεξεργαστές.
- 1 ενδιάμεση μνήμη (L2 cache).

Κάθε πολυεπεξεργαστής αποτελείται από:

- 32 ή 48 πυρήνες.
- 4 ή 8 ειδικές μονάδες εκτέλεσης μαθηματικών συναρτήσεων (Special Function Units, SFU).

- 2 warp schedulers.
- 2 ενδιάμεσες μνήμες (constant cache, texture cache).
- 1 ενδιάμεση μνήμη (L1 cache).
- 1 γρήγορα προσπελάσιμη μνήμη (shared μνήμη).
- 32768 32-bit καταχωρητές (registers).

Ο αριθμός των πυρήνων ενός πολυεπεξεργαστή και εκείνος των SFUs εξαρτάται από την υπολογιστική δυνατότητα της GPU. Έτσι, κάρτες υπολογιστικής δυνατότητας 2.0, όπως οι Tesla M2050, έχουν 32 πυρήνες και 4 SFUs, ενώ κάρτες δυνατότητας 2.1 έχουν 48 πυρήνες και 8 SFUs.

Οι warp schedulers ορίζουν στα warps τις εργασίες που θα εκτελέσουν. Τα warps που εκτελούνται σε έναν πολυεπεξεργαστή χαρακτηρίζονται από έναν τοπικό αύξοντα αριθμό. Δηλαδή τα threads 0 έως 31 ανήκουν στο warp 0 κ.ο.κ. Έτσι ο πρώτος από τους δύο warp schedulers χειρίζεται τα warps με περιττό αριθμό ενώ ο δεύτερος με άρτιο. Επιπλέον, οι 32 (ή 48) πυρήνες του πολυεπεξεργαστή χωρίζονται σε δύο ομάδες των 16 (ή 24). Έτσι, ο πρώτος warp scheduler ορίζει στα warps που χειρίζεται εκείνος τη χρήση της πρώτης ομάδας πυρήνων για την εκτέλεση αριθμητικών πράξεων μεταξύ ακεραίων, ή πραγματικών αριθμών απλής ακρίβειας. Αντίστοιχα ενεργεί ο δεύτερος warp scheduler αξιοποιώντας τη δεύτερη ομάδα πυρήνων. Σε περίπτωση που απαιτείται η εκτέλεση αριθμητικών πράξεων μεταξύ πραγματικών αριθμών διπλής ακρίβειας χρησιμοποιούνται από τους warp schedulers το σύνολο των πυρήνων ανά πολυεπεξεργαστή. Γίνεται αντιληπτό, ότι όταν ένα warp εκτελεί πράξεις διπλής ακρίβειας, κανένα άλλο warp του πολυεπεξεργαστή δεν μπορεί να χρησιμοποιήσει τους πυρήνες του πολυεπεξεργαστή. Αντίθετα, επιτρέπεται η χρήση των SFUs ή η προσπέλαση των μνημών της GPU από τα threads ενός άλλου warp. Στις κάρτες γραφικών 2.0 οι warp schedulers αναθέτουν 2 οδηγίες σε 2 warps. Αντίθετα, σε κάρτες γραφικών 2.1, οι warp schedulers αναθέτουν 2 οδηγίες ανά warp, δηλαδή συνολικά 4 οδηγίες σε 2 warps.

Το γεγονός ότι οι warp schedulers αναθέτουν την εκτέλεση των αριθμητικών πράξεων μεταξύ πραγματικών αριθμών διπλής ακρίβειας στο σύνολο των πυρήνων του πολυεπεξεργαστή ενώ εκείνες μεταξύ πραγματικών αριθμών απλής ακρίβειας στους μισούς πυρήνες, κάνει την ταχύτητα εκτέλεσης αριθμητικών πράξεων διπλής ακρίβειας τη μισή εκείνων με απλή ακρίβεια. Για παράδειγμα, μια Tesla M2050 χαρακτηρίζεται από ταχύτητα εκτέλεσης αριθμητικών πράξεων απλής ακρίβειας ίση με 1030 Gflops και διπλής ακρίβειας ίση με 515 Gflops.

Οι ενδιάμεσες μνήμες constant, texture cache επιταχύνουν την ανάγνωση δεδομένων από την constant και την texture μνήμη, αντίστοιχα, που βρίσκονται στον ίδιο χώρο με την κεντρική μνήμη της GPU. Η διάσταση της texture cache εξαρτάται από τον τύπο της κάρτας γραφικών και είναι ανάμεσα σε 6 και 8 KB ανά πολυεπεξεργαστή. Η constant cache έχει διάσταση 8 KB και η constant μνήμη 64 KB.

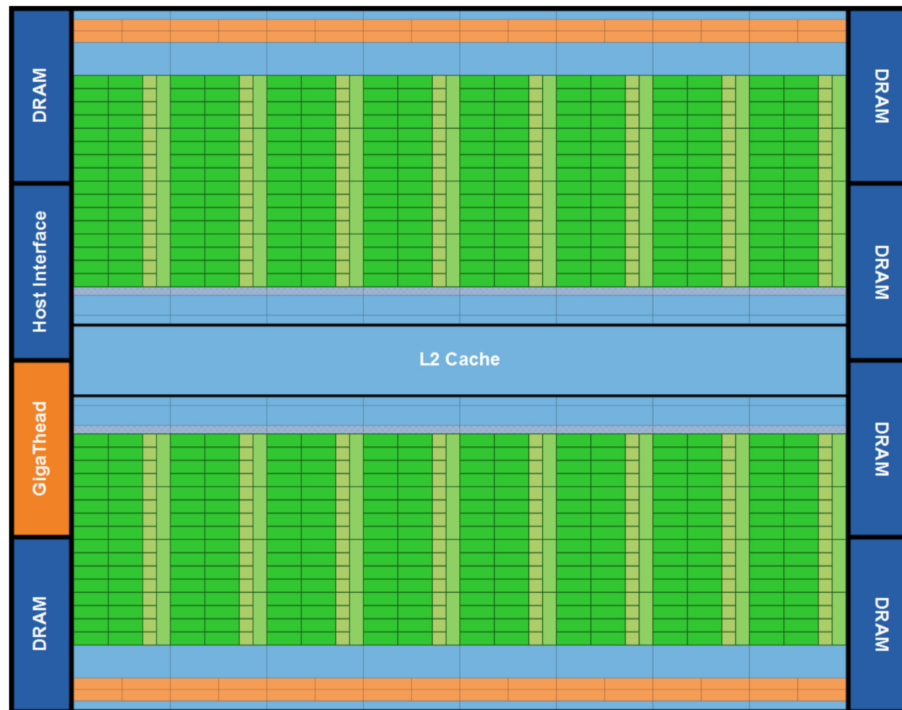
Οι ενδιάμεσες μνήμες L1, L2 cache επιταχύνουν την πρόσβαση στην κεντρική μνήμη της GPU και στις τοπικές ανά thread. Υπενθυμίζεται ότι σε κάθε thread αντιστοιχεί συγκεκριμένος χώρος της κεντρικής μνήμης. Ο χώρος αυτός ανά thread αποτελεί την τοπική ανά thread μνήμη. Στις κάρτες γραφικών αρχιτεκτονικής Fermi το μέγεθος της τοπικής ανά thread μνήμης είναι 512 KB. Το συνολικό μέγεθος της L2 cache για όλους τους πολυεπεξεργαστές είναι 768 KB. Η L1 βρίσκεται στον ίδιο χώρο με τη shared μνήμη και δίνεται η δυνατότητα στον προγραμματιστή να επιλέξει ανάμεσα σε 16 KB L1 και 48 KB shared (αποτελεί προεπιλογή) ή ανάποδα. Η επιλογή γίνεται σύμφωνα με τις απαιτήσεις του kernel σε shared μνήμη. Έτσι, στην περίπτωση που ένα kernel χρησιμοποιεί λιγότερα των 16 KB από τη shared μνήμη ανά πολυεπεξεργαστή, συμφέρει ο προγραμματιστής να μοιράσει τα συνολικά 64 KB ανά πολυεπεπεξεργαστή σε 16 KB για τη shared και 48 KB για την L1 cache. Υπενθυμίζεται ότι η shared μνήμη επιτρέπει την επικοινωνία μεταξύ των threads του ίδιου block.

Κάθε πολυεπεξεργαστής περιέχει 32768 32-bit registers. Με βάση τις απαιτήσεις του kernel σε registers και KB shared μνήμης ανά πολυεπεξεργαστή, καθορίζεται ο αριθμός των blocks που διανέμεται στους πολυεπεξεργαστές. Ο μέγιστος αριθμός threads που μπορεί γενικά να χειριστεί ένας πολυεπεξεργαστής είναι 1536.

Συνοψίζοντας, οι Tesla M2050 περιέχουν συνολικά 448 πυρήνες ή 14 πολυεπεξεργαστές. Δηλαδή, μπορούν να χειρίζονται έως και $14 \times 1536 = 21504$ threads.

3.5 Περιγραφή των ειδών μνήμης της GPU

Το μέγεθος των ενδιάμεσων (cache) μνημών των GPUs είναι πολύ μικρότερο σε σχέση με εκείνο των σύγχρονων CPUs. Ενδεικτικά αναφέρεται ότι κάθε blade server του cluster διασυνδεδεμένων GPUs του ΕΘΣ, έχει 2 quad core CPUs. Κάθε πυρήνας μίας CPU έχει 12288 KB cache μνήμη. Το μέγεθος αυτό είναι πολύ μεγαλύτερο σε σχέση με το συνολικό μέγεθος των cache μνημών ανά πολυεπεξεργαστή μίας GPU. Οι cache μνήμες μίας GPU είναι η constant και η texture cache και οι L1, L2 cache (αν η GPU είναι υπολογιστικής δυνατότητας



Σχήμα 3.3: Σχεδιάγραμμα της αρχιτεκτονικής Fermi, [20]. Σε αυτήν, 16 πολυεπεξεργαστές μοιράζονται την ίδια L2 cache που επιταχύνει τη προσπέλαση της κεντρικής μνήμης της κάρτας γραφικών και της τοπικής ανά thread που βρίσκεται στον ίδιο χώρο. Ο δίαυλος επικοινωνίας (Host Interface) επιτρέπει τη μεταφορά δεδομένων από την κεντρική μνήμη της κάρτας γραφικών σε εκείνη του υπολογιστή και ανάποδα. Μία ειδική μονάδα (GigaThread) διανέμει τα blocks στους διαθέσιμους πολυεπεξεργαστές. Κάθε πολυεπεξεργαστής έχει 32 πυρήνες, 4 ειδικές μονάδες εκτέλεσης μαθηματικών συναρτήσεων, 2 μονάδες χειρισμού των warps, registers των 32-bit και 64 KB shared και L1 cache.

2.x). Δεν θα ήταν λάθος να προστεθεί στις μνήμες που αναφέρθηκαν προηγουμένως και η γρήγορη shared μνήμη, καθώς η ταχύτητα πρόσβασης σε αυτή είναι ίδια με την ταχύτητα πρόσβασης σε cache μνήμες. Μία κάρτα γραφικών αρχιτεκτονικής Fermi έχει 8 KB constant cache, 8 KB texture cache, 64 KB για τη shared και την L1 cache και 48 KB L2 cache ανά πολυεπεξεργαστή. Συνολικά, δηλαδή, έχει 128 KB cache μνημών.

Η μικρή έκταση των cache μνήμων της GPU τονίζει τη σημασία της επιλογής του αποδοτικότερου τρόπου προσπέλασης των μνημών της GPU και καθιστά άρρηκτα συνδεδεμένη την απόδοση ενός GPU-κώδικα με θέματα διαχείρισης των μνημών της κάρτας γραφικών.

3.5.1 Κεντρική μνήμη (global memory)

Η κεντρική μνήμη της GPU (global memory) αποτελεί τη μεγαλύτερη σε έκταση μνήμη της κάρτας γραφικών και είναι προσπελάσιμη από όλα τα threads. Χαρακτηρίζεται, όμως, από υψηλό χρόνο προσπέλασης. Συνεπώς, η χρήση της γρήγορα προσπελάσιμης shared μνήμης, ή των cached constant και texture μνημών αντί της κεντρικής, φυσικά όταν αυτό γίνεται, αυξάνει κατακόρυφα την απόδοση του GPU-κώδικα. Οι shared, constant και texture μνήμες παρουσιάζονται αναλυτικά στις επόμενες παραγράφους. Επιπλέον η χρήση της κεντρικής μνήμης πρέπει να περιορίζεται, κατά το δυνατό, σε ένα kernel.

Οι GPUs αντιμετωπίζουν την κεντρική μνήμη ως μία αλληλουχία από τμήματα μήκους 32, 64, ή 128 Bytes. Αυτό πρακτικά σημαίνει ότι ο δίαυλος μεταφοράς δεδομένων από την κεντρική μνήμη στους registers των πολυεπεξεργαστών, μεταφέρει τμήματα της κεντρικής μνήμης μήκους 32, 64, ή 128 Bytes. Επομένως, όσο μικρότερο είναι το πλήθος των τμημάτων της κεντρικής μνήμης στα οποία έχουν πρόσβαση τα threads ενός warp, τόσο λιγότερες φορές θα μεταφέρει δεδομένα ο δίαυλος επικοινωνίας. Σημειώνεται ότι η διάσταση ενός ακεραίου είναι 2 Bytes, η διάσταση ενός πραγματικού αριθμού απλής ακρίβειας 4 Bytes και εκείνη ενός διπλής ακρίβειας 8 Bytes. Συνεπώς, η συνολική διάσταση των ακεραίων, πραγματικών αριθμών απλής, ή διπλής ακρίβειας που χειρίζονται και τα 16 threads ενός half–warp είναι 32, 64, ή 128 Bytes, αντίστοιχα.

3.5.2 Constant μνήμη

Η constant μνήμη βρίσκεται στον ίδιο χώρο με την κεντρική μνήμη της GPU. Αντίθετα, όμως, με την κεντρική μνήμη είναι cached, δηλαδή η πρόσβαση σε αυτήν επιταχύνεται από την constant cache, και επιτρέπει στα threads που εκτελούνται στη GPU μόνο να διαβάζουν (read-only) από αυτή. Επομένως, ενδείκνυται η αποθήκευση σταθερών ποσοτήτων στη μνήμη αυτή. Η αποθήκευση των ποσοτήτων αυτών από τη στιγμή που τα threads της GPU δεν μπορούν να γράψουν σε αυτή, γίνεται από τη CPU, αντιγράφοντας δεδομένα από την κεντρική μνήμη του υπολογιστή στην constant μνήμη της GPU. Υπενθυμίζεται ότι η διάσταση της constant μνήμης είναι 64 KB και εκείνη της constant cache 8 KB ανεξάρτητα της υπολογιστικής δυνατότητας της GPU.

Η ανάγνωση δεδομένων που είναι προσωρινά αποθηκευμένα στην constant cache (cache hit) γίνεται πρακτικά ακαριαία, αφού ο χρόνος προσπέλασης μίας cache μνήμης είναι πολύ μικρός. Αν τα δεδομένα δεν βρίσκονται στην constant cache (cache miss), τότε εκείνα μεταφέρονται από την constant μνήμη στην constant cache (παίρνοντας τη θέση κάποιων άλλων) και διαβάζονται από τα

threads μέσω της constant cache. Δηλαδή, πρακτικά, ο χρόνος ανάγνωσης της τιμής μίας σταθεράς που δεν βρίσκεται στην constant cache είναι ίδιος με το χρόνο προσπέλασης της κεντρικής μνήμης.

Για να επιτευχθεί η μέγιστη ταχύτητα προσπέλασης της constant μνήμης, πρέπει τα threads του ίδιου warp (για GPUs 2.x) να διαβάζουν την ίδια θέση της constant μνήμης. Σε αντίθετη περίπτωση, το σύνολο των αναγνώσεων εντός του warp πραγματοποιείται σειριακά σε ομάδες ανάγνωσης ίδιων θέσεων της constant μνήμης.

Συνοψίζοντας, ο χρόνος ανάγνωσης δεδομένων από την constant μνήμη είναι πολύ μικρός όταν τα ζητούμενα δεδομένα βρίσκονται στην constant cache. Αντίθετα, είναι μεγαλύτερος ή ίσος του χρόνου προσπέλασης της κεντρικής μνήμης ανάλογα με το πλήθος των διαφορετικών θέσεων της constant μνήμης που διαβάζουν τα threads του ίδιου warp. Περισσότερες θέσεις μνήμης προς ανάγνωση σημαίνει μεγαλύτερο χρόνο ανάγνωσης. Σημειώνεται ότι, αν όλα τα warps που εκτελούνται στον ίδιο πολυεπεξεργαστή διαβάζουν την ίδια θέση της constant μνήμης, τότε μόνο το πρώτο warp θα διαβάσει την τιμή της ζητούμενης σταθεράς από την constant μνήμη (αργή ανάγνωση) και όλα τα υπόλοιπα από την constant cache (γρήγορη ανάγνωση).

3.5.3 Texture μνήμη

Η texture μνήμη είναι ακόμα ένα είδος μνήμης μονο για ανάγνωση, που μπορεί να αυξήσει την απόδοση όταν οι αναγνώσεις στη μνήμη ακολουθούν συγκεκριμένα μοτίβα. Ειδικότερα όταν τα μοτίβα αυτά επιδεικνύουν χωρική εγγύτητα (spatial locality). Παρά το ότι αρχικά είχε σχεδιαστεί για παραδοσιακές εφαρμογές γραφικών, μπορεί να χρησιμοποιηθεί και σε υπολογιστικές εφαρμογές με εξαιρετικά αποτελέσματα.

Η texture μνήμη δεν καταλαμβάνει συγκεκριμένο χώρο εσωτερικά της GPU. Αντίθετα, ο προγραμματιστής ορίζει δυναμικά το χώρο αυτής. Πρακτικά ο προγραμματιστής επιλέγει τμήματα της κεντρικής μνήμης της GPU που έχουν ήδη δεσμευτεί για την αποθήκευση δεδομένων. Τα τμήματα αυτά της κεντρικής μνήμης ονομάζονται textures και ορίζουν την texture μνήμη. Σημειώνεται ότι από τη στιγμή που έχει οριστεί ένα τμήμα της κεντρικής μνήμης ως texture, το τμήμα συνεχίζει να είναι προσπελάσιμο ως κομμάτι της κεντρικής μνήμης από τα threads της GPU, και ισχύουν όσα αναλύθηκαν στην παράγραφο 3.5.1.

Η προσπέλαση των textures γίνεται μέσω ορισμένων ειδικών συναρτήσεων, τις texture fetches. Η θέση ενός texture στην κεντρική μνήμη καθορίζεται από μεταβλητές τύπου texture, τις texture references, οι οποίες ορίζονται κατά την

επιλογή των τμημάτων της κεντρικής μνήμης που θα αποτελούν την texture μνήμη. Περισσότερες από μία texture references μπορούν να δείχνουν στο ίδιο texture. Επίσης, επιτρέπεται ένα texture να περιέχει κοινές θέσεις μνήμης με ένα άλλο texture.

Τονίζεται ότι οι texture fetches επιτρέπουν μόνο την ανάγνωση των δεδομένων που είναι αποθηκευμένα στα textures. Όπως αναφέρθηκε προηγουμένως όμως, οι θέσεις μνήμης ενός texture μπορούν να προσπελαστούν ως θέσεις της κεντρικής μνήμης. Συνεπώς τα threads που εκτελούνται σε μία GPU μπορούν να ανανεώνουν θέσεις της κεντρικής μνήμης. Αν οι ίδιες θέσεις αποτελούν τμήμα ενός texture, τότε οι ανανεωμένες τιμές μπορούν να διαβαστούν μέσω των texture fetches.

Πλεονεκτήματα χρήσης της texture μνήμης

Η ανάγνωση δεδομένων από την texture μνήμη επιταχύνεται από την texture cache. Σημειώνεται ότι, όπως συμβαίνει με όλες τις cached μνήμες έτσι και με την texture, όταν ζητείται η ανάγνωση ενός δεδομένου το οποίο δεν βρίσκεται στην cache (στην texture cache στην περίπτωση της texture μνήμης), τότε εκείνο μεταφέρεται από την κεντρική μνήμη στην cache και η ανάγνωση γίνεται από την cache μνήμη. Δηλαδή, η ανάγνωση μίας θέσης της texture μνήμης (δηλαδή μίας θέσης ενός texture) από ένα thread, γίνεται είτε μέσω της texture cache (γρήγορη ανάγνωση, cache hit), είτε από το texture (αργή ανάγνωση, cache miss). Η διάσταση της texture cache εξαρτάται από το μοντέλο της κάρτας γραφικών και είναι 6 με 8 KB ανά πολυεπεξεργαστή.

Στην ουσία, ο προγραμματιστής ορίζει μία cache μνήμη, την texture cache, να μεσολαβεί στην ανάγνωση ενός τμήματος της κεντρικής μνήμης. Το ποιο θα είναι αυτό το τμήμα της κεντρικής μνήμης αποτελεί επιλογή του προγραμματιστή. Σημειώνεται ότι στις GPUs 2.x η κεντρική μνήμη είναι cached. Συνεπώς, η αυθαίρετη επιλογή των τμημάτων της κεντρικής μνήμης που θα σχηματίσουν τα textures μπορεί να προκαλέσει ελάττωση (αντί της επιδιωκόμενης αύξησης) της παράλληλης απόδοσης ενός GPU-κώδικα. Αντίθετα, η προσεκτική επιλογή, μπορεί να αυξήσει κατακόρυφα την παράλληλη απόδοση ενός GPU-κώδικα. Προτείνεται η χρήση της texture μνήμης για την ανάγνωση δεδομένων που δεν ανανεώνονται συχνά κατά τη διάρκεια της αριθμητικής επίλυσης ενός προβλήματος και που η προσπέλαση αυτών στην κεντρική ή την constant μνήμη δεν τηρεί τα αντίστοιχα βέλτιστα πρότυπα πρόσβασης.

3.5.4 Shared μνήμη

Η shared μνήμη είναι μνήμη ταχείας προσπέλασης και επιτρέπει την επικοινωνία μεταξύ των threads του ίδιου block. Στις GPUs 2.x ο προγραμματιστής επιλέγει ανάμεσα σε 48 KB shared μνήμης και 16 KB L1 cache ανά πολυεπεξεργαστή, ή αντίστροφα. Η ταχύτητα προσπέλασης της shared μνήμης είναι περίπου ίδια με εκείνη μίας cache μνήμης.

3.5.5 Τοπική μνήμη (local memory)

Σε κάθε thread αντιστοιχεί ένα τμήμα της κεντρικής μνήμης, η λεγόμενη τοπική μνήμη (local memory). Ως τμήμα της κεντρικής μνήμης, η τοπική χαρακτηρίζεται από υψηλούς χρόνους προσπέλασης ενώ χρησιμοποιείται κυρίως από την ίδια την κάρτα γραφικών όταν οι απαιτήσεις ενός kernel σε registers υπερβαίνουν τους διαθέσιμους ανά πολυεπεξεργαστή. Είναι δομημένη με τέτοιο τρόπο ώστε, σε περίπτωση χρήσης της από τα threads του ίδιου warp, εκείνα να διαβάζουν/αποθηκεύουν σε θέσεις του ίδιου τμήματος μνήμης, όπως δηλαδή υπαγορεύει το βέλτιστο πρότυπο προσπέλασης της κεντρικής μνήμης. Έτσι επιτυγχάνεται ο ελάχιστος δυνατός χρόνος προσπέλασης. Στις GPUs 2.x η πρόσβαση στην τοπική μνήμη επιταχύνεται από τις L1, L2 cache. Η διάσταση της τοπικής ανά thread μνήμης είναι 512 KB για GPUs 2.x.

