ΕΘΝΙΚΟ ΚΑΙ ΚΑΠΟΔΙΣΤΡΙΑΚΟ ΠΑΝΕΠΙΣΤΗΜΙΟ ΑΘΗΝΩΝ

ΣΧΟΛΗ ΘΕΤΙΚΩΝ ΕΠΙΣΤΗΜΩΝ
ΤΜΗΜΑ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΤΗΛΕΠΙΚΟΙΝΩΝΙΩΝ

ΔΙΑΤΜΗΜΑΤΙΚΟ ΠΡΟΓΡΑΜΜΑ ΜΕΤΑΠΤΥΧΙΑΚΩΝ ΣΠΟΥΔΩΝ ΣΤΟΝ ΗΛΕΚΤΡΟΝΙΚΟ ΑΥΤΟΜΑΤΙΣΜΟ

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

Υβριδική υλοποίηση σε OpenMP και CUDA για την επίλυση της εξίσωσης διάχυσης θερμότητας με χρήση της τοπικής Τροποποιημένης μεθόδου SOR

Πέτρος Ν. Γιαννακόπουλος

Α.Μ.: 2012504

Επιβλέπων: **Ιωάννης Κοτρώνης,** Επίκουρος Καθηγητής

ΑΘΗΝΑ

ΟΚΤΩΒΡΙΟΣ 2014

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

Υβριδική υλοποίηση σε OpenMP και CUDA για την επίλυση της εξίσωσης διάχυσης θερμότητας με χρήση της τοπικής Τροποποιημένης μεθόδου SOR

Πέτρος Ν. Γιαννακόπουλος
Α.Μ.: 2012504

ΕΠΙΒΛΕΠΩΝ: **Ιωάννης Κοτρώνης,** Επίκουρος Καθηγητής

Οκτώβριος 2014

ΠΕΡΙΛΗΨΗ

Αντικείμενο της παρούσας διπλωματικής εργασίας είναι η υβριδική παράλληλη υλοποίηση της μεθόδου SOR για την αριθμητική επίλυση της Εξίσωσης Διάχυσης Θερμότητας που θα εκμεταλλεύεται τη CPU και τη GPU για την επιτάχυνση του χρόνου επίλυσης, με χρήση υβριδικού OpenMP – CUDA κώδικα. Αφετηρία είναι δύο προυπάρχουσες ξεχωριστές υλοποιήσεις της LMSOR, σε OpenMP και CUDA, με χρήση διάταξης κόκκινων και μαύρων σημείων (red-black ordering) και χρήση 2 συνόλων παραμέτρων ω_{ij} και ω_{2ij} για το stencil 5 σημείων. Οι γραμμές του πλέγματος διαμοιράζονται με στατικό τρόπο σε CPU και GPU και καθεμία πραγματοποιεί τους υπολογισμούς στο δικό της κομμάτι με ανταλλαγή των συνοριακών γραμμών και υπολογισμό της ολικής σύγκλισης σε κάθε επανάληψη. Η υλοποίηση για CPU (OpenMP) χρησιμοποιεί και τις επεκτάσεις SSE2, κάτι που ενσωματώθηκε και στην υβριδική υλοποίηση για το μέρος των υπολογισμών που πραγματοποιούνται στη CPU. Για τις γραμμές του πλέγματος που υπολογίζονται στη GPU έχουν υλοποιηθεί 3 διαφοροποιήσεις των πυρήνων (kernels) για χρήση κύριας (global), κοινής (shared) ή μνήμης υφής (texture memory), σύμφωνα και με την αρχική υλοποίηση μόνο για GPU. Η επίδοση που επιτεύχθηκε για τους υπολογισμούς με την υβριδική υλοποίηση ήταν, για μεγάλα και μεσαίου μεγέθους προβλήματα, ικανοποιητική και πολύ κοντά στην αθροιστική επίδοση των αρχικών υλοποιήσεων μόνο για GPU και CPU.

ΘΕΜΑΤΙΚΗ ΠΕΡΙΟΧΗ: Επιστημονικοί Υπολογισμοί

ΛΕΞΕΙΣ ΚΛΕΙΔΙΑ: επαναληπτικές μέθοδοι, R/B SOR, OpenMP – CUDA hybrid, SSE2, GPU computing, CUDA, OpenMP

ABSTRACT

The subject of the present Thesis is a hybrid parallel implementation of the of the SOR method for the numerical solution of the Convection Diffusion equation which will take advantage of both the CPU and GPU for the acceleration of the time required to reach a solution, using hybrid OpenMP – CUDA code. The start point is two preexisting separate implementations of LMSOR, in OpenMP and CUDA, employing red-black ordering using two sets of parameters ω_{ij} and ω_{2ij} for the 5 point stencil. Grid lines are distributed statically between the CPU and GPU and each performs the calculations on its own segment with border rows exchange and computation of the aggregate convergence taking place on every iteration. The CPU implementation (OpenMP) also employs SSE2 extensions, something that was also incorporated in the hybrid implementation for the part of the computations pertaining to the CPU. For the grid lines computed on the GPU, 3 variations of the kernels have been implemented for the use of global, shared or texture memory, in accordance with the initial GPU-only implementation. The performance achieved for computations with this hybrid implementation was, for large and medium-sized problems, satisfactory and quite close to the aggregate performance of the initial CPU-only and GPU-only implementations.

SUBJECT AREA: Scientific Computing

KEYWORDS: iterative methods, R/B SOR, OpenMP – CUDA hybrid, SSE2, GPU computing, CUDA, OpenMP

ΕΥΧΑΡΙΣΤΙΕΣ

Ευχαριστώ τον επιβλέποντα καθηγητή Δρ. Ι. Κοτρώνη και τον υποψήφιο διδάκτορα Η. Κωνσταντινίδη για την βοήθειά τους και τη γρήγορη επίλυση οποιονδήποτε αποριών ή άλλων σκοπέλων εμφανίστηκαν.

ΠΕΡΙΕΧΟΜΕΝΑ

1. ΕΙΣΑΓΩΓΗ **9**

1.1 Εισαγωγή στη CUDA **10**
1.1.1 Πυρήνες (Kernels) 15
1.1.2 Ιεραρχία νημάτων 15
1.1.3 Ιεραρχία μνήμης 17
1.1.4 Ετερογενής Προγραμματισμός 18
1.1.5 Υπολογιστική Δυνατότητα 19

1.2 Εισαγωγή στο OpenMP **19**

1.3 Η τοπική μέθοδος SOR **20**

2. ΥΛΟΠΟΙΗΣΗ ΚΑΙ ΜΕΤΡΗΣΕΙΣ **23**

2.1 Υλοποίηση **23**

2.2 Μετρήσεις **25**
2.2.1 Μικρό πρόβλημα (N = 504) 26
2.2.2 Μεσαίο πρόβλημα (N = 2054) 29
2.2.3 Μεγάλο πρόβλημα (N = 6144) 32
2.2.4 Μεγάλο πρόβλημα που δε χωράει στη μνήμη της GPU (N = 8194) 34

3. ΣΥΜΠΕΡΑΣΜΑΤΑ **37**

ΠΙΝΑΚΑΣ ΟΡΟΛΟΓΙΑΣ **39**

ΣΥΝΤΜΗΣΕΙΣ – ΑΡΚΤΙΚΟΛΕΞΑ – ΑΚΡΩΝΥΜΙΑ **40**

ΑΝΑΦΟΡΕΣ **41**

ΚΑΤΑΛΟΓΟΣ ΣΧΗΜΑΤΩΝ

Σχήμα 1: Πράξεις κινητής υποδιαστολής ανα δευτερόλεπτο για CPU και GPU [4]........11

Σχήμα 2: Εύρος ζωνης της μνήμης για CPU και GPU [4] 11

Σχήμα 3: Αρχιτεκτονική CPU έναντι GPU [4] 12

Σχήμα 4: Αυτόματη κλιμάκωση με τον αριθμό των διαθέσιμων SMs [4]........................15

Σχήμα 5: Δισδιάστατο πλέγμα απο blocks νημάτων [4]..16

Σχήμα 6: Ιεραρχεία μνήμης [4]...17

Σχήμα 7: Ετερογενής Προγραμματισμός [4] 18

Σχήμα 8: Νηματικό μοντέλο fork-join 20

Σχήμα 9: Εύρος ζώνης συναρτήσει του διαμοιρασμού γραμμών του πλέγματος σύμφωνα με το πόσες απο αυτές ανατίθενται στη CPU (N = 504) 27

Σχήμα 10: Εύρος ζώνης συναρτήσει του αριθμού γραμμών που ανταλλάσονται σε ίσο αριθμό επαναλήψεων (N = 504) 28

Σχήμα 11: Συνολικός χρονος υπολογισμών, Χρόνος υπολογισμών της GPU και χρόνος αδράνειας της GPU συναρτήσει του αριθμού γραμμών που ανατίθενται στη CPU (N = 504) 29

Σχήμα 12: Εύρος ζώνης συναρτήσει του διαμοιρασμού γραμμών του πλέγματος σύμφωνα με το πόσες απο αυτές ανατίθενται στη CPU (N = 2054) 30

Σχήμα 13: Εύρος ζώνης συναρτήσει του αριθμού γραμμών που ανταλλάσονται σε ίσο αριθμό επαναλήψεων (N = 2054) 30

Σχήμα 14: Συνολικός χρονος υπολογισμών, Χρόνος υπολογισμών της GPU και χρόνος αδράνειας της GPU συναρτήσει του αριθμού γραμμών που ανατίθενται στη CPU (N = 2054) 31

Σχήμα 15: Εύρος ζώνης συναρτήσει του διαμοιρασμού γραμμών του πλέγματος σύμφωνα με το πόσες απο αυτές ανατίθενται στη CPU (N = 6144) 32

Σχήμα 16: Εύρος ζώνης συναρτήσει του αριθμού γραμμών που ανταλλάσονται σε ίσο αριθμό επαναλήψεων (N = 6144) 33

Σχήμα 17: Συνολικός χρονος υπολογισμών, Χρόνος υπολογισμών της GPU και χρόνος αδράνειας της GPU συναρτήσει του αριθμού γραμμών που ανατίθενται στη CPU (N = 6144) 34

Σχήμα 18: Εύρος ζώνης συναρτήσει του διαμοιρασμού γραμμών του πλέγματος σύμφωνα με το πόσες απο αυτές ανατίθενται στη CPU (N = 8194) 35

Σχήμα 19: Συνολικός χρονος υπολογισμών, Χρόνος υπολογισμών της GPU και χρόνος αδράνειας της GPU συναρτήσει του αριθμού γραμμών που ανατίθενται στη CPU (N = 8194) 36

1. ΕΙΣΑΓΩΓΗ

Οι επεξεργαστές γραφικών (GPUs) χρησιμοποιούνται όλο και περισσότερο για την επίλυση προβλημάτων μεγάλου υπολογιστικού φορτίου. Διαθέτοντας μεγαλύτερη υπολογιστική ισχύ, προσφέρουν σε αρκετές περιπτώσεις σημαντική επιτάχυνση των υπολογισμών συγκριτικά με τις CPUs που χρησιμοποιούνταν παραδοσιακά. Αν και ο βασικός προορισμός των επεξεργαστών γραφικών είναι, όπως λέει και το όνομά τους, η επεξεργασία γραφικών, έχουν αναπτυχθεί προγραμματιστικά περιβάλλοντα για την ανάπτυξη εφαρμογών που θα εκμεταλλεύονται τη μαζικά παράλληλη (massively parallel) φύση τους και το υψηλό διαθέσιμο εύρος ζώνης (bandwidth) τους για τη πραγματοποίηση γενικότερων υπολογισμών. Έτσι, οι σημερινοί επεξεργαστές γραφικών έχουν στην ουσία εξελιχθεί σε επεξεργαστές γραφικών γενικής χρήσης (GPGPUs – General Purpose GPUs). Τα δύο πιο ευρέως χρησιμοποιούμενα περιβάλλοντα είναι το κλειστό πρότυπο της CUDA (Compute Unified Development Architecture) που υποστηρίζεται μόνο απο την NVidia και το ανοιχτό πρότυπο του OpenCL (Open Computing Language) που υποστηρίζεται απο πολλούς κατασκευαστές συμπεριλαμβανομένης και της NVidia.

Απο την άλλη, οι σύγχρονες CPUs περικλείουν έναν ολοένα αυξανόμενο αριθμό πυρήνων σε ένα chip και σε συνδυασμό με τεχνολογίες όπως το Hyper-Threading, η CPU έχει μετατραπεί, σε αντίθεση με το παρελθόν, σε μια πολυνηματική μονάδα επεξεργασίας. Επιπλέον υπάρχει η δυνατότητα να υπάρχουν πάνω απο μία CPUs σε ένα σύστημα ή και η διασύνδεση πολλών συστημάτων μαζί που το καθένα έχει μία ή περισσότερες CPUs. Έτσι έχουν αναπτυχθεί πρότυπα για την εκμετάλλευση της προσφερόμενης απο τις CPUs παραλληλίας με πιο ευρέως διαδεδομένα το προγραμματιστικό μοντέλο του OpenMP (Open Multi-Processing) για αρχιτεκτονικές κοινής μνήμης (shared memory) και το μοντέλο του MPI (Message Passing Interface) για αρχιτεκτονικές κατανεμημένης μνήμης (distributed memory). Συνδυασμός των δύο προτύπων είναι επίσης δυνατός.

