Πολυτεχνείο Κρήτης

Τμήμα Ηλεκτρονικών Μηχανικών

και Μηχανικών Ηλεκτρονικών Υπολογιστών

Παραλληλοποίηση Εφαρμογών σε Συστήματα Πολυπύρηνων Αρχιτεκτονικών με έμφαση στην Αρχιτεκτονική *CUDA*

Αναστάσιος Κατσούλας

Εξεταστική επιτροπή

Ιωάννης Παπαευσταθίου, καθηγητής (επιβλέπων)

Απόστολος Δόλλας, καθηγητής

Διονύσιος Πνευματικάτος, καθηγητής

Χανιά, Σεπτέμβρης 2011

``To put it quite blunty: as long as there were no machines, programming was no problem at all, when we had a few weak computers, programming became a mild problem, and now we have gigantic computers, programming has become an equally gigantic problem."

--E. Dijkstra, 1972 Turing Award Lecture

Περίληψη

Η αυξανόμενη ζήτηση για μεγαλύτερη απόδοση καθώς και η επίτευξη του φυσικού ορίου όσον αφορά την κατανάλωση ενέργειας και παραγωγή θερμότητας στο σχεδιασμό ολοκληρωμένων επεξεργαστικών συστημάτων, έχει ως αποτέλεσμα την αναζήτηση και ανάπτυξη νέων πολυπύρηνων συστημάτων, διαφορετικών αρχιτεκτονικών σε σχέση με τη σχεδιαστική προσέγγιση που ακολουθήθηκε τα τελευταία σαράντα πέντε (45) χρόνια. Στην εργασία αυτή θα εξετάσουμε την ετερόγενη πολυπύρηνη αρχιτεκτονική του Cell/Be (Cell Broadband Engine) της IBM, καθώς και την μαζικά παράλληλη αρχιτεκτονική της NVIDIA, ονόματι CUDA (Compute Unified Device Architecture).

Η πλατφόρμα του Cell Broadband Engine (Cell) είναι μια ετερόγενη πολυπύρηνη αρχιτεκτονική που αναπτύχθηκε από τις εταιρίες

IBM, Sony και Toshiba και είναι ικανή να επιτύχει εντυπωσιακές επιδόσεις σε υπολογιστικά απαιτητικές εφαρμογές. Χρησιμοποιείται μαζικά στην πλατφόρμα του Playstation3, καθώς και σε συστήματα εξυπηρετητών από την IBM (blade servers). Στην πλατφόρμα αυτή προσπαθήσαμε άκαρπα να παραλληλοποιήσουμε μια εφαρμογή αναγνώρισης δαχτυλικών αποτυπωμάτων που αναπτύχθηκε απο το National Institute of Standards and Technology (NIST). Οι λόγοι που δεν κατέστησαν δυνατή την ολοκλήρωση του εγχειρήματος θα αναλυθούν στη συνέχεια.

Η παράλληλη αρχιτεκτονική CUDA αναπτύχθηκε από την εταιρία NVIDIA και προσφέρει μεγάλη αύξηση στην απόδοση σε υπολογιστικά απαιτητικές εφαρμογές σε σχέση με την x86 αρχιτεκτονική, εκμεταλλευόμενη την επεξεργαστική ισχύ που προσφέρει το υποσύστημα της κάρτας γραφικών ενός υπολογιστή. Χρησιμοποιήσαμε την ισχύ αυτή για την παραλληλοποίηση ενός συστήματος πεπερασμένων διαφορών Crank-Nicholson για την επίλυση της μερικής διαφορικής εξίσωσης Black-Scholes, για την πρόβλεψη τιμών μετοχών. Τα αποτελέσματα από την προσπάθεια αυτή ήταν εντυπωσιακά από πλευράς απόδοσης και θα αναπτυχθούν στην πορεία της παρουσίασης μας.

Ευχαριστίες

Αρχικά θα ήθελα να ευχαριστήσω τη μητέρα μου Ευαγγελία και την αδερφή μου Χάρις για την στήριξη τους και τις θυσίες τους όλα αυτά τα χρόνια.

Ευχαριστώ πολύ τον επιβλέποντα καθηγητή κ.Ιωάννη Παπαευσταθίου για την κατανόηση που επέδειξε αλλά και την πολύτιμη καθοδήγηση του κατά την εκπόνηση της εργασίας αυτής. Επίσης ευχαριστώ τον καθηγητή κ Απόστολο Δόλλα και τον καθηγητή κ. Διονύση Πνευματικάτο για την ενασχόληση τους με την εργασία μου.

Τέλος ευχαριστώ πολύ την Αλεξάνδρα, τους φίλους μου Γιώργο και Πέτρο για όλες τις στιγμές που περάσαμε μαζί σε αυτό το μακρύ ταξίδι.

Περιεχόμενα

Κατάλογος σχημάτων

Listings

Κατάλογος πινάκων

1 Εισαγωγή

1.1 Γενικά

Τα τελευταία είκοσι (20) χρόνια βιώνουμε μια ραγδαία αύξηση στην απόδοση και στις δυνατότητες των υπολογιστικών συστημάτων. Βασικός λόγος είναι η τεχνολογική πρόοδος στην ανάπτυξη ολοκληρωμένων κυκλωμάτων ευρείας κλίμακας (VLSI - Very Large Scale Integration) που επιτρέπει μεγαλύτερο αριθμό τρανζίστορ και μεγαλύτερες συχνότητες χρονισμού σε ένα ολοκληρωμένο κύκλωμα.

Όμως η ανάπτυξη αυτή, δεν είναι δίχως τίμημα. Η αύξηση της πολυπλοκότητας έχει ως αποτέλεσμα, την αύξηση της κατανάλωσης ενέργειας καθώς και την έκκληση θερμότητας, στοιχεία που διαδραματίζουν βασικό ρόλο στην περαιτέρω εξέλιξη των ολοκληρωμένων κυκλωμάτων. Επίσης, ο φυσικός περιορισμός που εισάγεται στην περαιτέρω μείωση των τρανζίστορ καθώς και στην χωρητικότητα και ταχύτητα της μνήμης ενός υπολογιστικού συστήματος σε συνδυασμό με την διαρκή αναζήτηση για περαιτέρω απόδοση και ρεαλισμό στις υπολογιστικές επιστήμες, έχουν οδηγήσει την επιστημονική και βιομηχανική κοινότητα στην αναζήτηση εναλλακτικών λύσεων σε όρους απόδοσης και αποδοτικότητας των ολοκληρωμένων κυκλωμάτων.

Το κυρίαρχο χαρακτηριστικό τα τελευταία χρόνια στη σχεδίαση ολοκληρωμένων συστημάτων είναι η παραλληλοποίηση. Περισσότεροι υπολογιστικοί πόροι, που εργάζονται ταυτόχρονα για την εκτέλεση μιας εργασίας.

Η τάση που επικρατεί πλέον στην αρχιτεκτονική υπολογιστών είναι η ενσωμάτωση πολλών επεξεργαστικών μονάδων σε ένα ολοκληρωμένο κύκλωμα γεγονός που εν δυνάμει πολλαπλασιάζει την υπολογιστική ισχύ μιας επεξεργαστικής μονάδας. Με την τεχνολογική πρόοδο τα ολοκληρωμένα αυτά κυκλώματα γίνονται

αποδοτικότερα, φθηνότερα και πιο αξιόπιστα κάνοντας τις παράλληλες υπολογιστικές πλατφόρμες ευρέως διαδεδομένες.

Ένα ακόμη βασικό χαρακτηριστικό, όσον αφορά την επίτευξη μεγαλύτερης απόδοσης, είναι η αποθήκευση των δεδομένων. Τα δεδομένα, τα οποία επεξεργάζονται όλο και με μεγαλύτερους ρυθμούς πρέπει να αποθηκευτούν σε κάποιο σημείο στον υπολογιστή γεγονός που συνδέει την παράλληλη επεξεργασία με την τοπικότητα των δεδομένων (data locality) και την επικοινωνία (communication).

Επομένως, γίνεται σαφές ότι σε όρους απόδοσης το βάρος μετατοπίζεται στη σχεδίαση του λογισμικού. Συγκεκριμένα, το λογισμικό πρέπει να συντονίσει μια πληθώρα υπολογιστικών και επικοινωνιακών δραστηριοτήτων μεταξύ πολλαπλών επεξεργαστικών πυρήνων, διαφόρων επιπέδων μνημών και άλλων λειτουργικών μονάδων ώστε να αυξηθεί η αποδοτικότητα και ο συγχρονισμός, δηλαδή το ποσοστό των υπολογιστικών πράξεων (operations) που εκτελούνται ταυτόχρονα.

Αυτό είναι και το θέμα της συγκεκριμένης πτυχιακής εργασίας. Συγκεκριμένα, προσπαθούμε να επιτύχουμε μεγαλύτερη απόδοση για την εφαρμογή που χρησιμοποιούμε, με τον μερικό επανασχεδιασμό του λογισμικού ώστε να εκμεταλλευτούμε στο έπακρο τους υπολογιστικούς πόρους που διαθέτουμε από το εκάστοτε πολυπύρηνο σύστημα.