3.6 Συνεργασία CPU-GPU

Έχει ήδη αναφερθεί ότι η κλήση ενός kernel από τη CPU είναι ασύγχρονη. Δηλαδή, ο έλεγχος επιστρέφει στη CPU πριν την ολοκλήρωση της εκτέλεσης του kernel. Αυτό πρακτικά σημαίνει ότι η CPU μπορεί να εκτελεί τμήμα ενός κώδικα χαμηλού ή ακόμα και μηδενικού βαθμού παραλληλίας, επεξεργαζόμενη δεδομένα που βρίσκονται στην κεντρική μνήμη του υπολογιστή, ταυτόχρονα με την εκτέλεση ενός kernel στη GPU. Έτσι επιτυγχάνεται η αξιοποίηση όλων των διαθέσιμων υπολογιστικών πόρων και ως εκ τούτου, αυξάνεται η παράλληλη απόδοση του κώδικα.

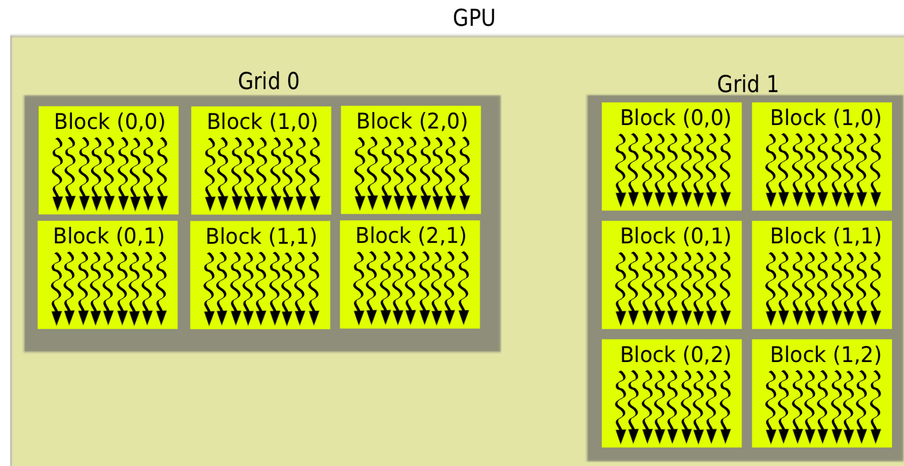
3.7 Ασύγχρονη επεξεργασία δεδομένων σε μία GPU

Το περιβάλλον της CUDA επιτρέπει την ομαδοποίηση των εργασιών που εκτελεί μία GPU σε streams. Στην ουσία, κάθε stream αποτελείται από μία σειρά από εντολές προς τη GPU, οι οποίες εκτελούνται η μία μετά την άλλη. Εφόσον, εντολές που ανήκουν σε διαφορετικά streams εκτελούνται ασύγχρονα, είναι δυνατή η ταυτόχρονη αντιγραφή δεδομένων από την κεντρική μνήμη του υπολογιστή σε εκείνη της GPU (ή αντίστροφα), με την εκτέλεση ενός kernel στη GPU ή η εκτέλεση δύο ή περισσότερων kernels στην ίδια GPU ταυτόχρονα. Για να γίνει αυτό πρέπει οι προς εκτέλεση από τη GPU εντολές να ανήκουν σε διαφορετικά streams. Σημειώνεται ότι για την ανάθεση της αντιγραφής δεδομένων από την κεντρική μνήμη του υπολογιστή σε εκείνη της GPU, ή αντίστροφα, πρέπει οι θέσεις της κεντρικής μνήμης του υπολογιστή να είναι pinned[1]. Επιπλέον, μόνο ορισμένες GPUs 2.x επιτρέπουν την ταυτόχρονη εκτέλεση διαφορετικών kernels (επιτρέπεται η ταυτόχρονη εκτέλεση έως και 16 kernels). Στις περιπτώσεις αυτές, κάθε kernel εκτελείται σε διαφορετικό grid από threads. Στο σχήμα 3.4 φαίνεται ο σχηματισμός δύο grids (Grid0, Grid1) στα οποία εκτελούνται παράλληλα δύο ανεξάρτητα kernels στην ίδια κάρτα γραφικών. Ο αριθμός των threads ανά block και εκείνος των blocks στο grid μπορεί να διαφέρουν ανάμεσα στα grids. Προφανώς, απαραίτητη προϋπόθεση για την παράλληλη εκτέλεση περισσότερων του ενός kernels στην ίδια κάρτα γραφικών είναι να επαρκούν οι διαθέσιμοι πολυεπεξεργαστές της κάρτας.

3.8 Atomic functions

Οι σημερινές GPUs αποτελούν μονάδες παράλληλης επεξεργασίας κοινής μνήμης (shared memory). Η χρήση κοινής μνήμης από τα threads της GPU πρέπει να λαμβάνεται σοβαρά υπόψη από τον προγραμματιστή, ώστε εκείνος να αποτρέπει την ταυτόχρονη ανάθεση τιμών από διαφορετικά, ταυτόχρονα εκτελούμενα, threads στην ίδια θέση μνήμης. Στην περίπτωση που ο προγραμματιστής δεν μπορεί να εγγυηθεί το παραπάνω, το περιβάλλον προγραμματισμού της CUDA περιέχει μία σειρά από ειδικές συναρτήσεις (atomic functions) που εξασφαλίζουν την ανανέωση μίας θέσης της κεντρικής ή της shared μνήμης από ένα

[1]Pinned ή Page-locked μνήμη ονομάζεται η μνήμη του υπολογιστή η οποία δεσμεύεται και μπορεί να είναι απευθείας προσβάσιμη από την GPU με την χαρακτηριστική ιδιότητα ότι απαγορεύει στον υπολογιστή να τη μετακινήσει στο σκληρό δίσκο (paging ή swap) σε περίπτωση που αυτή γεμίσει. Εξ' ου και το pinned (=καρφιτσωμένη).



Σχήμα 3.4: Ταυτόχρονη εκτέλεση δύο kernels σε μία GPU. Κάθε kernel έχει τη δική του κατανομή threads στα blocks και των blocks στο grid. Μόνο ορισμένες GPUs 2.x υποστηρίζουν την ταυτόχρονη εκτέλεση δύο ή περισσότερων kernels. Βασική προϋπόθεση είναι να επαρκούν οι πολυεπεξεργαστές της κάρτας γραφικών.

thread, αποτρέποντας την ταυτόχρονη ανανέωση της ίδιας θέσης μνήμης από ένα άλλο thread που πιθανώς εκτελείται την ίδια στιγμή. Οι συναρτήσεις αυτές μπορούν να χειρίζονται ακέραιους, πραγματικούς αριθμούς απλής ακρίβειας και μόλις στην 4η έκδοση της CUDA και πραγματικούς αριθμούς διπλής ακρίβειας. Φυσικά, η χρήση των συναρτήσεων αυτών πρέπει να γίνεται με σύνεση καθώς συγχρονίζουν τα threads που εκτελούνται ταυτόχρονα, μειώνοντας, ως εκ τούτου, την απόδοση του GPU-κώδικα.

3.9 Προγραμματισμός με CUDA

Για να γίνει πιο εύκολα κατανοητός ο προγραμματισμός στο περιβάλλον της CUDA δίνεται ως παράδειγμα ([18]) ο κώδικας μιας εφαρμογής που υπολογίζει το εσωτερικό γινόμενο δύο διανυσμάτων. Η σχέση (3.1) δίνει το εσωτερικό γινόμενο δύο διανυσμάτων $\vec{A}$, $\vec{B}$ διάστασης N:

$$\vec{A} \cdot \vec{B} = \sum_{j=0}^{N} A_j B_j \tag{3.1}$$

Λαμβάνοντας υπ΄ όψη πως σε μία GPU πολλά threads οργανωμένα σε blocks εκτελούνται παράλληλα, μπορεί κάθε thread να αναλάβει ένα τμήμα του υπολογισμού. Αν threadsPerBlock είναι ο αριθμός των threads ανά block και blocksPerGrid ο αριθμός των blocks στο grid, τότε εκτελούνται παράλληλα $\mathsf{nbThreads} = \mathsf{threadsPerBlock} \cdot \mathsf{blocksPerGrid}$ threads. Έτσι ο υπολογισμός του εσωτερικού γινομένου γίνεται ως:

$$\vec{A} \cdot \vec{B} = \sum_{thr=0}^{nbThreads} S_{thr} \tag{3.2}$$

με S_{thr} το επιμέρους κομμάτι του εσωτερικού γινομένου που υπολογίζεται από κάθε thread. Δηλαδή, το thread i υπολογίζει:

$$S_{thr} = \sum_{k=0}^{m} A_{i+k \cdot nthr} B_{i+k \cdot nthr} \tag{3.3}$$

όπου το m εξαρτάται από τη διάσταση N των διανυσμάτων. Ουσιαστικά, πρέπει το m να είναι τέτοιο ώστε $i + m \cdot \mathsf{threadsPerBlock} \leq N$ δηλαδή:

$$m = floor\left(\frac{N - i}{\mathsf{threadsPerBlock}}\right) \tag{3.4}$$

Στην πράξη, όπως θα φανεί παρακάτω, η σχέση (3.3) υλοποιείται με έναν επαναληπτικό βρόχο `while( )` οπότε δεν χρειάζεται να καθοριστεί το m.

Το S_{thr} από κάθε thread αποθηκεύεται στη shared μνήμη του block που ανήκει το thread. Τα threads του ίδιου block (όπως θα αναλυθεί στη συνέχεια) συνεργάζονται στην άθροιση των S_{thr} του block τους. Το αποτέλεσμα αυτής της άθροισης (S_{block}) αποθηκεύεται στη global μνήμη. Η CPU αναλαμβάνει το άθροισμα των S_{block} ώστε να προκύψει το τελικό αποτέλεσμα S:

$$S = \sum S_{block} \tag{3.5}$$

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  // Handle NULL assignments (malloc, new, etc)
5  #define HANDLE_NULL(call) \
6      do { \
7          if ((call) == NULL) { \
8              fprintf(stderr, "Resource allocation error in %s " \
9                      "at line %d\n", __FILE__, __LINE__); \
10             exit(EXIT_FAILURE); \
11         } \
12     } while(0)
13
14 #define imin(a,b) (a<b?a:b)
15 #define sum_squares( x ) ( x*(x+1)*(2*x+1)/6 )
16
17 static void HandleError(cudaError_t err, const char *file,
18                         int line);
19
20 // Handle GPU errors
21 static void HandleError(cudaError_t err, const char *file,
22                         int line)
23 {
24     if (err != cudaSuccess)
25     {
26         fprintf(stderr, "%s in %s at line %d\n",
27                 cudaGetErrorString( err ), file, line);
28         exit(EXIT_FAILURE);
29     }
30 }
31 #define HANDLE_ERROR(err) (HandleError(err, __FILE__, __LINE__))
32
33 const int N = 33 * 1024 * 1024;
34 const int threadsPerBlock = 128;
35 const int blocksPerGrid = imin(32,
36           (N+threadsPerBlock-1)/threadsPerBlock);
37
38 __global__ void dot(double *a, double *b, double *c);
39
40 // ========================================================
41 __global__ void dot(double *a, double *b, double *c)
42 // ========================================================
43 {
44     __shared__ double cache[threadsPerBlock];
45     int tid = threadIdx.x + blockIdx.x * blockDim.x;
46     int cacheIndex = threadIdx.x;
47
48     double   temp = 0.0;
49     while (tid < N)
50     {
51         temp += a[tid] * b[tid];
52         tid  += blockDim.x * gridDim.x;
53     }
54     // set the cache values
55     cache[cacheIndex] = temp;
56
57     // synchronize threads in this block
58     __syncthreads();
```

```
59
60      // for reductions, threadsPerBlock must be a power of 2
61      // because of the following code
62      int i = blockDim.x/2;
63      while (i != 0)
64      {
65          if (cacheIndex < i) cache[cacheIndex] += cache[cacheIndex + i];
66          __syncthreads();
67
68          i /= 2;
69      }
70      if (cacheIndex == 0) c[blockIdx.x] = cache[0];
71  }
72  // ========================================================
73  int main( )
74  // ========================================================
75  {
76      double *a, *b, c, *partial_c;
77      double *dev_a, *dev_b, *dev_partial_c;
78
79      // allocate memory on the cpu side
80      int isize = N * sizeof(double);
81      HANDLE_NULL(a          = (double *) malloc(isize));
82      HANDLE_NULL(b          = (double *) malloc(isize));
83      isize      = blocksPerGrid * sizeof(double);
84      HANDLE_NULL(partial_c = (double *) malloc(isize));
85
86      // allocate memory on the GPU
87      isize = N * sizeof(double);
88      HANDLE_ERROR(cudaMalloc(&dev_a, isize));
89      HANDLE_ERROR(cudaMalloc(&dev_b, isize));
90
91      isize = blocksPerGrid * sizeof(double);
92      HANDLE_ERROR(cudaMalloc(&dev_partial_c, isize));
93
94      // fill in the host memory with data
95      for (int i = 0; i < N; i++) { a[i] = i; b[i] = i*2; }
96
97      // copy the arrays 'a' and 'b' to the GPU
98      isize = N * sizeof(double);
99
100     cudaMemcpyKind kind = cudaMemcpyHostToDevice;
101     HANDLE_ERROR(cudaMemcpy(dev_a, a, isize, kind));
102     HANDLE_ERROR(cudaMemcpy(dev_b, b, isize, kind));
103
104     dot<<<blocksPerGrid,threadsPerBlock>>>( dev_a, dev_b,
105                                             dev_partial_c );
106
107     // copy the array 'c' back from the GPU to the CPU
108     kind  = cudaMemcpyDeviceToHost;
109     isize = blocksPerGrid * sizeof(double);
110
111     HANDLE_ERROR(cudaMemcpy(    partial_c,
112                             dev_partial_c, isize, kind));
113
114     // finish up on the CPU side
115     c = 0.0;
116     for (int i = 0; i < blocksPerGrid; i++) c += partial_c[i];
117
118     printf( "Does GPU value %.6g = %.6g?\n", c,
119             2 * sum_squares( (double)(N - 1) ) );
120
```