Εκτός απο τις εφαρμογές που εκμεταλλεύονται τη παραλληλία της GPU ή της CPU για τη γρηγορότερη πραγματοποίηση υπολογισμών, αναπτύσσονται και εφαρμογές υβριδικού κώδικα που εκμεταλλεύονται και τις δύο μαζί για υπολογισμούς με στόχο την επίτευξη πληρέστερης αξιοποίησης των υπολογιστικών πόρων και μεγαλύτερης επιτάχυνσης. Υπάρχει η δυνατότητα: 1) για ενα ομογενές πρόβλημα, ο κατάλληλος διαμοιρασμός των δεδομένων σε GPU και CPU και η πραγματοποίηση υπολογισμών ή 2) για ένα ετερογενές πρόβλημα, η ανάθεση, για παράδειγμα, ενός μέρους που θα

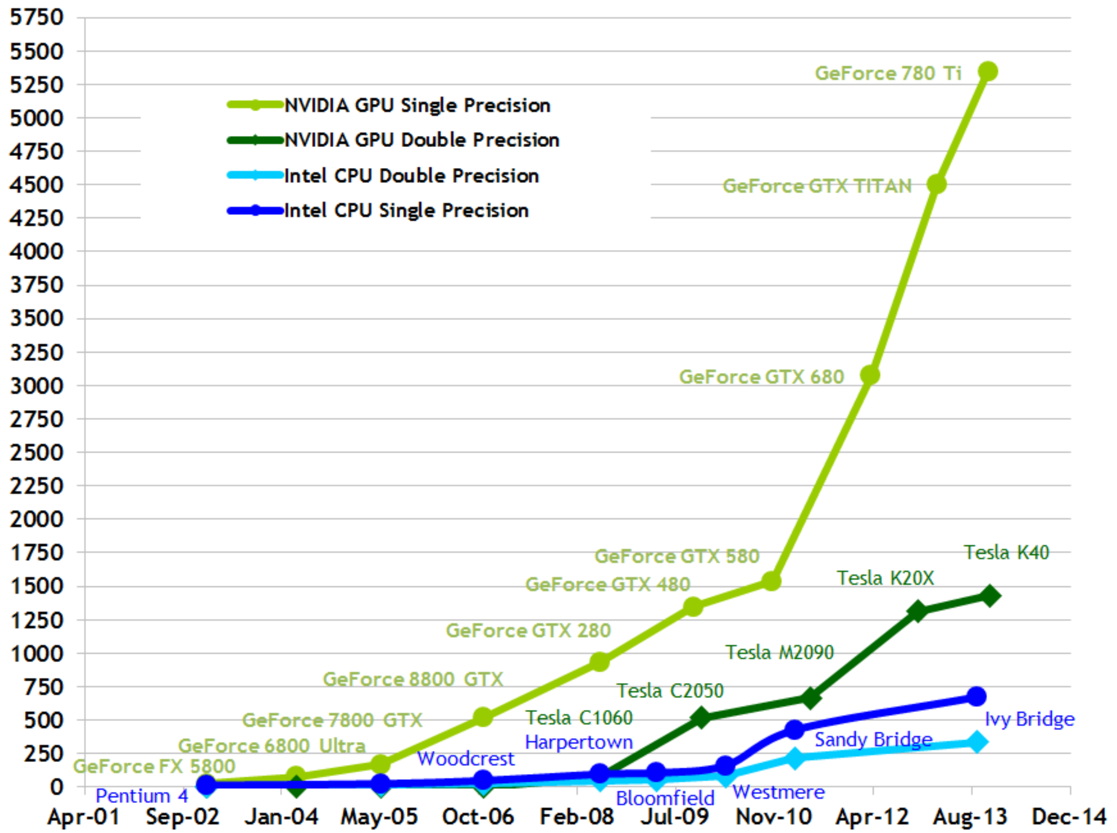
μπορούσε να υπολογιστεί πιο αποδοτικά στη GPU (ιδιαίτερα παραλληλοποιήσιμο) σε αυτήν και η ανάθεση ενός μέρους που δε θα μπορούσε να υπολογιστεί αποδοτικά στη GPU (λιγότερο παραλληλοποιήσιμο) στη CPU. Οι υβριδικές υλοποιήσεις αναπτύσσονται, στη συντριπτική πλειονότητά τους, με ένα συνδυασμό CUDA ή OpenCL και OpenMP ή MPI. Το OpenMP και MPI μπορούν, όπως προαναφέρθηκε να συνδυαστούν και έτσι να προκύψει μια υβριδική υλοποίηση π.χ. CUDA – OpenMP – MPI.

Η παρούσα εργασία αφορά τη 1η περίπτωση ομογενούς προβλήματος και η υλοποίηση έχει αναπτυχθεί με συνδυασμό OpenMP – CUDA. Στόχος είναι η επίλυση της εξίσωσης διάχυσης θερμότητας 2ας τάξης με χρήσης της τοπικής μεθόδου SOR, τροποποιημένης ώστε να είναι κατάλληλη για παράλληλη υλοποίηση πάνω σε μία συνδεόμενων με πλέγμα (mesh connected) διάταξη επεξεργαστών [1].

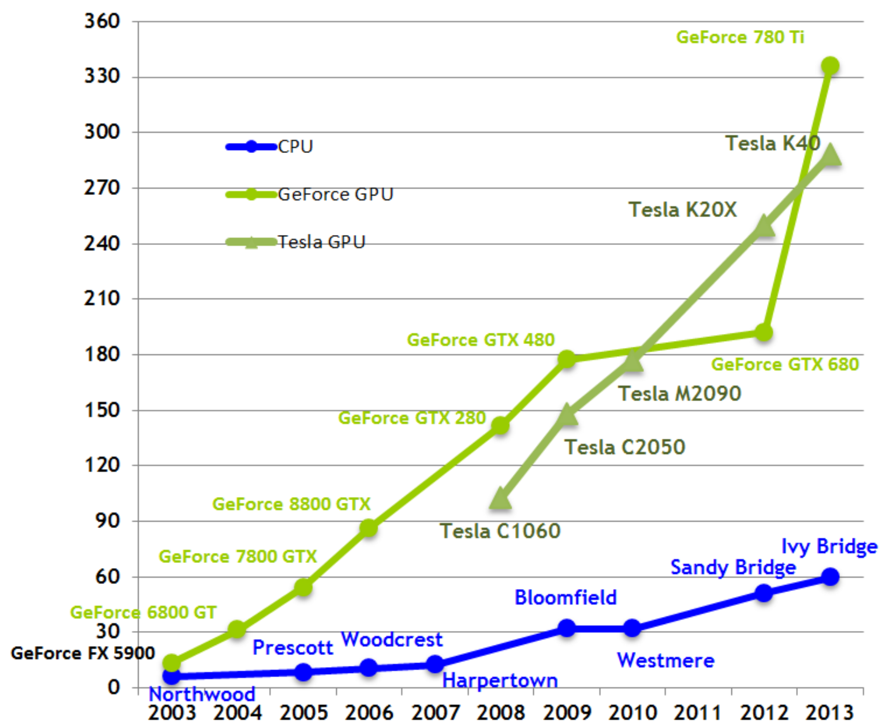
Ακολουθούν κάποια εισαγωγικά στοιχεία για τη CUDA και το OpenMP καθώς και για τη Τοπική Τροποποιημένη μέθοδο SOR.

1.1 Εισαγωγή στη CUDA

Όπως προαναφέρθηκε, ο επεξεργαστής γραφικών έχει εξελιχθεί σε ένα μαζικά παράλληλο, πολυνηματικό (multithreaded), πολυπύρηνο (manycore) επεξεργαστή με ιδιαίτερα μεγάλη υπολογιστική ισχύ και εύρος ζώνης, όπως διαφαίνεται απο το Σχήμα 1 και Σχήμα 2 τα οποία παρουσιάζουν την εξέλιξη της θεωρητικής υπολογιστικής ισχύος και του θεωρητικού εύρους ζώνης των GPUs και των CPUs απο το 2001 μέχρι σήμερα.

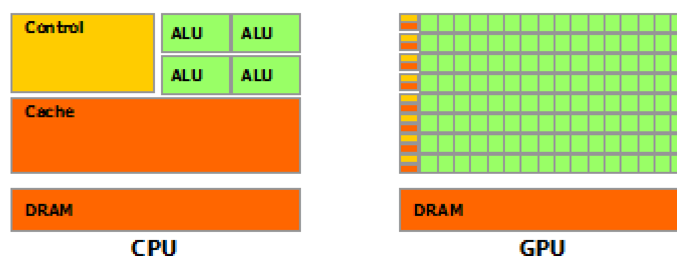


Σχήμα 1: Πράξεις κινητής υποδιαστολής ανα δευτερόλεπτο για CPU και GPU [4]



Σχήμα 2: Εύρος ζωνης της μνήμης για CPU και GPU [4]

Ο λόγος για τη σημαντική υπεροχή του επεξεργαστή γραφικών στη ταχύτητα εκτέλεσης πράξεων κινητής υποδιαστολής είναι το γεγονός ότι είναι εξειδικευμένος για την αντιμετώπιση προβλημάτων που μπορούν να εκφραστούν ως υπολογισμοί με παραλληλισμό δεδομένων (data parallel), δηλαδή οι ίδιες εντολές εκτελούνται σε πολλά διαφορετικά δεδομένα, με μεγάλο φόρτο αριθμητικών πράξεων σε σχέση με το φόρτο για τις προσβάσεις μνήμης (λόγος υπολογισμών/προσβάσεις μνήμης – compute/memory accesses ratio). Επειδή η ίδια σειρά εντολών εκτελείται για κάθε στοιχείο δεδομένων, είναι μικρή η ανάγκη για ύπαρξη περίπλοκης μονάδας ελέγχου ροής (flow control). Επιπρόσθετα, λόγω της εκτέλεσης των εντολών σε πολλά στοιχεία δεδομένων και του υψηλού φόρτου υπολογισμών, οι καθυστερήσεις για τις προσβάσεις μνήμης μπορούν να επικαλυφθούν με υπολογισμούς μειώνοντας την απαίτηση για χρήση μεγάλης προσωρινής μνήμης (cache) για αποθήκευση δεδομένων. Έτσι μεγαλύτερος αριθμός transistors μπορεί να αφιερωθεί για υπολογιστικές μονάδες, όπως φένεται στο Σχήμα 3:



Σχήμα 3: Αρχιτεκτονική CPU έναντι GPU [4]

Η αρχιτεκτονική της GPU προέκυψε απο την ανάγκη απόδοσης γραφικών (graphics rendering), που είναι εκ φύσεως μια διεργασία με υψηλό υπολογιστικό φόρτο και υψηλό βαθμό παραλληλισμού. Ουσιαστικά, οι GPUs στις αρχές της δεκαετίας του 2000 ήταν σχεδιασμένες να παράγουν ένα χρώμα για κάθε εικονοστοιχείο (pixel) στην οθόνη με τη χρήση προγραμματιζόμενων αριθμητικών μονάδων (AUs) γνωστές ως pixel shaders. Σε γενικές γραμμές, κάθε pixel shader χρησιμοποιεί τις συντεταγμένες του (x,y) στην οθόνη σε συνδυασμό με κάποιες επιπλέον πληροφορίες ώστε να συνδυάσει δίαφορες εισόδους για τον υπολογισμό του τελικού χρώματος. Οι επιπλέον πληροφορίες μπορεί να είναι χρώμα, συντεταγμένες υφών, ή άλλες παράμετροι που θα περνούσαν στον shader όταν θα έτρεχε [2].

Επειδή όμως οι υπολογισμοί που πραγματοποιούνταν πάνω στα χρώματα και τις υφές που έπαιρνε ως είσοδο ο pixel shader ήταν υπο τον έλεγχο του προγραμματιστή, κάποιοι ερευνητές παρατήρησαν ότι στη θέση αυτών των εισόδων θα μπορούσαν να

είναι οποιαδήποτε αριθμητικά δεδομένα που να αντιπροσωπεύουν κάτι άλλο εκτός απο χρώμα. Έτσι οι pixel shaders μπορούσαν να επαναπρογραμματιστούν έτσι ώστε να πραγματοποιούν αυθαίρετους υπολογισμούς σε αυτά τα δεδομένα εισόδου. Τα αποτελέσματα θα επέστρεφαν ώς το τελικό «χρώμα» ενός pixel, με τη διαφορά τώρα ότι στη πραγματικότητα θα είναι το αποτέλεσμα του υπολογισμού που ο προγραμματιστής είχε ορίσει στον συγκεκριμένο shader για το συγκεκριμένο στοιχείο δεδομένων. Τα τελικά αποτελέσματα μπορούσαν να διαβαστούν απο τους ερευνητές με τη GPU να «νομίζει» πως επεξεργάζεται γραφικά σε όλη τη διαδικασία [2].

Χάρη στη μεγάλη αριθμητική διαμεταγωγή (throughput) των GPUs, τα αρχικά αποτελέσματα αυτών των πειραμάτων ήταν ενθαρρυντικά. Το προγραμματιστικό μοντέλο όμως ήταν ιδιαίτερα περιοριστικό για να έχει ευρεία αποδοχή. Οι πόροι ήταν πολύ περιορισμένοι, καθώς τα προγράμματα μπορούσαν να δεχτούν δεδομένα εισόδου μόνο απο ένα περιορισμένο αριθμό χρωμάτων και μονάδων υφών (texture units). Υπήρχαν σοβαροί περιορισμοί στο πως και που ο προγραμματιστής μπορούσε να γράψει αποτελέσματα στη μνήμη. Επιπρόσθετα, ήταν σχεδόν αδύνατο να προβλεφθεί η συμπεριφορά μιας συγκεκριμένες GPU απέναντι σε δεδομένα κινητής υποδιαστολής, εάν μπορούσε καν να τα διαχειριστεί, έτσι οι περισσότεροι επιστημονικοί υπολογισμοί που απαιτούν τέτοιο είδους δεδομένα αδυνατούσαν να τρέξουν σε GPUs. Τέλος οι δυνατότητες για απασφαλμάτωση (debugging) GPU κώδικα ήταν ελάχιστες ή ανύπαρκτες [2].

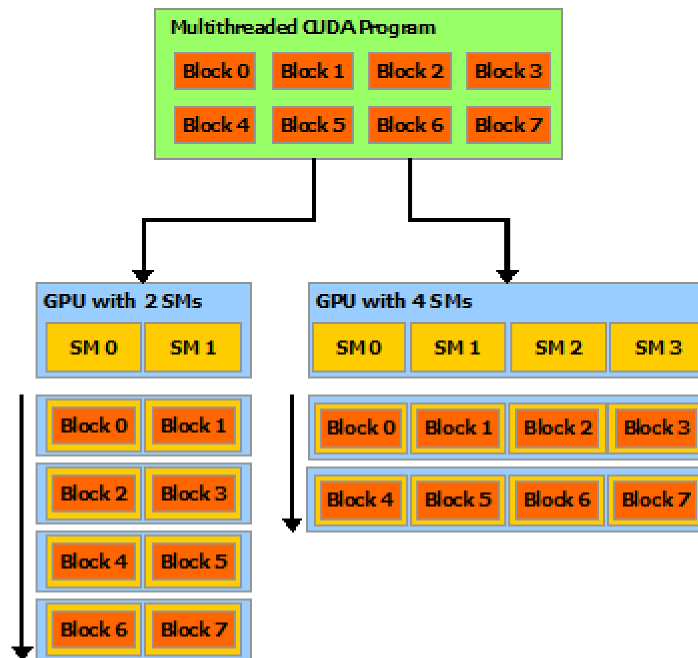
Πέρα απο τους παραπάνω περιορισμούς, ο μοναδικός τρόπος αλληλεπίδρασης με τη GPU ήταν το OpenGL ή το DirectX. Αυτό σήμαινε ότι η αποθήκευση δεδομένων θα έπρεπε να γίνεται σε υφές γραφικών και η εκτέλεση των υπολογισμών με τη κλήση συναρτήσεων του OpenGL ή του DirectX αλλά και ο ίδιος ο κώδικας για τους υπολογισμούς θα έπρεπε να είναι γραμμένος σε γλώσσες προγραμματισμού οι οποίες προορίζονταν μόνο για υλοποίηση γραφικών [2]. Όλοι αυτοί οι περιορισμοί σε πόρους και η μεγάλη δυσκολία και περιορισμοί του προγραμματιστικού μοντέλου αποδείχτηκαν σημαντικό εμπόδιο για την ευρύτερη αποδοχή και αξιοποίηση των υπολογιστικών δυνατοτήτων των GPUs απο τους ερευνητές.