Το σύστημα που αρχικά χρησιμοποιήθηκε είναι ο επεξεργαστής Cell BroadBand Engine που αναπτύχθηκε από τις εταιρίες Sony, IBM και Toshiba. Πρόκειται για έναν πολυπύρηνο επεξεργαστή ετερογενής αρχιτεκτονικής που ανήκει στην οικογένεια επεξεργαστών Power/PowerPc. Ο Cell BroadBand Engine χρησιμοποιείται στην κονσόλα παιχνιδιών Playstation3 αλλά και σε εξυπηρετητές που αναπτύσσονται από την εταιρία IBM (IBM BladeCenter) [1] [2].

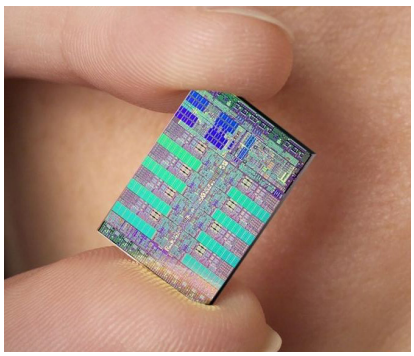
Τα χαρακτηριστικά του συγκεκριμένου επεξεργαστή, τον καθιστούν ιδανικό

για υπολογιστικά δύσκολες (computation-intensive) εφαρμογές όπως εφαρμογές επεξεργασίας ήχου και εικόνας, bioinformatics, cluster και distributed computing, multimedia και vector processing.

Για τους λόγους αυτούς, επιλέξαμε τη συγκεκριμένη πλατφόρμα στην οποία χρησιμοποιήσαμε μια εφαρμογή αναγνώρισης δαχτυλικών (Biometric Image Software - NBIS) αποτυπωμάτων του National Institue of Standards and Technology (NIST) με σκοπό να επιτύχουμε μεγαλύτερη απόδοση σε σύγκριση με την x86 αρχιτεκτονική. Για λόγους που θα γίνουν αντιληπτοί στην πορεία της ανάλυσης της εργασίας αυτής, η προσπάθεια αυτή δεν στέφθηκε με επιτυχία.

Σχήμα 1: Η κονσόλα παιχνιδιών Playstation 3

Στη συνέχεια στραφήκαμε στην μαζικά παράλληλη αρχιτεκτονική που αναπτύχθηκε απο την εταιρία NVidia, CUDA. Η αρχιτεκτονική αυτή εκμεταλλεύεται τη δύναμη που παρέχει το υποσύστημα της κάρτας γραφικών (GPU) ενός υπολογιστή για να προσφέρει επεξεργαστική δύναμη της τάξης πολλών GFLOPS (Giga Floating Point Operations Per Second). Οι εκατοντάδες πυρήνες απο τους οποί-



Σχήμα 2: Ο επεξεργαστής Cell BroadBand Engine

ους αποτελούνται οι σύγχρονες κάρτες γραφικών καθώς και η ευκολία και συμβατότητα που προσφέρει η αρχιτεκτονική CUDA, τη καθιστούν ιδανική για εφαρμογές που απαιτούν εκατοντάδες υπολογισμούς το δευτερόλεπτο. Τη πλατφόρμα αυτή χρησιμοποιήσαμε για την επίλυση της μερικής διαφορικής εξίσωσης Black-Scholes με τη μέθοδο Odd-Even Reduction. Ο σκοπός ήταν να χρησιμοποιήσουμε τη μέθοδο αυτή κατάλληλα τροποποιημένη για την κάρτα γραφικών και να επιτύχουμε τη μεγαλύτερη δυνατή απόδοση παραλληλοποιώντας τα πιο υπολογιστικά απαιτητικά τμήματα της εφαρμογής.

1.2 Οργάνωση του κειμένου

Στα επόμενα κεφάλαια της εργασίας θα χρησιμοποιηθεί η τεχνική *bottom - up*, δηλαδή από την πιο απλή μονάδα στη πιο σύνθετη, την αρχιτεκτονική από τις επιμέρους πλατφόρμες που χρησιμοποιήθηκαν, την ανάλυση των εφαρμογών , στην ανάλυση των τεχνικών που χρησιμοποιήθηκαν και στην παρουσίαση των αποτελεσμάτων που προέκυψαν από την ανάπτυξη και τις δοκιμές.

Πιο συγκεκριμένα στο **Κεφάλαιο 2** αναλύεται σε τρεις ενότητες, στο *υλικό*, και στο *λογισμικό* που χρησιμοποιήθηκαν καθώς και στους λόγους που δεν κατέ-

στησαν δυνατό το μερικό επανασχεδιασμό της εφαρμογής . Στην πρώτη ενότητα παρουσιάζουμε την αρχιτεκτονική του Cell Broadband Engine και αναλύουμε τα τεχνικά χαρακτηριστικά και τις δυνατότητες του. Επίσης γίνεται αναφορά και στην κονσόλα παιχνιδιών Playstation3 που χρησιμοποιήθηκε στη συγκεκριμένη εργασία. Στη δεύτερη ενότητα "*Λογισμικό*" παρουσιάζεται η βιβλιοθήκη εφαρμογών του ινστιτούτου NIST, Biometric Image Software καθώς και το λογισμικό που χρησιμοποιήθηκε κατά την ενασχόληση μας με τη συγκεκριμένη εργασία και τέλος στη τρίτη ενότητα αναλύονται οι λόγοι της εγκατάλειψης του όλου εγχειρήματος.

Το *Κεφάλαιο 3* εισάγει τον αναγνώστη στη πλατφόρμα CUDA. Αναλύει την αρχιτεκτονική των τελευταίων γενεών καρτών γραφικών και περιγράφει το υλικό και το λογισμικό που χρησιμοποιήθηκαν για την ολοκλήρωση της εργασίας.

Το *Κεφάλαιο 4* περιγράφει την εφαρμογή που χρησιμοποιήσαμε για την πρόβλεψη τιμών μετοχών. Συγκεκριμένα αναλύεται το μαθηματικό μοντέλο, παρουσιάζονται οι διαφορικές εξισώσεις τους συστήματος καθώς και οι μέθοδοι μερικής λύσης αυτών.

Στο *Κεφάλαιο 5* παρουσιάζεται η προσέγγιση που ακολουθήσαμε για την υλοποίηση της εφαρμογής στο GPU (Graphics Processing Unit), τα δίαφορα στάδια βελτιστοποίησης καθώς και το κέρδος σε απόδοση αυτών. Τέλος συγκρίνουμε τα αποτελέσματα και την απόδοση του συστήματος σε σχέση με ένα σύστημα που ακολουθεί την x86 Intel αρχιτεκτονική.

Στο *Κεφάλαιο 6* αναλύονται τα συμπεράσματα από το όλο εγχείρημα και προτείνονται επεκτάσεις του συστήματος.

Τέλος, στο παράρτημα του *Κεφαλαίου 7* υπάρχουν αναφορές για το λογισμικό και λειτουργικό σύστημα κατά τη διάρκεια ανάπτυξης της εργασίας. Παρατίθεται

ο πηγαίος κώδικας που αναπτύχθηκε κατά τη διάρκεια εκπόνησης της εργασίας στις παραγράφους 7.3 και 7.4.

2 IBM Cell/B.E και NIST Institute

Το κεφάλαιο αυτό εισάγει τον αναγνώστη στη πλατφόρμα καθώς και στην βιομετρική εφαρμογή που χρησιμοποιήθηκαν. Παρουσιάζεται η αρχιτεκτονική του επεξεργαστή Cell B.E. καθώς και του υλικού (hardware) και αναλύεται η εφαρμογή δαχτυλικών αποτυπωμάτων του ινστιτούτου NIST. Τέλος, αναφέρονται οι λόγοι που μας οδήγησαν στην απόφαση να εγκαταλείψουμε το όλο εγχείρημα.

2.1 Υλικό

2.1.1 Ιστορική αναδρομή

Το 2001 οι εταιρίες Sony, Toshiba και IBM ανακοίνωσαν το σχηματισμό της κοινοπραξίας STI (STI Alliance) με σκοπό την ανάπτυξη ενός επεξεργαστή εξειδικευμένου για εφαρμογές πολυμέσων. Έπειτα απο τρία (3) χρόνια ο πρώτος επεξεργαστής Cell είναι πραγματικότητα, χρονισμένος σε συχνότητα μεγαλύτερη των τέσσερα (4) GHz. Το 2006 η εταιρία Sony Computer Entertainment λανσάρει το Playstation 3 που είναι και η πρώτη εμπορική προσφορά του Cell/B.E. ενώ ταυτόχρονα η IBM παρέχει το λογισμικό CBE Software Development Kit (SDK) και ανακοινώνει τον καινούργιο της υπερυπολογιστή, Roadrunner. Το 2008 ο Roadrunner αποτελούμενος από 12.960 Cell επεξεργαστές και 12.960 Opterons γίνεται ο ταχύτερος υπερυπολογιστής του κόσμου, επιτυγχάνοντας απόδοση της τάξης των 1.026 Petaflops.

Αρχικά ο επεξεργαστής Cell/B.E. προοριζόταν για εφαρμογές σε κονσόλες παιχνιδιών καθώς και για συσκευές πολυμέσων, όμως η μοναδική αρχιτεκτονική του τον έκανε κατάλληλο για εφαρμογές σε διάφορους βιομηχανικούς κλάδους όπως στην υγεία, εκπαίδευση, θετικών και οικονομικών επιστημών κ.α.