```
121     // free memory on the gpu side
122     HANDLE_ERROR(cudaFree(dev_a));
123     HANDLE_ERROR(cudaFree(dev_b));
124     HANDLE_ERROR(cudaFree(dev_partial_c));
125
126     // free memory on the cpu side
127     free(a);
128     free(b);
129     free(partial_c);
130
131     return 0;
132 }
```

Κώδικας 3.1: (dot–1gpu.cu) Παράδειγμα κώδικα CUDA που υπολογίζει παράλληλα το εσωτερικό γινόμενο 2 διανυσμάτων.

Αναλυτικότερη περιγραφή του κώδικα του παραδείγματος

Η κάρτα γραφικών περιορίζει τον αριθμό των blocks που μπορούν να χρησιμοποιηθούν σε 65535. Παρόμοιος περιορισμός υπάρχει και για τον αριθμό των threads ανά block που για τις κάρτες σημερινής τεχνολογίας είναι 512. Για να είναι δυνατός ο υπολογισμός του εσωτερικού γινομένου δύο διανυσμάτων ($\vec{A}$, $\vec{B}$) με διάσταση N μεγαλύτερη από $65535 \cdot 512$, όπως συμβαίνει και στο παράδειγμα (γραμμή 33), κάθε thread δεν υπολογίζει μόνο το γινόμενο δύο συνιστωσών (A_k, B_k) αλλά ένα επιμέρους άθροισμα (S_{thr}, σχέση 3.3).

Επιλέγεται 1Δ κατανομή των threads στα blocks και των blocks στο grid. Συγκεκριμένα ο αριθμός των threads ανά block ορίζεται ίσος με 128 (μεταβλητή threadsPerBlock, γραμμή 36) και ο αριθμός των blocks στο grid είναι η μικρότερη τιμή ανάμεσα στο 32 και το πηλίκο της ευκλείδειας διαίρεσης της διάστασης $N + \text{threadsPerBlock} - 1$ και του αριθμού των threads ανά block (γραμμές 35-36). Γίνεται αντιληπτό ότι στην περίπτωση που ο αριθμός των blocks στο grid ισούται με τη δεύτερη τιμή, τότε κάθε thread υπολογίζει το γινόμενο μόνο δύο συνιστωσών (δηλαδή $S_{thr} = A_k \cdot B_k$), που όμως δεν είναι η περίπτωση του παραδείγματος εξαιτίας της μεγάλης τιμής της διάστασης N που έχει επιλεχθεί. Έχει προστεθεί όμως στον κώδικα και αυτή η επιλογή για λόγους γενίκευσης.

Στις γραμμές 81-84 δεσμεύονται θέσεις μνήμης του υπολογιστή για τα διανύσματα $\vec{A}$, $\vec{B}$ και τα μερικά αθροίσματα S_{block}. Η δέσμευση των θέσεων της κεντρικής μνήμης της GPU για τους αντίστοιχους πίνακες γίνεται στις γραμμές 88-92. Παρατηρείται ότι οι ίδιες ποσότητες είναι απαραίτητο να υπάρχουν τόσο στη μνήμη του υπολογιστή όσο και στην κεντρική της GPU. Τα διανύσματα $\vec{A}$, $\vec{B}$ αρχικοποιούνται στη CPU στη γραμμή 95 και αντιγράφονται από τη μνήμη του υπολογιστή στην κεντρική της GPU (γραμμές 101-102).

Η δήλωση του αριθμού των threads ανά block και των blocks στο grid ξεχωρίζει στην κλήση του kernel υπολογισμού του εσωτερικού γινομένου (γραμμή 104). Η πληροφορία αυτή δίνεται στη GPU εντός `dot<<< ... >>>( )`. Επισημαίνεται ότι το kernel παίρνει ως ορίσματα τις θέσεις μνήμης των αποθηκευμένων στη GPU πινάκων. Η εκτέλεση του kernel γίνεται ασύγχρονα ως προς τη CPU η οποία συνεχίζει εκτελώντας τις υπόλοιπες εντολές του αλγορίθμου.

Οι συνιστώσες των διανυσμάτων που επεξεργάζεται ένα thread ορίζονται με βάση τον αύξοντα αριθμό του thread στο block και τον αύξοντα αριθμό του block στο grid (γραμμές 45, 52). Υπενθυμίζεται ότι έχει επιλεχθεί 1Δ κατανομή τόσο των threads στα blocks, όσο και των blocks στο grid. Για τον λόγο αυτό στις γραμμές 45 και 52 εμφανίζεται μόνο το επίθεμα `.x` στις μεταβλητές `threadIdx`, `BlockIdx` και `BlockDim`. Η τελευταία αποτελεί τη διάσταση του κάθε block μετρούμενη σε threads. Το σχήμα 3.5 απεικονίζει την κατανομή των threads στα blocks.

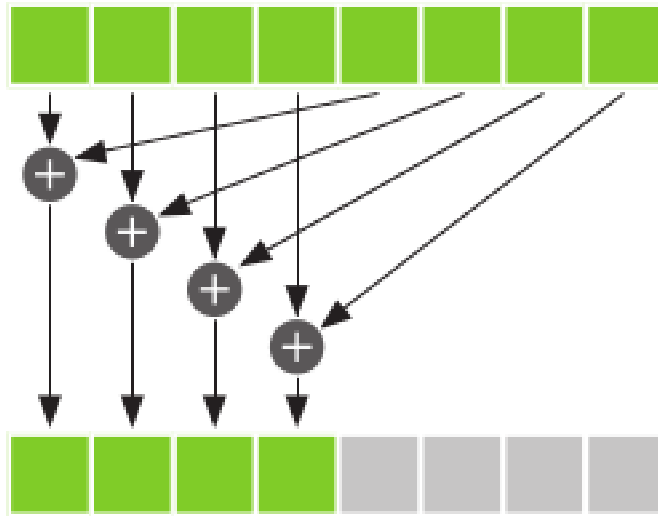
Block 0	Thread 0	Thread 1	Thread 2	Thread 3
Block 1	Thread 0	Thread 1	Thread 2	Thread 3
Block 2	Thread 0	Thread 1	Thread 2	Thread 3
Block 3	Thread 0	Thread 1	Thread 2	Thread 3

Σχήμα 3.5: 1Δ κατανομή των blocks στο grid και των threads στα blocks, [18]

Ιδιαίτερη έμφαση δίνεται στη γραμμή 49 του κώδικα, όπου γίνεται έλεγχος για το αν ο δείκτης της προς επεξεργασία συνιστώσας (tid) είναι μικρότερος της διάστασης N. Έτσι, όταν ο δείκτης `tid` υπερβεί τη διάσταση των διανυσμάτων δεν γίνεται ο υπολογισμός και αποφεύγεται η πρόσβαση σε μη δεσμευμένη θέση μνήμης (segmentation fault).

Στη γραμμή 44 δεσμεύεται στη shared memory ένας πίνακας (`cache[ ]`) διάστασης `threadsPerBlock` για την αποθήκευση των S_{thr} ανά thread του block. Η κλήση της `__syncthreads()` (γραμμή 58) συγχρονίζει τα threads του block και εξασφαλίζει ότι έχουν υπολογίσει τα αντίστοιχα S_{thr}. Τα τελευταία αθροίζονται στις γραμμές 62-69.

Η άθροιση των S_{thr} στηρίζεται στη γενική ιδέα ότι κάθε thread προσθέτει δύο τιμές του πίνακα `cache[ ]` και αποθηκεύει το αποτέλεσμα της άθροισης σε μια από αυτές. Με την ολοκλήρωση αυτού του βήματος έχουν ανανεωθεί οι τιμές των μισών θέσεων και έχουν αθροιστεί τα μισά S_{thr}. Το βήμα αυτό επαναλαμβάνεται `log2(threadsPerBlock)` φορές, έως ότου προκύψει το συνολικό άθροισμα των στοιχείων του πίνακα `cache[ ]`. Στο παράδειγμα, για 128 threads ανά block η άθροιση των S_{thr} του block, όπως περιγράφηκε παραπάνω, πραγματοποιείται σε $log_2(128) = 7$ βήματα. Το τελικό άθροισμα (S_{block}) βρίσκεται στην πρώτη θέση του πίνακα `cache[ ]` και αποθηκεύεται στην κεντρική μνήμη από το thread 0 του block (πίνακας `c[ ]`, γραμμή 70). Η διαδικασία άθροισης των S_{thr} φαίνεται γραφικά στο σχήμα 3.6.



Σχήμα 3.6: Βήμα μείωσης (reduction) αθροισμάτων [18]

Για την άθροιση των S_{block} στη GPU απαιτείται μια ανάλογη διαδικασία. Είναι παράλογο, όμως, να χρησιμοποιούνται 480 αριθμητικές μονάδες για την άθροιση μόλις 32 αριθμών. Οπότε, επιστρέφεται ο έλεγχος στην CPU και της ανατίθεται να ολοκληρώσει την άθροιση των 32 στοιχείων του πίνακα `c[ ]`. Οι τιμές αυτών αντιγράφονται από την κεντρική μνήμη της GPU στη μνήμη του υπολογιστή (γραμμή 111) και αθροίζονται στη γραμμή 116. Σημειώνεται ότι η αντιγραφή δεδομένων από την κεντρική μνήμη της GPU στη μνήμη του υπολογιστή συγχρονίζει τη GPU με τη CPU. Έτσι εξασφαλίζεται η ολοκλήρωση του kernel `dot( )` πριν την έναρξη της αντιγραφής.

Η αποδέσμευση των θέσεων μνήμης (GPU και CPU) γίνεται στις γραμμές 122-129.

Κεφάλαιο 4

Προγραμματισμός με νήματα (POSIX threads)

4.1 Εισαγωγή

Σε πολυεπεξεργαστικές αρχιτεκτονικές κοινής μνήμης (shared memory architectures - SMP **S**ymmetric **M**ulti**P**rocessors) τα νήματα χρησιμοποιούνται για να υλοποιηθεί η παραλληλία. Το συγκεκριμένο κεφάλαιο θα μπορούσε να σταθεί αυτούσιο έξω από οποιαδήποτε συζήτηση για CUDA, OpenCL, ή υπολογισμούς σε κάρτες γραφικών. Το αντικείμενο που πραγματεύεται είναι ένα μοντέλο ασύγχρονου/παράλληλου προγραμματισμού το οποίο έχει εισαχθεί εδώ και παραπάνω από μια δεκαετία, και κυριάρχησε όταν άρχισαν να γίνονται εμπορικά διαθέσιμοι οι πρώτοι πολυπύρηνοι επεξεργαστές (multi-core CPUs). Με τα νήματα (threads) έγινε δυνατή η υλοποίηση πολυνηματικών (multithreaded) εφαρμογών που εκμεταλλεύονται για τις υπολογιστικές τους απαιτήσεις όλους τους διαθέσιμους πυρήνες μιας CPU. Εκτενής κάλυψη του πολυνηματικού προγραμματισμού γίνεται στα [21], [22].

Η ιδέα (προγραμματίζοντας με νήματα) είναι να αξιοποιούνται, στο πλαίσιο μίας μόνο διεργασίας, όλοι οι διαθέσιμοι επεξεργαστές (ή πυρήνες) που μοιράζονται την ίδια μνήμη (RAM), χωρίς να χρειάζεται η δημιουργία (`fork( )`) νέων διεργασιών με το κόστος (overhead) και επικοινωνία (inter-process communication) που συνεπάγεται. Κάτι παρόμοιο μπορεί να γίνει και με το OpenMP (Open Multiprocessing) που όμως λειτουργεί εντελώς διαφορετικά, σε υψηλότερο επίπεδο, κρύβοντας από τον προγραμματιστή πολλές λεπτομέρειες και στερώντας του σημαντικές επιλογές. Βέβαια, ενώ η μετατροπή ενός κώδικα με POSIX threads απαιτεί αφιέρωση και αρκετές εργατοώρες, η ίδια μετατροπή

με OpenMP κοστίζει μόνο λίγες παραπάνω γραμμές κώδικα. Γι΄ αυτό το λόγο, τα POSIX threads πολύ συχνά χαρακτηρίζονται ως η assembly γλώσσα του παράλληλου προγραμματισμού.

Εδώ, μπορεί επίσης να επισημανθεί ότι το MPI (Message Passing Interface) δεν έχει σχέση με τα νήματα. Το MPI είναι πρωτόκολλο μετάδοσης μηνυμάτων μεταξύ διαφορετικών κόμβων (από μητρική σε μητρική κάρτα) και διαπρέπει σε κατανεμημένες-ετερογενείς (distributed) αρχιτεκτονικές. Μπορεί, καταχρηστικά, να χρησιμοποιηθεί και σε ένα μόνο κόμβο, όμως σ΄ αυτή την περίπτωση λειτουργεί με τη λογική `fork( )`, δηλαδή φτιάχνει τόσες διεργασίες όσοι και οι πυρήνες και η απόδοσή του είναι πολύ χαμηλότερη από αυτή των νημάτων.

4.2 Χρήσιμες έννοιες

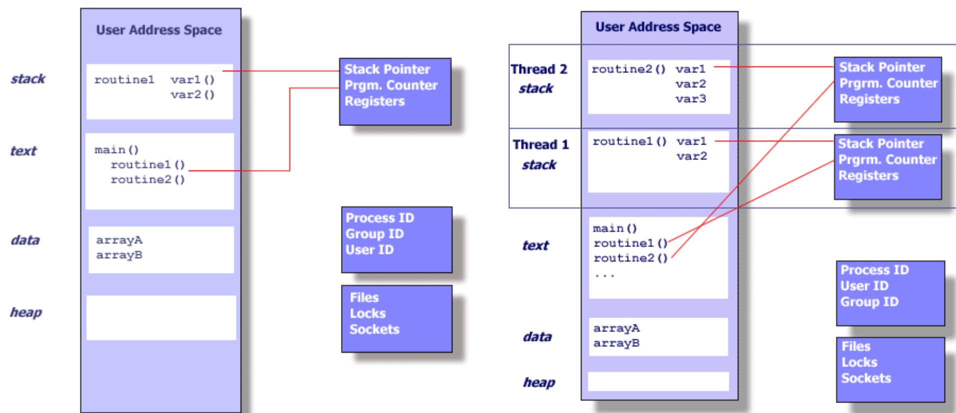
4.2.1 POSIX

Είναι το ακρωνύμιο Portable Operating System Interface και αφορά μια οικογένεια προτύπων, καθορισμένων από την IEEE, προκειμένου να τηρείται (όσο αυτό είναι δυνατό) μια συμβατότητα μεταξύ λειτουργικών συστημάτων. Το POSIX ορίζει το Application Programming Interface (API), τα κελύφη (command line shells) και τα απαραίτητα εργαλεία για επίτευξη συμβατότητας σε επίπεδο λογισμικού μεταξύ των κλασικών παραλλαγών του UNIX και των υπολοίπων λειτουργικών συστημάτων. Εν ολίγοις, είναι η «ετικέτα» που σημαδεύει τι είναι συμβατό με το κλασικό UNIX. Για παράδειγμα, το παραδοσιακό UNIX και όλοι οι απόγονοί του που φτάνουν ως τις μέρες μας (FreeBSD, Mac OS X, Solaris, GNU/Linux) συμμορφώνονται πλήρως με τα πρότυπα POSIX. Παραδείγματα όσων δεν συμμορφώνονται ή που για να συμμορφωθούν χρειάζονται πρόσθετα συμβατότητας (compatibility features) είναι το Symbian OS και τα Windows NT/2000/XP/2003/Vista/7.

4.2.2 Νήμα (thread)

Προκειμένου να μη δημιουργηθεί σύγχυση με τη χρήση του όρου thread, τονίζεται εξαρχής ότι στο παρόν κεφάλαιο ο όρος thread δεν έχει την ίδια σημασία που έχει στην ορολογία των καρτών γραφικών και της CUDA. Υπάρχουν όμως ομοιότητες μεταξύ των δύο που θα φανούν στον ορισμό.

Πριν γίνει κατανοητό τί είναι νήμα πρέπει να γίνει αναδρομή στην διεργασία UNIX. Οι διεργασίες περιέχουν πληροφορία για τους πόρους του εκτελέσιμου προγράμματος και την κατάσταση εκτέλεσης και συμπεριλαμβάνουν:



(α΄) Μια τυπική διεργασία UNIX (β΄) Μια διεργασία UNIX με νήματα

Σχήμα 4.1: Σχέση διεργασιών-νημάτων

- Ταυτότητα της διεργασίας και του χρήστη που την ξεκίνησε (Process ID, user/group ID)
- Εντολές του προγράμματος
- Καταχωρητές (Registers)
- Στοίβα μνήμης (stack)
- Σωρός μνήμης (heap)
- Περιγραφείς αρχείων (file descriptors)
- Σήματα (signals)
- Κοινές βιβλιοθήκες (shared libraries)
- Εργαλεία ενδοδιεργασιακής επικοινωνίας (inter-process communication)

Τα νήματα υπάρχουν και λειτουργούν μέσα στο πλαίσιο μιας διεργασίας, χρησιμοποιώντας τους πόρους της. Επιπλέον, μπορούν να εκτελεστούν ως ανεξάρτητες οντότητες, διότι χρειάζονται να αντιγράψουν μόνο τα ελάχιστα εκείνα στοιχεία της διεργασίας που τους επιτρέπουν να λειτουργούν ως εκτελέσιμος κώδικας.

Αυτή η ανεξάρτητη ροή ελέγχου επιτυγχάνεται διότι ένα νήμα διατηρεί ανεξάρτητα: δείκτη στοίβας (Stack pointer), καταχωρητές (Registers), ιδιότητες χρονοπρογραμματισμού (όπως προτεραιότητα εκτέλεσης) και σύνολο εκκρεμών ή μπλοκαρισμένων σημάτων.

Επειδή τα νήματα μιας διεργασίας μοιράζονται πόρους, αλλαγές που γίνονται από ένα νήμα πάνω σε κοινούς πόρους (όπως το κλείσιμο ενός αρχείου) αντανακλώνται και στα υπόλοιπα. Επιπλέον, δύο δείκτες που έχουν ίδια τιμή «δείχνουν» στον ίδιο χώρο μνήμης. Τέλος, η ανάγνωση κι εγγραφή στις ίδιες θέσεις μνήμης είναι δυνατή, οπότε απαιτείται να προβλεφθεί από τον προγραμματιστή ο απαραίτητος συντονισμός.

Νήμα (thread), λοιπόν, είναι μια **στοιχειώδης οντότητα λογισμικού** που μπορεί να εκτελέσει μια εργασία σε έναν υπολογιστή. Το νήμα είναι μικρότερο, γρηγορότερο και πιο εύκολα διαχειρίσιμο από μια τυπική διεργασία (η δημιουργία ενός νήματος έχει σαφώς μικρότερο κόστος-overhead από της διεργασίας). Στην πραγματικότητα, στα σύγχρονα λειτουργικά συστήματα, η έννοια της διεργασίας αντιπροσωπεύει μόνο δεδομένα - χώρο διευθύνσεων μνήμης (address space), περιγραφείς αρχείων (file descriptors) - συν ένα ή περισσότερα νήματα (threads) τα οποία καλούνται να κάνουν «κάτι» με όλα αυτά τα δεδομένα.

Η έννοια thread λοιπόν ως «πακέτο εργασίας» (work package), όπως ορίζεται εδώ, προϋπήρχε του προγραμματισμού σε κάρτες γραφικών οπότε και γίνεται φανερό από πού εμπνεύστηκαν οι αρχιτέκτονες της CUDA τη χρήση του όρου thread στις κάρτες γραφικών για να περιγράψουν αυτό που αντιπροσωπεύει κι εκεί: τη στοιχειώδη μονάδα εργασίας. Στο εξής, όταν χρειάζεται να ξεκαθαριστεί σε ποιον τύπο thread αναφέρεται το κείμενο θα χρησιμοποιείται το host-thread για τα threads του επεξεργαστή και μόνο thread για τα threads της κάρτας γραφικών. Προφανώς, στο παρόν κεφάλαιο δεν τίθεται τέτοιο θέμα διότι γίνεται αναφορά μόνο στα host-threads.

Η προγραμματιστική προσέγγιση που χρησιμοποιεί νήματα εφαρμόζεται με μεγάλη επιτυχία σε μια τεράστια ποικιλία προγραμματιστικών προβλημάτων:

- Μεγάλης κλίμακας, υπολογιστικά απαιτητικά προγράμματα.
- Εφαρμογές υψηλών επιδόσεων και κώδικας βιβλιοθηκών που μπορούν να εκμεταλλεύονται επεξεργαστές πολλαπλών πυρήνων. Για παράδειγμα, μία εφαρμογή μπορεί να αναθέτει σε ένα νήμα τον σχεδιασμό του γραφικού της περιβάλλοντος και την αλληλεπίδραση με τον χρήστη και σε ένα άλλο νήμα την κυρίως εργασία που επιτελεί.
- Εφαρμογές που εκτελούν εντατικές διεργασίες εισόδου/εξόδου (I/O) με αργές εξωτερικές «συσκευές» (δίκτυα, ανθρώπους) και επωφελούνται με την ανάθεση αυτών σε ένα νήμα ενώ ταυτόχρονα ένα ή περισσότερα άλλα νήματα συνεχίζουν τη σημαντική δουλειά της εφαρμογής.

4.2.3 POSIX threads

Το μοντέλο threads που χρησιμοποιείται ευρύτατα, αλλά και στην παρούσα εργασία, ονομάζεται POSIX threads ή συντομογραφικά pthreads. Η επίσημη ονομασία του είναι POSIX 1003.1c-1995 standard. Διεπιφάνειες για pthreads συμπεριλαμβάνονται εγγενώς στα λειτουργικά συστήματα που αναφέρθηκαν παραπάνω και συμμορφώνονται πλήρως με τα πρότυπα POSIX. Στα Windows, το Win32 API της Microsoft, δηλαδή η πρωταρχική διεπιφάνεια προγραμματισμού για εφαρμογές που τρέχουν σε Windows, υποστηρίζει ένα πολύ διαφορετικό μοντέλο threads τηρώντας όμως τις ίδιες βασικές αρχές. Ένας προγραμματιστής μυημένος στη λογική του ταυτοχρονισμού, συγχρονισμού, χρονοπρογραμματισμού (scheduling) και του παράλληλου προγραμματισμού μπορεί να προσαρμοστεί σε οποιοδήποτε μοντέλο threads υιοθετώντας διαφορετικό συντακτικό και γραμματική κάθε φορά.

Στο σημείο αυτό είναι χρήσιμο να δοθούν κάποιοι ορισμοί:

Ασυγχρονία (asynchrony) Οποιεσδήποτε δύο λειτουργίες είναι ασύγχρονες (asynchronous) όταν μπορούν να διεκπεραιωθούν ανεξάρτητα η μία από την άλλη. Μία από τις μεγαλύτερες παρεξηγήσεις του όρου που οδηγεί σε κακή χρήση είναι ότι δεν υπάρχει λόγος/ανάγκη ασυγχρονίας εκτός κι αν μπορούν να εντοπιστούν σε ένα πρόγραμμα περισσότερες από μια δραστηριότητες που μπορούν να εξελίσσονται ταυτόχρονα. Αν κάποιος μπορεί να ξεκινήσει μια ασύγχρονη λειτουργία αλλά μετά δεν μπορεί να κάνει τίποτα παρά μόνο να περιμένει την ολοκλήρωσή της, δεν κερδίζει τίποτα τελικά.

Ταυτοχρονισμός (concurrency) Βάσει της έννοιας του «ταυτόχρονου» που μπορεί να βρεθεί σε ένα λεξικό, αναφέρεται σε πράγματα που συμβαίνουν στον ίδιο χρόνο. Όμως στην ανάπτυξη λογισμικού και ειδικότερα στα νήματα χρησιμοποιείται για πράγματα που μπορεί να φαίνεται πως συμβαίνουν στον ίδιο χρόνο, όμως μπορεί να συμβαίνουν σειριακά και τόσο γρήγορα ώστε να δίνουν την εντύπωση του ταυτόχρονου. Ο ταυτοχρονισμός περιγράφει τη συμπεριφορά των νημάτων ή των διεργασιών σε συστήματα με έναν επεξεργαστή.

Μονοεπεξεργαστής (uniprocessor) Με την έννοια **μονοεπεξεργαστής** δηλώνεται ένας υπολογιστής με μία μόνο ορατή από την σκοπιά του προγραμματιστή μονάδα εκτέλεσης (processor/execution unit). Συμπεριλαμβάνει ακόμα κι έναν μεγάλων επιδόσεων διανυσματικό επεξεργαστή με ενσωματωμένους συνεπεξεργαστές (coprocessors) για μαθηματικές πράξεις ή I/O.

Πολυεπεξεργαστής (multiprocessor) Με την έννοια **πολυεπεξεργαστής** δηλώνεται ένας υπολογιστής με περισσότερους από έναν επεξεργαστές που μοιράζονται ένα κοινό σύνολο εντολών (instruction set) και πρόσβαση σε ολόκληρη την φυσική μνήμη (shared memory). Άρα, πολυεπεξεργαστής θεωρούνται 2 μονοπύρηνοι επεξεργαστές πάνω στην ίδια μητρική κάρτα που έχουν ισότιμη πρόσβαση στην ίδια μνήμη, αλλά θεωρείται επίσης κι ένας επεξεργαστής με πολλούς πυρήνες.

Παραλληλία (parallelism) περιγράφει ταυτόχρονες ακολουθίες ενεργειών που εξελίσσονται παράλληλα. Με άλλα λόγια, η παραλληλία είναι πιο κοντά στον ορισμό του «ταυτόχρονου» όπως τον δίνει ένα λεξικό κι όχι όπως χρησιμοποιείται στην ανάπτυξη λογισμικού. Αληθινή παραλληλία μπορεί να υπάρξει **μόνο σε ένα πολυεπεξεργαστικό περιβάλλον**, αλλά ταυτοχρονισμός μπορεί να υπάρξει σε μονοεπεξεργαστές και πολυεπεξεργαστές από κοινού. Ο ταυτοχρονισμός μπορεί να λειτουργεί σε μονοεπεξεργαστές επειδή είναι κατ΄ ουσία η ψευδαίσθηση της παραλληλίας.

Εδώ να σημειωθεί ότι ένα πολυνηματικό πρόγραμμα ΔΕΝ ΑΠΑΙΤΕΙ τουλάχιστον τόσους πυρήνες όσα είναι και τα νήματα που δημιουργεί προκειμένου να εκτελεστεί. Αν, για παράδειγμα, είναι γραμμένο με 4 νήματα και ο επεξεργαστής στον οποίο κληθεί να εκτελεστεί είναι 2πύρηνος ή ακόμα και μονοπύρηνος, τότε τα νήματα από μόνα τους θα «βολευτούν» με τη λογική της χρονοδιαίρεσης (time division scheduling, δηλαδή εκ περιτροπής ανάθεση στους διαθέσιμους πυρήνες). Το ίδιο θα συμβεί όταν υπάρχουν 4 νήματα και 4 πυρήνες αλλά κάποιοι απ΄ αυτούς δεν είναι διαθέσιμοι διότι χρησιμοποιούνται από άλλες διεργασίες (λ.χ του λειτουργικού συστήματος ή προγράμματα άλλων χρηστών). Είναι προφανές ότι, αν δεν συνέβαιναν τα παραπάνω, θα υπήρχε πολύ μεγάλη δυσκαμψία στον προγραμματισμό και εκτέλεση πολυνηματικών προγραμμάτων. Βέβαια, είναι καλό να διασφαλίζεται η αντιστοιχία νημάτων με διαθέσιμους πυρήνες, ειδικά όταν μετράται η επιτάχυνση ενός πολυνηματικού παράλληλου προγράμματος σε σχέση με ένα σειριακό.

Υπάρχουν επίσης και νεότερες αρχιτεκτονικές πολυπύρηνων επεξεργαστών (της Intel® προς το παρόν) που αυτήν την αντιστοιχία την πάνε ένα βήμα πιο πέρα. Υποστηρίζοντας μια τεχνολογία που λέγεται Hyper-threading™ [23], μπορούν να αναθέτουν μέχρι και 2 νήματα ανά πυρήνα, τα οποία εκτελούνται παράλληλα. Έτσι ένας π.χ. 6πύρηνος επεξεργαστής μπορεί να συμπεριφέρεται σαν 12πύρηνος όταν εκτελεί προγράμματα γραμμένα με 12 νήματα.

4.3 Λειτουργίες ελέγχου του ταυτοχρονισμού

Οποιοδήποτε ταυτόχρονο σύστημα πρέπει να παρέχει ένα βασικό σύνολο λειτουργιών που χρειάζονται για να δημιουργούν **πλαίσια ταυτόχρονης εκτέλεσης** (concurrent execution contexts) και να τα ελέγχουν κατά τη διάρκεια εκτέλεσης μιας εφαρμογής. Παρουσιάζονται οι τρεις πιο σημαντικές πτυχές ενός ταυτόχρονου συστήματος:

1. **Πλαίσιο εκτέλεσης** (execution context) είναι η κατάσταση μιας ταυτόχρονης οντότητας. Ένα ταυτόχρονο σύστημα πρέπει να παρέχει μεθόδους δημιουργίας ή καταστροφής πλαισίων εκτέλεσης και να διατηρεί τις καταστάσεις τους ανεξάρτητα. Πρέπει να είναι σε θέση να αποθηκεύει την κατάσταση ενός πλαισίου και να μεταπηδά σε άλλο σε ανύποπτο χρόνο, για παράδειγμα όταν κάτι χρειάζεται να περιμένει να συμβεί ένα εξωτερικό γεγονός. Πρέπει επίσης να μπορεί να συνεχίσει την εκτέλεση ενός πλαισίου από εκεί που το άφησε, με τα ίδια περιεχόμενα καταχωρητών (registers) σε δεύτερο χρόνο.

2. **Χρονοπρογραμματισμός** (scheduling) καθορίζει ποιό πλαίσιο (ή σύνολο πλαισίων) πρέπει να εκτελεστεί μια συγκεκριμένη χρονική στιγμή και αλλάζει μεταξύ πλαισίων όταν είναι απαραίτητο.

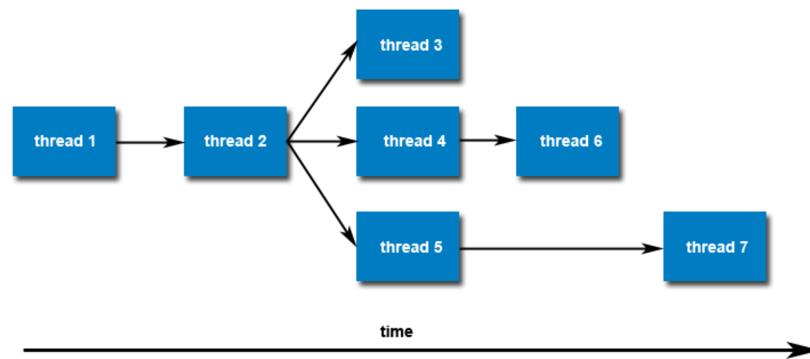
3. **Συγχρονισμός** (synchronization) παρέχει μηχανισμούς στα πλαίσια ταυτόχρονης εκτέλεσης ώστε να συντονίζεται η χρήση κοινών πόρων μεταξύ τους. Και σε αυτό το σημείο γίνεται κατάχρηση της λέξης σύμφωνα με τον ορισμό που δίνεται στα λεξικά, όπου αναφέρονται σε «κάτι που προκαλεί δύο ή περισσότερα πράγματα να συμβούν στον ίδιο χρόνο» ενώ στο λεξιλόγιο των νημάτων πρόκειται για «κάτι που **αποτρέπει** δύο ή περισσότερα πράγματα από το να συμβούν στον ίδιο χρόνο».

4.4 Υλοποίηση

Η ανάπτυξη ενός πολυνηματικού προγράμματος σε pthreads προαπαιτεί χρήση της γλώσσας C/C++. Κατά τη μετάφραση του προγράμματος γίνεται σύνδεση του αντικείμενου αρχείου (object file) με τη βιβλιοθήκη libpthread η οποία είναι προεγκατεστημένη σε κάθε POSIX λειτουργικό σύστημα και περιέχει όλες τις συναρτήσεις και τους τύπους μεταβλητών των pthreads. Ειδικότερα για τη C++, υπάρχει μια επιπλέον υλοποίηση-τύλιγμα (wrapper implementation) των

pthreads ώστε να εκμεταλλεύονται πιο αποτελεσματικά κάποιες δυνατότητές της, στην ανοικτού κώδικα βιβλιοθήκη Boost (libboost) [24], τα Boost Threads. Ωστόσο, κάτι τέτοιο αυξάνει τις εξαρτήσεις βιβλιοθηκών του προγράμματος οπότε στην παρούσα εργασία, δεν χρησιμοποιήθηκε.

Τα threads (εκτός του αρχικού) δημιουργούνται με κλήση της συνάρτησης `pthread_create( )`. Αυτή η συνάρτηση μπορεί να κληθεί πολλές φορές από οπουδήποτε μέσα σε έναν πολυνηματικό κώδικα. Άπαξ και δημιουργηθούν, τα νέα threads είναι ισότιμα (peers) μεταξύ τους και μπορούν με τη σειρά τους να δημιουργήσουν νέα threads. Δεν υπάρχει καμία ιεραρχία ή αλληλεξάρτηση μεταξύ των threads.



Σχήμα 4.2: Δημιουργία και ισοτιμία threads

Όταν εκτελείται ένα πρόγραμμα γραμμένο σε C, το σημείο εισόδου είναι η συνάρτηση `main( )`. Όταν το πρόγραμμα είναι πολυνηματικό, ακριβώς στο σημείο εισόδου της εφαρμογής (τη στιγμή που ξεκινά να εκτελεστεί η συνάρτηση `main( )`) δημιουργείται αυτόματα το πρώτο thread που ονομάζεται και αρχικό ή κεντρικό (initial/main) thread. Το αρχικό thread μπορεί να εκτελέσει οποιαδήποτε λειτουργία των pthreads. Μπορεί να προσδιορίσει την ταυτότητά του καλώντας για παράδειγμα τη συνάρτηση `pthread_self( )` ή να τερματίσει τη λειτουργία του καλώντας την `pthread_exit( )`. Αν το αρχικό thread αποθηκεύσει την ταυτότητά του κάπου που να είναι προσβάσιμη από ένα άλλο thread, τότε εκείνο μπορεί για παράδειγμα να περιμένει το αρχικό thread να τερματίσει ή μπορεί και να αποδεσμεύσει το αρχικό thread.

Το μόνο χαρακτηριστικό που καθιστά το αρχικό thread ξεχωριστό είναι ότι συμπεριφέρεται ταυτόχρονα ως παραδοσιακή διεργασία UNIX, οπότε αν η `main( )` τερματίσει χωρίς να ληφθεί κάποια ιδιαίτερη φροντίδα, τότε τα υπόλοιπα threads που έχουν δημιουργηθεί απλά «εξατμίζονται» χωρίς να ολοκληρώσουν την δουλειά τους. Όταν η διεργασία τερματίσει, όλα της τα threads μαζί με την αποθηκευμένη κατάστασή τους και ότι τους είχε ανατεθεί να κάνουν

απλά εξαφανίζονται.

Παρά το ότι η «εξάτμιση» των threads (threads evaporation) με τον παραπάνω τρόπο είναι ορισμένες φορές βολική, τις περισσότερες φορές μια διεργασία ζει περισσότερο από τα επιμέρους threads που δημιουργεί. Προκειμένου λοιπόν να απελευθερωθούν οι πόροι του συστήματος που χρησιμοποιούνταν από threads που έχουν ολοκληρωθεί, πρέπει πάντα ένα thread αφού έχει τελειώσει τη δουλειά του να αποδεσμεύεται με κλήση της συνάρτησης `pthread_detach( )`. Ένα thread μπορεί να αποδεσμευτεί οποτεδήποτε από μόνο του ή από οποιοδήποτε άλλο thread που γνωρίζει την ταυτότητά του. Αν χρειάζεται να κρατηθεί η τιμή που επιστρέφει ένα thread ή αν πρέπει να γνωρίζει η εφαρμογή πότε ένα thread τερμάτισε τότε η `pthread_detach( )` δεν είναι κατάλληλη. Γι΄ αυτό το σκοπό υπάρχει η `pthread_join( )`, η οποία μπλοκάρει την συνάρτηση από την οποία κλήθηκε, μέχρι το thread (η ταυτότητα του οποίου αποτελεί το πρώτο όρισμα της) να τερματίσει και τότε - αν χρειάζεται - αποθηκεύει την επιστρεφόμενη τιμή του στο δεύτερο όρισμά της (όταν δεν χρειάζεται, το δεύτερο όρισμα τίθεται απλώς NULL). Η κλήση της `pthread_join( )` αποδεσμεύει το thread που της δίνεται αυτόματα.

Ένα παράδειγμα χρήσης pthreads δίνεται με κώδικα. Πρόκειται για πρόγραμμα που υπολογίζει το εσωτερικό γινόμενο δύο διανυσμάτων. Στην αρχή δίνεται η σειριακή εκδοχή του (κώδικας 4.1) και στη συνέχεια η παράλληλη υλοποιημένη με pthreads (κώδικας 4.2):