Το Νοέμβριο του 2006, η NVidia παρουσίασε τη CUDA, μία πλατφόρμα για παράλληλους υπολογισμούς γενικής χρήσης και ένα προγραμματιστικό μοντέλο που αξιοποιεί τις δυνατότητες της GPU για την επίλυση υπολογιστικών προβλημάτων [4]. Η GeForce 8800 GTX, που παρουσιάστηκε την ίδια χρονιά, ήταν η πρώτη GPU σχεδιασμένη με βάση την αρχιτεκτονικη CUDA. Η αρχιτεκτονική αυτή περιελάμβανε

πολλά νέα στοιχεία σχεδιασμένα αποκλειστικά για πραγματοποίηση γενικών υπολογισμών με τη GPU και στοχεύοντας στην αναίρεση πολλών απο τους περιορισμούς που δεν επέτρεπαν στους προηγούμενους επεξεργαστές γραφικών να είναι ουσιαστικά χρήσιμοι για πραγματοποίηση γενικών υπολογισμών [2].

Η CUDA επιτρέπει στους προγραμματιστές να χρησιμοποιούν την ευρέως διαδεδομένη γλώσσα C ώς μια γλώσσα προγραμματισμού υψηλού επιπέδου για τον προγραμματισμό της GPU για πραγματοποίηση γενικών υπολογισμών, προσθέτωντας έναν αριθμό επιπλέον λέξεων-κλειδιά που επιτρέπουν την αξιοποίηση των δυνατοτήτων της CUDA αρχιτεκτονικής. Έτσι η CUDA C έγινε η 1η γλώσσα αποκλειστικά σχεδιασμένη για πραγματοποίηση υπολογισμών γενικής χρήσης στις GPUs [2]. Στον πυρήνα του προγραμματιστικού μοντέλου της CUDA βρίσκονται τρείς απλοποιήσεις: μια ιεραρχία απο ομάδες νημάτων, κοινές μνήμες, και συγχρονισμός μεταξύ των νημάτων, τα οποία παρουσιάζονται στον προγραμματιστή ως ένα ελάχιστο σύνολο απο επεκτάσεις της γλώσσας C. Οι απλοποιήσεις αυτές προσφέρουν παραλληλισμό δεδομένων υψηλής διακριτότητας και παραλληλισμό νημάτων, εμφωλευμένων μέσα σε παραλληλισμό δεδομένων χαμηλής διακριτότητας και παραλληλισμό διεργασιών. Οδηγούν το προγραμματιστή στο να διαχωρίσει το πρόβλημα σε μικρότερα υποπροβλήματα τα οποία μπορούν να λυθούν ανεξάρτητα και παράλληλα απο blocks νημάτω, και κάθε υποπρόβλημα σε μικρότερα κομμάτια που μπορούν να λυθούν απο κοινού παράλληλα απο όλα τα νήματα μέσα σε ένα block [4].

Ο διαχωρισμός αυτός διατηρεί την εκφραστικότητα της γλώσσας επιτρέποντας στα νήματα να συνεργάζονται για την επίλυση κάθε υποπροβλήματος, και ταυτόχρονα επιτρέπει αυτόματη κλιμάκωση (scalability), εφόσον κάθε block νημάτων μπορεί να προγραμματιστεί να τρέξει σε οποιοδήποτε απο τους διαθέσιμους πολυεπεξεργαστές (multiprocessors) μέσα στη GPU, με οποιαδήποτε σειρά, παράλληλα ή σειριακά, έτσι ώστε ένα πρόγραμμα σε CUDA να μπορεί να εκτελεστεί σε οποιοδήποτε αριθμό πολυεπεξεργαστών όπως φαίνεται στο Σχήμα 4. Μία GPU είναι σχεδιασμένη γύρο απο ένα σύνολο Streaming Multiprocessors (SMs). Ένα πολυνηματικό (multithreaded) πρόγραμμα διαχωρίζεται σε blocks νημάτων που εκτελούνται ανεξάρτητα το ένα απο το άλλο, έτσι ώστε η GPU με τους περισσότερους SMs εκτελεί αυτόματα το πρόγραμμα σε λιγότερο χρόνο απο ότι μια GPU με λιγότερους SMs [4].



Σχήμα 4: Αυτόματη κλιμάκωση με τον αριθμό των διαθέσιμων SMs [4]

Παρακάτω παρουσιάζονται συνοπτικά όρισμένες κύριες έννοιες του προγραμματιστικού μοντέλου της CUDA οι οποίες χρησιμοποιούνται σε κάθε πρόγραμμα:

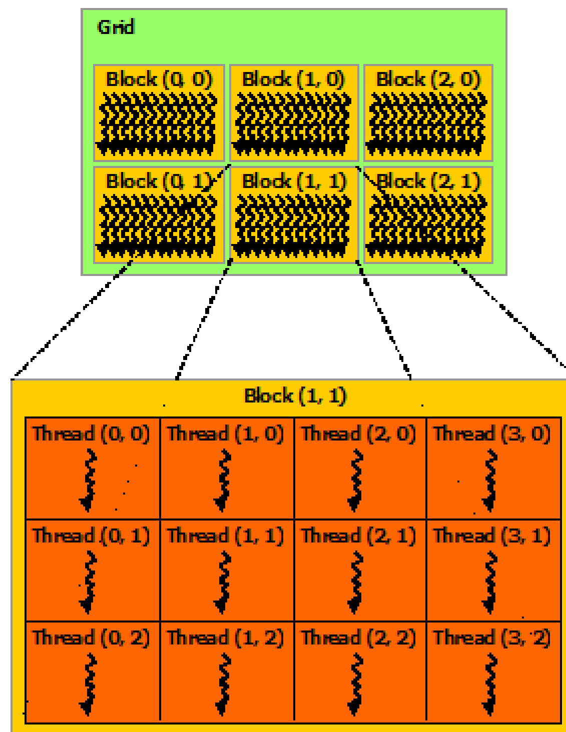
1.1.1 Πυρήνες (Kernels)

Η CUDA C επιτρέπει στον προγραμματιστή να ορίζει συναρτήσεις της C, που λέγονται πυρήνες (kernels). Με τη κλήση ενός kernel, ο κώδικας μέσα σε αυτόν εκτελείται N φορές παράλληλα απο N διαφορετικά νήματα στη GPU. Οι kernels αποτελούν τον πρωταρχικό τρόπο εκτέλεσης κώδικα στη GPU [4].

1.1.2 Ιεραρχία νημάτων

Τα νήματα μπορούν να σχηματίζουν μονοδιάστατα, δισδιάστατα ή τρισδιάστατα blocks. Αυτό επιτρέπει ένα διαισθητικό τρόπο αντιστοίχησης των διαστάσεων ενός προβλήματος με το νηματικό πλέγμα (grid) επεξεργασίας. Υπάρχει ένα άνω όριο στον αριθμό των νημάτων μέσα σε ένα block, εφόσον όλα τα νήματα ενός block πρέπει να βρίσκονται μέσα στον ίδιο SM και πρέπει να μοιράζονται τους περιορισμένους πόρους μνήμης αυτού. Στις τελευταίες GPUs με Compute Capability 3.0, κάθε block μπορεί να περιέχει μέχρι 1024 νήματα. Όμως, κάθε kernel μπορεί να εκτελείται απο πολλαπλά block ίδιου σχήματος, έτσι ώστε ο συνολικός αριθμός των νημάτων να είναι ίσος με τον αριθμό των νημάτων ανα block επί τον αριθμό των block. Τα blocks οργανώνονται σε

ένα μονοδιάστατο, δισδιάστατο ή τρισδιάστατο πλέγμα οπως φένεται στο Σχήμα 5. Ο αριθμός των blocks σε ένα πλέγμα συνήθως καθορίζεται απο το μέγεθος τον δεδομένων προς επεξεργασία ή τον αριθμό των SMs στο σύστημα.

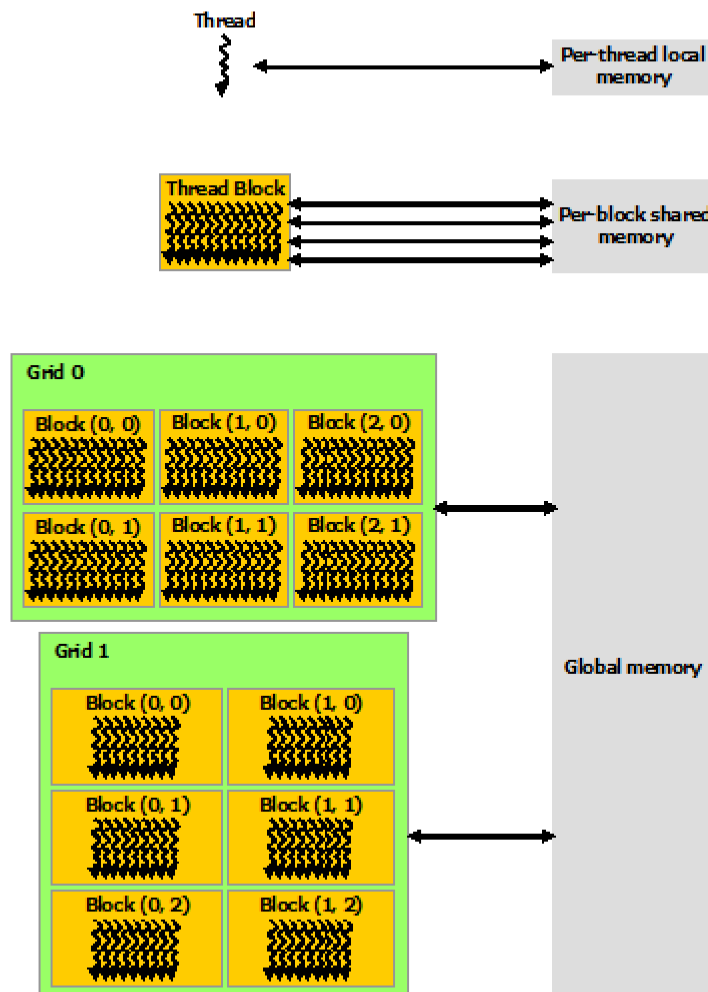


Σχήμα 5: Δισδιάστατο πλέγμα απο blocks νημάτων [4]

Κάθε block νημάτων απαιτείται να εκτελείται ανεξάρτητα, δηλαδή να είναι δυνατό να εκτελούνται με οποιαδήποτε σειρά, παράλληλα η σειριακά. Η απαίτηση αυτή επιτρέπει στα blocks νημάτων να προγραμματίζονται για εκτέλεση με οποιαδήποτε σειρά σε οποιοδήποτε αριθμό πυρήνων, επιτρέποντας στους προγραμματιστές τη συγγραφή κώδικα η απόδοση του οποίου κλιμακώνει με τον αριθμό των πυρήνων. Τα νήματα μέσα σε ένα block μπορούν να συνεργάζονται με το να μοιράζονται δεδομένα μέσω κάποιας κοινής (shared) μνήμης και με το να συγχρονίζουν την εκτέλεσή τους. Πιο συγκεκριμένα, ο προγραμματιστής μπορεί να θεσπίσει σημεία συγχρονισμού στο kernel καλώντας την αντίστοιχη συνάρτηση η οποία παίζει το ρόλο ενός φράγματος (barrier) στην εκτέλεση που όλα τα νήματα μέσα στο block θα πρέπει να έχουν φτάσει πριν επιτραπεί να συνεχίσουν την εκτέλεσή τους. Η shared memory είναι μία μνήμη χαμηλής καθυστέρησης κοντά στον επεξεργαστή, αντίστοιχη της L1 cache στη CPU και επιτρέπει τη πολύ γρηγορότερη πρόσβαση των threads στα δεδομένα που βρίσκονται σε αυτή.

1.1.3 Ιεραρχία μνήμης

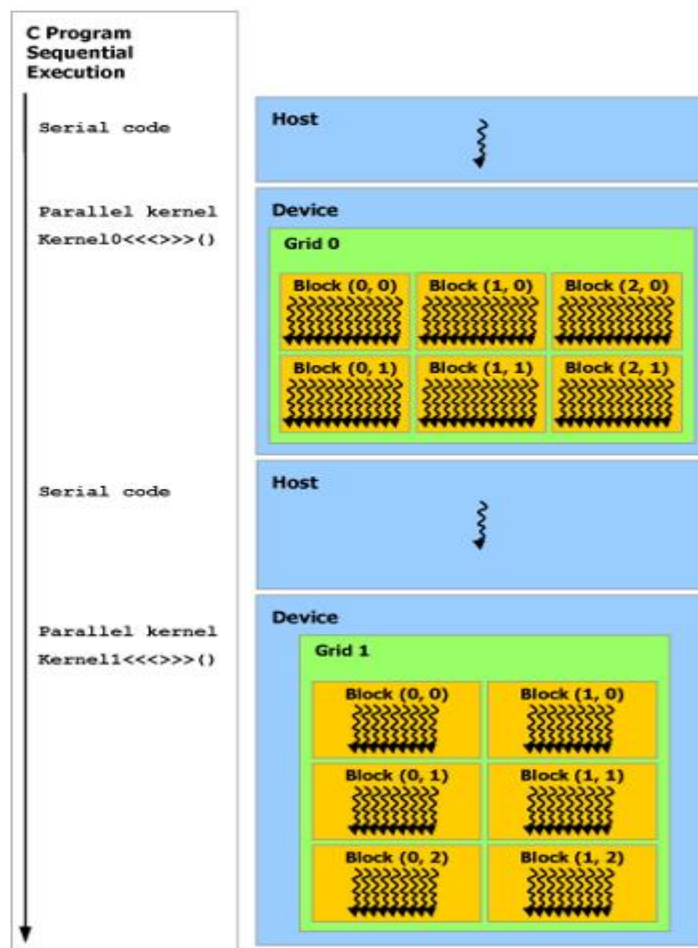
Τα thread στη CUDA μπορούν να προσπελάσουν τα δεδομένα απο πολλαπλούς χώρους μνήμης (memory spaces) όπως φαίνεται στο Σχήμα 6. Κάθε νήμα έχει μια δική του ιδιωτική τοπική μνήμη (local memory). Επίσης, κάθε block έχει τη κοινή μνήμη (shared memory) που είναι ορατή σε όλα τα νήματα του block και έχει τον ίδιο χρόνο ζωής με το block. Τέλος, όλα τα νήματα έχουν πρόσβαση στην ίδια κύρια μνήμη (global memory). Υπάρχουν επίσης δύο επιπρόσθετοι τύποι μνήμης μόνο για ανάγνωση (read-only) προσβάσιμοι απο όλα τα threads: η διαρκής μνήμη (constant memory) και η μνήμη υφών (texture memory). Οι global, constant και texture μνήμες είναι βελτιστοποιημένες για διαφορετικές χρήσεις. Τέλος, σε αντίθεση με τη local και shared μνήμη, οι global, constant και texture μνήμες εξακολουθούν να «ζούν» και μετά το τέλος της εκτέλεσης ενός kernel και είναι προσβάσιμες απο όλους τους kernels μιας εφαρμογής.