Ο Cell/B.E. είναι προγραμματίσιμος μέσω γλωσσών προγραμματισμού υψηλού επιπέδου όπως C/C++ και Fortran. Παρόλα αυτά για να εκμεταλλευτεί κανείς στο έπακρο τις δυνατότητες του επεξεργαστή αυτού θα πρέπει οι εφαρμογές να είναι προσεχτικά σχεδιασμένες τόσο στη δομή όσο και στο τρόπο που χειρίζονται τα δεδομένα.

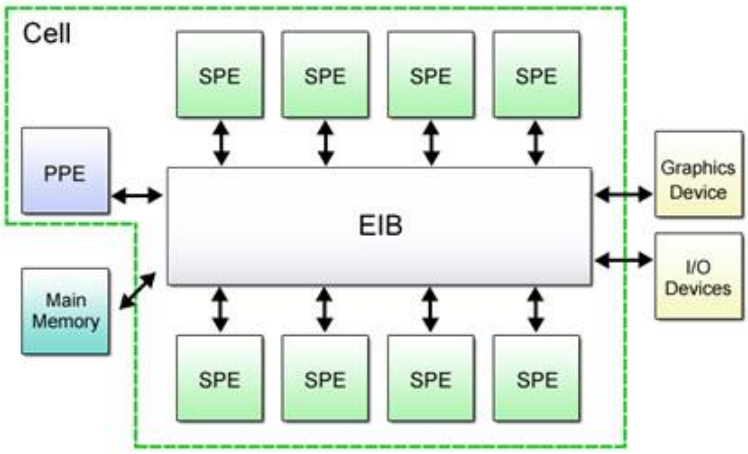
2.1.2 Cell Processor

Η αρχιτεκτονική του Cell Broadband Engine (Cell B.E) ορίζει μια επεξεργαστική δομή των 64-bit Power Architecture με μοναδικά χαρακτηριστικά, ικανή να υποστηρίξει υπολογιστικά έντονο παραλληλοποιήσιμο πηγαίο κώδικα.

Η πρώτη υλοποίηση της αρχιτεκτονικής είχε ως αποτέλεσμα μια συσκευή (single device) ετερόγεννων επεξεργαστικών μονάδων. Ο επεξεργαστής αποτελείται από ένα διανυσματικό (vector) επεξεργαστή *Power Processor Element (PPE)* με δύο επίπεδα μνήμης cache και οχτώ (8) ανεξάρτητους πυρήνες, *Synergistic Processor Elements (SPEs)*, κάθε ένα από το οποίο έχει τη δική του πολυεπίπεδη δομή αποθήκευσης.

Πέρα απο τη παραλληλοποίηση στο επίπεδο του επεξεργαστή (processor-level parallelism), κάθε επεξεργαστικό στοιχείο έχει *SIMD (Single Instruction Multiple Data)* μονάδες που μπορούν να επεξεργαστούν απο τέσσερις (4) λέξεις (words) εώς δεκαέξι (16) χαρακτήρες ανά επεξεργαστικό κύκλο καθώς και μονάδες για Direct Memory Access (DMA),οι οποίες προσφέρουν ένα πλούσιο σετ εντολών DMA για επικοινωνία μεταξύ όλων των επεξεργαστικών μονάδων του Cell/B.E.

Στο σχήμα που ακολουθεί φαίνονται οι βασικές μονάδες του Cell. Συγκεκριμένα η μονάδα PPE,οι 8 SPEs μονάδες, ο δίαυλος EIB (*Element Interconnect Bus*) και οι μονάδες γραφικών και εισόδου/εξόδου.



Σχήμα 3: Η αρχιτεκτονική δομή του Cell/B.E.

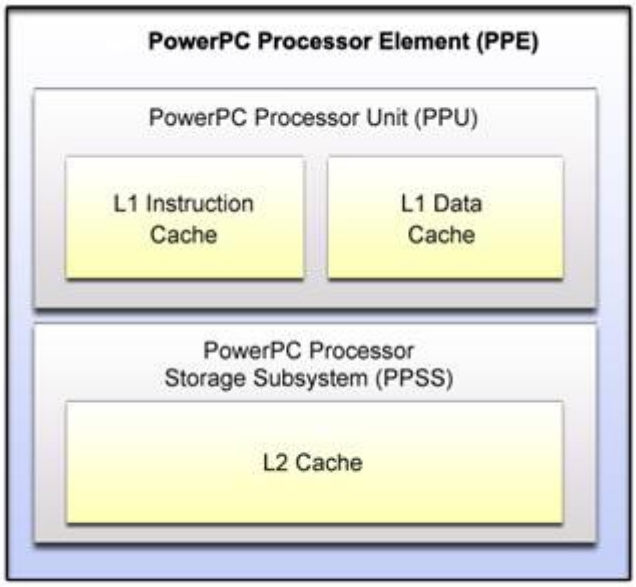
2.1.3 Power Processor Element

Η μονάδα *Power Processor Element (PPE)* είναι ένας επεξεργαστής των 64-bit που ανήκει στην οικογένεια επεξεργαστών Power Architecture. Η μονάδα αυτή αποτελεί το κέντρο ελέγχου του Cell, μιας και "τρέχει" το λειτουργικό σύστημα , χειρίζεται interrups, περιέχει και διαχειρίζεται τα 512KB μνήμης L2 cache. Επίσης, διανέμει το φόρτο εργασίας (workload) μεταξύ των μονάδων SPEs και συντονίζει τη λειτουργία τους.

Όπως διακρίνεται και στο σχήμα που ακολουθεί, η μονάδα PPE αποτελείται απο δύο (2) λειτουργικά μπλοκ. Το *PowerPC Processor Unit* (PPU) και το *PowerPC Processor Storage Subsystem* (PPSS). Το σετ εντολών της μονάδας PPU βασίζεται στην 64-bit αρχιτεκτονική PowerPC 970 που χρησιμοποιήθηκε κυρίως στους επεξεργαστές G5 Power Mac της εταιρίας Apple. Περιέχει 32KB + 32KB μνήμης cache L1 για εντολές και δεδομένα. Τέλος, είναι ικανή και για SIMD (*Single Instruction, Multiple Data*) επεξεργασία χάριν στη υπομονάδα VMX της

IBM που περιέχει και επιτρέπει την ταυτόχρονη εκτέλεση δύο (2) thread τα οποία μοιράζονται μεταξύ τους τους πόρους της μονάδας.

Η υπομονάδα PPSS περιέχει τη μνήμη cache επιπέδου L2 μαζί με registers και queues για την εγγραφή και ανάγνωση δεδομένων. Η μνήμη αυτή αποτελεί και την μοναδική μνήμη που μοιράζεται μεταξύ των διαφόρων επεξεργαστικών μονάδων της συσκευής.



Σχήμα 4: Η δομή της μονάδας PPE

2.1.4 Synergistic Processor Elements

Η μεγάλη επιτυχία του Cell βασίζεται στις μονάδες SPEs. Κάθε SPE αποτελείται απο ένα *Synergistic Processor Unit* (SPU), που έχουν σαν μοναδικό σκοπό τη γρήγορη εκτέλεση SIMD πράξεων. Κάθε SPU αποτελείται από δύο παράλληλα pipelines που εκτελούν εντολές χρονισμένα στα 3.1GHz. Σε μόλις λίγους επεξερ-

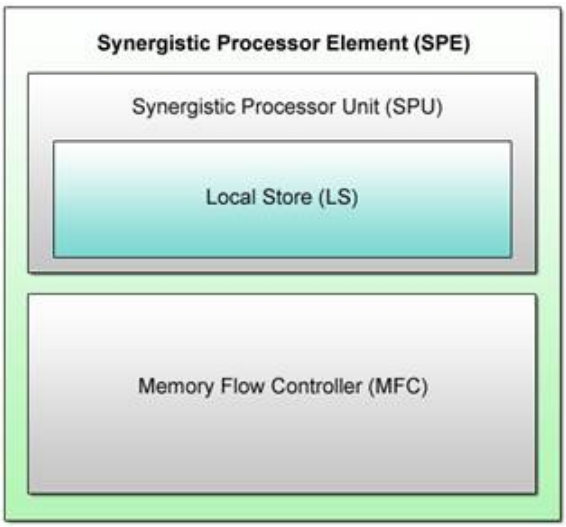
γαστικούς κύκλους το ένα pipeline μπορεί να επεξεργαστεί διανύσματα εντολών των 128-bit ενώ παράλληλα το άλλο "διαβάζει" επιπλέον διανύσματα απο το υποσύστημα μνήμης. Οι εντολές και τα δεδομένα των SPU's αποθηκεύονται σε ένα ενιαίο χώρο ονόματι *Local Store (LS)* το οποίο δεν είναι μνήμη cache.

Η μνήμη αυτή σε συνδυασμό με το αρχείο καταχωρητών (128 128-bit registers) είναι και τα μόνα επίπεδα μνήμης που έχει πρόσβαση η μονάδα SPU. Στο σχήμα που ακολουθεί διακρίνεται ο ελεγκτής μνήμης *Memory Flow Controller (MFC)*.