```c
#include <stdio.h>
#include <stdlib.h>

// Handle NULL assignments (malloc, new, etc)
#define HANDLE_NULL(call) \
    do { \
        if ((call) == NULL) { \
            fprintf(stderr, "Resource allocation error in %s " \
                    "at line %d\n", __FILE__, __LINE__); \
            exit(EXIT_FAILURE); \
        } \
    } while(0)

#define sum_squares( x ) ( x*(x+1)*(2*x+1)/6 )

#define N ( 33 * 1024 * 1024 )

double dot(const int size, double *a, double *b);

// =========================================================
double dot(const int size, double *a, double *b)
// =========================================================
{
    double psum = 0.0;
    for (int i = 0; i < size ; i++) psum += a[i] * b[i];
    return psum;
}
// =========================================================
```

```
int main()
// =========================================================
{
    // allocate vectors a, b
    double *a, *b;
    HANDLE_NULL(a = new double[N]);
    HANDLE_NULL(b = new double[N]);

    // fill the vectors a, b
    for (int i = 0; i < N; i++) {a[i] = i; b[i] = i*2;}

    double product = dot(N, a, b);

    // Cleanup: free memory space
    delete[] a; delete[] b;

    printf("Value calculated:  %.6g\n", product);
    printf("Is it  %.6g ?\n", 2 * sum_squares( (double)(N-1) ) );

    exit( EXIT_SUCCESS );
}
```

Κώδικας 4.1: (dot–1cpu.cpp) Παράδειγμα κώδικα που υπολογίζει σειριακά, με ένα επαναληπτικό βρόχο, το εσωτερικό γινόμενο 2 διανυσμάτων.

Ο κώδικας 4.1 μεταφράζεται σε εκτελέσιμο με την εντολή:
`g++ dot-1cpu.cpp -o dot-1cpu.run`

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <pthread.h>
4
5 // Handle NULL assignments (malloc, new, etc)
6 #define HANDLE_NULL(call) \
7     do { \
8         if ((call) == NULL) { \
9             fprintf(stderr, "Resource allocation error in %s " \
10                     "at line %d\n", __FILE__, __LINE__); \
11             exit(EXIT_FAILURE); \
12         } \
13     } while(0)
14
15 #define N ( 33 * 1024 * 1024 )
16
17 #define sum_squares( x ) ( x*(x+1)*(2*x+1)/6 )
18
19 struct DataStruct
20 {
21     int     size;
22     double *a;
23     double *b;
24     double product;
25 };
26
27 double dot(const int size, double *a, double *b);
28 void* routine(void *pvoidData);
29
30 // ========================================================
31 double dot(const int size, double *a, double *b)
32 // ========================================================
33 {
34     double  psum = 0.0;
35     for (int i = 0; i < size ; i++) psum += a[i] * b[i];
36     return psum;
37 }
38 // ========================================================
39 void* routine(void *pvoidData)
40 // ========================================================
41 {
42     DataStruct  *data = (DataStruct*)pvoidData;
43
44     const int size = data->size;
45     double *a, *b;
46
47     a = data->a;
48     b = data->b;
49
50     data->product = dot(size, a, b);
51
52     return 0;
53 }
54 // ========================================================
55 int main()
56 // ========================================================
57 {
58     // allocate vectors a, b
59     double *a, *b;
60     HANDLE_NULL(a = new double[N]);
61     HANDLE_NULL(b = new double[N]);
62
```

```
63      // fill the vectors a, b
64      for (int i = 0; i < N; i++) {a[i] = i; b[i] = i*2;}
65
66      // define and form the data set for each thread
67      DataStruct data[3];
68
69      for (int i = 0; i < 3; i++)
70      {
71          data[i].size = N/3;
72          data[i].a    = a + i * (N/3);
73          data[i].b    = b + i * (N/3);
74      }
75      // define the pthread instances
76      pthread_t hostThreads[2];
77
78      // useful variable for error checking
79      int res;
80
81      // Create 2 threads
82      for (int i = 0; i < 2; i++)
83      {
84          res = pthread_create(&(hostThreads[i]), NULL,
85                               routine, (void *)(&data[i]));
86          if (res != 0)
87          {
88              printf("thread creation failed\n");
89              exit(EXIT_FAILURE);
90          }
91      }
92      // No need to create another thread
93      // Use initial thread (main) as 3rd thread
94      routine( (void *) (&data[2]) );
95
96      // Join threads
97      for (int i = 0; i < 2; i++)
98      {
99          res = pthread_join(hostThreads[i], NULL);
100         if (res != 0)
101         {
102             printf("pthread_join failed\n");
103             exit(EXIT_FAILURE);
104         }
105     }
106     // Cleanup: free memory space
107     delete[] a; delete[] b;
108
109     printf("Value calculated: %.6g\n", data[0].product +
110                                         data[1].product +
111                                         data[2].product);
112
113     printf("Is it %.6g ?\n", 2 * sum_squares((double)(N-1)));
114
115     exit(EXIT_SUCCESS);
116 }
```

Κώδικας 4.2: (dot–3cpu.cpp) Παράδειγμα κώδικα που υπολογίζει παράλληλα το εσωτερικό γινόμενο 2 διανυσμάτων, με διάσπαση του καθενός σε 3 μικρότερα υποδιανύσματα και ανάθεσή τους σε 3 νήματα.

Ο κώδικας 4.2 μεταφράζεται σε εκτελέσιμο με την εντολή:

```
g++ dot-3cpu.cpp -o dot-3cpu.run -lpthread
```

Σχετικά με τον παράλληλο κώδικα για 3 threads, μπορούν να γίνουν οι εξής παρατηρήσεις:

Γραμμή 3: Κάθε πολυνηματικός κώδικας πρέπει να περιέχει την κεφαλίδα (header) της βιβλιοθήκης libpthread. Εκεί βρίσκονται όλα τα πρωτότυπα συναρτήσεων (function prototypes) όπως η `pthread_create( )` και οι ειδικοί τύποι μεταβλητών όπως ο `pthread_t`. Αυτό είναι το αρχείο `pthread.h`. Προκειμένου όλα τα παραπάνω να συνδεθούν με το εκτελέσιμο, ο linker πρέπει να ξέρει πού θα τα βρει. Αυτό του το δίνει ο προγραμματιστής με χρήση του διακόπτη `-lpthread` στην εντολή μετάφρασης. Αν παραλειφθεί, ο linker θα παραπονεθεί με μηνύματα σφάλματος ότι δεν τα βρίσκει.

Γραμμές 19-25: Επειδή όταν δημιουργείται ένα thread (βλ. παρακάτω), τα δεδομένα που θα επεξεργαστεί του ανατίθενται μόνο με ένα όρισμα, πρέπει όλα να «πακεταριστούν» σε ένα. Αυτό γίνεται με τον ορισμό μιας δομής δεδομένων της C (`struct`). Αυτή η δομή έχει ως μέλη το μέγεθος των 2 υποδιανυσμάτων, τις διευθύνσεις τους, και το εσωτερικό γινόμενο όπου θα αποθηκευτεί το αποτέλεσμα. Το όνομά της είναι `DataStruct` και αποτελεί νέο τύπο δεδομένων. Αυτό σημαίνει πως, όταν οριστεί μια μεταβλητή τύπου `DataStruct`, θα δημιουργηθεί χώρος για όλα τα παραπάνω.

Γραμμές 39-53: Ορισμός (σώμα) της συνάρτησης `routine( )`. Κατά τη δημιουργία ενός thread (βλ. παρακάτω), η εργασία που πρέπει να εκτελέσει, του ανατίθεται με τη μορφή μιας συνάρτησης. Έχει επικρατήσει, ειδικά σε περιπτώσεις που όλα τα threads εκτελούν την ίδια εργασία, αυτή η συνάρτηση να ονομάζεται `routine( )`, παρά το ότι μπορεί να έχει όποιο όνομα θέλει ο προγραμματιστής. Αυτή η συνάρτηση, πάντα επιστρέφει μια ουδέτερη διεύθυνση `void*` και πάντα παίρνει ως όρισμα επίσης μια ουδέτερη διεύθυνση `void*`. Επειδή υπάρχει αυτός ο περιορισμός, αμέσως μετά την είσοδο στη συνάρτηση, λαμβάνεται φροντίδα (γραμμή 42) να μετατραπεί (cast) η διεύθυνση που έχει εισέλθει ως όρισμα, στον κατάλληλο τύπο (εδώ `DataStruct`). Στις γραμμές 44-48 δίνεται στο thread η πληροφορία για τον δικό του «χώρο δεδομένων», δηλαδή από ποιο μέχρι ποιο στοιχείο των ενιαίων διανυσμάτων θα εργαστεί. Αυτό γίνεται πιο εύκολα κατανοητό παρακάτω. Στη γραμμή 50 το κάθε thread καλεί τη συνάρτηση `dot( )` για να υπολογίσει το δικό του εσωτερικό γινόμενο.

Γραμμή 55: η συνάρτηση `main( )` είναι το σημείο εισόδου σε κάθε πρόγραμμα C/C++. Εδώ αρχίζουν όλα. Μάλιστα, επειδή το συγκεκριμένο πρόγραμμα είναι πολυνηματικό, η είσοδος στη `main( )` σηματοδοτεί την αυτόματη δημιουργία του αρχικού thread όπως έχει αναφερθεί και σε προηγούμενη παράγραφο.

Γραμμές 60,61: Δεσμεύεται χώρος N θέσεων στη μνήμη για τα ενιαία

(μεγάλα) διανύσματα `a, b`.

Γραμμή 64: Τα στοιχεία των ενιαίων (μεγάλων) διανυσμάτων παίρνουν τιμές:
$\vec{a} = (0, 1, 2, ..., N-1)$ και $\vec{b} = (0, 2, 4, ..., 2 \cdot (N-1))$

Γραμμές 67-74: Στη γραμμή 67 ορίζεται ένας πίνακας 3 στοιχείων τύπου `DataStruct`. Το κάθε στοιχείο αυτού του πίνακα (που μελλοντικά θα τροφοδοτηθεί σε κάποιο thread), ως μεταβλητή τύπου `DataStruct` περιέχει κενές - προς το παρόν - θέσεις (μέλη) για το μέγεθος των υποδιανυσμάτων, τις διευθύνσεις μνήμης τους και το εσωτερικό γινόμενό τους. Στις γραμμές 69-74 ακολουθεί το γέμισμα αυτών των θέσεων με τιμές. Κάθε μέγεθος υποδιανύσματος, λοιπόν, ισούται με N/3. Για το στοιχείο `data[0]`, το υποδιάνυσμα `a` αρχίζει στη διεύθυνση μνήμης που ταυτίζεται με τη διεύθυνση μνήμης του ενιαίου `a`, δηλαδή $a+0\cdot(N/3)$. Για το στοιχείο `data[1]`, το υποδιάνυσμα `a` αρχίζει μία διεύθυνση μνήμης δίπλα από εκεί που τελειώνει του `data[0]`, δηλαδή $a+1\cdot(N/3)$, κ.ο.κ. Τα αντίστοιχα ισχύουν για τα υποδιανύσματα `b`. Γίνεται σαφές λοιπόν ότι τα ενιαία διανύσματα `a,b` χωρίζονται στα 3, και κάθε thread αναλαμβάνει το 1/3 που του αναλογεί.

Γραμμή 76: Ορίζεται ένας πίνακας 2 στοιχείων, τύπου `pthread_t`. Το κάθε στοιχείο αυτού του πίνακα περιέχει ταυτόχρονα την «ταυτότητα» και την «χειρολαβή» ενός thread (thread handle), δηλαδή την μεταβλητή από την οποία ο προγραμματιστής μπορεί να το «πιάσει» και να το διαχειριστεί. Ως τύπος, ο `pthread_t`, είναι «αδιαφανής» (opaque type). Δεν είναι εφικτό, δηλαδή, να τυπωθεί η τιμή του στην οθόνη. Το ίδιο συμβαίνει με τους περισσότερους, αν όχι όλους, τύπους μεταβλητών που ορίζονται στα pthreads (π.χ `pthread_mutex_t` `pthread_cond_t`) που θα παρουσιαστούν σε επόμενες παραγράφους. Τέλος, το γιατί ο πίνακας είναι διάστασης 2 κι όχι 3 (αν δεν έχει γίνει ήδη κατανοητό) θα εξηγηθεί στη συνέχεια.

Γραμμές 82-91: Βρόχος μέσα στον οποίο γεννιούνται τα threads και αναλαμβάνουν εργασία. Στη γραμμή 84, η `pthread_create( )` λαμβάνει ως ορίσματα (με σειρά εμφάνισης): την διεύθυνση της ταυτότητας του thread που πρόκειται να γεννηθεί, `NULL` που σημαίνει ότι δεν αποδίδεται σ΄ αυτό το thread κάποιο ιδιαίτερο χαρακτηριστικό (attribute), το όνομα-διεύθυνση της συνάρτησης που θα εκτελέσει το νέο thread και τέλος, το όρισμα αυτής της συνάρτησης, μια διεύθυνση στοιχείου του πίνακα `data[]` που πρέπει να μετατραπεί προηγουμένως σε ουδέτερη διεύθυνση με τον τελεστή cast. Η κλήση της `pthread_create( )` επιστρέφει έναν ακέραιο ο οποίος στις γραμμές 86-90 εξετάζεται για να διαπιστωθεί αν η δημιουργία του thread ήταν επιτυχής ή όχι.

Γραμμή 94: Επειδή η δημιουργία ενός thread έχει κάποιο ελάχιστο κόστος (overhead), και με την προϋπόθεση ότι η περίπτωση το επιτρέπει, αποφεύγεται η

δημιουργία 3ου thread. Το 3ο thread δεν χρειάζεται να δημιουργηθεί, εφόσον υπάρχει ήδη από την αρχή του προγράμματος και είναι η ίδια η συνάρτηση `main( )`, δηλαδή το αρχικό thread. Αυτός είναι και ο λόγος που ο πίνακας `hostThreads[]` ορίστηκε με διάσταση 2 αντί για 3.

Γραμμές 97-105: Βρόχος στον οποίο το αρχικό thread περιμένει τα άλλα 2 να τελειώσουν το έργο τους, ώστε να συναντηθούν (join) όλα μαζί. Η συνάρτηση `pthread_join( )` δέχεται δύο ορίσματα, την ταυτότητα του thread που αναμένει και την τιμή που αυτό επιστρέφει μόλις τελειώσει. Επειδή στο συγκεκριμένο πρόγραμμα τα threads δεν επιστρέφουν τίποτα, το δεύτερο όρισμα είναι αδιάφορο κι ως εκ τούτου μπαίνει στη θέση του `NULL`. Όπως έχει αναφερθεί, μόλις τα θυγατρικά threads φτάσουν στην `pthread_join( )` καταστρέφονται και οι πόροι που χρησιμοποιούσαν αποδεσμεύονται από το λειτουργικό σύστημα. Η κλήση της `pthread_join( )` επιστρέφει έναν ακέραιο ο οποίος στις γραμμές 100-103 εξετάζεται για να διαπιστωθεί αν η κλήση ήταν επιτυχής ή όχι.

Γραμμές 109-111: Τα επιμέρους εσωτερικά γινόμενα που έχουν υπολογιστεί από τα threads αθροίζονται πριν τυπωθούν στην οθόνη, ώστε να προκύψει το εσωτερικό γινόμενο των μεγάλων διανυσμάτων.

Σύνοψη: Αυτό το παράδειγμα εντάσσεται στο μοντέλο «ομάδας εργασίας» (βλ. παράγραφο 4.7 και σχήμα 4.5). Μια σημαντική παρατήρηση είναι πως (όπως φαίνεται κι από την συνάρτηση `dot( )`) το κάθε thread κάνει τους υπολογισμούς του σειριακά (με ένα βρόχο) συσσωρεύοντας ένα άθροισμα γινομένων, απλώς τους εκτελεί σε μικρότερο όγκο δεδομένων. Αυτό έχει μεγάλη διαφορά από την έκδοση εσωτερικού γινομένου για CUDA (κώδικας 3.1) όπου υπολογίζονταν ταυτόχρονα όλα τα γινόμενα των στοιχείων και μετά αθροίζονταν. Σε μια επόμενη έκδοση εσωτερικού γινομένου, γίνεται υβριδισμός της CUDA έκδοσης με την pthreads έκδοση ώστε να προκύψουν ακόμα μεγαλύτερα οφέλη.

4.5 Συγχρονισμός

Σε ένα πολυνηματικό πρόγραμμα οποιουδήποτε βαθμού πολυπλοκότητας, προκύπτει η ανάγκη να μοιραστούν δεδομένα μεταξύ των threads ή να εκτελεστούν διάφορες λειτουργίες με μια συγκεκριμένη σειρά. Τα παραπάνω εντάσσονται στην έννοια του συγχρονισμού.

4.5.1 Mutexes

Τα περισσότερα πολυνηματικά προγράμματα χρειάζεται να μοιράζουν δεδομένα μεταξύ των threads. Αν δεν προηγηθεί προσεκτικός σχεδιασμός αυτό γίνεται αντικείμενο συγκρούσεων (shared data conflicts). Για παράδειγμα, τη στιγμή που ένα thread τροποποιεί την τιμή μιας μεταβλητής, ένα άλλο thread επιχειρεί να τη διαβάσει. Όταν περισσότερα του ενός threads προσπαθούν να αποκτήσουν πρόσβαση ταυτόχρονα σε κοινά δεδομένα, προκύπτει πάντα ένας αγώνας (race condition) για το ποιό thread θα προλάβει πρώτο να κάνει κάτι με αυτά τα δεδομένα. Η έκβαση αυτού του αγώνα είναι στοχαστική και διαφέρει από εκτέλεση σε εκτέλεση, οπότε και δίνονται διαφορετικά αποτελέσματα ανά περίπτωση από το πρόγραμμα.

Ο πιο λογικός και γενικός τρόπος συγχρονισμού μεταξύ των threads είναι να εξασφαλιστεί ότι όλες οι προσβάσεις μνήμης σε μοιρασμένα δεδομένα είναι **αμοιβαία αποκλειόμενες**. Αυτό σημαίνει ότι μόνο ένα thread επιτρέπεται να γράφει κάποια χρονική στιγμή, τα υπόλοιπα πρέπει να περιμένουν τη σειρά τους. Τα POSIX threads παρέχουν μία μέθοδο αμοιβαίου αποκλεισμού, που ονομάζεται **mutex**. Η λέξη αυτή αποτελεί συγχώνευση των λέξεων **mut**ual **ex**clusion. Το mutex είναι μια μεταβλητή τύπου `pthread_mutex_t` η οποία μπορεί να κλειδώνει και να ξεκλειδώνει.

Ο συγχρονισμός δεν είναι σημαντικός μόνο όταν τροποποιούνται δεδομένα ταυτόχρονα. Είναι απαραίτητος κι όταν ένα thread χρειάζεται να διαβάσει δεδομένα που έχουν προηγουμένως τροποποιηθεί από ένα άλλο thread δηλαδή όταν η σειρά με την οποία γράφονται και διαβάζονται έχει σημασία. Όταν ένα thread κλειδώνει ένα mutex γύρω από ένα κομμάτι κώδικα που τροποποιεί ή διαβάζει κοινά δεδομένα, τότε μόνο αυτό επιτρέπεται να εκτελέσει τις εντολές αυτού του τμήματος κώδικα. Τα υπόλοιπα μπλοκάρουν μέχρι να ξεκλειδώσει το mutex.