Σχήμα 6: Ιεραρχεία μνήμης [4]

1.1.4 Ετερογενής Προγραμματισμός

Το προγραμματιστικό μοντέλο της CUDA υποθέτει ότι τα νήματα εκτελούνται σε μία ξεχωριστή, σε φυσικό επίπεδο, μονάδα που λειτουργεί ως συνεπεξεργαστής (coprocessor) στον κεντρικό επεξεργαστή που τρέχει το πρόγραμμα σε C. Αυτό γίνεται όταν, για παράδειγμα, οι kernels εκτελούνται στη GPU (παράλληλος κώδικας) και το υπόλοιπο C πρόγραμμα (σειριακός κώδικας) εκτελείται στη CPU. Βέβαια με χρήση κάποιου παράλληλου προγραμματιστικού μοντέλου για CPU μπορούμε να έχουμε παράλληλο κώδικα και εκεί, όπως γίνεται στη παρούσα εργασία. Το προγραμματιστικό μοντέλο της CUDA υποθέτει επίσης ότι τόσο η CPU όσο και η GPU διατηρούν τους δικούς τους ξεχωριστούς χώρους μνήμης. Για το λόγο αυτό, ένα πρόγραμμα διαχειρίζεται τους global, constant και texture χώρους μνήμης, οι οποίοι είναι ορατοί στους kernels, μέσω κλήσεων συναρτήσεων της CUDA. Αυτό περιλαμβάνει δέσμευση και αποδέσμευση μνήμης της GPU καθώς και μεταφορά δεδομένων μεταξύ CPU και GPU.



Σχήμα 7: Ετερογενής Προγραμματισμός [4]

1.1.5 Υπολογιστική Δυνατότητα

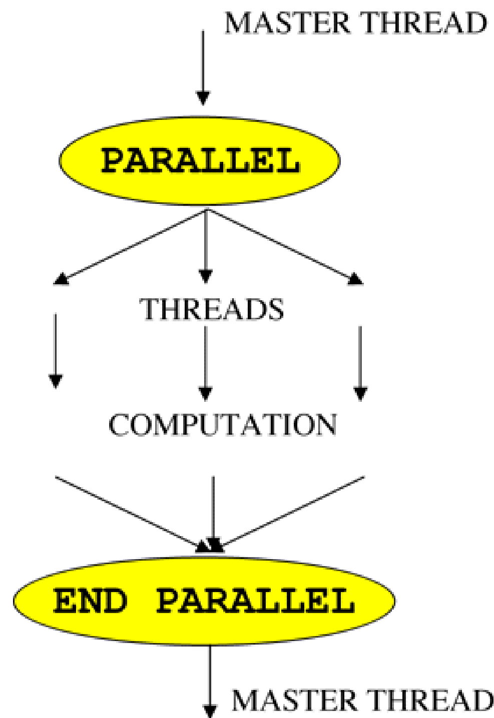
Η υπολογιστική δυνατότητα μιας GPU αντιπροσωπεύεται απο έναν αριθμό έκδοσης. Αυτός ο αριθμός έκδοσος ταυτοποιεί τα χαρακτηριστικά που υποστηρίζονται απο το υλικό της GPU και χρησιμοποιείται απο τις εφαρμογές κατά την εκτέλεση για πληροφορηθούν ποιές δυνατότητες υλικού και εντολές είναι διαθέσιμες στη παρούσα GPU. Η υπολογιστική δυνατότητα αποτελείται απο ένα κύριο και ένα δευτερεύων αριθμό έκδοσης (x.y): Οι GPUs με τον ίδιο κύριο αριθμό έκδοσος ανήκουν στην ίδια αρχιτεκτονική. Ο κύριος αριθμός έκδοσης είναι 5 για GPUs της αρχιτεκτονικής Maxwell, 3 για GPUs της αρχιτεκτονικής Kepler και 2 για GPUs της αρχιτεκτονικής Fermi. Ο δευτερεύων αριθμός έκδοσης αντιστοιχεί σε βελτιώσεις στην υπάρχουσα αρχιτεκτονική, που πιθανώς να περιλαμβάνουν και νέα χαρακτηριστικά.

1.2 Εισαγωγή στο OpenMP

Το OpenMP είναι ένα προγραμματιστικό περιβάλλον για τη συγγραφή πολυνηματικών εφαρμογών. Αποτελείται απο ένα σύνολο οδηγιών προς τον compiler, ρουτίνες βιβλιοθηκών και μεταβλητές περιβάλλοντος για παράλληλες εφαρμογές. Απλοποιεί ιδιαίτερα τη διαδικασία συγγραφής πολυνηματικών εφαρμογών στη C, τη C++ και τη Fortran. Η διαχείριση του προτύπου γίνεται απο τη μη κερδοσκοπική κοινοπραξία OpenMP Architecture Review Board (ή OpenMP ARB), την οποία σχηματίζουν απο κοινού μεγάλες εταιρίες υλικού και λογισμικού, συμπεριλαμβανομένων των AMD, IBM, Intel, Cray, HP, Fujitsu, NVidia, NEC, Red Hat, Texas Instruments, Oracle και άλλες [5]. Το OpenMP χρησιμοποιεί ένα φορητό, κλιμακωτό μοντέλο που παρέχει στους προγραμματιστές μια απλή και ευέλικτη διεπαφή για την υλοποίηση παράλληλων εφαρμογών.

Το OpenMP αποτελεί μια υλοποίηση του πολυνηματισμού, μία μέθοδος παραλληλισμού όπου ένα κυρίως νήμα (master thread) διχαλώνεται (fork) σε ένα καθορισμένο αριθμό υπονημάτων (slave threads) με μια διεργασία να μοιράζεται αναμεταξύ τους. Τα νήματα τρέχουν παράλληλα με το περιβάλλον εκτέλεσης να αντιστοιχίζει νήματα σε διαφορετικούς επεξεργαστές. Το κομμάτι του κώδικα που θέλουμε να τρέξει παράλληλα μαρκάρεται ανάλογα, με μία οδηγία προς τον προεπεξεργαστή (preprocessor) η οποία θα σημάνει τη δημιουργία νημάτων πρίν την εκτέλεση αυτού του κομματιού. Κάθε νήμα χαρακτηρίζεται μοναδικά απο έναν άυξοντα αριθμό, ξεκινώντας απο το κυρίως νήμα. Μετά την εκτέλεση του παράλληλου κώδικα, τα νήματα συνενώνονται (join) πίσω στο

κυρίως νήμα που εξακολουθεί να υπάρχει μέχρι το τέλος της εκτέλεσης του προγράματος.



Σχήμα 8: Νηματικό μοντέλο fork-join

Τυπικά, κάθε νήμα εκτελεί το παράλληλο τμήμα του κώδικα ανεξάρτητα. Είναι δυνατή η χρήση δομών διαμοιρασμού εργασίας (work-sharing constructs) για το διαμοιρασμό μιας εργασίας ανάμεσα στα νήματα με τέτοιο τρόπο ώστε κάθε νήμα να εκτελεί το κομμάτι του κώδικα που αντιστοιχεί σε αυτό. Με τρόπο αυτό είναι δυνατό να επιτευχθεί και παραλληλισμός σε επίπεδο διεργασίας και παραλληλισμός σε επίπεδο δεδομένων.

Τέλος, το περιβάλλον εκτέλεσης του OpenMP εκχωρεί νήματα σε επεξεργαστές ανάλογα με τη χρήση, το φόρτο του συστήματος και άλλους παράγοντες. Ο αριθμός των νημάτων μπορεί να καθοριστεί απο το χρήστη χρησιμοποιώντας τις ανάλογες συναρτήσεις του OpenMP αλλά, ανεξάρτητα απο αυτό, δε μπορούν να δημιουργηθούν περισσότερα νήματα από όσα επιτρέπονται απο το σύστημα.

1.3 Η τοπική μέθοδος SOR

Όπως έχει αναφερθεί, στόχος της παρούσας υλοποίησης είναι η επίλυση της Εξίσωσης Διάχυσης Θερμότητας 2ας τάξης

$$\Delta u - f(x,y)\frac{\partial y}{\partial x} - g(x,y)\frac{\partial y}{\partial x} = 0 \quad (1)$$

σε ένα χώρο $\Omega = \{(x,y)\} \mid 0 \le x \le 1,\ 0 \le y \le 1\}$, όπου η $u = u(x,y)$ όρίζεται στο όριο $\partial\Omega$. Η διακριτή μορφή της εξίσωσης σε ένα ορθογώνιο πλέγμα $M_1 \times M_2 = N$ αγνώστων μέσα στο χώρο Ω είναι

$$u_{ij} = l_y u_{i-1j} + r_{ij} u_{i+1j} + t_{ij} u_{ij+1} + b_{ij} u_{ij-1}, \qquad i = 1,2,\ldots,M_1,\ j = 1,2,\ldots,M_2 \qquad (2)$$

όπου

$$l_{ij} = \frac{k^2}{2(k^2+h^2)}\left(1 + \frac{1}{2} h f_{ij}\right), \quad r_{ij} = \frac{k^2}{2(k^2+h^2)}\left(1 - \frac{1}{2} h f_{ij}\right)$$

$$t_{ij} = \frac{k^2}{2(k^2+h^2)}\left(1 - \frac{1}{2} k g_{ij}\right), \quad b_{ij} = \frac{k^2}{2(k^2+h^2)}\left(1 + \frac{1}{2} k g_{ij}\right)$$

με $h = 1/(M_1 + 1)$, $k = (M_2 + 1)$, $f_{ij} = f(ih, jk)$ και $g_{ij} = g(ih, jk)$.

Για συγκεκριμένη διάταξη των σημείων του πλέγματος, η διακριτή εξίσωση δίνει ένα μεγάλο, αραιό, γραμμικό σύστημα εξισώσεων τάξης N της μορφής

$$Au = b.$$

Η επαναληπτική (iterative) μέθοδος της Διαδοχικής Υπερομαλοποίησης (Successive OverRelaxation – SOR), η οποία έχει τη μορφή

$$u_{ij}^{(n+1)} = (1-\omega) u_{ij}^{(n)} + \omega(l_{ij} u_{i-1j}^{(n+1)} + r_{ij} u_{ij+1}^{(n)} + t_{ij} u_{ij+1}^{(n)} + b_{ij} u_{ij-1}^{(n+1)}) \qquad (3)$$

είναι σημαντική για την επίλυση μεγάλων γραμικών συστημάτων. Όμως, η μέθοδος SOR είναι σειριακή με τη πρωταρχική μορφή της.

Για το λόγο αυτό, έχουν μελετηθει πολλές παράλληλες εκδόσεις της με χρήση διάφορων τεχνικών. Μία απο αυτές είναι η τοπική μέθοδος SOR (Local SOR) που προτάθηκε απο τους Ehrlich, Botta και Veldman [6,7,8] σε μία προσπάθεια να επιταχυνθεί ο ρυθμός σύγκλισης της SOR. Βασίζεται στην ιδέα του διαφορετικού παράγοντα ομαλοποίησης ω_{ij}. Έχει βρεθεί ότι, σε συνδυασμό με red-black αναδιάταξη, η μέθοδος αυτή είναι κατάλληλη για παράλληλη υλοποίηση πάνω σε διάταξη πλεγματικά συνδεόμενων επεξεργαστών.

Η μέθοδος SOR για τη διακριτοποίηση 5 σημείων της (2) δίνεται απο την (3). Μπορούμε να επιλέξουμε να καλούμε ένα σημείο του πλέγματος (i,j) κόκκινο (red) όταν I + j είναι άρτιο και μαύρο (black) όταν i + j είναι περιττό. Με τη χρήση δύο διαφορετικών συνόλων παραμέτρων ω_{1ij} και ω_{2ij} για τα κόκκινα και μαύρα σημεία του πλέγματος αντίστοιχα και αν $\Omega = diag(\omega_1, \omega_2, \ldots, \omega_N)$, $\omega_i = 1,2,\ldots,N$ πραγματικοί αριθμοί η (3) γίνεται:

$$
\begin{aligned}
u_{ij}^{(n+1)} &= (1-\omega_{1ij})u_{ij}^{(n)} + \omega_{1ij} J_{ij} u_{ij}^{(n)}, \ \text{για } i+j \text{ άρτιο} \\
u_{ij}^{(n+1)} &= (1-\omega_{2ij})u_{ij}^{(n)} + \omega_{2ij} J_{ij} u_{ij}^{(n)}, \ \text{για } i+j \text{ περιττό}
\end{aligned} \quad (4)
$$

όπου

$$J_{ij} u_{ij}^{(n)} = l_{ij} u_{i-1j}^{(n)} + r_{ij} u_{i+1j}^{(n)} + t_{ij} u_{ij+1}^{(n)} + b_{ij} u_{ij-1}^{(n)}$$

και J_{ij} είναι ο τοπικός τελεστής Jacobi. Οι παράμετροι ω_{1ij}, ω_{2ij} καλούνται τοπικοί παράμετροι ομαλοποίησης και η (4) καλείται τοπική Τροποποιημένη μέθοδος SOR (Local Modified SOR – LMSOR) [1].