Η πιο σημαντική εργασία του ελεγκτή αυτού είναι η πρόσβαση DMA. Συγκεκριμένα, όταν η μονάδα PPU μεταφέρει δεδομένα σε ένα SPU, δίνει στον ελεκτή MFC τη διεύθυνση μνήμης στη μνήμη του συστήματος και τη διεύθυνση στη μονάδα αποθήκευσης LS και έτσι ο ελεγκτής αρχίζει να μετακινεί bytes. Αντίστοιχα λειτουργεί και όταν ένα SPU θέλει να μεταφέρει δεδομένα ή λίστες δεδομένων στη μονάδα αποθήκευσης του LS. Με το τρόπο αυτό επιτυγχάνεται αποδοτικά η μεταφορά δεδομένων από μη συνεχόμενα τμήματα μνήμης χωρίς να επιβαρύνεται ο κεντρικός δίαυλος ή να χάνονται επεξεργαστικοί κύκλοι για την μεταφορά δεδομένων.

2.1.5 Element Interconnect Bus

Ο EIB αποτελεί το δίαυλο επικοινωνίας μεταξύ των διαφόρων υποσυστημάτων του Cell/B.E. Λειτουργικά, αποτελείται απο τέσσερα (4) δακτύλιους, δύο (2) μεταφοράς δεδομένων με φορά αντίθετης των δεικτών του ρολογιού (PPE > SPE1 >SPE3 > SPE5 > SPE7 > IOIF1 > IOIF0 > SPE6 > SPE4 > SPE2 > SPE0 > MIC) και δύο που μεταφέρουν δεδομένα με φορά σύμφωνη με τη φορά των δεικτών του ρολογιού. Κάθε δακτύλιος έχει πλάτος 16 bytes και υποστηρίζει μέχρι τρείς (3) ταυτόχρονες μεταφορές δεδομένων.



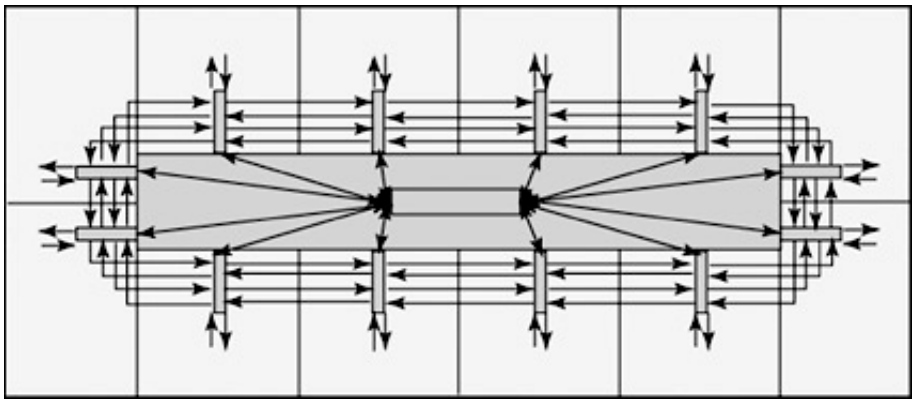
Σχήμα 5: Η δομή της μονάδας SPE

Κάθε μεταφορά DMA μπορεί να μεταφέρει όγκο δεδομένων των 1,2,4,6 και 16 bytes καθώς και πολλαπλάσια των 16 bytes μέχρι το μέγιστο των 16 KB. Ανεξαρτήτως μεγέθους, κάθε μεταφορά DMA αποτελείται από οκτώ (8) τμήματα των 128 bytes στο δίαυλο EIB. Τέλος, ο δίαυλος EIB έχει μέγιστη διαμεταγωγή (peak bandwidth) της τάξης των 204.8 GB/s για μεταφορά δεδομένων μεταξύ των SPE's.

Τα SPU's δεν έχουν άμεση πρόσβαση στο δίαυλο EIB. Αντιθέτως, κάθε SPU επικοινωνεί με το δίαυλο μέσω του *Memory Flow Controller (MFC)*. Πρόκειται για ένα συνεπεξεργαστή (coprocessor) που είναι σχεδιασμένος για να παρέχει επικοινωνία με το δίαυλο EIB. Το πλεονέκτημα της σχεδίασης αυτής είναι ότι ο ελεγκτής MFC ολοκληρώνει τη μεταφορά των δεδομένων χωρίς τα SPU's να διακόψουν την επεξεργασία τους.

2.1.6 Input/Output Interface

Η μονάδα *Input/Output Interface (IOIF)* είναι ο συνδετικός κρίκος μεταξύ του επεξεργαστή Cell και των εξωτερικών περιφερειακών. Βασίζεται στη τεχνολογία Rambus: FlexIO. Οι συνδέσεις FlexIO (FlexIO connections) μπορούν να επιτύχουν ρυθμούς δεδομένων έως 76.8 GB/s. Στο σχήμα που ακολουθεί φαίνεται ο δίαυλος EIB.



Σχήμα 6: Ο δίαυλος EIB

2.1.7 Playstation 3

Ο πιο οικονομικός και εύκολος τρόπος για να αποκτήσει κανείς πρόσβαση στον επεξεργαστή Cell είναι μέσω του Playstation 3 και της δυνατότητας που προσφέρει στο χρήστη η ανοιχτή αρχιτεκτονική του.

Συγκεκριμένα, μέσω της επιλογής που παρέχει για την εγκατάσταση ενός Linux λειτουργικού συστήματος μπορεί κανείς να χρησιμοποιήσει τη κονσόλα ως κανονικό υπολογιστή και να εκμεταλλευτεί τις δυνατότητες που παρέχει ο επεξεργαστής Cell. Τα μειονεκτήματα από μια τέτοιου είδους χρήση είναι ότι η μνήμη του συστήματος περιορίζεται στα 256MB, από τα οποία μόνο τα 200 είναι διαθέσιμα

για το λειτουργικό σύστημα και εφαρμογές, καθώς επίσης ο χρήστης έχει πρόσβαση μόνο σε έξι (6) από τα οκτώ (8) SPU's. Το ένα είναι απενεργοποιημένο σε επίπεδο υλικού και το άλλο προορίζεται αυστηρά για χρήση από το λειτουργικό της κονσόλας.

Όπως ήδη έχουμε αναφέρει το Playstation 3 μπορεί να δεχτεί την εγκατάσταση κάποιας διανομής Linux. Συγκεκριμένα, στο Playstation 3 βρίσκεται εγκατεστημένο ένα λειτουργικό σύστημα, το *Game OS* πάνω από το οποίο μπορεί να εγκατασταθεί η διανομή Linux. Το Linux λειτουργικό σύστημα έχει πρόσβαση στο υλικό (hardware) της κονσόλας μέσω των hypervisor calls που παρέχει το Game OS. Με το τρόπο αυτό ο πυρήνας (kernel) επικοινωνεί με το υλικό μέσω vitrualized interrups.

Ενδεικτικά κάποιες εμπορικές διανομές που υποστηρίζουν το υλικό του Playstation 3 είναι:

- Gentoo Linux [10]
- OpenSuse [11]
- Ubuntu Linux [12]
- Yellow Dog Linux [13]
- Fedora Linux [14]

2.2 Λογισμικό

Στην ενότητα αυτή παρουσιάζεται η βιομετρική εφαρμογή του ινστιτούτου NIST, NFIS2 καθώς επίσης και το λογισμικό που χρησιμοποιήθηκε.

2.2.1 NIST Biometric Image Software - NFIS2

Η δεύτερη έκδοση του λογισμικού αναγνώρισης δαχτυλικών αποτυπωμάτων αποτελείται από μια συλλογή εφαρμογών, βοηθητικών προγραμμάτων εικόνας και βιβλιοθηκών πηγαίου κώδικα γραμμένα στη γλώσσα προγραμματισμού ``C'' και έχουν αναπτυχθεί για το λειτουργικό σύστημα Linux. Για την ανάπτυξή τους χρησιμοποιήθηκαν τα εργαλεία της GNU gcc και gmake. Χωρίζεται σε επτά (7) κύρια πακέτα, τα οποία είναι:

1. NFSEG

   Ένα σύστημα που απομονώνει τα αποτυπώματα από κάρτες δαχτυλικών αποτυπωμάτων.

2. PCASYS

   Ένα νευρωνικό δίκτυο ταξινόμησης προτύπων (pattern classification).

3. MINDTCT

   Ένα σύστημα ανίχνευσης των μικρολεπτομερειών ενός αποτυπώματος.

4. NFIQ

   Ένα νευρωνικό δίκτυο κατηγοριοποίησης της ποιότητας του δαχτυλικού αποτυπώματος.

5. BOZORTH3

   Ένα σύστημα ταυτοποίησης των μικρο λεπτομερειών ενός αποτυπώματος.

6. AN2K

   Μία εφαρμογή αναφοράς (reference implementation) του προτύπου ANSI/NIST-ITL 1-2000 ``*Data Format for the Interchange of Fingerprint, Facial, Scar*

Mark and Tattoo (SMT) Information"

7. IMGTOOLS Μία συλλογή βοηθητικών προγραμμάτων εικόνας, περιλαμβανομένων κωδικοποιητών και αποκωδικοποιητών για Baseline και Losslesss JPEG και WSQ.

Στη συνέχεια, στην ενότητα 2.2.2 θα αναφερθούμε στο πακέτο PCASYS, που είναι και το πακέτο άμεσου ενδιαφέροντος για υλοποίηση στον επεξεργαστή Cell[15].