Ένα παράδειγμα χρήσης mutex παρουσιάζεται παρακάτω. Το συγκεκριμένο παράδειγμα, ως έμπνευση του συγγραφέα, προτιμήθηκε από όσα υπάρχουν ήδη στη βιβλιογραφία διότι είναι το απλούστερο δυνατό και το πιο αφαιρετικό. Σε ένα πολυνηματικό πρόγραμμα, που εκτελείται στον κεντρικό server μίας τράπεζας

για να ενημερώνει τους λογαριασμούς των καταθετών, προβλέπονται τα εξής:

- Δουλειά του thread1 είναι να εκτελεί μια εντολή ανάληψης.
- Δουλειά του thread2 είναι να εκτελεί μια εντολή κατάθεσης.
- Κοινό χαρακτηριστικό της δουλειάς τους είναι ότι πρέπει να εκτελέσουν μια διαδικασία ανάγνωσης/τροποποίησης/εγγραφής (read-modify-write operation) στη μεταβλητή υπολοίπου.

```
// thread 1

bal  = GetBalance(account);
bal -= withdrawal;
SetBalance(account, bal);
```

thread 1

```
// thread 2

bal  = GetBalance(account);
bal += deposit;
SetBalance(account, bal);
```

thread 2

Έστω οικογενειακός λογαριασμός στον οποίο η τράπεζα έχει εκδόσει κάρτες ανάληψης για όσα μέλη της οικογένειας είναι συνδικαιούχοι. Αν από σύμπτωση τύχει να πάνε ταυτόχρονα σε διαφορετικά ATM 2 μέλη της οικογένειας, το ένα για να κάνει κατάθεση 100 € και το άλλο για να κάνει ανάληψη 100 €, θα δημιουργηθεί ένα race condition των 2 threads για την μεταβλητή υπολοίπου με δύο πιθανά σενάρια:

Βήμα	Ποσό
thread1 διαβάζει 1500	1500
thread2 διαβαζει 1500	1500
thread1 υπολογίζει 1500 − 100	1500
thread2 υπολογίζει 1500 + 100	1500
thread1 γράφει 1400	1400
thread2 γράφει 1600	1600

Βήμα	Ποσό
thread1 διαβάζει 1500	1500
thread2 διαβαζει 1500	1500
thread1 υπολογίζει 1500 − 100	1500
thread2 υπολογίζει 1500 + 100	1500
thread2 γράφει 1600	1600
thread1 γράφει 1400	1400

Και τα δύο παραπάνω σενάρια προφανώς καταλήγουν σε λάθος αποτέλεσμα. Στο αριστερό, το λάθος είναι προς όφελος των καταθετών ενώ στο δεξιά προς όφελος της τράπεζας. Πρέπει να επισημανθεί ότι το ποιο από τα δύο σενάρια θα συμβεί τελικά είναι καθαρά στοχαστικό. Μπορεί να λειτουργήσει ακόμα και το σωστό σενάριο παρά το ότι οι 2 συναλλαγές γίνονται ταυτόχρονα. Αυτό είναι το αποτέλεσμα της έλλειψης συντονισμού (ή καλύτερα συγχρονισμού) μεταξύ των threads.

Με τη χρήση mutex θα αποφευχθεί αυτή η κούρσα. Αρχικά, και οπωσδήποτε πριν δημιουργηθούν τα threads, ορίζεται μια μεταβλητή τύπου `pthread_mutex_t`. Στη συνέχεια αρχικοποιείται με την κλήση της `pthread_mutex_init( )`.

```
// before threads creation

pthread_mutex_t mtx;
pthread_mutex_init(&mtx);
```

αρχικοποίηση μεταβλητής mutex

Τα threads χειρίζονται τη μεταβλητή mutex έτσι ώστε το καθένα να εκτελεί μια ολοκληρωμένη συναλλαγή (atomic transaction) και στο μεταξύ το άλλο να περιμένει όπως φαίνεται παρακάτω:

```
// thread 1 function
// enclosed in atomic transaction

pthread_mutex_lock(&mtx);

bal  = GetBalance(account);
bal -= withdrawal;
SetBalance(account, bal);

pthread_mutex_unlock(&mtx);
```

thread 1

```
// thread 2 function
// enclosed in atomic transaction

pthread_mutex_lock(&mtx);

bal  = GetBalance(account);
bal += deposit;
SetBalance(account, bal);

pthread_mutex_unlock(&mtx);
```

thread 2

Εδώ φαίνεται παράλληλα πώς εξελίσσονται οι λειτουργίες των threads με τη χρήση mutex. Το τελικό αποτέλεσμα είναι και το αναμενόμενο.

Βήμα	Ποσό
thread1 κλειδώνει το mutex	1500
thread1 διαβάζει 1500	1500
thread1 υπολογίζει $1500 - 100$	1500
thread1 γράφει 1400	1400
thread1 ξεκλειδώνει το mutex	1400
...	1400
...	1400
...	1400
...	1500
...	1500

Βήμα	Ποσό
thread2 βρίσκει το mutex κλειδωμένο	1500
thread2 σε αναμονή...	1500
thread2 σε αναμονή...	1500
thread2 σε αναμονή...	1400
thread2 σε αναμονή...	1400
thread2 κλειδώνει το mutex	1400
thread2 διαβάζει 1400	1400
thread2 υπολογίζει $1400 + 100$	1400
thread2 γράφει 1500	1500
thread2 ξεκλειδώνει το mutex	1500

Μέγεθος ή έκταση ενός mutex

Με τον όρο «μέγεθος» ή «έκταση» ενός mutex υπονοείται η έκταση του κώδικα που επηρεάζεται από το κλείδωμα/ξεκλείδωμα του mutex. Ένας απλός και πρωτόγονος τρόπος να προστατευτεί ένα πρόγραμμα από συγκρούσεις μεταξύ των threads κατά την πρόσβαση σε κοινά δεδομένα, είναι να τοποθετείται και να κλειδώνεται ένα mutex στην αρχή κάθε συνάρτησης που θα εκτελεστεί από τα threads και να ξεκλειδώνεται στο τέλος της συνάρτησης. Έτσι καθίσταται σειριακή η συνάρτηση και αποτρέπεται οποιαδήποτε σύγκρουση. Αργά ή γρήγορα βέβαια γίνεται φανερό πως αυτή η προσέγγιση ακυρώνει οποιοδήποτε κέρδος από τη χρήση των threads. Αυτό το mutex που κλειδώνει μια μεγάλη

περιοχή κώδικα έχει επικρατήσει να λέγεται big mutex σε αντιδιαστολή με ένα mutex που κλειδώνει μόνο 2 γραμμές κώδικα. Κατ΄ επέκταση, ένα mutex που προστατεύει 2 μεταβλητές πρέπει να λογίζεται ως «μεγαλύτερο» από ένα άλλο mutex που προστατεύει μόνο μία. Στο ερώτημα: «πόσο μεγάλο πρέπει να είναι ένα mutex;» η απάντηση είναι «όσο μεγάλο χρειάζεται αλλά όχι μεγαλύτερο».

Όταν χρειάζεται να προστατευτούν από ταυτόχρονη πρόσβαση 2 κοινές μεταβλητές, υπάρχουν 2 στρατηγικές. Η λεπτή (fine-grained) είναι να τοποθετηθεί ένα μικρό mutex για κάθε μία, και η χονδροειδής (coarse-grained) να τοποθετηθεί ένα μεγάλο mutex και για τις δύο. Το ποιά στρατηγική είναι καλύτερη δεν είναι πάντα προφανές και εξαρτάται από πολλούς παράγοντες οι οποίοι μάλιστα μπορεί να αλλάζουν κατά το γράψιμο του κώδικα, αναλόγως με το πόσα threads χρειάζονται πρόσβαση στις μεταβλητές και πώς τις χρησιμοποιούν.

Παρακάτω φαίνονται μερικοί από αυτούς τους παράγοντες:

- Τα mutexes δεν είναι «δωρεάν». Το κλείδωμα και ξεκλείδωμά τους παίρνει χρόνο. Οπότε, κώδικας που στο σύνολό του χρησιμοποιεί λιγότερα mutexes θα τρέχει γρηγορότερα από κώδικα με περισσότερα. Το συμπέρασμα είναι ότι πρέπει να χρησιμοποιούνται τα λιγότερα δυνατά, με το καθένα να προστατεύει μια λογική περιοχή.

- Τα mutexes εκ της φύσεώς τους, κάνουν την εκτέλεση σειριακή. Αν πολλά threads χρειάζεται συχνά να κλειδώνουν ένα mutex τότε θα περνάνε τον περισσότερο χρόνο τους περιμένοντας. Αν τα τμήματα δεδομένων που προστατεύει ένα mutex δεν σχετίζονται μεταξύ τους, μπορεί να αυξηθεί η απόδοση χωρίζοντας το μεγάλο mutex σε μικρότερα.

- Οι δύο παραπάνω παράγοντες προφανώς αλληλοσυγκρούονται. Οπότε η βέλτιστη χρήση των mutexes είναι αποτέλεσμα μελέτης και μετρήσεων του κώδικα αλλά και εμπειρίας.

4.5.2 Condition Variables (μεταβλητές συνθήκης)

Μία μεταβλητή συνθήκης (τύπος `pthread_cond_t`) χρησιμοποιείται για να επικοινωνήσει πληροφορία σχετικά με την **κατάσταση** κοινών δεδομένων. Για παράδειγμα για να στείλει σήμα ότι μια λίστα δεν είναι πλέον άδεια. Όταν ένα thread έχει αμοιβαία αποκλειόμενη πρόσβαση σε μια μοιρασμένη κατάσταση, ίσως ανακαλύψει ότι δεν υπάρχει τίποτα να κάνει μέχρι κάποιο άλλο thread να αλλάξει αυτή την κατάσταση. Η κατάσταση μπορεί να είναι σωστή και συνεπής (δηλαδή δεν παραβιάζεται κάποια αρχή από αυτές που περιγράψαμε παραπάνω), απλώς τυχαίνει να μην έχει κανένα ενδιαφέρον για το thread.

Αν, για παράδειγμα, ένα thread (έστω `thread1`) που διαβάζει δεδομένα από μια λίστα, βρει τη λίστα άδεια, τότε πρέπει να περιμένει μέχρι να προστεθεί μια καταχώρηση στη λίστα (έστω από το `thread2`). Ας υποτεθεί ότι η πρόσβαση στη λίστα προστατεύεται από ένα mutex. Τότε, η σειρά είναι η εξής:

1. Το `thread1` πρέπει πρώτα να κλειδώσει το mutex.
2. Επιχειρεί να διαβάσει ένα στοιχείο της λίστας αλλά ανακαλύπτει ότι η λίστα είναι άδεια.
3. Ξεκλειδώνει το mutex πριν αρχίσει να περιμένει πότε το `thread2` θα προσθέσει μια καταχώρηση, διότι αν δεν το ξεκλειδώσει τότε ούτε το `thread2` ούτε κανένα άλλο thread δεν θα μπορέσει ποτέ να καταχωρήσει οτιδήποτε στη λίστα.
4. Περιμένει να γίνει η πρώτη καταχώρηση. Αυτό μπορεί να σημαίνει ότι μπαίνει σε λήθαργο μέχρι το `thread2` να κάνει την καταχώρηση και να το ξυπνήσει. Εδώ εντοπίζεται το πρώτο σοβαρό πρόβλημα: απ' τη στιγμή που το `thread1` θα ξεκλειδώσει το mutex μέχρι τη στιγμή που θα μπει σε λήθαργο, μεσολαβεί ένα ελάχιστο χρονικό διάστημα κατά το οποίο είναι ενεργό.
5. Μόλις το `thread2` φθάσει μέσα στο ελάχιστο χρονικό διάστημα και κλειδώσει με τη σειρά του το mutex για να κάνει την καταχώρησή του, θα κοιτάξει προηγουμένως αν το `thread1` κοιμάται περιμένοντάς το, αλλά θα το βρει ξύπνιο, οπότε θα θεωρήσει ότι δεν χρειάζεται να το ξυπνήσει για να το ενημερώσει για την αλλαγή κατάστασης της λίστας.
6. Το `thread2` κάνει την καταχώρηση και ξεκλειδώνει το mutex.
7. Τελικά το `thread1` θα περιμένει για πάντα να συμβεί η πρώτη καταχώρηση χωρίς να γνωρίζει ότι έχει ήδη συμβεί. Αυτή η κατάσταση σε ένα πολυνηματικό πρόγραμμα ονομάζεται **deadlock**.

8. Ακόμα χειρότερα, το παραπάνω deadlock μπορεί να μην εκδηλώνεται σε όλα τα τρεξίματα του προγράμματος. Είναι στοχαστικό. Δηλαδή θα υπάρχουν φορές που η διαδικασία θα γίνεται σωστά (διότι θα τυχαίνει να μην καταφθάνει το `thread2` σε αυτό το ελάχιστο χρονικό παράθυρο) και το πρόγραμμα θα λειτουργεί κανονικά και φορές που ο χρήστης απλώς θα περιμένει στο άπειρο χωρίς να βλέπει αποτελέσματα. Ωστόσο το πρόγραμμα θα φαίνεται να λειτουργεί κανονικά διότι το deadlock θα είναι μέρος της λειτουργίας του προγράμματος (ένα στοχαστικό λογικό σφάλμα).

Το συμπέρασμα είναι ότι οι λειτουργίες του ξεκλειδώματος και της έναρξης αναμονής πρέπει να είναι ατομικές (atomic) με την έννοια ότι κανένα άλλο thread δεν πρέπει να κλειδώσει το mutex πριν μπει σε λανθάνουσα κατάσταση αυτό που περιμένει. Αυτός είναι κι ο λόγος ύπαρξης των μεταβλητών συνθήκης. Μια μεταβλητή συνθήκης είναι ένας μηχανισμός σηματοδότησης που σχετίζεται πάντα με ένα mutex και κατ΄ επέκταση και με τα δεδομένα που προστατεύει αυτό το mutex. Όταν ένα thread αλλάζει την τιμή μιας μεταβλητής συνθήκης, στέλνει σήμα σε ένα (signaling) ή περισσότερα (broadcast) threads να ξυπνήσουν. Έτσι η μεταβλητή συνθήκης του παραπάνω παραδείγματος παίρνει δύο τιμές, γεμάτη/άδεια.

4.6 Ορατότητα μνήμης μεταξύ threads

Τα pthreads παρέχουν μερικούς βασικούς κανόνες για την ορατότητα μνήμης:

1. Οποιεσδήποτε μεταβλητές μπορεί να «δει» ένα thread τη στιγμή που καλεί τη συνάρτηση `pthread_create( )` μπορούν να είναι ορατές από το νέο thread μετά τη γέννησή του. Οποιαδήποτε μεταβλητή οριστεί μετά από την κλήση της `pthread_create( )` δεν είναι ορατή από το νέο thread.

2. Οποιεσδήποτε μεταβλητές μπορεί να «δει» ένα thread τη στιγμή που ξεκλειδώνει ένα mutex είτε άμεσα είτε περιμένοντας με μια μεταβλητή συνθήκης είναι ορατές από οποιοδήποτε thread που αργότερα κλειδώνει το ίδιο mutex.

3. Οποιεσδήποτε μεταβλητές μπορεί να «δει» ένα thread τη στιγμή που αναμεταδίδει με σήμα την τιμή μιας μεταβλητής συνθήκης, είναι ορατές από οποιοδήποτε thread ξυπνά απ΄ αυτό το σήμα.

4.7 Μοντέλα εργασίας με threads

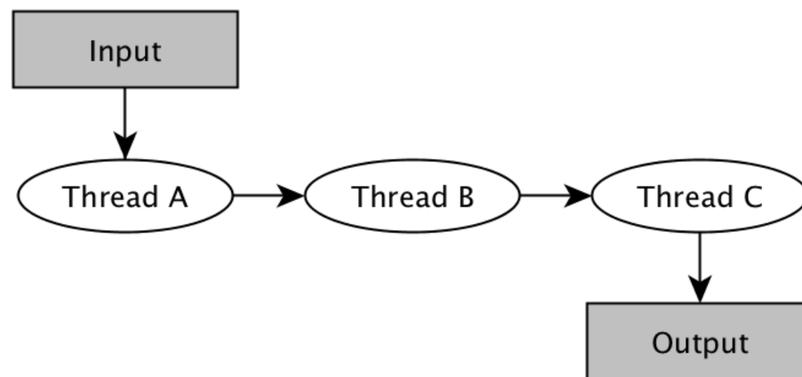
Υπάρχουν άπειροι συνδυασμοί για τη συνεργασία μεταξύ threads αλλά οι κυριότεροι είναι:

- Γραμμή παραγωγής (Pipeline ή assembly line)
- Πελάτης/Εξυπηρετητής (Client-Server)
- Ομάδα εργασίας (Work Crew)

Όλα τα παραπάνω μοντέλα μπορούν να συνδυαστούν αυθαίρετα και να τροποποιηθούν ώστε να ταιριάζουν σε συγκεκριμένα προβλήματα.

4.7.1 Γραμμή παραγωγής

Στη γραμμή παραγωγής (Σχήμα 4.3), ένα ρεύμα από «τεμάχια δεδομένων» επεξεργάζονται σειριακά από ένα διατεταγμένο σύνολο threads. Κάθε thread κάνει με τη σειρά του μια συγκεκριμένη δουλειά πάνω στο «τεμάχιο» περνώντας το στο επόμενο.

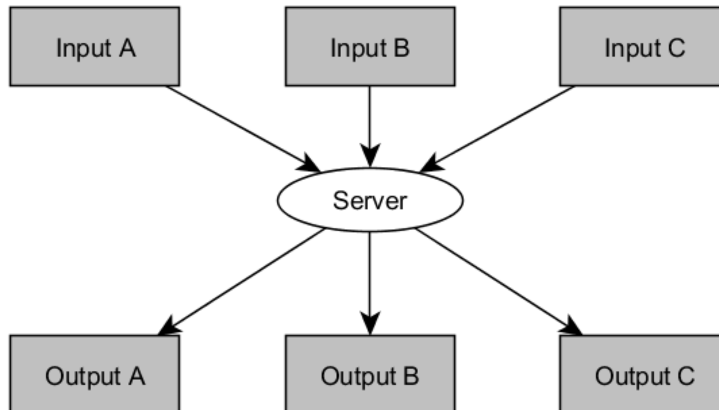


Σχήμα 4.3: Γραμμή παραγωγής. Κάθε thread επανειλημμένα εκτελεί την ίδια λειτουργία πάνω σε μια ακολουθία δεδομένων, περνώντας το αποτέλεσμα στο επόμενο thread.

4.7.2 Πελάτης/Εξυπηρετητής (Client-Server)

Στο μοντέλο αυτό (Σχήμα 4.4), ένας πελάτης ζητάει από έναν εξυπηρετητή να κάνει μια δουλειά πάνω σε ένα σύνολο δεδομένων. Ο εξυπηρετητής εκτελεί τη

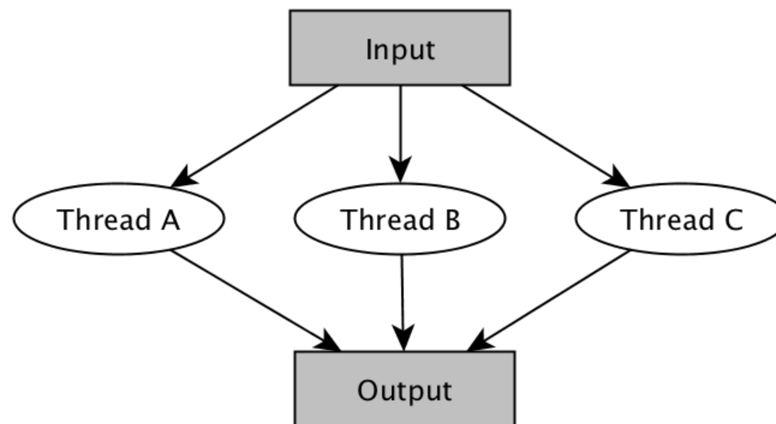
λειτουργία ανεξάρτητα, ο πελάτης μπορεί είτε να περιμένει, είτε να προχωρήσει με μια άλλη εργασία και να επανέλθει για το αποτέλεσμα όταν το χρειαστεί. Παρ΄ όλο που είναι απλούστερο για τον πελάτη να περιμένει, αυτό δεν είναι σχεδόν ποτέ χρήσιμο διότι δεν δίνει κανένα πλεονέκτημα ταχύτητας.



Σχήμα 4.4: Πελάτης/εξυπηρετητής. Ένας πελάτης «υπογράφει συμβόλαιο» με έναν ανεξάρτητο εξυπηρετητή για μια δουλειά. Συχνά το «συμβόλαιο» είναι ανώνυμο - δημιουργείται δηλαδή ένα αίτημα μέσω μιας διεπιφάνειας που απλώς τοποθετεί το αίτημα στην ουρά μέχρι την ικανοποίησή του.

4.7.3 Ομάδα εργασίας

Σε μια ομάδα εργασίας (Σχήμα 4.5), τα δεδομένα επεξεργάζονται ανεξάρτητα από μια ομάδα threads. Η «παράλληλη αποσύνθεση» ενός βρόχου συχνά εμπίπτει σ΄ αυτήν την κατηγορία. Σ΄ αυτήν την κατηγορία εμπίπτει επίσης κι ο κώδικας της παρούσας εργασίας. Ένα σύνολο δεδομένων (το ενιαίο πλέγμα) διαμερίζεται σε κομμάτια (υποχωρία) και μοιράζεται στα threads. Επειδή όλα τα threads μιας ομάδας εργασίας εκτελούν την ίδια λειτουργία, το καθένα πάνω στο κομμάτι του, δηλαδή σε διαφορετικά δεδομένα, αυτό το μοντέλο παράλληλης επεξεργασίας είναι γνωστό και ως **SIMD** (Single Instruction Multiple Data). Ωστόσο, τα threads μιας ομάδας εργασίας δεν είναι υποχρεωτικό να ακολουθούν το μοντέλο SIMD. Μπορούν να εκτελούν εντελώς διαφορετικές λειτουργίες σε διαφορετικά δεδομένα, μοντέλο **MIMD** (Multiple Instruction Multiple Data).



Σχήμα 4.5: Ομάδα εργασίας. Κάθε thread αναλαμβάνει μια δουλειά στο δικό του υποσύνολο δεδομένων. Μπορούν να κάνουν όλα τα threads την ίδια δουλειά (SIMD) ή διαφορετική (MIMD).

Κεφάλαιο 5

Συστοιχίες καρτών γραφικών

Η διαθέσιμη μνήμη, των σημερινών GPUs (3 GB για την TESLA M2050) δεν επαρκεί για την επίλυση προβλημάτων αεροδυναμικής μεγάλης κλίμακας. Για την επίλυση τέτοιων προβλημάτων, μπορούν να χρησιμοποιηθούν συστοιχίες καρτών γραφικών. Ως **συστοιχία καρτών γραφικών** (στο εξής συστοιχία GPUs ή GPU cluster) νοείται ένας υπολογιστικός κόμβος με πολλές GPUs ή, πολλοί υπολογιστικοί κόμβοι (διασυνδεδεμένοι μεταξύ τους) με πολλές ή από μία GPUs [7]. Το αντικείμενο της παρούσας εργασίας αφορά έναν υπολογιστικό κόμβο με πολλές GPUs.

Ωστόσο, ο προγραμματισμός σε συστοιχίες GPUs, έχει πλεονεκτήματα, ακόμα κι όταν εφαρμόζεται σε περιπτώσεις που η μνήμη μιας GPU είναι επαρκής. Αυτό βέβαια ισχύει με την προϋπόθεση ότι το κόστος της επικοινωνίας μεταξύ των GPUs που εμπλέκονται δεν θα είναι τέτοιο που να καθιστά την παράλληλη επιτάχυνση μικρότερη από αυτήν της μίας GPU.