2. ΥΛΟΠΟΙΗΣΗ ΚΑΙ ΜΕΤΡΗΣΕΙΣ

2.1 Υλοποίηση

Έχουν ήδη αναπτυχθεί παράλληλες υλοποιήσεις της μεθόδου LMSOR για CPU και GPU ξεχωριστά [1]. Η παράλληλη υλοποίηση για CPU βασίζεται στις δομές που προσφέρονται απο το προγραμματιστικό μοντέλο του OpenMP για αρχιτεκτονική κοινής μνήμης. Η υλοποίηση για GPU αναπτύχθηκε με το προγραμματιστικό περιβάλλον της CUDA και είναι ένα μαζικά παράλληλο πρόγραμμα, όπου μόνο λίγα σημεία προς υπολογισμό ανατίθενται σε κάθε νήμα για την επίτευξη παραλληλισμού υψηλής διακριτότητας (fine-grained parallelism).

Αφετηρία για την υβριδική υλοποίηση είναι οι προυπάρχουσες υλοποιήσεις της μεθόδου LMSOR για CPU και GPU ξεχωριστά που δουλεύουν πάνω σε ένα τετράγωνο πλέγμα N×N. Ο κώδικας των πρότυπων υλοποιήσεων ενσωματώνεται ως έχει στην υβριδική υλοποίηση με ελάφριες τροποποιήσεις για την επεξεργασία ενός παραλληλόγραμου υποπλέγματος του αρχικού τετράγωνου πλέγματος το οποίο δέχονταν ως είσοδο οι υλοποιήσεις αυτές. Στόχος είναι ο κατάλληλος διαμοιρασμός του φόρτου υπολογισμών μεταξύ της CPU και της GPU, η παράλληλη εκτέλεσή τους σε CPU – GPU και η ελαχιστοποίηση του απαιτούμενου overhead για το συγχρονισμό και την ανταλλαγή δεδομένων.

Ξεκινώντας με ένα τετράγωνο πλέγμα μεγέθους N×N, αναθέτουμε τις πρώτες NCPU γραμμές για υπολογισμό στη CPU και τις υπόλοιπες (N-NCPU) για υπολογισμό στη GPU. Έτσι έχουμε χωρίσει το πλέγμα σε 2 κομμάτια μεγέθους NCPU×N και (N-NCPU)×N. Ο τρόπος υπολογισμού της LMSOR μεθόδου στη GPU και CPU χωριστά είναι ο ίδιος με τις πρότυπες αντίστοιχες υλοποιήσεις. Αρχικά, σύμφωνα και με τις πρότυπες υλοποιήσεις, πραγματοποιείται red-black αναδιάταξη (reordering) των δεδομένων στη GPU και στη CPU πριν και μετά το τέλος των υπολογισμών. Στη πρότυπη υλοποίηση για GPU υπάρχει η δυνατότητα για επαναυπολογισμό (recomputation) των πινάκων l, r, b, t ή και του πίνακα ω στη GPU για τη μείωση των απαιτήσεων σε μνήμη ενώ στη πρότυπη υλοποίηση για CPU υπάρχει μόνο η δυνατότητα επαναυπολογισμού των πινάκων l, r, b, t. Στην υβριδική υλοποίηση, σύμφωνα και με τις πρότυπες υλοποιήσεις, υπάρχει η δυνατότητα πραγματοποίησης επαναυπολογισμών στο μέρος του πλέγματος στη CPU και στο μέρος στη GPU αλλά όχι ανεξάρτητα, δηλαδή αν επιλεγεί μερικός (RECALC_LEVEL = 1) ή ολικός (RECALC_LEVEL = 2) επαναυπολογισμός στη GPU, θα πρέπει να γίνεται και επαναυπολογισμός στη CPU (PRECALC = 1) και αντίστροφα.

Οι υπολογισμοί σε κάθε επανάληψη προχωρούν παράλληλα σε CPU και GPU. Μέσω του OpenMP δημιουργούνται στη CPU όσα νήματα (threads) επιτρέπεται απο το σύστημα. Όπως και στην αρχική υλοποίηση για CPU μόνο, το κάθε νήμα υπολογίζει το δικό του τετράγωνο κομμάτι (block) δεδομένων με τη διαφορά τώρα ότι το κυρίως νήμα (master thread) αναλαμβάνει το ρόλο της διαχείρισης της GPU επιπλέον του υπολογισμού του δικού του block. Εκμεταλλευόμενοι το γεγονός ότι στη CUDA οι κλήσεις των kernels απο το νήμα της CPU (host thread) είναι ασύγχρονες, δηλαδή ο έλεγχος επιστρέφεται άμεσα στο νήμα μετά τη κλήση του kernel, επιτυγχάνουμε καλύτερη επικάλυψη των υπολογισμών σε CPU και GPU εφόσον το master thread μπορεί άμεσα (με ελάχιστο overhead) να ξεκινήσει τον υπολογισμό του δικού του block δεδομένων χωρίς να περιμένει τη GPU, όπως γίνεται και με τα υπόλοιπα νήματα (slave threads) που δεν έχουν αυτο το ρόλο. Οι αντιγραφές δεδομένων μεταξύ CPU και GPU μπορούν επίσης να είναι ασύγχρονες κάτι που εκμεταλλευόμαστε στον υπολογισμό της νέας σύγκλισης της GPU σε κάθε επανάληψη. Οι υπολογισμοί στη CPU ξεκινούν ταυτόχρονα με τη κλήση των kernels για τους υπολογισμούς στη GPU. Στο τέλος κάθε επανάληψης και πρίν απο τον έλεγχο σύγκλισης και την ανταλλαγή γραμμών πραγματοποιείται συγχρονισμός CPU-GPU με τη μία να περιμένει την άλλη να τελειώσει τους υπολογισμούς της. Προφανώς η κατανομή του φόρτου εργασίας (load balancing) σε CPU και GPU πρέπει να είναι κατάλληλη ώστε να ελαχιστοποιείται αυτός ο χρόνος αναμονής για την επίτευξη των βέλτιστων επιδόσεων. Για τον έλεγχο σύγκλισης, στο τέλος κάθε επανάληψης η CPU και η GPU υπολογίζουν τη σύγκλιση στο δικό τους κομμάτι του πλέγματος. Μετά τα αποτελέσματα συνδυάζονται για να προκύψει η σύγκλιση για ολόκληρο το πλέγμα. Κάθε ένα προκαθορισμένο αριθμό επαναλήψεων πραγματοποιείται ανταλλαγή ενός προκαθορισμένου αριθμού συνοριακών γραμμών (R_EXCH) μεταξύ CPU και GPU για τη διατήρηση του υπολογισμού της LMSOR σε ολόκληρο το πλέγμα. Η ανταλλαγή αυτή πραγματοποιείται απο CPU → GPU με τα red σημεία του κομματιού του πλέγματος στη CPU να αντιγράφονται στα red σημεία του πλέγματος στη GPU και αντίστοιχα τα black σημεία στη CPU να αντιγράφοναι στα black σημεία στη GPU καθώς και αντίστροφα απο GPU → CPU. Πραγματοποιούνται δηλαδή συνολικά 2 + 2 = 4 αντιγραφές πάνω απο το δίαυλο PCI-E. Ο χρόνος που απαιτείται για τις αντιγραφές επηρρεάζει φυσικά αρνητικά την απόδοση και πρέπει να είναι όσο το δυνατόν μικρότερος. Επιπλέον, μεγαλύτερο R_EXCH οδηγεί σε πρόσθετους υπολογισμούς που μπορούν επίσης να επηρρεάσουν αρνητικά την απόδοση.

2.2 Μετρήσεις

Θέλουμε να εξετάσουμε την απόδοση της υλοποίησης για μικρού, μεσαίου και μεγάλου μεγέθους προβλήματα. Για το σκοπό αυτό γίνονται μετρήσεις για 3 ενδεικτικά μεγέθη πλέγματος (N = 504, 2054, 6144). Δοκιμάζεται ακόμη ένα πλέγμα αρκετό μεγάλο (N = 8194) ώστε να μη χωράει στη μνήμη της GPU οπότε δεν είναι δυνατόν να λυθεί μόνο με αυτήν. Στόχος είναι να εξετάσουμε τα οφέλη της υβριδικής υλοποίησης σε μια τέτοια περίπτωση. Για κάθε μέγεθος πλέγματος κρατάμε το εύρος ζώνης και το χρόνο που επιτυγχάνεται αν η επίλυση γίνει μόνο στη GPU και μόνο στη CPU τρέχοντας τις αντοίστιχες υλοποιήσεις. Το αθροιστικό εύρος ζώνης απο τις δύο υλοποίησεις (aggregate bandwidth) θα είναι πρακτικά το μέγιστο δυνατό (Max. Bandwidth) που θα ανεμενόταν να επιτευχθεί απο την υβριδική υλοποίηση εάν είχε 100% απόδοτικότητα (efficiency).

Πραγματοποιούνται μετρήσεις για την εύρεση της βέλτιστης κατανομής φόρτου και του βέλτιστου αριθμού και συχνότητας ανταλλαγής γραμμών για τα οποία αποδίδει καλύτερα η υβριδική υλοποίηση. Για το σκοπό αυτό το πρόγραμμα εκτελείται μερικές φορές, αυξάνοντας σε κάθε εκτέλεση τον αριθμό των γραμμών του πλέγματος που ανατίθενται για υπολογισμό στη CPU (NCPU) μέχρι να παρατηρηθεί μείωση της απόδοσης. Η τιμή του NCPU για την οποία επιτυγχάνεται το μεγαλύτερο εύρος ζώνης θεωρείται η βέλτιστη κατανομή φόρτου. Μόλις βρεθεί η βέλτιστη κατανομή φόρτου προχωράμε στην εύρεση του βέλτιστου αριθμού και συχνότητας ανταλλαγής γραμμών, αυξάνοντας σε κάθε εκτέλεση τον αριθμό των γραμμών που ανταλλάσονται μεταξύ CPU και GPU σε κάθε ίσο αριθμό επαναλήψεων (R_EXCH). Το έυρος ζώνης που θα προκύψει μετά απο τις δύο προαναφερθείσες μετρήσεις θα είναι το μέγιστο εύρος ζώνης της υβριδικής υλοποίησης και συγκρίνεται με το αναμενόμενο μέγιστο δυνατό ώστε να έχουμε μια εικόνα της αποδοτικότητάς της. Ακολουθεί σύγκριση του εύρους ζώνης και του χρόνου εκτέλεσης της υβριδικής υλοποίησης με αυτό της πιο γρήγορης αρχικής υλοποίησης (GPU ή CPU) για να δούμε τι κέρδος έχουμε.

Για να επιβεβαιώσουμε την βέλτιστη κατανομή γραμμών που προέκυψε απο τις δοκιμές κάνουμε ανάλυση της εφαρμογής κατα το χρόνο εκτέλεσης (runtime) με χρήση του nvidia profiler για τις ίδιες κατανομές γραμμών που δοκιμάσαμε προηγουμένως. Στόχος είναι να πάρουμε τους χρόνους υπολογισμών και μεταφοράς δεδομένων της GPU ώστε να βρούμε το χρόνο αδράνειάς της μέσω της σύγκρισης με το συνολικό χρόνο εκτέλεσης της εφαρμογής. Η κατανομή γραμμών για την οποία ο χρόνος αδράνειας θα

είναι ο ελάχιστος θα είναι η βέλτιστη κατανομή και αναμένεται να είναι η ίδια με αυτή που θα έχει βρεθεί απο τις δοκιμές.

Οι μετρήσεις πραγματοποιήθηκαν για τιμή του συντελεστή Reynold Re = 10.0. Πραγματοποιείται μερικός επαναυπολογισμός στη GPU (RECALC_LEVEL = 1) και στη CPU (PRECALC = 1). Για τους υπολογισμούς στη GPU χρησιμοποιείται ο 2^{ος} kernel για χρήση texture memory ενώ για τους υπολογισμούς στη CPU γίνεται χρήση των επεκτάσεων SSE2. Όλες οι μετρήσεις πραγματοποιήθηκαν σε 64bit Ubuntu Linux. Το compilation της υβριδικής υλοποίησης έγινε με τον nvcc compiler του CUDA toolkit έκδοσης 5.5 με όλα τα απαραίτητα optimization flags ενεργά (`-O2 -fomit-framepointer -ftree-vectorize -msse2 –msse -funroll-loops -fassociative-math -fno-signed-zeros -fno-trapping-math -fnosignaling-nans`) και τα απαραίτητα flags για χρήση του OpenMP (`-fopenmp` κατά το στάδιο του compilation και `–lgomp` κατά το στάδιο του linking). Για μεγάλα πλέγματα απαιτείται η υποστήριξη προγραμμάτων που το συνολικό μεγεθός τους στη μνήμη (κώδικας + δεδομένα) ξεπερνά τα 2 GB. Για το σκοπό αυτό, εκτός φυσικά απο ένα 64-bit περιβάλλον, απαιτείται και η υποστήριξη απο τον compiler κάτι που γίνεται με τη δήλωση του flag –mcmodel=medium. Για όλα τα flags που αφορούν τον host compiler (gcc) προηγείται η εντολή `–Xcompiler` ώστε να περαστούν απευθείας απο τον nvcc σε αυτόν.

Το hardware που χρησιμοποιήθηκε για τις δοκιμές ήταν ένας Intel Core i7-3770K (3.5Ghz) και μία NVidia GTX 680.

2.2.1 Μικρό πρόβλημα (N = 504)

Για μικρό μέγεθος προβλήματος που χωράει στη cache της CPU αναμένεται να δούμε τα πλεονεκτήματα που αυτή προσφέρει στη ταχύτητα υπολογισμών της CPU. Αντιθέτως, η εκμετάλλευση των πόρων της GPU δεν αναμένται να είναι ιδανική με μικρό μεγέθος προβλήματος καθώς ο υπολογιστικός φόρτος είναι μικρός.