2.2.2 PCASYS

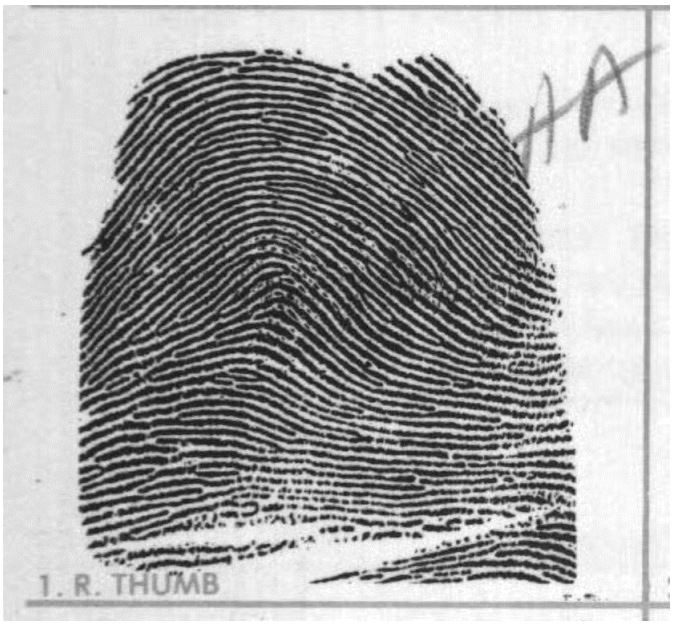
Ένα αυτόματο σύστημα αναγνώρισης δαχτυλικών αποτυπωμάτων αποτελείται από μια βάση δεδομένων με κάρτες δαχτυλικών αποτυπωμάτων από όλα τα δάκτυλα ενός ανθρώπου (ten print cards) με την οποία συγκρίνεται το αποτύπωμα ενδιαφέροντος.

Η σύγκριση βασίζεται στην ταυτοποίηση των καταλήξεων των κορυφογραμμών καθώς και στις μικρολεπτομέρειες των αποτυπωμάτων. Όμως, επειδή η απαίτηση για μεγάλη λεπτομέρεια έχει σαν αποτέλεσμα τα αρχεία των δαχτυλικών αποτυπωμάτων να είναι πολύ μεγάλα, η εξαντλητική αναζήτηση αυτών είναι υπολογιστικά πολύ απαιτητική και μη πρακτική. Επομένως, σε αυτή τη περίπτωση, η διαδικασία ταυτοποίησης μπορεί να γίνει πιο αποδοτική κατηγοριοποιώντας και διαμερίζοντας τα αποτυπώματα.

Από τη στιγμή που αναγνωριστεί η κατηγορία κάθε αποτυπώματος σε μια κάρτα αποτυπωμάτων, η αναζήτηση μπορεί να περιοριστεί στην κλάση αποτυπωμάτων που ανήκουν στη συγκεκριμένη κατηγορία μειώνοντας σημαντικά τον αριθμό των συγκρίσεων που πρέπει να πραγματοποιήσει το σύστημα ταυτοποίησης.

Τα δαχτυλικά αποτυπώματα ενός ανθρώπου μπορούν να κατηγοριοποιηθούν σε έξι βασικές κατηγορίες αναλόγως τα χαρακτηριστικά τους. Συγκεκριμένα, διακρίνονται σε αποτυπώματα με καμάρα (arch), αριστερού και δεξιού βρόχου (left, right loop), αποτυπώματα ουλής (scar), με εκτεταμένη καμάρα (tended arch) και τέλος αποτυπώματα με σπείρες (whorl).

Το PCASYS είναι μια εφαρμογή που επενεργεί πάνω στα αποτυπώματα και τα διαχωρίζει με βάση τις πιο πάνω κατηγορίες. Στο σχήμα που ακολουθεί διακρίνεται ένα αποτύπωμα αντίχειρα με καμάρες.



Σχήμα 7: Arch fingerprint

Η βασική μέθοδος που χρησιμοποιεί το πρόγραμμα είναι αρχικά, να εξάγει από το προς κατηγοριοποίηση αποτύπωμα ένα δισδιάστατο πίνακα που περιέχει τον προσανατολισμό των κοιλάδων και κορυφογραμμών του αποτυπώματος και έπειτα να συγκρίνει τον πίνακα αυτών με πίνακες αναφοράς.

Επιγραμματικά, τα βασικά βήματα του αλγορίθμου είναι :

- Segmentor

  Η συγκεκριμένη ρουτίνα εκτελεί την πρώτη επεξεργασία πάνω στα δεδομένα. Συγκεκριμένα, έχει ως είσοδο την εικόνα με το δαχτυλικό αποτύπωμα και αποκόπτει οτιδήποτε πέρα από το αποτύπωμα ώστε να μειώσει τη ποσότητα των προς επεξεργασία δεδομένων από τις υπόλοιπες ρουτίνες στη συνέχεια.

- Image Enhancement

  Το βήμα αυτό βελτιώνει την αποκομένη εικόνα του προηγούμενο βήματος. Αυτό επιτυγχάνεται αρχικά με την εφαρμογή μετασχηματισμού Fourier ωστε να απεικονιστούν τα δεδομένα στο πεδίο της συχνότητας. Στη συνέχεια εφαρμόζεται μια μη-γραμμική συνάρτηση που ενισχύει την ισχύ του χρήσιμου σήματος ως προς το θόρυβο και τέλος, μέσω του αντίστροφου μετασχηματισμού Fourier, τα δεδομένα απεικονίζονται στην αρχική τους χωρική μορφή.

- Ridge-Valley Orientation Detector

  Στο βήμα αυτό, ο αλγόριθμος ανιχνεύει τον προσανατολισμό των κορυφών και κοιλάδων του αποτυπώματος και παράγει ως έξοδο ένα πίνακα με τους περιφερειακούς μέσους όρους των προσανατολισμών,

- Karhunen-Loeve Transformation

  Το μέγεθος των πινάκων κάνει πρακτικά αδύνατη τη χρησιμοποίηση τους ως είσοδο στο νευρωνικό δίκτυο, οπότε και εφαρμόζεται ο εν λόγω αλγόριθμος που μετασχηματίζει τα πολυδιάστατα διανύσματα σε διανύσματα χαμηλότερης διάστασης.

- Probabilistic Neural Network Classifier

  Στο βήμα αυτό δίνεται ως είσοδο το διάνυσμα που προκύπτει έπειτα από το Karhunen-Loeve μετασηματισμό, και το νευρωνικό δίκτυο το κατηγοριοποιεί ανάλογα με τη κλαση στην οποία ανήκει.

2.2.3 IBM Cell/B.E. SDK

Το Software Development Kit (SDK) της εταιρίας IBM είναι ένα πλήρες πακέτο εργαλείων που απλοποιούν το προγραμματισμό του επεξεργαστή Cell. Είναι διαθέσιμο μόνο για το λειτουργικό σύστημα Linux και παρέχει στο χρήστη μεταγλωττιστή (gcc compiler) και αποσφαλματωτή (debugger) καθώς και σε εργαλεία ανάλυσης (oprofile) ώστε να γράφει πηγαίο κώδικα για την αρχιτεκτονική του Cell καθώς επίσης πρόσβαση σε βιβλιοθήκες και εργαλεία αλλά και σε ένα προσομοιωτή (simulator) ώστε ο χρήστης να μπορεί να ``τρέξει'' εκτελέσιμα του Cell σε Intel x86 αρχιτεκτονική. Τα εργαλεία και το λογισμικό του SDK παρέχονται με τρεις άδειες χρήσης, τις GPL, LGPL και την άδεια της IBM, ILAR [16, 17, 18].

2.3 Συμπεράσματα

Ο επεξεργαστής Cell/B.E αποτελεί ένα λαμπρό επίτευγμα από αρχιτεκτονικής απόψεως στη επιστήμη των υπολογιστών. Πάρα αυτά κατά τη διάρκεια εκπόνησης της εργασίας συναντήσαμε σημαντικά εμπόδια τα οποία μας οδήγησαν στην απόφαση να εγκαταλείψουμε το Cell αλλά και το λογισμικό που χρησιμοποιούσαμε.

2.3.1 Προβλήματα κατά την εγκατάσταση

Το πρόβλημα που αρχικά αντιμετωπίσαμε είχε να κάνει με την εγκατάσταση του απαραίτητου λογισμικού στο υπολογιστικό σύστημα που χρησιμοποιούσαμε αλλά και στην κονσόλα παιχνιδιών Playstation 3.

Τα πακέτα αυτά, από την IBM, ήταν σε μορφή πακέτου .rpm (RedHat Package Manager) και υποστηρίζουν εγγενώς το λειτουργικό σύστημα Red Hat Linux καθώς και παρεμφερής διανομές που υποστηρίζουν το συγκεκριμένο σύστημα διαχείρισης λογισμικού σε μορφή πακέτων (packages). Οπότε το πρώτο πρόβλημα που είχαμε ήταν η ασυμβατότητα με τη διανομή που είχαμε επιλέξει να εργαστούμε.

Αρχικά, η προσέγγιση που ακολουθήσαμε ήταν η λύση ενός virtual machine στο οποίο εγκαταστήσαμε μια συμβατή διανομή Linux. Λύση όμως που γρήγορα εγκαταλείφθηκε, καθώς το virtual machine κατανάλωνε πολύτιμους υπολογιστικούς πόρους τους συστήματος και ο εξομοιωτής, που είναι ούτως ή άλλως αρκετά αργός σε όρους χρόνου εκτέλεσης, ήταν σχεδόν μη ανταποκρίσιμος.