Συστήματα με πολλές GPUs εξελίσσονται και γίνονται ολοένα και πιο διαδεδομένα τα τελευταία χρόνια. Πλέον, υπάρχουν συστήματα πολλών GPUs που εγκαθίστανται σε οικιακούς υπολογιστές. Κάρτες γραφικών όπως η GeForce GTX 690 της NVIDIA ή η Radeon HD 7990 της AMD θεωρούνται, την περίοδο που γράφεται αυτό το κείμενο, ως οι «ναυαρχίδες» των πολυτελών παιχνιδομηχανών και ενσωματώνουν 2 GPUs σε μία κάρτα γραφικών. Στην κατηγορία του επιστημονικού προγραμματισμού, η NVIDIA TESLA S1070 ενσωματώνει 4 GPUs πάνω σε μία κάρτα.

Η συστοιχία GPUs της ΜΠΥΡ&Β του ΕΘΣ, στην οποία και δοκιμάστηκε η παρούσα εργασία, αποτελείται από 4 υπολογιστικούς κόμβους, ο καθένας εκ των οποίων διαθέτει 3 TESLA M2050.

5.1 Χειρισμός πολλών GPUs

Όταν άρχισε να γράφεται ο κώδικας για την παρούσα εργασία, η τότε τρέχουσα έκδοση του περιβάλλοντος προγραμματισμού της CUDA (3.x) επέβαλε κάθε GPU να ελέγχεται από ένα CPU-thread. Επομένως, ο κώδικας έπρεπε να είναι πολυνηματικός είτε με χρήση pthreads (κεφάλαιο 4) είτε με directives OpenMP ([25, 26]). Άρα προβλεπόταν η δημιουργία μέσω pthreads των CPU-threads καθένα απ΄ τα οποία χειριζόταν μια GPU. Επελέγη η προσέγγιση των pthreads διότι δίνει πολύ μεγαλύτερο έλεγχο στον προγραμματιστή και έχει υψηλότερη απόδοση από το OpenMP. Λίγους μήνες αργότερα, έγινε διαθέσιμη η 4η έκδοση του περιβάλλοντος προγραμματισμού της CUDA (τρέχουσα έως την περίοδο που γράφεται αυτό το κείμενο) που επιτρέπει σε ένα CPU-thread τη διαχείριση πολλών GPUs. Ωστόσο, ο κώδικας της εργασίας ήταν ήδη πολυνηματικός και αφού είχε ήδη αποκτηθεί η εμπειρία προγραμματισμού με pthreads, αποφασίστηκε να συνεχιστεί έτσι.

5.2 Προσπέλαση μνήμης του υπολογιστή

Στην ενότητα 3.5 παρουσιάστηκαν οι μνήμες εσωτερικά μίας GPU. Τα threads μίας GPU όμως έχουν τη δυνατότητα της απευθείας πρόσβασης και στη μνήμη του υπολογιστή. Συνήθως, ζητείται από τα threads μίας GPU η προσπέλαση της μνήμης του υπολογιστή, όταν χρειάζεται η συνεργασία CPU-GPU ή η επικοινωνία μεταξύ threads διαφορετικών GPUs. Αυτό είναι και το χαρακτηριστικό που τις καθιστά απαραίτητες για τον προγραμματισμό συστοιχίας καρτών γραφικών.

Το περιβάλλον προγραμματισμού της CUDA επιτρέπει τη δέσμευση θέσεων στη μνήμη του υπολογιστή (RAM), καθιστώντας ταυτόχρονα τις θέσεις αυτές άμεσα προσβάσιμες από τα threads μίας GPU. Προκειμένου να γίνει αυτό, εξασφαλίζεται οτί αυτό το τμήμα μνήμης υπολογιστή που θα δεσμευτεί δεν πρόκειται μελλοντικά να μετακινηθεί στον σκληρό δίσκο. Η προσωρινή μετακίνηση τμημάτων μνήμης RAM σε ειδικά διαμορφωμένη περιοχή του σκληρού δίσκου (swap για LINUX ή Page File για Windows), είναι μια τακτική που χρησιμοποιούν όλα τα λειτουργικά συστήματα όταν κινδυνεύει να γεμίσει η RAM και να μην υπάρχει χώρος για τις διεργασίες. Αυτή η μετακίνηση ονομάζεται paging. Γι΄ αυτό η μνήμη υπολογιστή που δεσμεύεται από την GPU ονομάζεται Page-Locked, ή Pinned (καρφιτσωμένη) μνήμη. Προφανώς, η πρόσβαση σε pinned θέσεις μνήμης είναι αρκετά πιο αργή σε σχέση με την πρόσβαση στη global μνήμη της GPU. Είναι όμως πιο γρήγορη από τις συνεχείς αντιγραφές

από/προς την όχι page-locked μνήμη του υπολογιστή. Πρέπει να τονιστεί πως η χρήση της page-locked μνήμης δεν είναι πανάκεια. Αν κατά τη λειτουργία του προγράμματος το λειτουργικό σύστημα «ξεμείνει» από μνήμη, απαγορεύεται να μετακινήσει στο δίσκο τις θέσεις που δεσμεύτηκαν από τη GPU με αποτέλεσμα να καταστρέφονται όποιες διεργασίες είναι εκείνη τη στιγμή ενεργές. Άρα η χρήση της page-locked μνήμης πρέπει να είναι λελογισμένη και να έχει εκτιμηθεί εκ των προτέρων το παραπάνω ρίσκο. Η ταχύτητα προσπέλασης pinned θέσεων μνήμης εξαρτάται από την ταχύτητα μεταφοράς δεδομένων του διαύλου (PCIe). Το βέλτιστο πρότυπο προσπέλασης pinned θέσεων μνήμης είναι ίδιο με εκείνο της κεντρικής μνήμης της GPU.

Στον παράλληλο GPU-κώδικα που αναπτύχθηκε, η page-locked μνήμη αξιοποιείται για την αποθήκευση των δεδομένων ροής μόνο των κόμβων επικοινωνίας μεταξύ των υποχωρίων. Αυτό λύνει το πρόβλημα της μεταφοράς δεδομένων ροής μεταξύ των GPUs αφού όλες μπορούν να έχουν πρόσβαση στη μνήμη του υπολογιστή. Παράδειγμα δέσμευσης page-locked μνήμης δίνεται με τη χρήση της αντίστοιχης συνάρτησης `cudaHostAlloc()` στον κώδικα 5.1 της επόμενης παραγράφου.

5.3 Συνδυασμός pthreads με CUDA

Για να γίνει εύκολα κατανοητό το πως συνδυάζονται τα pthreads με CUDA, μπορεί να επεκταθεί το παράδειγμα του εσωτερικού γινομένου διανυσμάτων για N GPUs. Στην προκειμένη περίπτωση $N = 3$.

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>

// Handle NULL assignments (malloc, new, etc)
#define HANDLE_NULL(call) \
    do { \
        if ((call) == NULL) { \
            fprintf(stderr, "Resource allocation error in %s " \
                    "at line %d\n", __FILE__, __LINE__); \
            exit(EXIT_FAILURE); \
        } \
    } while(0)

static void HandleError(cudaError_t err, const char *file,
                        int line);

// Handle GPU errors
static void HandleError(cudaError_t err, const char *file,
                        int line)
{
```

```
    if (err != cudaSuccess)
    {
        fprintf(stderr, "%s in %s at line %d\n",
                cudaGetErrorString( err ), file, line);
        exit(EXIT_FAILURE);
    }
}
#define HANDLE_ERROR(err) (HandleError(err, __FILE__, __LINE__))

#define   N      (33*1024*1024)
#define   NGPU   3

#define sum_squares(x) (x*(x+1)*(2*x+1)/6)
#define imin(a,b) (a<b?a:b)

const int threadsPerBlock = 256;
const int blocksPerGrid = imin(32,
          (N/NGPU+threadsPerBlock-1)/threadsPerBlock);

__global__ void dot( int size, double *a, double *b, double *c );

// =========================================================
__global__ void dot( int size, double *a, double *b, double *c )
// =========================================================
{
    __shared__ double cache[threadsPerBlock];
    int tid = threadIdx.x + blockIdx.x * blockDim.x;
    int cacheIndex = threadIdx.x;
    double temp = 0.0;

    while (tid < size)
    {
        temp += a[tid] * b[tid];
        tid  += blockDim.x * gridDim.x;
    }
    // set the cache values
    cache[cacheIndex] = temp;

    // synchronize threads in this block
    __syncthreads();

    // for reductions, threadsPerBlock must be a power of 2
    // because of the following code
    int i = blockDim.x/2;
    while (i != 0)
    {
        if (cacheIndex < i) cache[cacheIndex] += cache[cacheIndex + i];
        __syncthreads();
        i /= 2;
    }
    if (cacheIndex == 0)
        c[blockIdx.x] = cache[0];
}
struct DataStruct
{
    int     deviceID;
    int     size;
    int     offset;
    double   *a;
    double   *b;
    double   returnValue;
};
```

```
void* routine(void *pvoidData);

// ===========================================================
void* routine(void *pvoidData)
// ===========================================================
{
    DataStruct *data = (DataStruct*)pvoidData;

    HANDLE_ERROR(cudaSetDevice(data->deviceID));

    int    size = data->size;
    double *a, *b, c, *partial_c;
    double *dev_a, *dev_b, *dev_partial_c;

    // allocate memory on the CPU side for partial sums
    a = data->a;
    b = data->b;
    HANDLE_NULL(partial_c = (double*)
                malloc(blocksPerGrid*sizeof(double)));

    // Get a pointer useful to the GPU
    HANDLE_ERROR(cudaHostGetDevicePointer(&dev_a, a, 0));
    HANDLE_ERROR(cudaHostGetDevicePointer(&dev_b, b, 0));

    // allocate memory on the GPU side for partial sums
    HANDLE_ERROR(cudaMalloc((void**)&dev_partial_c,
                 blocksPerGrid*sizeof(double)));

    // offset 'a' and 'b' to where this GPU gets its data
    dev_a += data->offset;
    dev_b += data->offset;

    dot<<<blocksPerGrid,threadsPerBlock>>>( size, dev_a, dev_b,
                                            dev_partial_c );

    // copy the array 'c' back from the GPU to the CPU
    HANDLE_ERROR(cudaMemcpy(partial_c, dev_partial_c,
                 blocksPerGrid*sizeof(double),
                 cudaMemcpyDeviceToHost));

    // finish up on the CPU side
    c = 0;
    for (int i = 0; i < blocksPerGrid; i++) c += partial_c[i];
    HANDLE_ERROR(cudaFree(dev_partial_c));

    // free memory on the CPU side
    free(partial_c);

    data->returnValue = c;
    return 0;
}
// ===========================================================
int main()
// ===========================================================
{
    double *a, *b;
    HANDLE_ERROR(cudaSetDevice(0));
    HANDLE_ERROR(cudaSetDeviceFlags(cudaDeviceMapHost));

    HANDLE_ERROR(cudaHostAlloc((void**)&a, N*sizeof(double),
                 cudaHostAllocWriteCombined |
```

```
                 cudaHostAllocPortable |
                 cudaHostAllocMapped));

HANDLE_ERROR(cudaHostAlloc((void**)&b, N*sizeof(double),
                 cudaHostAllocWriteCombined |
                 cudaHostAllocPortable      |
                cudaHostAllocMapped));

// fill in the host memory with data
for (int i = 0; i < N; i++) { a[i] = i; b[i] = i*2; }

DataStruct  data[NGPU];

// prepare for multithread
for (int i = 0; i < NGPU; i++)
{
    data[i].deviceID = i;
    data[i].offset   = i * (N/NGPU);
    data[i].size     = N/NGPU;
    data[i].a        = a;
    data[i].b        = b;
}
int res = 0;

// define the pthread instances
pthread_t hostThreads[NGPU-1];

for (int i = 0; i < NGPU-1; i++)
{
    res = pthread_create(&(hostThreads[i]), NULL,
                         routine, (void *)(&data[i]));
    if (res != 0)
    {
        printf("thread creation failed\n");
        exit( EXIT_FAILURE );
    }
}
// No need to create another thread
// Use initial thread (main) as 3rd thread
routine( &(data[0]) );

// Join threads
for (int i = 0; i < NGPU-1; i++)
{
    res = pthread_join(hostThreads[i], NULL);
    if (res != 0)
    {
        printf("pthread_join failed\n");
        exit(EXIT_FAILURE);
    }
}
// free memory on the CPU side
HANDLE_ERROR(cudaFreeHost(a));
HANDLE_ERROR(cudaFreeHost(b));

double prod = 0.0;
for (int i = 0; i < NGPU; i++) prod += data[i].returnValue;

printf("Value calculated:  %.6g\n", prod);
printf("Value check:  %.6g\n", 2 * sum_squares((double)(N-1)));