GPU-only version bandwidth = 59.07 GB/s
CPU-only version bandwidth = 70.45 GB/s

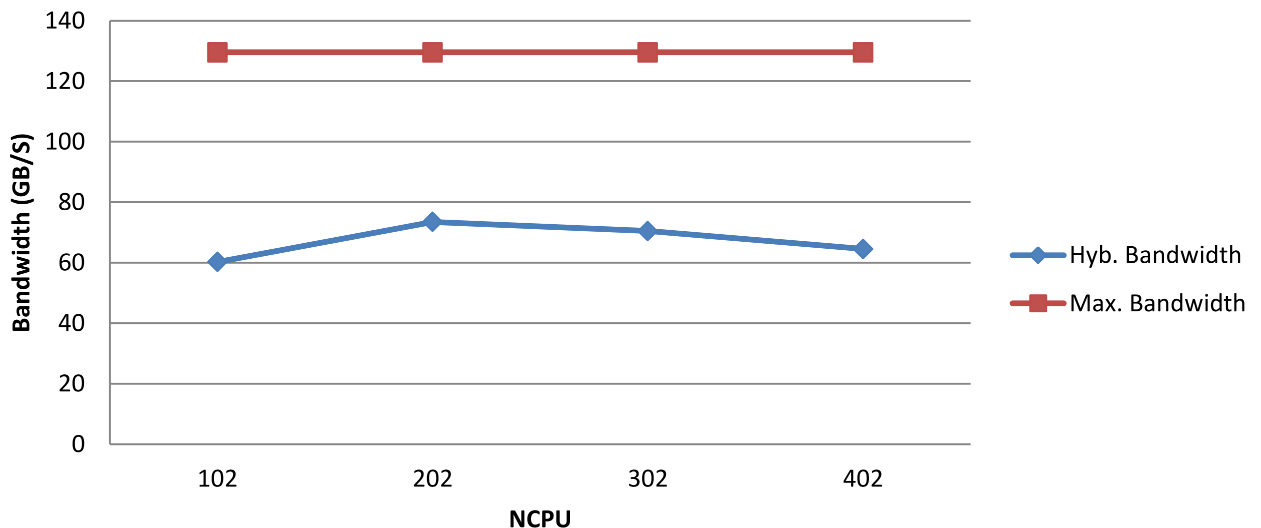
Aggregate bandwidth = 129.52 GB/s

Για μικρό μέγεθος προβλήματος το οποίο χωράει όλο στη cache της CPU διαφαίνεται το πλεονέκτημα που προσφέρει ο υψηλός ρυθμός μεταγωγής της cache. Αντιθέτως,

βλέπουμε ότι η GPU δεν αποδίδει τόσο καλά σε περιπτώσεις χαμηλού υπολογιστικού φόρτου.

Πίνακας 1

NCPU	Hybrid ver. bandwidth (GB/s)
102	60.25
202	73.45
302	70.44
402	64.50



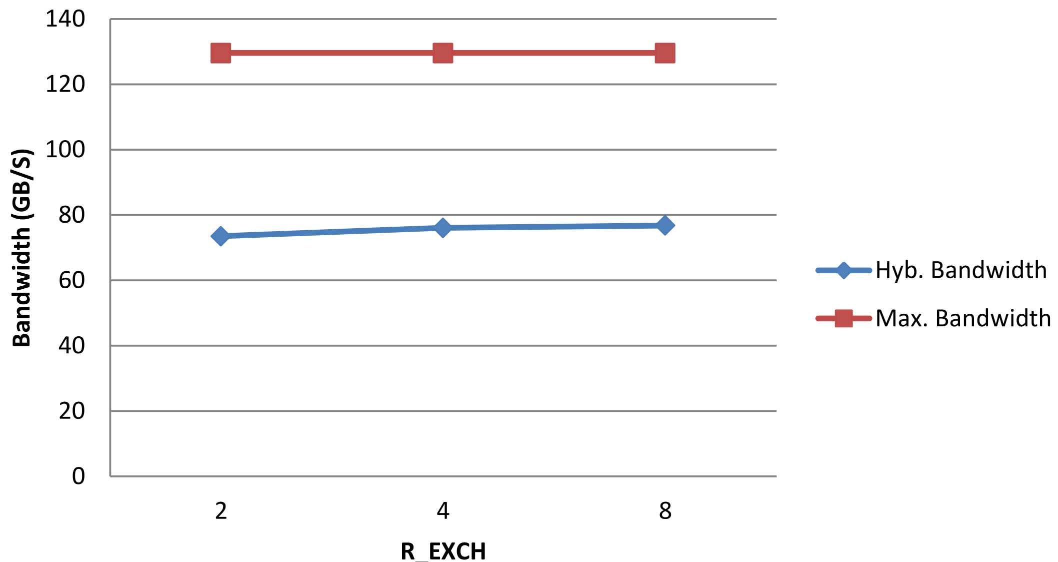
Σχήμα 9: Εύρος ζώνης συναρτήσει του διαμοιρασμού γραμμών του πλέγματος σύμφωνα με το πόσες απο αυτές ανατίθενται στη CPU (N = 504)

Επιλέγουμε NCPU = 202 για το οποίο έχουμε τη μεγαλύτερη επιτάχυνση και τώρα μεταβάλλουμε το R_EXCH:

Πίνακας 2

R_EXCH	Hybrid ver. bandwidth (GB/s)
2	73.45
4	76.00
8	76.75

Μειώνοντας το ρυθμό ανταλλαγής γραμμών σε R_EXCH = 4 παρατηρούμε μια πολύ μικρή αύξηση της απόδοσης. Μεταξύ R_EXCH = 4 και 8 δεν υπάρχει ουσιαστική διαφορά.



Σχήμα 10: Εύρος ζώνης συναρτήσει του αριθμού γραμμών που ανταλλάσονται σε ίσο αριθμό επαναλήψεων (N = 504)

Η μέγιστη επιτάχυνση (76.75 GB/s) επιτυγχάνεται για NCPU = 202 και R_EXCH = 8 και είναι το 59.3 % της μέγιστης δυνατής (129.52 GB/s). Η επιτάχυνση είναι επίσης 9% μεγαλύτερη απο αυτή που επιτυγχάνεται μόνο με τη CPU (70.45 GB/s).

Οι χρόνοι εκτέλεσης έχουν ως εξής:

Πίνακας 3

Χρόνος εκτέλεσης μόνο με CPU (s)	Χρόνος εκτέλεσης με υβριδικό (s)
0.084389	0.072534

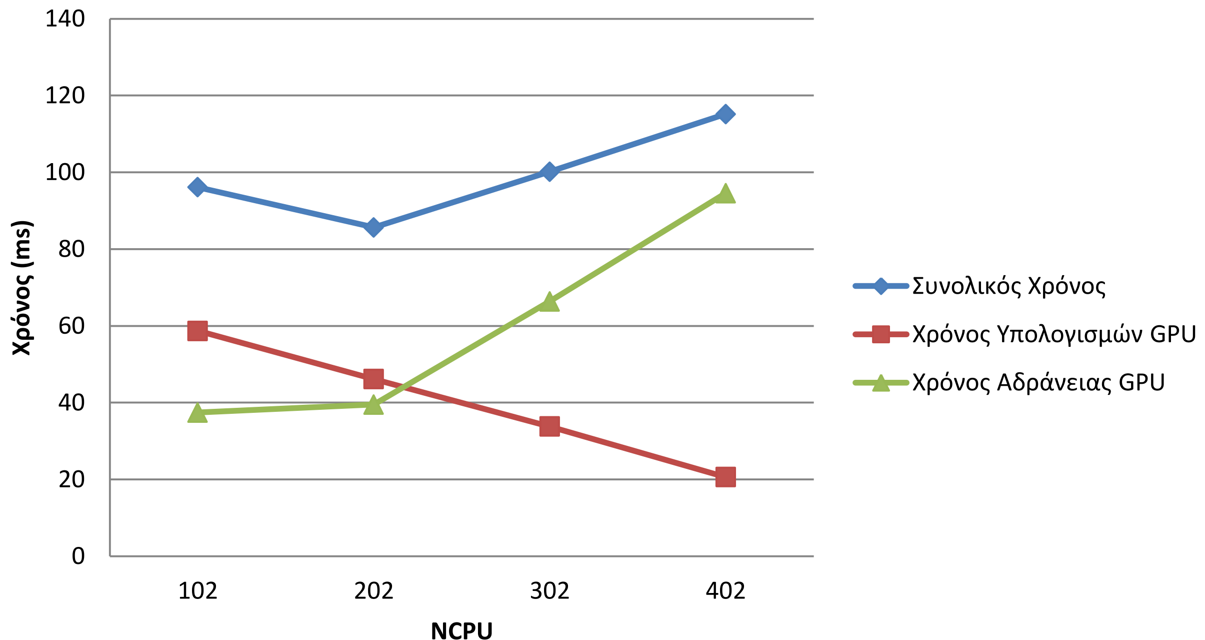
Με την υβριδική υλοποίηση επιτυγχάνεται ~16% καλύτερος χρόνος για μικρό πρόβλημα.

Πίνακας 4

NCPU	Συνολικός χρόνος (ms)	Χρόνος υπολογισμών GPU (ms)	Χρόνος αδράνειας GPU (ms)
102	96.09	58.68	37.41
202	85.62	46.12	39.50
302	100.14	33.78	66.35
402	115.15	20.59	94.56

Ο μικρότερος χρόνος αδράνειας προκύπτει για NCPU = 102. Ομως, για NCPU = 202, ο χρόνος αδράνειας είναι ~ 2 ms μεγαλύτερος αλλά ο χρόνος υπολογισμών της GPU είναι

~ 12 ms μικρότερος κι έτσι ο συνολικός χρόνος προκύπτει μικρότερος απο το χρόνο για NCPU = 101.



Σχήμα 11: Συνολικός χρονος υπολογισμών, Χρόνος υπολογισμών της GPU και χρόνος αδράνειας της GPU συναρτήσει του αριθμού γραμμών που ανατίθενται στη CPU (N = 504)

2.2.2 Μεσαίο πρόβλημα (N = 2054)

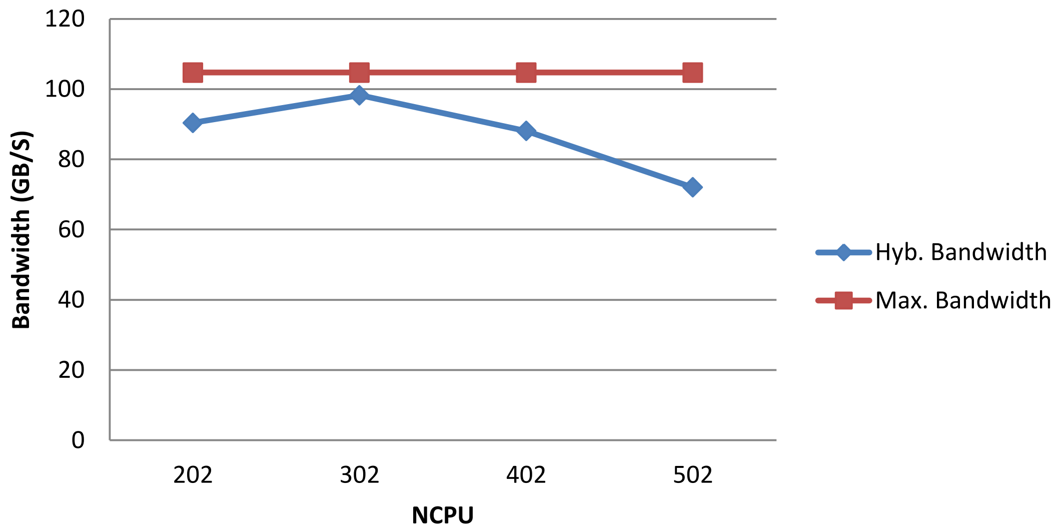
Για μεσαίο μέγεθος προβλήματος, το οποίο δε χωράει στη cache της CPU, αναμένεται μεγάλη πτώση στην απόδοση της CPU καθώς χάνεται το πλεονέκτημα της cache που υπήρχε για το μικρό πρόβλημα. Η εκμετάλλευση των πόρων της GPU αναμένεται να είναι καλύτερη λόγω του μεγαλύτερου υπολογιστικού φόρτου.

GPU version bandwidth = 85.51 GB/s
CPU version bandwidth = 19.21 GB/s

Aggregate bandwidth = 104.72 GB/s

Πίνακας 5

NCPU	Hybrid ver. bandwidth (GB/s)
202	90.37
302	98.24
402	88.05
502	72.00

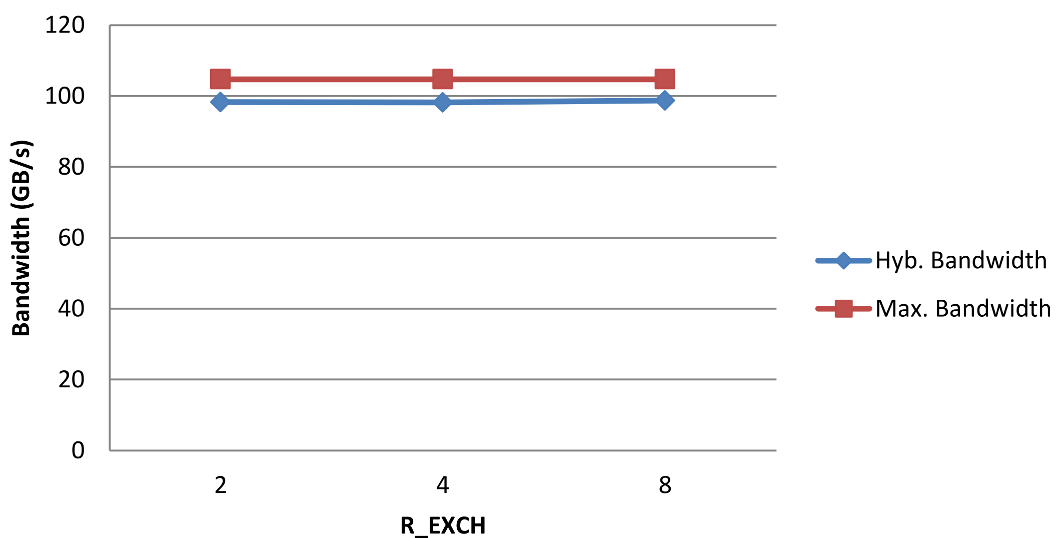


Σχήμα 12: Εύρος ζώνης συναρτήσει του διαμοιρασμού γραμμών του πλέγματος σύμφωνα με το πόσες απο αυτές ανατίθενται στη CPU (N = 2054)

Επιλέγουμε NCPU = 302 για το οποίο έχουμε τη μεγαλύτερη επιτάχυνση και μεταβάλλουμε τώρα το R_EXCH:

Πίνακας 6

R_EXCH	Hybrid ver. bandwidth (GB/s)
2	98.24
4	98.14
8	98.55



Σχήμα 13: Εύρος ζώνης συναρτήσει του αριθμού γραμμών που ανταλλάσονται σε ίσο αριθμό επαναλήψεων (N = 2054)

Η μέγιστη επιτάχυνση (98.55 GB/s) επιτυγχάνεται για NCPU = 302 και R_EXCH = 8, αν και η μεταβολή της επιτάχυνσης με το R_EXCH είναι στα όρια του στατιστικού λάθους και μπορούμε να θεωρήσουμε πως δεν παίζει ρόλο. Η επιτάχυνση που τελικά επιτυγχάνεται είναι το 94.1 % της μέγιστης δυνατής (104.72 GB/s). Η επιτάχυνση είναι επίσης 15% μεγαλύτερη απο αυτή που επιτυγχάνεται μόνο με τη GPU (85.51 GB/s).