Στη συνέχεια, προκειμένου να μπορέσουμε να εγκαταστήσουμε τα απαραίτητα πακέτα λογισμικού όπως τον μεταγλωττιστή, αποσφαλματωτή, τον προσομοιωτή Systemsim, αλλά και τις απαραίτητες βιβλιοθήκες, αποφασίσαμε πρώτα να τα τροποποιήσουμε.

Επομένως, αρχικά έπρεπε να τα μετατρέψουμε σε μορφή κατάλληλη για τη διανομή Linux που χρησιμοποιούσαμε. Το επιτύχαμε αυτό με τη βοήθεια του προγράμματος *Allien* που μετατρέπει πακέτα της μορφής rpm σε πακέτα της μορφής .deb (debian packages). Επιπλέον, έπρεπε να τροποποιήσουμε και τα scripts των εργαλείων προς εγκατάσταση ώστε να είναι συμβατά με τις βιβλιοθήκες του λει-

τουργικού συστήματος που χρησιμοποιούσαμε.

Τέλος, με τη βοήθεια ενός script μετατρέψαμε τα πακέτα αυτά από την i386 αρχιτεκτονική στην x64 αρχιτεκτονική ώστε να είναι συμβατά με την 64bit έκδοση του λειτουργικού συστήματος Ubuntu.

Όσον αφορά το Playstation, είχαμε την ατυχία η πρώτη συσκευή που χρησιμοποιήσαμε να έχει ελαττωματική κάρτα δικτύου. Το αποτέλεσμα ήταν ότι κατά τη διάρκεια ανανέωσης του λογισμικού (GameOS) να μην ολοκληρωθεί η εγκατάσταση και ουσιαστικά να καταστεί το σύστημα άχρηστο. Παρόλο που το γεγονός αυτό δεν μας επηρέασε άμεσα στην εξέλιξη της εργασίας μας, μας στοίχισε πολύτιμο χρόνο μέχρι να βρούμε μια καινούργια κονσόλα για να εργαστούμε.

2.3.2 Προβλήματα με την εφαρμογή

Το αμέσως επόμενο πρόβλημα, ακόμη πιο σημαντικό από τις δυσκολίες που είχαμε αντιμετωπίσει έως τώρα, είχε να κάνει με την εφαρμογή.

Συγκεκριμένα, το λογισμικό του ινστιτούτου NIST ήταν εγκαταλελειμμένο και αρκετά πολύπλοκο. Ενδεικτικά ο πηγαίος κώδικας της συγκεκριμένης εφαρμογής εφαρμογής είχε εκδοθεί τη χρονιά 1996.

Αυτό είχε σαν αποτέλεσμα να μπορεί να μεταγλωττιστεί ο πηγαίος κώδικας μόνο από τον μεταγλωττιστή της GNU gcc έκδοσης 3.4, κάτι που ερχόταν σε αντίθεση με το *Software Development Kit* που είχε ως επιλογή μόνο τους μεταγλωττιστές έκδοσης 4.1 ή 4.3.

Παρόλο που αυτό δεν θα αποτελούσε πρόβλημα για την x86 αρχιτεκτονική, ήταν πρόβλημα για την ανάπτυξη του πηγαίου κώδικα στο Playstation 3. Στο σημείο αυτό είχαμε τις εξής δύο εναλλακτικές λύσεις: Είτε να αναπτύξουμε την εργασία μας πάνω σε κάποια παλιότερη έκδοση του SDK είτε να τροποποιήσουμε

τον πηγαίο κώδικα της εφαρμογής ώστε να μεταγλωττίζεται από μεταγενέστερους μεταγλωττιστές.

Δυστυχώς και οι δύο (2) λύσεις αυτές κατέστησαν άκαρπες για διαφορετικούς λόγους η κάθε μια. Η περίπτωση της παλαιότερης έκδοσης SDK απορρίφθηκε λόγω ότι χρησιμοποιούσε την βιβλιοθήκη libspe της οποίας η ανάπτυξη και συντήρηση είχε εγκαταλειφθεί προς χάριν της libspe2. Το γεγονός αυτό αμέσως υπαγόρευε και την έκδοση του SDK που θα χρησιμοποιούσαμε.

Όσον αφορά τη περίπτωση τροποποίησης του πηγαίου κώδικα της εφαρμογής η πολυπλοκότητα και η δαιδαλώδης δομή αυτού το κατέστησαν απαγορευτικό καθώς δεν ήταν στα πλαίσια της συγκεκριμένης εργασίας η συντήρηση και ανάπτυξη αυτού του πηγαίου κώδικα. Στο σημείο αυτό αξίζει να αναφέρουμε ότι χρειάστηκε αρκετός καιρός, με τη βοήθεια του αποσφαλματωτή της GNU, *gdb*, προκειμένου να επιτύχουμε τη σωστή μεταγλώττιση και εκτέλεση της εφαρμογής για την x86 αρχιτεκτονική καθώς και για το PPE.

Χαρακτηριστικά, αναφέρουμε ότι η βιομετρική εφαρμογή αναγνώρισης δακτυλικών αποτυπωμάτων αποτελείται από 447 αρχεία πηγαίου κώδικα γραμμένα στη γλώσσα ``C'' και 74 αρχεία Makefile!

Τέλος, ένα ακόμη σημαντικό στοιχείο είναι ότι η εφαρμογή χρησιμοποιεί προς χάριν της απόδοσης πολλές στατικές βιβλιοθήκες στη x86 αρχιτεκτονική γεγονός που δεν επέτρεπε την εύκολη μεταφορά αυτών στην Power αρχιτεκτονική.

2.3.3 Προβλήματα με το υλικό

Πέρα όμως των προβλημάτων που περιγράφηκαν προηγουμένως και είχαμε με το *λογισμικό*, προβλήματα υπήρχαν και σε επίπεδο *υλικού*. Συγκεκριμένα, μια από τις μεγαλύτερες προκλήσεις για τον αποτελεσματικό προγραμματισμό του

Cell, όπως και κάθε πολυπύρηνης αρχιτεκτονικής σύστημα, είναι να γίνει η σωστή κατανομή του φόρτου εργασίας από τον προγραμματιστή σε κάθε πυρήνα ιδίως σε μια πλατφόρμα που στερείται κρυφής και εικονικής μνήμης.

Επιπλέον, ο προγραμματισμός του συγκεκριμένου επεξεργαστή απαιτεί εξειδικευμένη γνώση είτε σε γλώσσα μηχανής (assembly language) για τα SPE, είτε στους συγκεκριμένους τύπους δεδομένων της αρχιτεκτονικής καθώς και της διασύνδεσης (interface) των C/C++ προεκτάσεων αυτής.

Επίσης, απαιτείται εκτενής γνώση της διανυσματικής αριθμητικής λογικής μονάδας (ALU) καθώς και των χαρακτηριστικών του pipeline αυτής. Συνεπώς, ο πηγαίος κώδικας θα μπορούσαμε να πούμε ότι είναι ``κομμένος και ραμμένος'' για την συγκεκριμένη αρχιτεκτονική κάνοντας πολύ περιορισμένη τη φορητότητα του αλλά και αρκτετά μεγάλο το χρονικό διάστημα ανάπτυξης του. Ακόμη, και σε μία υποθετική ετερόγενη παραπλήσια αρχιτεκτονική με τρία (3) PPE αντί του ενός, δεν θα ήταν δυνατό να χρησιμοποιηθεί ο ίδιος κώδικας. Χαρακτηριστικό παράδειγμα της πολυπλοκότητας προγραμματισμού του Cell είναι το γνωστό πρόγραμμα ``Hello World'' που παρατίθεται στην ενότητα 7.3.

2.3.4 Η κατάργηση της ανοιχτής πλατφόρμας και η ακύρωση του Cell/B.E.

Εκτός όμως από τις παραπάνω δυσκολίες που συναντήσαμε υπήρχε και ένας πολύ βασικός λόγος ακόμη.

Η εταιρία IBM ανακοίνωσε ότι σταματά την ανάπτυξη του επεξεργαστή και δεν θα υπάρξει επόμενη έκδοση κάνοντας, κατά τα φαινόμενα το Cell/B.E. μια μη βιώσιμη πλατφόρμα.

Επιπλέον, η εταιρία Sony επικαλούμενη ρήγματα ασφαλείας στο *HyperVisor* που επέτρεπαν σε κάθε ενδιαφερόμενο να έχει άμεση πρόσβαση στον πυρήνα του

Λειτουργικού Συστήματος αλλά και στους οδηγούς των συσκευών του Playstation 3 αλλά και για οικονομικούς λόγους καθώς δεν θα έπρεπε πλέον να υποστηρίζει τους οδηγούς HyperVisor, αποφάσισε να εγκαταλείψει την υποστήριξη στο λειτουργικό σύστημα Linux. Στην επόμενη έκδοση firmware της κονσόλας, η εταιρία κλείδωσε το σύστημα της στερώντας έτσι τη δυνατότητα από την επιστημονική κοινότητα να συνεχίσει να αναπτύσσει λογισμικό και εργαλεία για τη συγκεκριμένη πλατφόρμα.