return 0;
```

```
}
```

Κώδικας 5.1: (dot–3gpu–pthreads.cu) Κώδικας που υπολογίζει το εσωτερικό γινόμενο διανυσμάτων με 3 GPUs χρησιμοποιώντας pthreads.

Δεδομένου ότι έχουν προηγηθεί οι κώδικες 4.2 και 3.1 για pthreads και CUDA αντίστοιχα, ίσως η μοναδική άξια σχολιασμού γραμμή να είναι η 93. Με την κλήση της συνάρτησης `cudaSetDevice( )` μέσα στη `routine( )` ενεργοποιείται η αντίστοιχη GPU που θα ελέγχεται από το pthread. Όποιες εκτελέσεις kernels ακολουθήσουν τη γραμμή αυτή, τις αναλαμβάνει η συγκεκριμένη ενεργή GPU.

Περίπου το ίδιο μοτίβο κυριαρχεί και στον κώδικα της παρούσας εργασίας, αν όπου διανύσματα τεθεί το ενιαίο πλέγμα και όπου υποδιανύσματα τεθούν τα υποχωρία του. Η διαφορά όμως με το παράδειγμα, είναι ότι στην εργασία η `routine( )` περιέχει την τοπολογία πλέγματος και τον επιλύτη σε CUDA και κάνει εντατική χρήση των τεχνικών συγχρονισμού.

5.4 Υλοποίηση της παραλληλίας

Στόχος είναι η παραλληλοποίηση σε συστοιχία GPUs του 3Δ επιλύτη Euler για δομημένα πλέγματα που αναπτύχθηκε στο πλαίσιο της διπλωματικής εργασίας [8]. Επισημαίνεται ότι με την έννοια της παραλληλοποίησης υπονοείται η παράλληλη εκτέλεση ίδιας σειράς εντολών σε διαφορετικές GPUs που η καθεμία χειρίζεται διαφορετικά δεδομένα (λογική SIMD). Ούτως ή άλλως, ο κώδικας για μία GPU θεωρείται παράλληλος, διότι κάθε GPU-thread αναλαμβάνει την επίλυση της ροής σε έναν κόμβο του υποχωρίου. Άρα στο εξής, ο κώδικας της παρούσας εργασίας, για συστοιχίες GPUs, θα αναφέρεται ως παράλληλος GPU κώδικας χωρίς αυτό να σημαίνει ότι ο κώδικας για μία GPU είναι σειριακός. Η ιδέα πίσω από την επέκταση σε συστοιχία καρτών γραφικών, είναι να γίνει, αρχικά, ένας διαμερισμός του πλέγματος σε υποχωρία, και να ανατεθεί ένα υποχωρίο ανά GPU. Ο υπάρχων GPU επιλύτης έπρεπε να επαναπρογραμματιστεί με στόχο την ανταλλαγή δεδομένων και τον συγχρονισμό των εργαζόμενων GPUs όταν αυτό είναι απαραίτητο. Τα παραπάνω αναλύονται στην επόμενη παράγραφο.

5.4.1 Τρόπος διαμερισμού υπολογιστικού πλέγματος

Η ανάγκη διαμερισμού του πλέγματος σε υποχωρία γεννάει την ανάγκη ανταλλαγής δεδομένων ροής ή «επικοινωνίας» μεταξύ των υποχωρίων. Οι επικοινωνίες μεταξύ GPU σημαίνουν συνεχείς μεταφορές/αντιγραφές δεδομένων μέσω του διαύλου PCIe. Ωστόσο, οι μεταφορές δεδομένων αποτελούν την κυριότερη αδυναμία των GPU δεδομένου ότι το εύρος ζώνης του διαύλου PCIe είναι περιοριστικό (για PCIe x16 Gen2 είναι μόλις 8 GB/sec έναντι 148 GB/sec για την εσωτερική global μνήμη μιας Tesla M2050). Γι' αυτό και η κυρίαρχη τάση (για την ακρίβεια, ένας από τους χρυσούς κανόνες) στον προγραμματισμό GPUs είναι να τοποθετούνται τα δεδομένα στη GPU και να διατηρούνται εκεί όσο αυτό είναι δυνατό (get the data on the GPU and keep it there [19]). Δεδομένου ότι κάτι τέτοιο έρχεται σε αντίθεση με την ανάγκη για επικοινωνία, η βέλτιστη λύση που υιοθετήθηκε ήταν ο διαμερισμός του πλέγματος σε **επικαλυπτόμενα υποχωρία** (overlapped subdomains). Για τον διαμερισμό χρησιμοποιήθηκε μια απλή εφαρμογή που αναπτύχθηκε στο πλαίσιο της παρούσας εργασίας.

Τα σχήματα 5.2 έως 5.5 δείχνουν τον διαμερισμό ενός δομημένου 2Δ υπολογιστικού πλέγματος σε 2 επικαλυπτόμενα υποχωρία. Στο σχήμα 5.3 φαίνεται η κατανομή κόμβων μεταξύ των υποχωρίων αν δεν υπήρχε επικάλυψη. Οι κόμβοι επαφής ή επικοινωνίας των 2 υποχωρίων, που θα εμπλακούν στην ανταλλαγή δεδομένων, χρωματίζονται γκρι (όπως στο σχήμα 5.4) διότι ανήκουν και στα δύο υποχωρία. Στο σχήμα 5.5 παρατηρείται η ταξινόμηση των κόμβων ανάμεσα σε κόμβους «αποστολής» (send) και «υποδοχής» (receive). Αυτοί ορίζονται ως κόμβοι επικοινωνίας, καθώς οι ροϊκές ποσότητες που αποθηκεύονται στους κόμβους αυτούς ανταλλάσσονται μεταξύ των συνεργαζόμενων GPUs.

Οι κόμβοι αποστολής περιέχουν την πληροφορία που θα στείλει ένα υποχωρίο στο γειτονικό του (και αποτελούν ταυτόχρονα κόμβους υποδοχής για το γειτονικό του), ενώ οι κόμβοι υποδοχής περιέχουν την πληροφορία που θα παραλάβει ένα υποχωρίο από το γειτονικό του (και αποτελούν ταυτόχρονα κόμβους αποστολής για το γειτονικό του). Αυτή η δυαδικότητα της ιδιότητας των κόμβων επιτρέπει και την επαλήθευση του αν έγιναν σωστά οι επικοινωνίες κατά τη διάρκεια της εκσφαλμάτωσης (debugging) του προγράμματος (ότι στέλνει ένα υποχωρίο πρέπει να το παραλαμβάνει το γειτονικό του και αντίστροφα).

Οι κόμβοι υποδοχής χρησιμοποιούνται, πέραν της επικοινωνίας μεταξύ των συνεργαζόμενων GPUs, μόνο για την αποθήκευση των ροϊκών μεταβλητών. Δηλαδή, μένουν εκτός της ολοκλήρωσης των εξισώσεων ροής, η οποία γίνεται στους όγκους ελέγχου που σχηματίζονται γύρω από τους «αυστηρά εσωτερικούς» κόμβους των υποχωρίων (δηλαδή τους κόμβους που θα αποτελούσαν

το υποχωρίο αν δεν υπήρχε η επικάλυψη). Συνεπώς, δεν αυξάνεται ο χρόνος επίλυσης ανά υποχωρίο, αφού δεν γίνονται διπλοί υπολογισμοί. Επιπλέον, από τη στιγμή που οι κόμβοι υποδοχής είναι «ανενεργοί» κατά τον υπολογισμό και ισολογισμό των αριθμητικών διανυσμάτων της ροής στους όγκους ελέγχου, δεν απαιτείται ο υπολογισμός και η αποθήκευση των ιακωβιανών μητρώων σε αυτούς. Άρα, ως προς τη μνήμη, το επιπλέον κόστος χρήσης επικαλυπτόμενων ζωνών είναι μικρό.

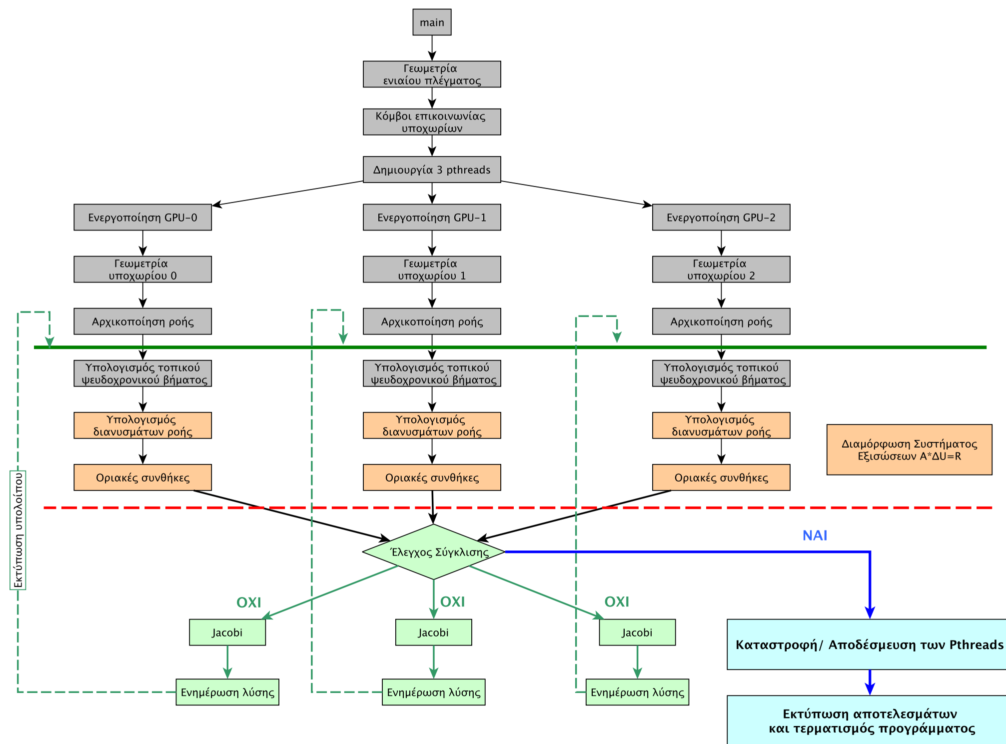
5.5 Ανάθεση παράλληλων εργασιών στις GPUs και συγχρονισμός

Εκτός από την ταυτοποίηση/αναγνώριση των κόμβων επικοινωνίας, υπάρχει το επόμενο στάδιο, της ανάθεσης εργασιών στις ενεργές GPUs. Στο διάγραμμα ροής του σχήματος 5.1 φαίνεται πώς γίνεται αυτό. Αρχικά υπολογίζονται ορισμένες γεωμετρικές ποσότητες ως προς το ενιαίο πλέγμα, όπως οι επιφάνειες εισόδου/εξόδου του υπολογιστικού χωρίου. Μετα προσδιορίζονται όλοι οι κόμβοι επικοινωνίας, προκειμένου αυτά τα δεδομένα να μεταδοθούν αργότερα στα υποχωρία. Στη συνέχεια δημιουργούνται τα pthreads και αναλαμβάνουν καθήκοντα. Το καθένα ενεργοποιεί την αντίστοιχη GPU, υπολογίζει τα γεωμετρικά χαρακτηριστικά του υποχωρίου που του ανατέθηκε (όγκους κελιών, κάθετα διανύσματα στα όρια των κελιών κτλ) και τελικά μεταφέρει τα δεδομένα αυτά στη μνήμη της GPU όπου και θα παραμείνουν μέχρι την ολοκλήρωση του προγράμματος. Έπειτα, αρχικοποιεί τη ροή σε όλους τους κόμβους και καλεί kernels που υπολογίζουν διαδοχικά το τοπικό ψευδοχρονικό βήμα, τα διανύσματα ατριβούς ροής (σχήμα Roe) και επιβάλουν τις οριακές συνθήκες. Στο τέλος αυτού του βήματος, έχουν σχηματιστεί τα μητρώα του προς επίλυση συστήματος και είναι αναγκαίο να γίνει ο πρώτος συγχρονισμός και επικοινωνία μεταξύ των GPUs (αυτό συμβολίζεται με τα βέλη που συγκλίνουν όλα στον ρόμβο του ελέγχου σύγκλισης).

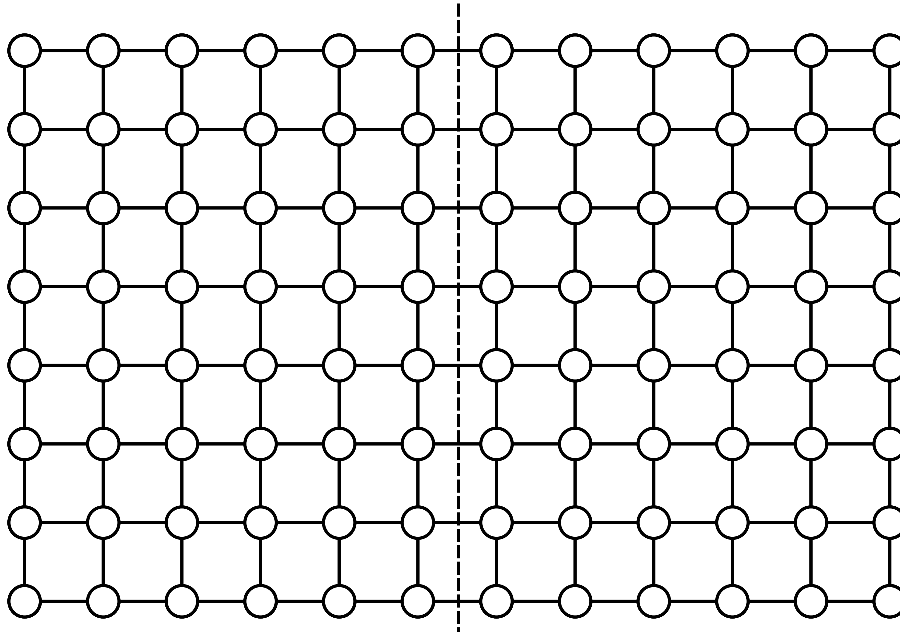
Αυτός επιτυγχάνεται με τη χρήση ενός pthread barrier που προγραμματίστηκε γι' αυτόν τον σκοπό. Το pthread barrier χρησιμοποιεί τις τεχνικές συγχρονισμού (mutexes και condition variables) που περιγράφηκαν στο κεφάλαιο 4. Στον κώδικα που προγραμματίστηκε, η ειδική αυτή συνάρτηση φέρει το όνομα `barrier_wait( )` και για ευκολία στη συνέχεια θα αναφέρεται έτσι. Η `barrier_wait( )` έχει την ίδια συμπεριφορά με τη συνάρτηση `__syncthreads( )` της CUDA (ενότητα 3.9). Υποχρεώνει τα CPU-threads (αντίστοιχα των threads ενός block) να περιμένουν στο σημείο που κλήθηκε η `barrier_wait( )` έως ότου φτάσουν όλα στο ίδιο σημείο. Μόλις συμβε-

ί αυτό συνεχίζουν την εκτέλεση των εντολών που βρίσκονται κάτω από την `barrier_wait( )`. Στο σχήμα 5.1 συμβολίζεται με διακεκομμένη κόκκινη γραμμή.

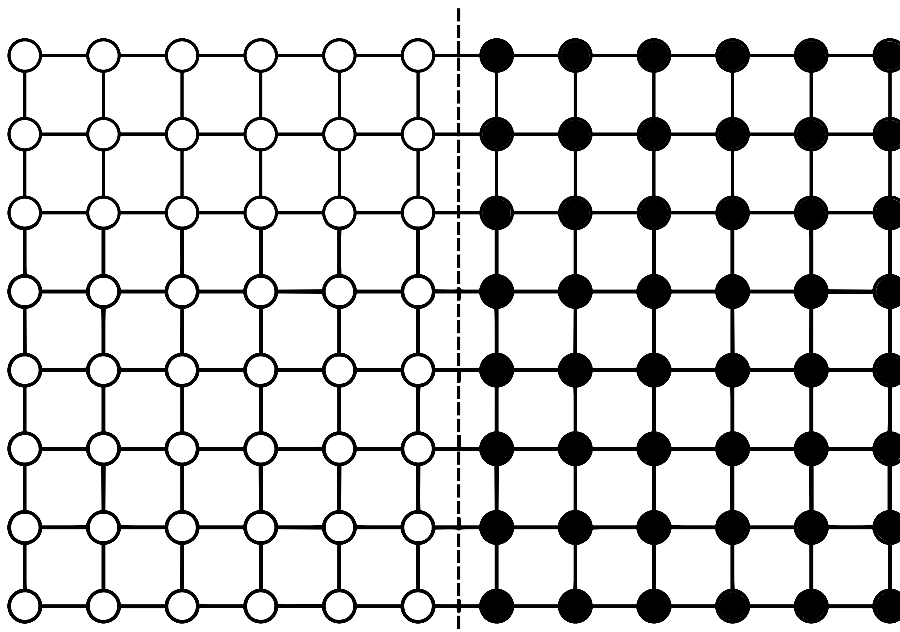
Μόλις φτάσουν όλα τα pthreads στο barrier, επικοινωνούν τα υπόλοιπα των εξισώσεών τους (residuals) ώστε να σχηματιστεί το ενιαίο υπόλοιπο. Αυτό τυπώνεται στην οθόνη από το pthread0. Γίνεται ο έλεγχος σύγκλισης και αν δεν υπάρχει σύγκλιση το κάθε pthread συνεχίζει με την επίλυση των εξισώσεων μέσω της σημειακά πεπλεγμένης μεθόδου Jacobi. Στο εσωτερικό της μεθόδου Jacobi απαιτούνται επιπρόσθετες επικοινωνίες και συγχρονισμοί με barriers, τα οποία δεν φαίνονται στο διάγραμμα λόγω δυσκολίας αναπαράστασης. Στη συνέχεια γίνεται ανανέωση της λύσης και επιστροφή του κάθε pthread στην αρχή του επαναληπτικού βρόχου, δηλαδή στο σημείο που συμβολίζεται με συμπαγή πράσινη γραμμή. Όταν ο κώδικας συγκλίνει, το κάθε pthread αποδεσμεύει το χώρο μνήμης που χρησιμοποιούσε για το πλέγμα και τα δεδομένα ροής στη GPU και στη συνέχεια καταστρέφεται. Το πρόγραμμα εκτυπώνει τα αποτελέσματα κάνοντας αναγωγή στο ενιαίο πλέγμα και τερματίζεται.



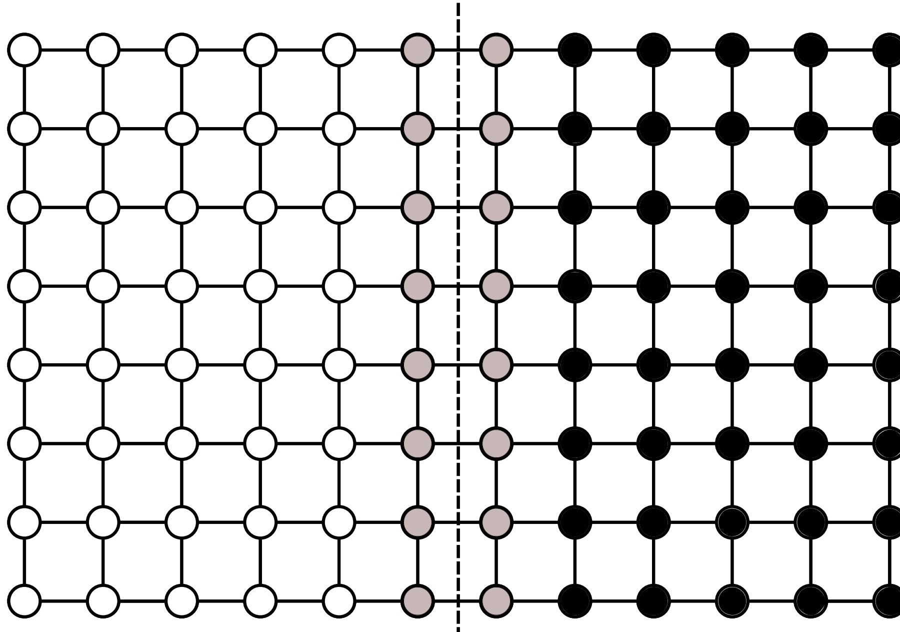
Σχήμα 5.1: Διάγραμμα ροής των εργασιών που εκτελούνται παράλληλα από 3 GPUs στον παράλληλο GPU-κώδικα



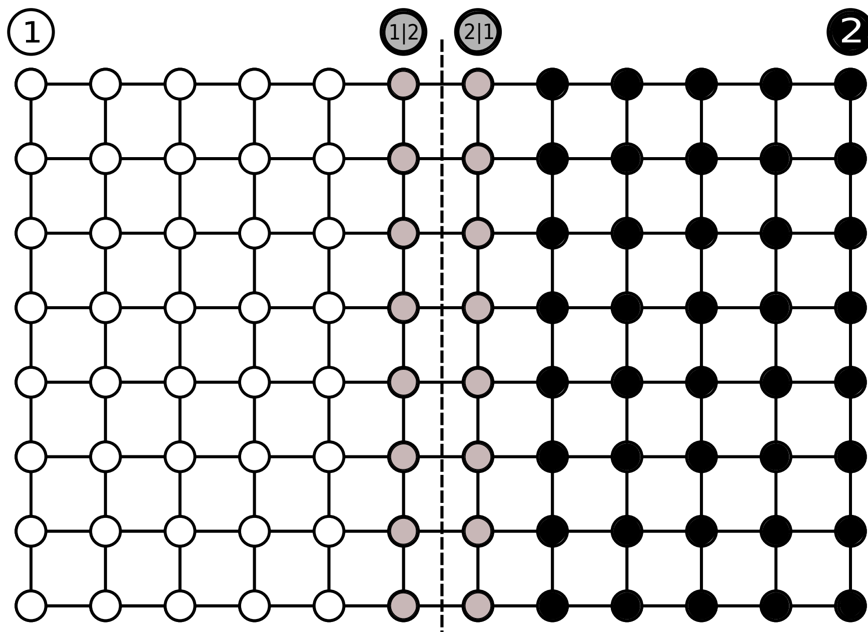
Σχήμα 5.2: Το ενιαίο αρχικό πλέγμα και η γραμμή διαμερισμού



Σχήμα 5.3: Η κατανομή κόμβων στα 2 υποχωρία αν δεν ήταν επικαλυπτόμενα



Σχήμα 5.4: Οι κόμβοι επαφής (επικοινωνίας) των 2 υποχωρίων χρωματίζονται γκρι διότι ανήκουν και στα 2 υποχωρία



Σχήμα 5.5: Αριθμούνται τα υποχωρία (1-άσπρο και 2-μαύρο). Οι κόμβοι με το σύμβολο 1/2 είναι «κόμβοι αποστολής» του 1 στο 2 και αντίστοιχα «υποδοχής» του 2 από το 1. Τα αντίστροφα ισχύουν για τους κόμβους 2/1

Κεφάλαιο 6

Αποτελέσματα και συγκριτικές επιδόσεις

Στο κεφάλαιο αυτό γίνεται παρουσίαση των αποτελεσμάτων του παράλληλου GPU-κώδικα που αναπτύχθηκε καθώς και οι συγκριτικές του επιδόσεις με τον αντίστοιχο κώδικα για μία GPU. Η ανάπτυξη, η πιστοποίηση και η αξιολόγηση του κώδικα πραγματοποιήθηκε στην συστοιχία καρτών γραφικών του ΕΘΣ, η οποία αποτελείται από 12 GPU TESLA M2050, οι οποίες έχουν 3 GB GDDR5 μνήμης, μέγιστη απόδοση 515 GFlops και μέχρι 148 GB/sec ταχύτητα μεταφοράς δεδομένων. Σε κάθε 3 GPU αντιστοιχούν 2 κύριοι επεξεργαστές Quad core Intel Xeon E5620@2.4 ghz, οι οποίοι και έχουν 12 MB λανθάνουσα μνήμη και 16 MB μνήμη RAM. Τα αποτελέσματα και η επιτάχυνση του επιλύτη για μία GPU ως προς τον σειριακό επιλύτη για μία CPU αναλύονται διεξοδικά στη διπλωματική εργασία [8].

Τα αποτελέσματα που παρουσιάζονται στο παρόν κεφάλαιο είναι ταυτόσημα με τα αποτελέσματα που προκύπτουν από τον επιλύτη για μία GPU ο οποίος είναι πιστοποιημένος επιλύτης του ΕΘΣ.

Στη συνέχεια παρουσιάζονται τα αποτελέσματα από δύο διαφορετικά προβλήματα. Πρώτα παρουσιάζονται τα αποτελέσματα από την επίλυση της ροής μέσα σε έναν αγωγό σχήματος S (S duct) και στη συνέχεια τα αποτελέσματα της επίλυσης της ροής μέσα σε έναν αγωγό με γωνία 90° (elbow duct). Τέλος, παρουσιάζεται η σύγκριση ανάμεσα στους χρόνους επίλυσης της μίας GPU και 2-3 GPUs, που είναι και το αντικείμενο της παρούσας διπλωματικής εργασίας.

6.1 Επίλυση ροής μέσα σε αγωγό σχήματος S

Ο αγωγός σχήματος S τετραγωνικής διατομής, μέσα στον οποίο και επιλύεται η ροή, φαίνεται στο σχήμα 6.1 σε πρόοψη και σε τριδιάστατη απεικόνιση. Η ροή επιλύθηκε για μηδενική γωνία εισόδου και $M_{2,is} = 0.6$, όπου $M_{2,is}$ είναι ο ισεντροπικός αριθμός Mach στην έξοδο της ροής. Στο σχήμα 6.2 φαίνεται το δομημένο πλέγμα που χρησιμοποιήθηκε για την επίλυση των εξισώσεων ροής. Το πλέγμα αποτελείται από $(n_x \times n_y) \times n_z = (30 \times 71) \times 151 = 321630$ κόμβους. Πλαισιώνεται με παρενθέσεις το πλήθος κόμβων επικοινωνίας στα όρια μεταξύ υποχωρίων. Εφόσον ο διαμερισμός έγινε κατά τη φορά της ροής (z), οι επιφάνειες επαφής/επικοινωνίας αποτελούνται από $n_x \times n_y$ κόμβους. Στο σχήμα 6.3 παρουσιάζεται το πεδίο του αριθμού Mach της ροής κατά μήκος του αγωγού.