Οι χρόνοι εκτέλεσης έχουν ως εξής:

Πίνακας 7

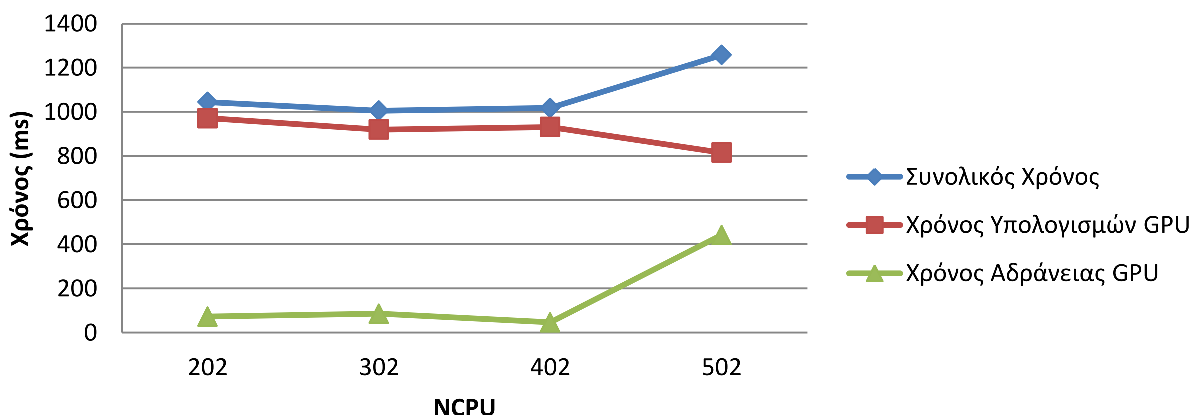
Χρόνος εκτέλεσης μόνο με GPU (s)	Χρόνος εκτέλεσης με υβριδικό (s)
1.143069	0.996451

Με την υβριδική υλοποίηση επιτυγχάνεται 15% καλύτερος χρόνος για μεσαίο πρόβλημα.

Πίνακας 8

NCPU	Συνολικός χρόνος (ms)	Χρόνος υπολογισμών GPU (ms)	Χρόνος αδράνειας GPU (ms)
202	1044.37	971.60	72.78
302	1005.43	919.60	85.83
402	1017.80	931.26	86.54
502	1257.93	815.67	442.26

Ο μικρότερος χρόνος αδράνειας προκύπτει για NCPU = 202. Ομως, για NCPU = 302, ο χρόνος αδράνειας είναι ~ 13 ms μεγαλύτερος αλλά ο χρόνος υπολογισμών της GPU είναι ~ 52 ms μικρότερος κι έτσι ο συνολικός χρόνος είναι μικρότερος απο το χρόνο για NCPU = 202.



Σχήμα 14: Συνολικός χρονος υπολογισμών, Χρόνος υπολογισμών της GPU και χρόνος αδράνειας της GPU συναρτήσει του αριθμού γραμμών που ανατίθενται στη CPU (N = 2054)

2.2.3 Μεγάλο πρόβλημα (N = 6144)

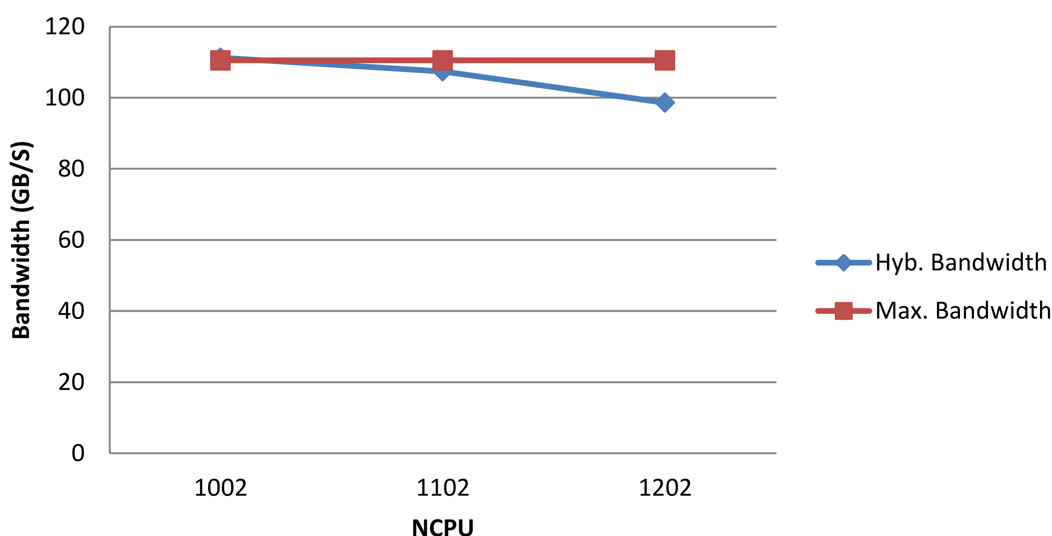
Για μεγάλο πρόβλημα αναμένεται να έχουμε τη καλύτερη αξιοποίηση των πόρων της GPU λόγω του μεγάλου υπολογιστικού φόρτου.

GPU version bandwidth = 92.83 GB/s
CPU version bandwidth = 17.67 GB/s

Aggregate bandwidth = 110.50 GB

Πίνακας 9

NCPU	Hybrid ver. bandwidth (GB/s)
1002	111.20
1102	107.41
1202	98.56

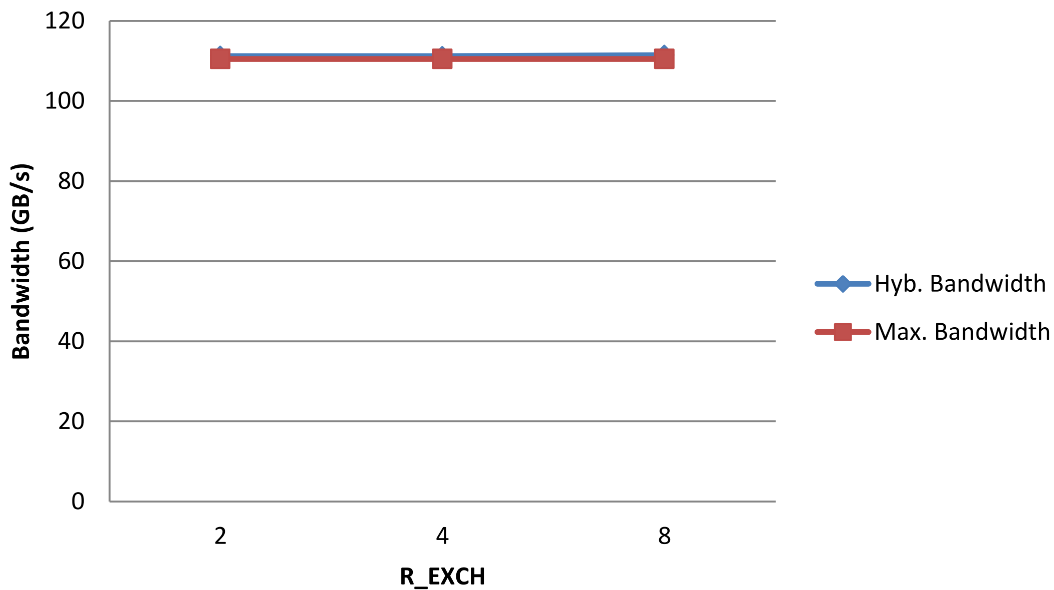


Σχήμα 15: Εύρος ζώνης συναρτήσει του διαμοιρασμού γραμμών του πλέγματος σύμφωνα με το πόσες απο αυτές ανατίθενται στη CPU (N = 6144)

Επιλέγουμε NCPU = 1002 για το οποίο έχουμε τη μεγαλύτερη επιτάχυνση και τώρα μεταβάλλουμε το R_EXCH:

Πίνακας 10

R_EXCH	Hybrid version bandwidth (GB/s)
2	111.20
4	111.17
8	111.43



Σχήμα 16: Εύρος ζώνης συναρτήσει του αριθμού γραμμών που ανταλλάσονται σε ίσο αριθμό επαναλήψεων (N = 6144)

Παρατηρούμε ότι και πάλι η διαφοροποίηση στην επιτάχυνση μεταξύ των διαφορετικών τιμών R_EXCH είναι μέσα στα όρια του στατιστικού λάθους. Η μέγιστη επιτάχυνση που επιτυγχάνεται (111.43 GB/s) με R_EXCH = 8 είναι το 100.8 % της μέγιστης δυνατής (110.50 GB/s). Η επιτάχυνση είναι επίσης 20% μεγαλύτερη απο αυτή που επιτυγχάνεται μόνο με τη GPU (92.83 GB/s).

Οι χρόνοι εκτέλεσης έχουν ως εξής:

Πίνακας 11

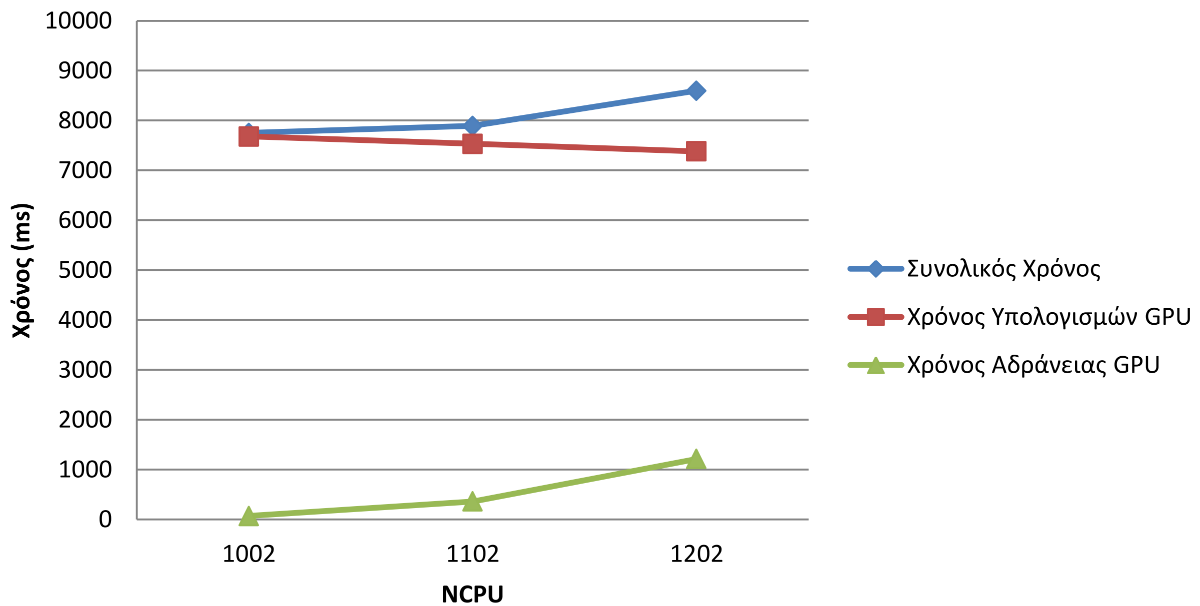
Χρόνος εκτέλεσης μόνο με GPU (s)	Χρόνος εκτέλεσης με υβριδικό (s)
9.135565	7.768092

Με την υβριδική υλοποίηση επιτυγχάνεται 18% καλύτερος χρόνος για μεσαίο πρόβλημα.

Πίνακας 12

NCPU	Συνολικός χρόνος (ms)	Χρόνος υπολογισμών GPU (ms)	Χρόνος αδράνειας GPU (ms)
1002	7750.93	7680.11	69.89
1102	7893.51	7532.36	361.16
1202	8596.08	7382.23	1213.86

Ο μικρότερος χρόνος αδράνειας προκύπτει για NCPU = 1002, όπως και ο μικρότερος συνολικός χρόνος.



Σχήμα 17: Συνολικός χρονος υπολογισμών, Χρόνος υπολογισμών της GPU και χρόνος αδράνειας της GPU συναρτήσει του αριθμού γραμμών που ανατίθενται στη CPU (N = 6144)

2.2.4 Μεγάλο πρόβλημα που δε χωράει στη μνήμη της GPU (N = 8194)

Για ένα πρόβλημα τόσο μεγάλο που να μην είναι δυνατός ο υπολογισμός του μόνο με τη GPU, εφόσον δε χωράει στη μνήμη της, αναμένεται να φανεί καθαρά το πλεονέκτημα της υβριδικής υλοποίησης εφόσον είναι δυνατή η ανάθεση κάποιον γραμμών του πλέγματος στη CPU για υπολογισμό έτσι ώστε το υπόλοιπο πλέγμα να μπορεί να υπολογιστεί με τη GPU. Με το τρόπο αυτό αναμένεται η επίτευξη σημαντικής επιτάχυνσης σχετικά με την εναλλακτική λύση του να υπολογιζόταν όλο το πλέγμα στη CPU.