Όλοι οι παραπάνω λόγοι μας οδήγησαν στην απόφαση να εγκαταλείψουμε τον επεξεργαστή Cell και να στραφούμε σε εναλλακτικά παράλληλα συστήματα. Χωρίς την υποστήριξη λογισμικού αλλά και την δυνατότητα εξέλιξης του επεξεργαστή Cell/B.E. η μόνη διέξοδος για προσιτή πλατφόρμα διανυσματικού προγραμματισμού αλλά και για υπολογιστικά συστήματα επιστημονικών υπολογισμών υψηλής απόδοσης βρίσκεται στο κόσμο των καρτών γραφικών και σε τεχνολογίες όπως οι Nvidia CUDA, AMD Cypress και Intel Larrabee.

Επιπλέον, η ευρεία διάδοση των υποσυστημάτων αυτών και η ευκολία προγραμματισμού τους, έκαναν τις κάρτες γραφικών την ιδανική πλατφόρμα για το μαζικά παράλληλο, χαμηλού κόστους σύστημα που αναζητούσαμε.

Στο επόμενο Κεφάλαιο γίνεται μια εισαγωγή στον αναγνώστη στη τεχνολογία *GPGPU (General-Purpose computing on graphics processing units - GP^2)* καθώς και στην αρχιτεκτονική CUDA.

3 *GPGPU and Compute Unified Device Architecture*

3.1 Γενικά

Η αυξανόμενη απαίτηση τα τελευταία χρόνια για υψηλής ευκρίνειας, σε πραγματικό χρόνο τρισδιάστατα γραφικά, είχε ως αποτέλεσμα την εξέλιξη του υποσυστήματος γραφικών σε ένα μαζικά παράλληλο, πολυνήματικό και πολυπύρηνο επεξεργαστή.

Είναι χαρακτηριστικό, ότι η βιομηχανία παιχνιδιών έχει ξεπεράσει σε απαίτηση για υψηλής απόδοσης επεξεργαστές άλλους βιομηχανικούς τομείς όπως της οικονομίας, υγείας και άμυνας. Ο παραλληλισμός των συστημάτων αυτών αυξάνεται συνεχώς υπακούοντας ακόμη και σήμερα στο νόμο του Moore. Ο λόγος που συμβαίνει αυτό σε σχέση με τους επεξεργαστές είναι επειδή η δουλειά ενός επεξεργαστή γραφικών είναι αρκετά πιο απλή: σε ένα απλοποιημένο μοντέλο αρκεί να έχει ως είσοδο ένα σύνολο απο πολύγωνα και να παράγει ως έξοδο ένα σύνολο από pixels. Επειδή τα πολύγωνα και τα pixels είναι ανεξάρτητα το ένα από το άλλο, μπορούν να επεξεργαστούν παράλληλα.

Οι δυνατότητες αυτών των επεξεργαστικών συστημάτων παρακίνησαν του ερευνητές να βρουν τρόπους ώστε να εκμεταλλευτούν αυτή την επεξεργαστική δύναμη για μη γραφικούς υπολογισμούς. Η ιδέα αυτή δεν ήταν καινούργια, όμως δεν υπήρχαν προγραμματιζόμενοι επεξεργαστές γραφικών μέχρι το 2003, όπου με την εξέλιξη των shaders, το υλικό της τότε εποχής ήταν ικανό για πολλαπλασιασμούς πινάκων.

Το μεγάλο όμως πρόβλημα για τους προγραμματιστές ήταν ότι για να αξιοποιήσουν την επεξεργαστική ισχύ των καρτών γραφικών για γενικούς υπολογισμούς έπρεπε να χρησιμοποιήσουν ένα από τα δύο API που ήταν διαθέσιμα, το Direct3D

και το OpenGL [19],[20]. Όμως, δεν ήταν όλοι οι προγραμματιστές γνώστες 3D γραφικών, γεγονός που καθιστούσε την τεχνολογία προσβάσιμη σε μικρό εύρος κοινού καθώς και αξιοποιήσιμη σε μικρό εύρος εφαρμογών. Έπρεπε να βρεθεί μια αναλογία μεταξύ του κόσμου των γραφικών και αυτού του παράλληλου προγραμματισμού.

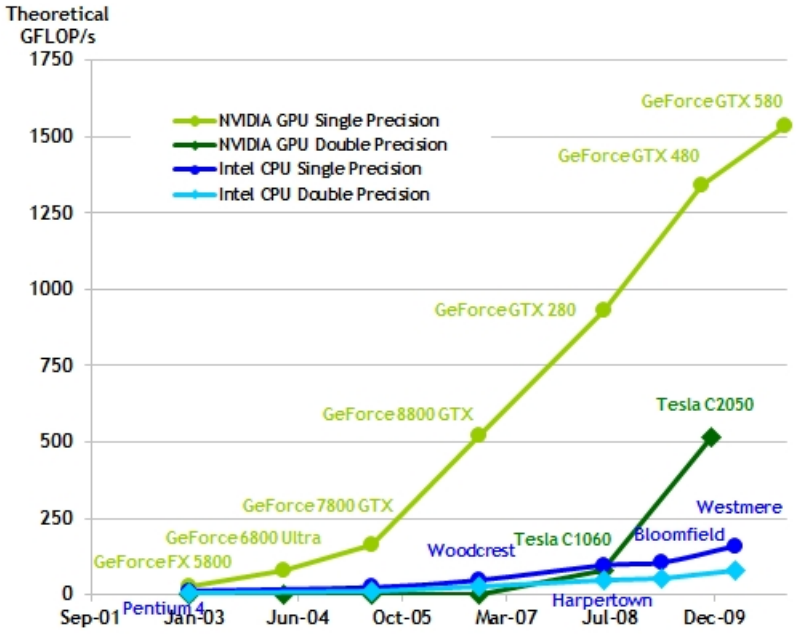
Το 2007 η εταιρία NVIDIA παρουσίασε τη τεχνολογία *Compute Unified Device Architecture (CUDA)*. Πρόκειται για ένα παράλληλο μοντέλου προγραμματισμού και μια πλατφόρμα λογισμικού που επιτρέπει στους προγραμματιστές να αναπτύξουν εφαρμογές που χρησιμοποιούν τον επεξεργαστή γραφικών, ξεπερνώντας τις δυσκολίες που υπήρχαν εως τώρα.

Παράλληλα με την αρχιτεκτονική αυτή, η NVIDIA προώθησε και μια καινούργια σειρά καρτών γραφικών που υλοποιούν την συγκεκριμένη αρχιτεκτονική. Οι συγκεκριμένες κάρτες ήταν πλέον ικανές όχι μόνο να επεξεργαστούν γραφικά, αλλά έγιναν μαζικά παράλληλα συστήματα που μπορούσαν να χρησιμοποιηθούν παράλληλα με τον επεξεργαστή, ως συνεπεξεργαστές. Ο επεξεργαστής θα αναλάμβανε τα προβλήματα που απαιτούσαν σειριακή επεξεργασία και η κάρτα γραφικών τα παράλληλα προβλήματα.

Οι πυρήνες του επεξεργαστή γραφικών με το CUDA, μπορούν πλέον να προγραμματιστούν με κάποιες προσθήκες στη γλώσσα προγραμματισμού C. Έτσι εξαλείφεται και η όποια δυσκολία και δυσπραγία υπήρχε από τη μεριά των προγραμματιστών.

Επιπλέον, επειδή οι κάρτες γραφικών υπάρχουν από σταθερούς και φορητούς υπολογιστές (desktops, laptops) μέχρι διακομιστές και σταθμούς εργασίας (servers, workstations), η τεχνολογία είναι ευρέως διαθέσιμη και με πολύ μικρό κόστος.

Ένα δείγμα του μεγέθους της απόδοσης της τεχνολογίας αυτής φαίνεται στα διαγράμματα που ακολουθούν, όπου παρουσιάζονται η απόδοση σε Gigaflops των καρτών γραφικών σε σχέση με μια κεντρική επεξεργαστική μονάδα, καθώς και κάποια παραδείγματα εφαρμογών που χρησιμοποιούν τη τεχνολογία CUDA καθώς και τα κέρδη σε απόδοση σε σχέση με ένα επεξεργαστή που ακολουθεί την Intel αρχιτεκτονική. [21]



Σχήμα 8: Performance over time

Example Application	*Url*	*Speedup*
Seismic Database	`http://www.headwave.com`	66x to 100x
Mobile Phone Antenna Simulation	`www.acceleware.com`	45x
Molecular Dynamics	`http://www.ks.uiuc.edu/Research/vmd`	21x to 100x
MRI processing	`http://www.bic-test.beckman.uiuc.edu`	245x to 415x

Πίνακας 1: Speedup of applications using cuda

3.2 Αρχιτεκτονική CUDA

Η υλοποίηση του CUDA αποτελείται τόσο από υλικό όσο και λογισμικό. Στην κατηγορία του υλικού εντάσσονται οι κάρτες γραφικών της εταιρίας. Οι κάρτες ικανές να υποστηρίξουν CUDA ξεκίνησαν από τη σειρά G80 και εκτείνονται μέχρι και σήμερα στις σειρές Geforce , την επαγγελματική Quadro και στη σειρά προοριζόμενη για επιστημονικούς υπολογισμούς Tesla.

Το λογισμικό αποτελείται από το κιτ προγραμματισμού SDK, το driver API, το runtime API καθώς και σε λογισμικό στο επίπεδο του πυρήνα του λειτουργικού συστήματος για την υποστήριξη υλικού.