6.2 Επίλυση ροής μέσα σε αγωγό γωνίας 90°

Ο αγωγός τετραγωνικής διατομής με γωνία 90°, μέσα στον οποίο και επιλύεται η ροή, φαίνεται στο σχήμα 6.4 σε πρόοψη και σε τριδιάστατη απεικόνιση. Η ροή επιλύθηκε για μηδενική γωνία εισόδου και $M_{2,is} = 0.6$, όπου $M_{2,is}$ είναι ο ισεντροπικός αριθμός Mach στην έξοδο της ροής. Στο σχήμα 6.5 φαίνεται το δομημένο πλέγμα που χρησιμοποιήθηκε για την επίλυση των εξισώσεων ροής. Το πλέγμα αποτελείται από $(n_x \times n_y) \times n_z = (64 \times 64) \times 250 = 1024000$ κόμβους. Πλαισιώνεται με παρενθέσεις το πλήθος κόμβων επικοινωνίας στα όρια μεταξύ υποχωρίων. Εφόσον ο διαμερισμός έγινε κατά τη φορά της ροής (z), οι επιφάνειες επαφής/επικοινωνίας αποτελούνται από $n_x \times n_y$ κόμβους. Στο σχήμα 6.6 παρουσιάζεται το πεδίο του αριθμού Mach της ροής κατά μήκος του αγωγού.

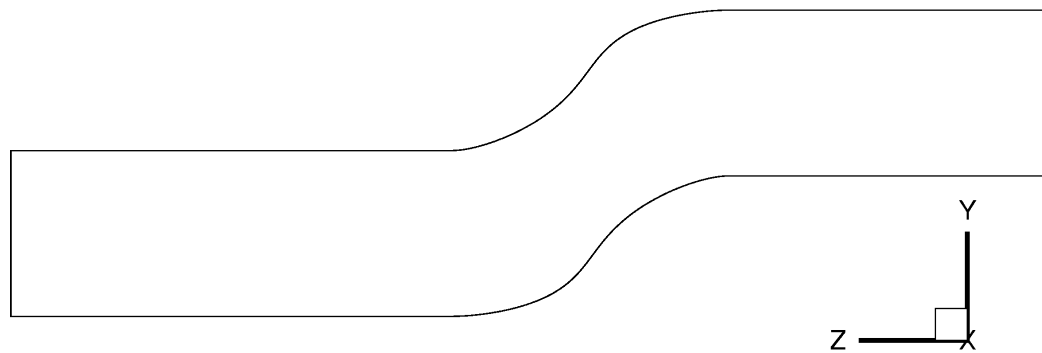
6.3 Συγκριτικές επιδόσεις ανάμεσα σε πολλές GPU και 1 GPU.

Στο σχήμα 6.7 παρουσιάζεται το διάγραμμα της επιτάχυνσης του παράλληλου GPU-κώδικα σε σχέση με τον κώδικα για μία GPU. Όπως φαίνεται και στο σχήμα, ο επιλύτης για συστοιχία 3 GPU είναι έως και 2.7 φορές πιο γρήγορος

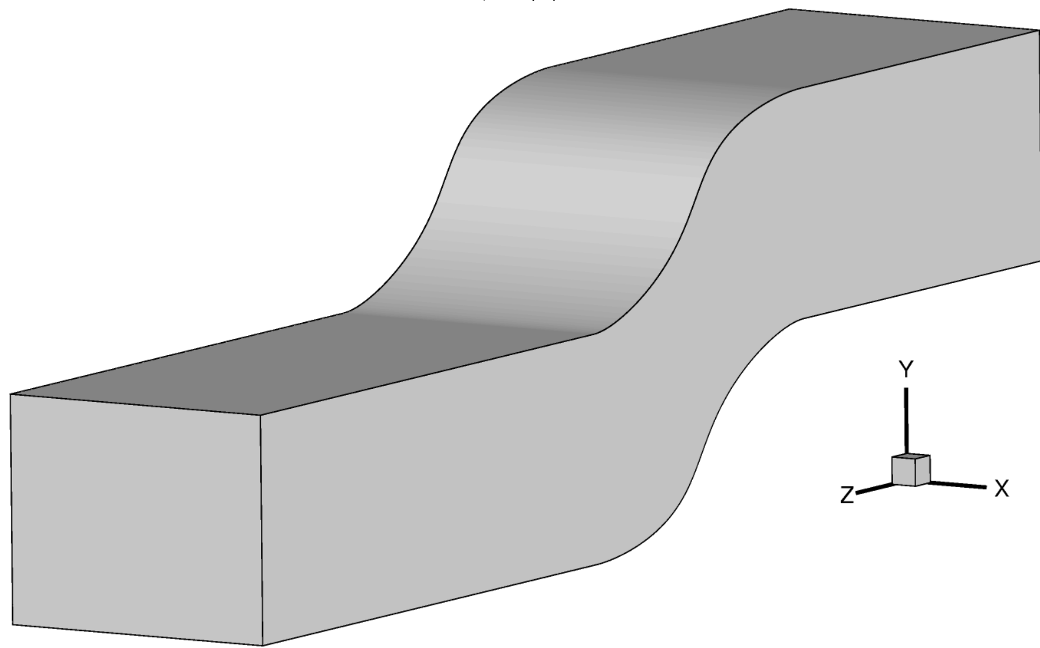
από τον επιλύτη για 1 GPU. Παρατηρείται πως στο πλέγμα της 2ης περίπτωσης η επιτάχυνση είναι μεγαλύτερη. Αυτό συμβαίνει διότι ο λόγος του αριθμού των κόμβων επικοινωνίας προς τους αυστηρά εσωτερικούς κόμβους (σχέσεις (6.1) (6.2)), είναι μικρότερος. Συνεπώς ο λόγος του χρόνου επικοινωνιών προς τον συνολικό χρόνο επίλυσης θα είναι μικρότερος.

$$\frac{64 \text{ x } 64}{64 \text{ x } 64 \text{ x } 250} = \frac{1}{250} \tag{6.1}$$

$$\frac{30 \text{ x } 71}{30 \text{ x } 71 \text{ x } 151} = \frac{1}{151} \tag{6.2}$$

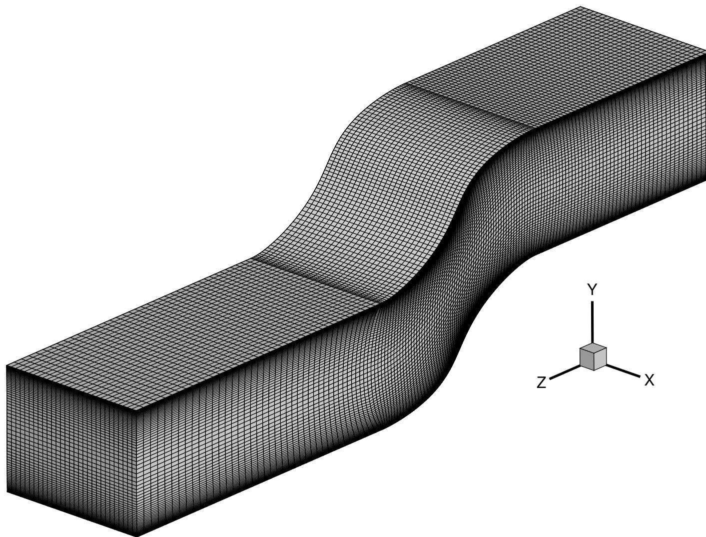


(α′) Πρόοψη αγωγού

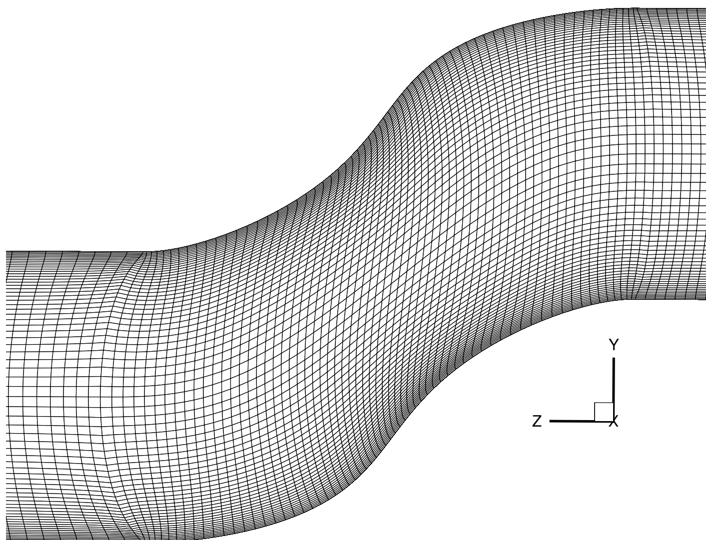


(β′) 3Δ απεικόνιση αγωγού

Σχήμα 6.1: Γεωμετρία αγωγού τύπου S

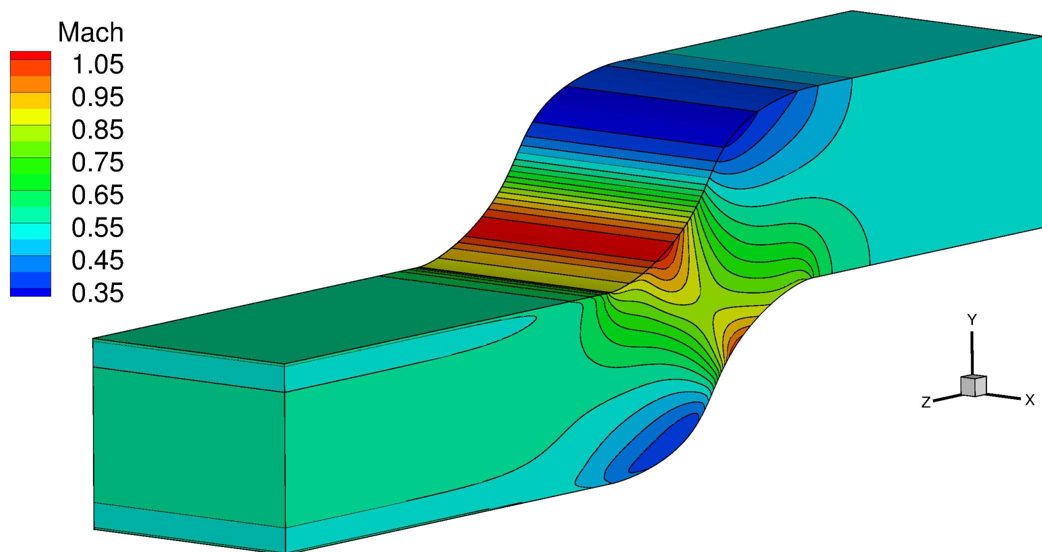


(α′) 3Δ απεικόνιση του πλέγματος του αγωγού

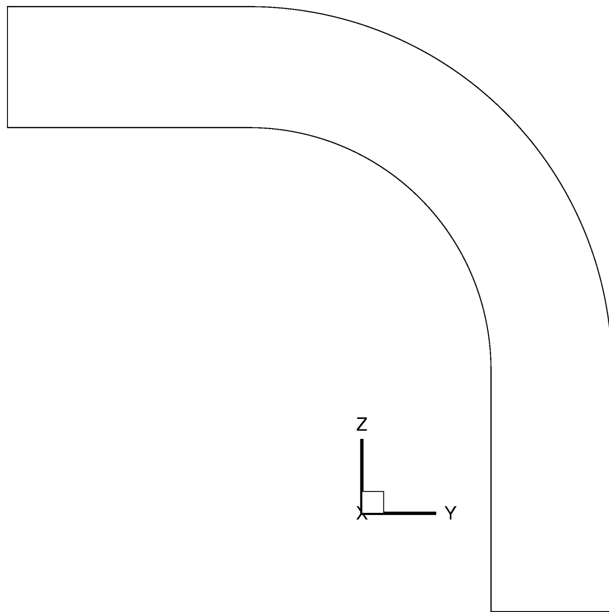


(β′) Μεγέθυνση στο σημείο στροφής της ροής

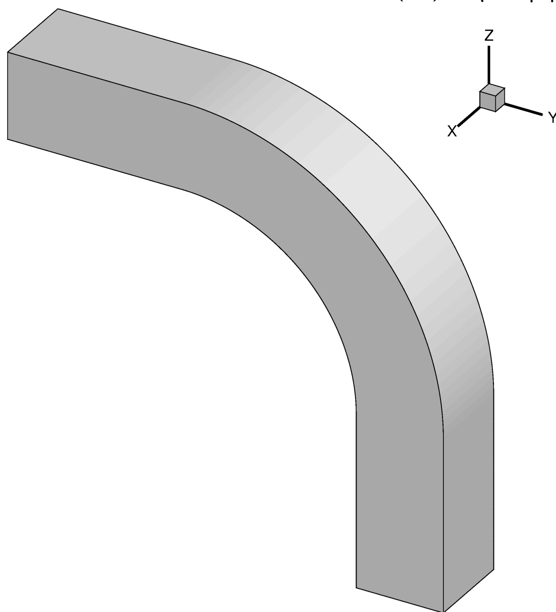
Σχήμα 6.2: Δομημένο υπολογιστικό πλέγμα $n_x \times n_y \times n_z = 30 \times 71 \times 151$ κόμβων για την επίλυση της ροής στον αγωγό τύπου S



Σχήμα 6.3: Πεδίο αριθμού Mach κατά μήκος του αγωγού

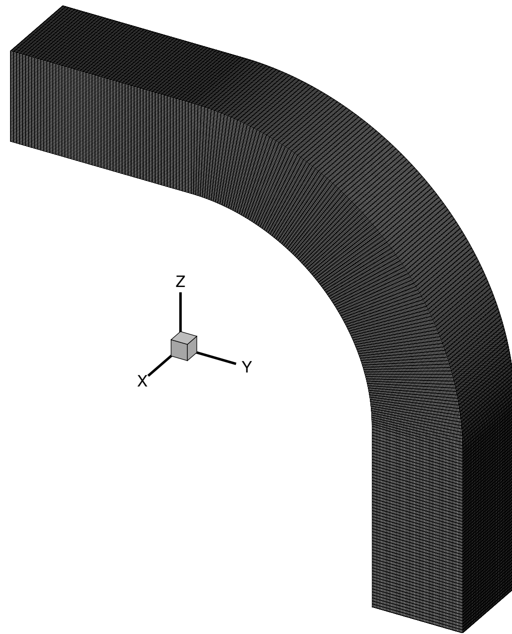


(α′) Πρόοψη αγωγού

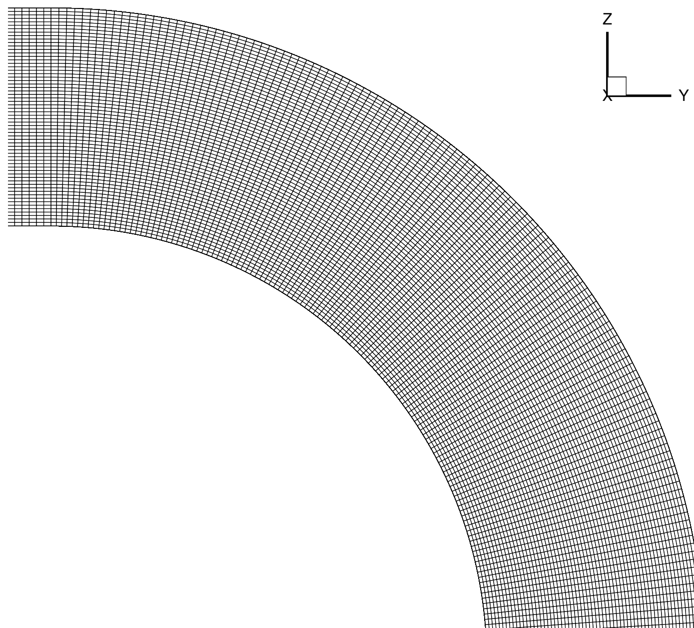


(β′) 3Δ απεικόνιση αγωγού

Σχήμα 6.4: Γεωμετρία αγωγού με γωνία 90°

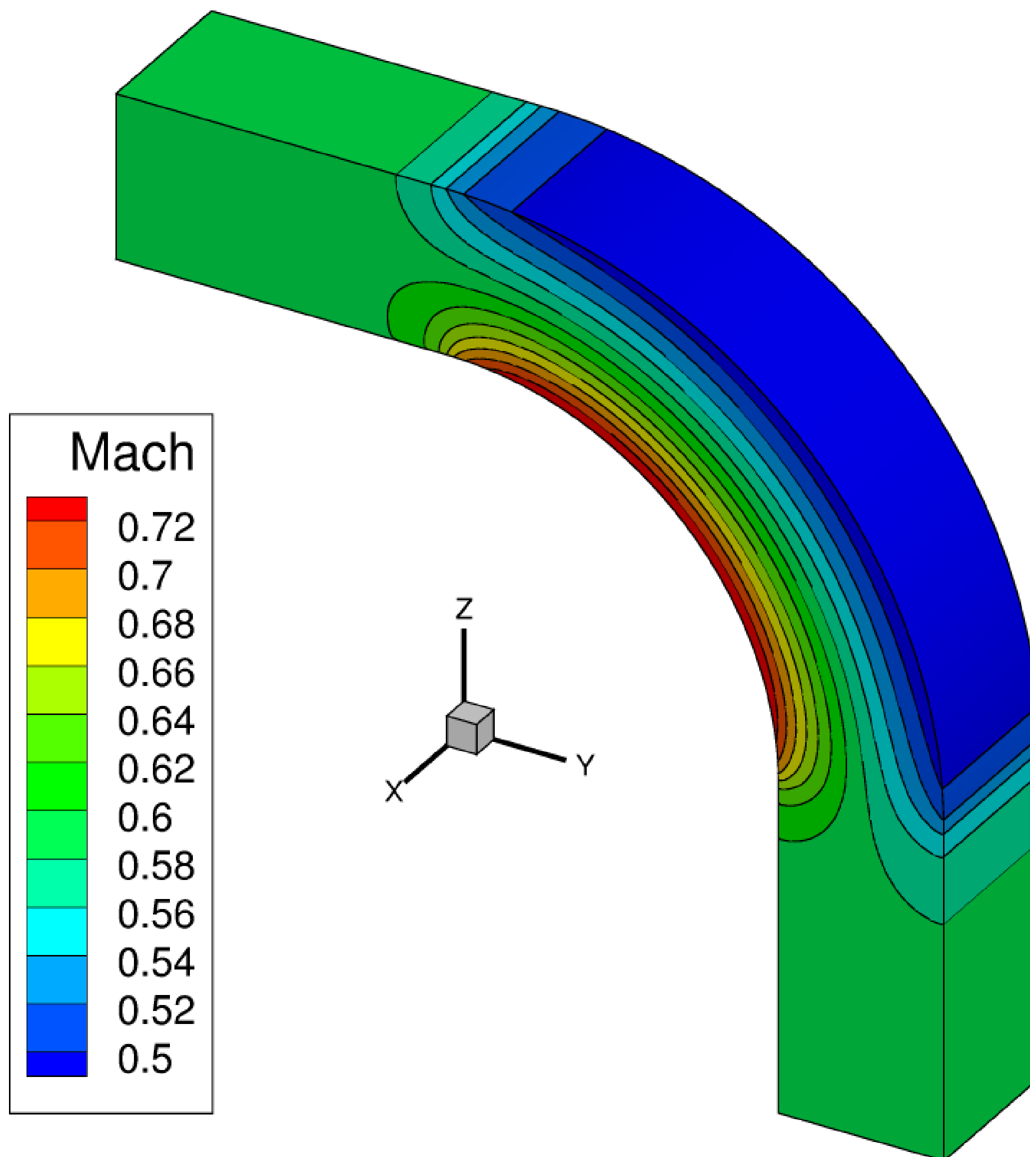


(α′) 3Δ απεικόνιση του πλέγματος του αγωγού

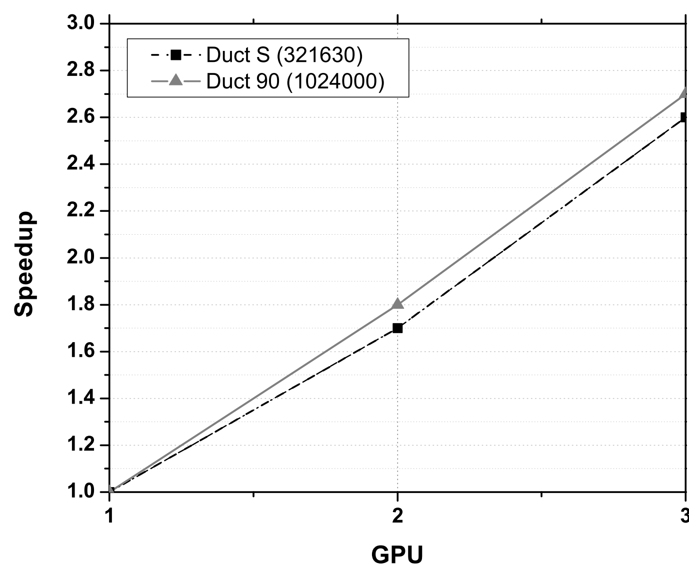


(β′) Μεγέθυνση στο σημείο στροφής της ροής

Σχήμα 6.5: Δομημένο υπολογιστικό πλέγμα $n_x \times n_y \times n_z = 64 \times 64 \times 250$ κόμβων για την επίλυση της ροής στον αγωγό γωνίας 90°



Σχήμα 6.6: Πεδίο αριθμού Mach κατά μήκος του αγωγού



Σχήμα 6.7: Διάγραμμα επιτάχυνσης επιλύτη εξισώσεων Euler για συστοιχία GPUs σε σχέση με 1 GPU.

Κεφάλαιο 7

Ανακεφαλαίωση και συμπεράσματα

Τα τελευταία χρόνια, η Μονάδα Παράλληλης Υπολογιστικής Ρευστοδυναμικής & Βελτιστοποίησης έχει επιτυχώς αναπτύξει κώδικες για την εκμετάλλευση των δυνατοτήτων των GPUs σε εφαρμογές υπολογιστικής ρευστοδυναμικής και βελτιστοποίησης. Στον τομέα της υπολογιστικής ρευστοδυναμικής, με τον οποίο και ασχολείται η παρούσα διπλωματική εργασία, έχουν προκύψει εντυπωσιακά αποτελέσματα σε επιλύτες ροής ατριβούς και συνεκτικού ρευστού με επιταχύνσεις έως και $\times 100$ [1], [2], [7].

Ωστόσο, η διαθέσιμη μνήμη, των σημερινών GPUs αποτελεί εμπόδιο για την επίλυση προβλημάτων αεροδυναμικής μεγάλης κλίμακας. Για την επίλυση τέτοιων προβλημάτων, χρησιμοποιούνται συστοιχίες καρτών γραφικών ώστε να αθροίζονται οι επιμέρους χωρητικότητες μνήμης. Σκοπός αυτής της διπλωματικής ήταν να επεκτείνει/τροποποιήσει έναν υπάρχοντα επιλύτη των 3Δ εξισώσεων Euler για συμπιεστές ροές ώστε να αξιοποιεί τη συστοιχία καρτών γραφικών του εργαστηρίου. Ο προγραμματισμός έγινε σε CUDA C/C++ και χρησιμοποιήθηκαν τεχνικές πολυνηματικού προγραμματισμού με POSIX threads ώστε να επιτευχθεί η συντονισμένη διαχείριση των GPUs της συστοιχίας.

Ο προγραμματισμός για συστοιχία GPUs, σε προβλήματα υπολογιστικής ρευστοδυναμικής, προϋποθέτει τον διαμερισμό του πλέγματος σε υποχωρία, που αποθηκεύονται το καθένα στη μνήμη διαφορετικής GPU. Επιπλέον, αυτό δημιουργεί την ανάγκη ανταλλαγής δεδομένων ροής μεταξύ υποχωρίων, όταν επιλύεται η ροή στους κόμβους επαφής τους. Αυτοί οι κόμβοι ονομάζονται και «κόμβοι επικοινωνίας». Σε κώδικα που προορίζεται για εκτέλεση στις GPUs, επικοινωνία σημαίνει μεταφορά δεδομένων από την κεντρική μνήμη της GPU στην μνήμη του υπολογιστή κι αντίστροφα. Αυτό είναι ότι χειρότερο για την

απόδοση των GPUs που το κυριότερο ίσως μειονέκτημά τους είναι οι χαμηλές ταχύτητες στη μεταφορά/αντιγραφή δεδομένων. Επιπλέον, αφού οι διαφορετικές GPU πρέπει να συντονίζονται και να επικοινωνούν μεταξύ τους στο τέλος κάθε ψευδοχρονικού βήματος, αυτή η επικοινωνία κοστίζει σε χρόνο εκτέλεσης. Επομένως, όσο μικρότερος είναι ο αριθμός των επικοινωνιών μεταξύ υποχωρίων, τόσο βελτιωμένη θα είναι η τελική επιτάχυνση του κώδικα.

Για να ελαχιστοποιηθούν οι επικοινωνίες, επιλέχθηκε η προσέγγιση των επικαλυπτόμενων υποχωρίων, κατά την οποία κάθε υποχωρίο «διεισδύει» στα γειτονικά του κατά μία σειρά κόμβων. Ουσιαστικά «υιοθετεί» τους κόμβους με τους οποίους τα γειτονικά του υποχωρία εφάπτονται σε αυτό. Έτσι, δημιουργούνται ζώνες επικάλυψης όπου το κάθε υποχωρίο αποθηκεύει τα δεδομένα ροής των γειτονικών κόμβων του σαν να ήταν δικοί του κόμβοι. Αυτά τα δεδομένα χρησιμοποιούν οι εσωτερικοί κόμβοι όταν χρειαστούν να υπολογίσουν τα διανύσματα ροής από/προς αυτούς. Έτσι οι επικοινωνίες, περιορίζονται πλέον μόνο στο βήμα που εφαρμόζεται η μέθοδος Jacobi για την επίλυση του συστήματος εξισώσεων.

Από την επίλυση ροών σε αγωγούς με χρήση έως και 3 GPUs, παρατηρήθηκε επιτάχυνση (σε σχέση με τον αντίστοιχο επιλύτη για μία GPU) έως και $\times 2.7$.

Βιβλιογραφία

[1] Asouti V. G., Trompoukis X. S, Kampolis I. C. and K. C., Giannakoglou: *Unsteady CFD computations using vertex-centered finite volumes for unstructured grids on Graphics Processing Units.* International journal for numerical methods in fluids, Int. J. Numer. Meth. Fluids (2010).

[2] Γεώργιος, Βαλσαμάκης: *Αριθμητική επίλυση μη-μόνιμου πεδίου ροής σε κάρτες γραφικών με απεικόνιση του σε «πραγματικό» χρόνο.* Διπλωματική εργασία. Εργαστήριο Θερμικών Στροβιλομηχανών Ε.Μ.Π., 2010.

[3] Ρήγας, Γεώργιος Δ.: *Προσομοίωση και χαμηλού κόστους βελτιστοποίηση του ενεργητικού ελέγχου ροής ρευστού γύρω από αεροτομή, σε κάρτες γραφικών.* Διπλωματική εργασία. Εργαστήριο Θερμικών Στροβιλομηχανών Ε.Μ.Π., 2010.

[4] Φούντης, Ελευθέριος Ι.: *Προγραμματισμός σε Κάρτες Γραφικών και Εφαρμογη στην Αεροδυναμική Βελτιστοποίηση.* Διπλωματική εργασία. Εργαστήριο Θερμικών Στροβιλομηχανών Ε.Μ.Π., 2009.

[5] Kampolis I. C., Trompoukis X. S., Asouti V. G., Giannakoglou K. C.: *CFD-based analysis and two-level aerodynamic optimization on graphics processing units.* Computer Methods in Applied Mechanics and Engineering, Volume 199, Issues 9-12, 15 January 2010, Pages 712-722.

[6] X. S. Trompoukis, V. G. Asouti, I. C. Kampolis, K. C. Giannakoglou: *CUDA implementation of Vertex-Centered, Finite Volume CFD methods on Unstructured Grids with Flow Control Applications.* GPU Computing Gems Vol. 2, Editor: Wei-mei Hwu, Morgan Kaufmann, 2011 (to be published).

[7] Τρομπούκης, Ξ.: *Αριθμητική επίλυση προβλημάτων αεροδυναμικής-αεροελαστικότητας σε επεξεργαστές καρτών γραφικών. Ανάπτυξη αριθμητικής μεθοδολογίας τεχνητής συμπιεστότητας για τον υπολογισμό μη*

μόνιμων ροών σε κινούμενα όρια. Διδακτορική διατριβή, Εργαστήριο Θερμικών Στροβιλομηχανών, Ε.Μ.Π., Αθήνα, 2012.

[8] Καββαδίας, Ι.: *Προγραμματισμός επιλύτη 3Δ εξισώσεων ροής ατριβούς ρευστού σε δομημένα πλέγματα, σε κάρτες γραφικών.* Διπλωματική Εργασία. Εργαστήριο Θερμικών Στροβιλομηχανών, Ε.Μ.Π., 2011.

[9] Τρομπούκης, Ξ.: *Υπολογιστική ανάλυση και παραμετρική διερεύνηση της τεχνικής συνεχούς αναρρόφησης για τον έλεγχο οριακών στρωμάτων.* Διπλωματική Εργασία. Εργαστήριο Θερμικών Στροβιλομηχανών, Ε.Μ.Π., 2002.

[10] Ζερβογιάννης, Θ.: *Μέθοδοι βελτιστοποίησης στην αεροδυναμική και τις στροβιλομηχανές με χρήση συζυγών τεχνικών, υβριδικών πλεγμάτων και του ακριβούς εσσιανού μητρώου.* Διδακτορική διατριβή, Εργαστήριο Θερμικών Στροβιλομηχανών, Ε.Μ.Π., Αθήνα, 2011.

[11] Roe, P.: *Approximate Riemann solvers, parameter vectors, and difference schemes.* Journal of Computational Physics, 43(2):357–372, 1981.

[12] van Albada, G. D., van Leer, B., and Roberts, W. W.: *A comparative study of computational methods in cosmic gas dynamics*, Astronomy and Astrophysics, 108:76-84, 1982.

[13] Pulliam, T. H. and Steget, J. L.: *Recent impovements in efficiency, accuracy and convergence for implicit approximate factorization algorithms*, AIAA-85-03060, 1985.

[14] Courant R., Friedrichs K. and Lewy H.: *Über die partiellen differenzengleichungen der mathematischen physik*, Mathematische Annalen 1928; 100:32-74.

[15] Steger, P. and Warming, R. F.: *Flux vector splitting of the inviscid gasdynamic equations with application to the finite-difference methods*, Journal of Computational Physics, 40:263-293, 1981.

[16] Fezoui, L. and Stoufflet, B.: *A class of implicit upwind schemes for euler simulations with unstructured meshes*, Journal of Computational Physics, 84:174-206, 1989.

[17] NVIDIA CUDA^{TM}: *NVIDIA CUDA C Programming Guide*, 4.0 edition, March 2011.

[18] Sanders, J. and Kandrot, E.: *CUDA by example: An introduction to General–Purpose GPU Programming.* Addison–Wesley, 2010.

[19] Farber, R.: *CUDA application design and development.* Morgan Kaufmann, 2012.

[20] *Fermi architecture, webpage.* http://www.nvidia.com/object/fermi_architecture.html.

[21] Butenhof, David R.: *Programming with POSIX threads.* Addison–Wesley Professional Computing Series, 1997.

[22] *POSIX threads programming.* https://computing.llnl.gov/tutorials/pthreads/.

[23] *Intel hyper-threading technology.* http://www.intel.com/content/www/us/en/architecture-and-technology/hyper-threading/hyper-threading-technology.html.

[24] *Boost threads documentation.* http://www.boost.org/doc/libs/1_51_0/doc/html/thread.html.

[25] *The OpenMP API specification for parallel programming.* http://openmp.org/wp/.

[26] Chapman, B., van der Jost, G., Pas, R., and Kuck, D.: *Using OpenMP: Portable shared memory programming.* The MIT Press, 2007.