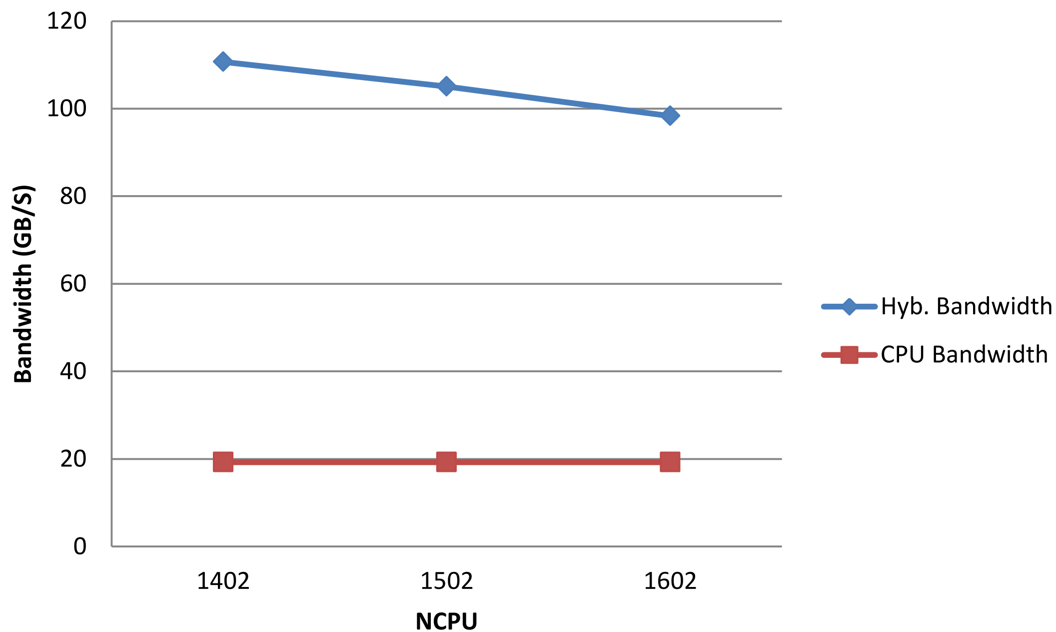
GPU version bandwidth = 0.00 GB/s (Μη επαρκής μνήμη GPU)
CPU version bandwidth = 19.28 GB/s

Aggregate bandwidth = 19.28 GB/s

Πίνακας 13

NCPU	Hybrid ver. bandwidth (GB/s)
1002	0.00 (μη επαρκής μνήμη GPU)
1302	0.00 (μη επαρκής μνήμη GPU)
1402	110.69
1502	105.06
1602	98.33

Οι μετρήσεις για N = 8194 έγιναν με R_EXCH = 8 που, σύμφωνα με τις προηγηθείσες μετρήσεις για τα μικρότερα πλέγματα, φένεται να δίνει την καλύτερη απόδοση.



Σχήμα 18: Εύρος ζώνης συναρτήσει του διαμοιρασμού γραμμών του πλέγματος σύμφωνα με το πόσες απο αυτές ανατίθενται στη CPU (N = 8194)

Για μεγέθη πλεγμάτων που δε χωρούν στη GPU, παρατηρούμε ότι η υβριδική υλοποίηση προσφέρει σημαντικά οφέλη στη ταχύτητα σε σχέση με την επίλυση μόνο με τη χρήση της CPU. Συγκεκριμένα η υβριδική υλοποίηση είναι 574% ταχύτερη απο την υλοποίηση μόνο για CPU.

Οι χρόνοι εκτέλεσης έχουν ως εξής:

Πίνακας 14

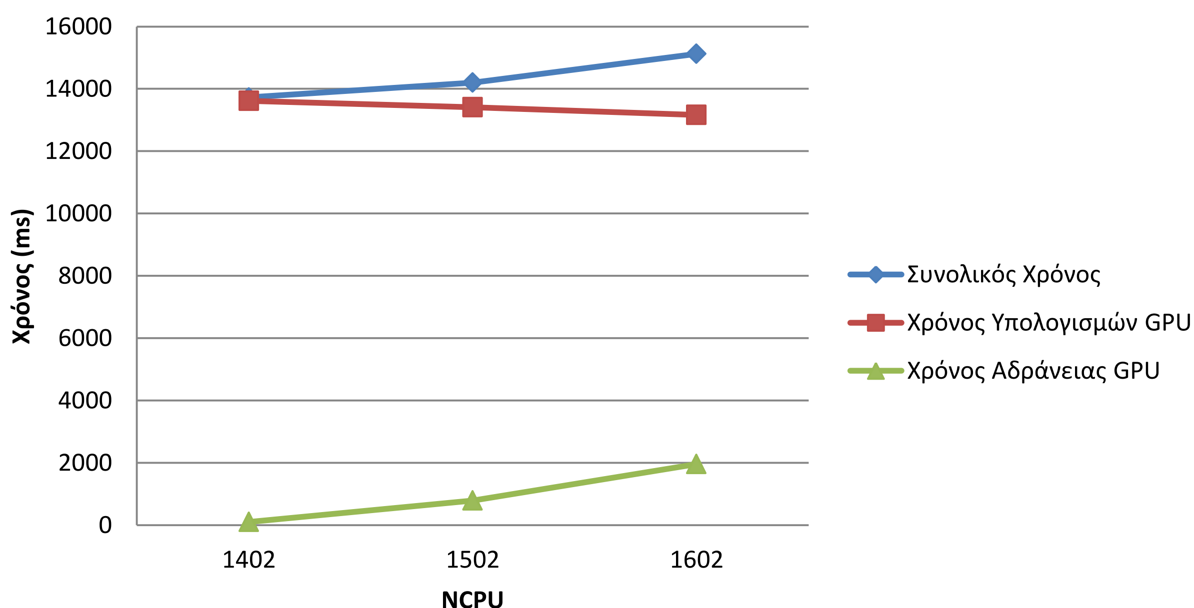
Χρόνος εκτέλεσης μόνο με CPU (s)	Χρόνος εκτέλεσης με υβριδικό (s)
77.411364	13.809092

Με την υβριδική υλοποίηση επιτυγχάνεται 561% καλύτερος χρόνος για μεγάλο πρόβλημα που δε μπορεί να λυθεί μόνο με τη GPU, επιτάχυνση που συμφωνεί και με αυτή του εύρους ζώνης.

Πίνακας 15

NCPU	Συνολικός χρόνος (ms)	Χρόνος υπολογισμών GPU (ms)	Χρόνος αδράνειας GPU (ms)
1402	13724.34	13618.00	106.34
1502	14204.13	13411.67	792.46
1602	15127.48	13163.64	1963.84

Ο μικρότερος χρόνος αδράνειας προκύπτει για NCPU = 1402, όπως και ο μικρότερος συνολικός χρόνος.



Σχήμα 19: Συνολικός χρονος υπολογισμών, Χρόνος υπολογισμών της GPU και χρόνος αδράνειας της GPU συναρτήσει του αριθμού γραμμών που ανατίθενται στη CPU (N = 6144)

3. ΣΥΜΠΕΡΑΣΜΑΤΑ

Οι υβριδικές υλοποιήσεις προσφέρουν αύξηση της ταχύτητας υπολογισμών για την επίλυση συστημάτων γραμμικών εξισώσεων με τη χρήση επαναληπτικών μεθόδων, επιτρέποντας την εκμετάλλευση τόσο της GPU όσο και της CPU και επιτυγχάνοντας έτσι πληρέστερη αξιοποίηση του διαθέσιμου υλικού σε ένα υπολογιστικό σύστημα. Η κλιμάκωση της υβριδικής υλοποίησης είναι πολύ καλή για μεσαία και μεγάλου μεγέθους πλέγματα και αγγίζει τη συνολική διαθέσιμη απόδοση της GPU και της CPU, όπως προκύπτει απο τις πρότυπες χωριστές υλοποιήσεις. Ειδικά για προβλήματα επαρκώς μεγάλα ώστε να μη μπορούν να επιλυθούν με τη GPU λόγω μη αρκούντως διαθέσιμου χώρου στη μνήμη, η υβριδική υλοποίηση προσφέρει σημαντικό πλεονέκτημα εφόσον ένα τμήμα του πλέγματος μπορεί να ανατεθεί στη CPU και το υπόλοιπο στη GPU. Έτσι επιτυγχάνεται μεγάλη επιτάχυνση σε σχέση με το να γίνονταν όλοι οι υπολογισμοί στη CPU. Για μικρά πλέγματα δεν αποκομίζουμε ιδιαίτερα οφέλη απο τη χρήση υβριδικής υλοποίησης.

Απο τα πειραματικά δεδομένα προκύπτει επίσης ότι ο αριθμός και συχνότητα ανταλλαγής γραμμών θα πρέπει να επιλέγεται τουλάχιστον ίσος με R_EXCH = 8 ενώ η κατανομή φόρτου, εκφρασμένη μέσω των γραμμών που ανατίθενται στη CPU (NCPU), θα μεταβάλλεται, όπως είναι ανεμενόμενο, ανάλογα με το μέγεθος του εκάστοτε προβλήματος και απαιτείται κάποιος πειραματισμός για την εύρεση της βέλτιστης κατανομής για συγκεκριμένο μέγεθος προβλήματος.

Σημεία που έχρηζαν ιδιαίτερης προσοχής κατα την υλοποίηση ήταν ο επιπλέον χρόνος (overhead) που απαιτούταν για τις ανταλλαγές δεδομένων καθώς και η επίτευξη της καλύτερης δυνατής επικάλυψης υπολογισμών της GPU και CPU, λαμβάνοντας υπόψη και το γεγονός ότι ένα απο τα thread αναλάμβανε και το ρόλο της διαχείρισης της GPU πέρα απο τον υπολογιστικό του ρόλο. Το overhead για τις ανταλλαγές δεδομένων, στο σύστημα που έγιναν οι δοκιμές, συνεισέφερε <1% του συνολικού χρόνου εκτέλεσης και μπορεί να θεωρηθεί αμελητέο ενώ μετά απο δοκιμές επιτεύχθηκε η υλοποίηση του κώδικα που προσφέρει τη βέλτιστη επικάλυψη υπολογισμών ελαχιστοποιώντας το χρόνο αναμονής.

Απο την ανάλυση του χρόνου εκτέλεσης, προκύπτει ότι, για μικρό και μεσαίο πρόβλημα, η απόδοση είναι εξαρτώμενη απο τη σχέση του χρόνου αδράνειας της GPU με το χρόνο υπολογισμών της. Δηλαδή, μικρότερος χρόνος υπολογισμών δε σημαίνει μικρότερο συνολικό χρόνο αν ο χρόνος αδράνειας είναι μεγάλος και αντίστοιχα, ενάς μικρότερος χρόνος αδράνειας δε σημαίνει απαραίτητα μικρότερο συνολικό χρόνο αν ο χρόνος

υπολογισμών είναι μεγάλος. Για μεγάλο πρόβλημα ο χρόνος εκτέλεσης φαίνεται να εξαρτάται μόνο απο το χρόνο αδράνειας.

Σε αυτή την εργασία χρησιμοποιείται μόνο μία CPU και μία GPU για τους υπολογισμούς. Η χρήση περισσότερων CPUs, αν υπάρχουν, στο ίδιο σύστημα είναι τεττριμένη και γίνεται αυτόματα απο το OpenMP, ενώ η χρήση περισσότερων GPUs θα απαιτούσε και την ανταλλαγή δεδομένων μεταξύ των GPUs και επιπλέον στρατηγικές επικάλυψης υπολογισμών. Με μία υβριδική υλοποίηση MPI – OpenMP – CUDA θα ήταν δυνατή η κατανομή των υπολογισμών σε πολλαπλά συστήματα – κόμβους (nodes) που το καθένα διαθέτει CPU(s) ή/και GPU(s) και η ανταλλαγή δεδομένων γίνεται μέσω κάποιου καναλιού επικοινωνίας (π.χ. Ethernet) για την επίλυση πολύ μεγάλων προβλημάτων. Μία τέτοια υλοποίηση θα ήταν σαφώς πιο περίπλοκη καθώς θα απαιτούσε πιο «έξυπνες» στρατηγικές επικάλυψης υπολογισμών και ανταλλαγής δεδομένων μεταξύ των διαφορετικών κόμβων λόγω της εμπλοκής των συνήθως πιο αργών, σε σχέση με τα ενδοκομβικά, καναλιών επικοινωνίας.

ΠΙΝΑΚΑΣ ΟΡΟΛΟΓΙΑΣ

Ξενόγλωσσος όρος	**Ελληνικός Όρος**
Hybrid	Υβριδικό
Kernel	Πυρήνας
Granularity	Διακριτότητα
Bandwidth	Εύρος Ζώνης
Compiler	Μεταγλωττιστής
Shared memory	Κοινή μνήμη
Distributed memory	Κατανεμημένη μνήμη
Local memory	Τοπική μνήμη
Constant memory	Διαρκής μνήμη
Texture memory	Μνήμη υφών
Memory space	Περιοχή μνήμης
Multithreaded	Πολυνηματικό
Iterative methods	Επαναληπτικές μέθοδοι
Data Parallelism	Παραλληλισμός δεδομένων
Task Parallelism	Παραλληλισμός διεργασιών
Scalability	Δυνατότητα κλιμάκωσης
Graphics Rendering	Απόδοση Γραφικών
Pixel	Εικονοστοιχείο
Debugging	Απασφαλμάτωση

ΣΥΝΤΜΗΣΕΙΣ – ΑΡΚΤΙΚΟΛΕΞΑ – ΑΚΡΩΝΥΜΙΑ

LMSOR	Local Modified Successive Over-Relaxation
CUDA	Compute Unified Device Architecture
OpenCL	Open Computing Language
OpenMP	Open Multi-Processing
MPI	Message Passing Interface
nvcc	NVidia cuda compiler
gcc	GNU compiler collection
SM	Streaming Multiprocessor

ΑΝΑΦΟΡΕΣ

[1] Y. Cotronis et al., A comparison of CPU and GPU implementations for solving the Convection Diffusion equation using the local Modified SOR method, Parallel Comput. (2014), http://dx.doi.org/10.1016/j.parco.2014.02.002.

[2] Jason Sanders, Edward Kandrot, CUDA by Example, Addison-Wesley (2010).

[3] Mark Harris, How to Overlap Data Transfers in CUDA C/C++, Dec. 2012; http://devblogs.nvidia.com/parallelforall/how-overlap-data-transfers-cuda-cc.

[4] NVIDIA CUDA C Programming Guide v6.5, NVidia, 2014.

[5] "About the OpenMP ARB and". OpenMP.org. 2013-07-11.

[6] L.W. Ehrlich, An ad-hoc SOR method, J. Comput. Phys. 42 (1981) 31–45.

[7] L.W. Ehrlich, The Ad-Hoc SOR Method: A Local Relaxation Scheme, in Elliptic Problem Solvers II, Academic Press, New York, 1984. pp. 257–269.

[8] E.F. Botta, A.E.P. Veldman, On local relaxation methods and their application to convection-diffusion equations, J. Comput. Phys. 48 (1981) 127–149.