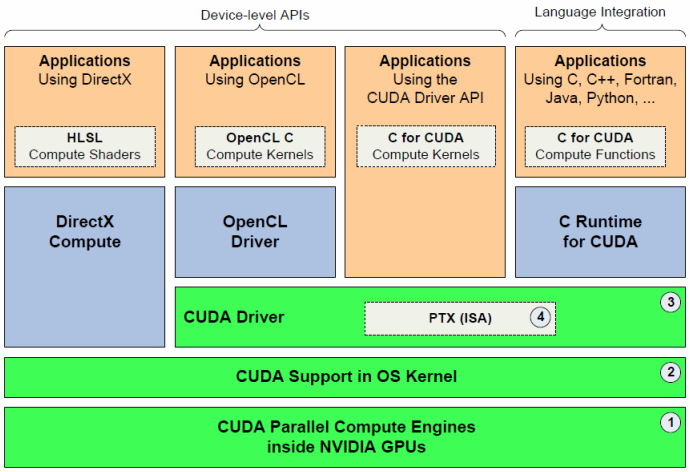
Το runtime API δεν χρησιμοποιείται εκτός και αν απαιτείται λεπτομερής και απόλυτος έλεγχος των CUDA threads.

Το driver API είναι αυτό που επιτρέπει τη χρησιμοποίηση της κάρτας γραφικών ως συνεπεξεργαστή και το SDK παρέχει τα απαραίτητα εργαλεία,παραδείγματα και βιβλιογραφία για την επιτυχή ανάπτυξη, μεταγλωτισσή και αποσφαλμάτωση εφαρμογών που μπορούν να χρησιμοποιήσουν την αρχιτεκτονική CUDA.

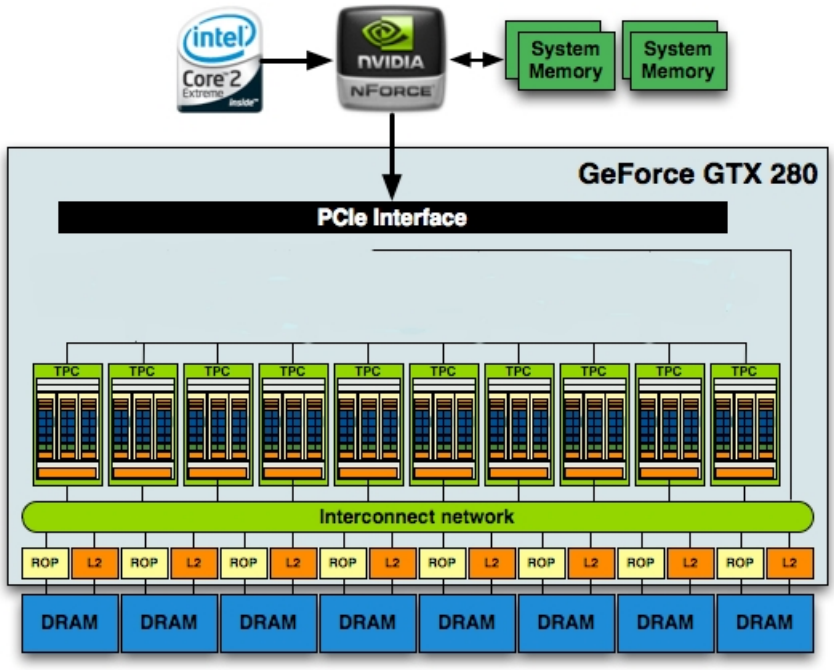
Η διασύνδεση των καρτών γραφικών με τα υπόλοιπα εξαρτήματα ενός υπολογιστή γίνεται μέσω του διαύλου *PCI Express (PCIe)* [22] ,[23]. Τον έλεγχο της επικοινωνίας μεταξύ της κάρτας γραφικών μέσω του διαύλου, του επεξεργαστή και του υποσυστήματος μνήμης αναλαμβάνει το ολοκληρωμένο σύστημα northbridge.[1] Στο παρακάτω σχήμα φαίνεται η συγκεκριμένη αρχιτεκτονική για τη σειρά καρτών γραφικών Geforce GT2$\times\times$.

Η NVIDIA ονομάζει κάθε επεξεργαστικό πυρήνα ως *SP (Streaming Processor)*. Κάθε SP είναι ένας πλήρης μικροεπεξεργαστής με pipeline, αποτελούμενος από

[1] Στις τελευταίες γενιές επεξεργαστών, ο ελεγκτής μνήμης (northbridge) περιέχεται μέσα στο ολοκληρωμένο κύκλωμα του επεξεργαστή.



Σχήμα 9: Η αρχιτεκτονική CUDA



Σχήμα 10: Διασύνδεση ολοκληρωμένων μέσω του διαύλου PCIe

δύο Αριθμητικές Λογικές Μονάδες (ALUs) και μια μονάδα κινητής υποδιαστολής (FPU) και δεν διαθέτει καθόλου μνήμη cache. Οκτώ τέτοιοι μικροεπεξεργαστές μαζί με δύο Ειδικές Μονάδες (Special Functions Units - SFUs) αποτελούν ένα Streaming Multiprocessor (SM). Ένα SM έχει και μια μικρή instruction cache, μια κρυφή μνήμη για ανάγνωση δεδομένων (read-only data cache) καθώς και 16KB κοινόχρηστης μνήμης (shared memory).

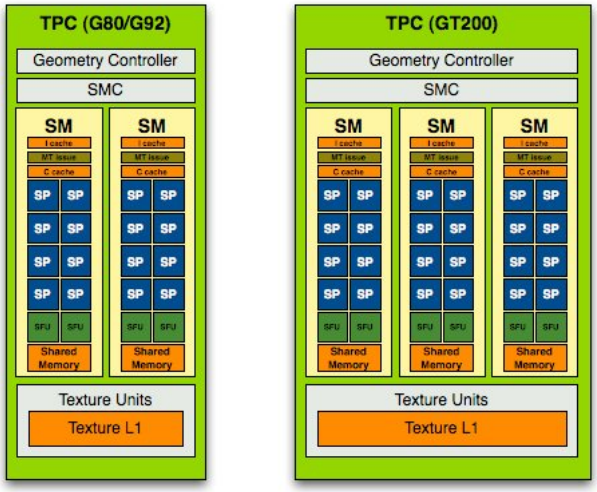
Τα 16KB μνήμης θα μπορούσαν να παρομοιαστούν με το local store του Cell processor. Η ομοιότητα τους έγκειται οτι και τα δύο χρησιμοποιούνται για την αποθήκευση δεδομένων ώστε να αποκρύπτεται η καθυστέρηση από τη μεταφορά δεδομένων (latency).

Η τεχνολογία CUDA ξεχωρίζει για την αρθρωτή σχεδίαση. Έτσι, η μονάδα που αποτελείται από κάποιο αριθμό SM καλείται Texture/Processor Cluster(TPC). Στην αρχιτεκτονική των καρτών της γενίας G80 ένα TPC αποτελούνταν από 2 SMs ενώ στη σειρά Geforce GT2$\times\times$ έχει 3 SMs.

Η μονάδα TPC αποτελείται επιπλέον από Texture Units στα οποία περιέχται texture μνήμη και ελεγκτής αυτής. Συνοψίζοντας, ένα TPC της γενίας Geforce 2$\times\times$ αποτελείται από 24 SPs και 6 SFUs.

Η αρθρωτή σχεδίαση ολοκληρώνεται με το Streaming Processor Array. Πρόκειται για μια σειρά αποτελούμενη από TPCs. Στη σειρά G80 αποτελούνταν από 8, ενώ στη σειρά Geforce GT2xx ο αριθμός έχει ανέβει στα 10. Έτσι η συνολική επεξεργαστική ισχύς της σειράς 2$\times\times$ έχει αυξηθεί κατά 87.5% σε σχέση με τη σειρά G80.

Τέλος, η GPU περιέχει διεκπεραιωτές (schedulers) και control logic για να διανέμει το φόρτο εργασίας σε ολόκληρη τη σειρά των πυρήνων. Συνοψίζοντας όλα τα παραπάνω η κάρτα της σειράς Geforce GTX 260, την οποία και χρησιμοποιή-



Σχήμα 11: Texture/Processor Cluster Units

σαμε στη εργασία μας, περιέχει 216 πυρήνες CUDA, 160KB local memory, μνήμη cache και αποτελείται από 1,4 δισεκατομμύρια τρανζίστορ κατασκευασμένα στα 65nm.

Κάθε SM είναι σχεδιασμένο να εκτελεί παράλληλα εκατοντάδες threads. Για να το επιτύχει αυτό, η NVIDIA χρησιμοποιήσε την αρχιτεκτονική *SIMT (Single-Instruction, Multiple-Thread)* που αναλύεται στη επόμενη υποενότητα.

3.2.1 SIMT Architecture

Κάθε SM δημιουργεί, δρομολογεί και εκτελεί νήματα (threads) σε ομάδες των τριάντα δύο (32) παράλληλων νημάτων που ονομάζονται warps.

Κάθε νήμα που αποτελεί ένα warp ξεκινά από την ίδια διεύθυνση προγράμματος (program address), αλλά έχουν το δικό τους μοναδικό μετρητή διεύθυνσης εντολών (instruction address counter) και τη δική τους κατάσταση στο αρχείο καταχωρητών (register state), οπότε και μπορούν να διακλαδωθούν και να εκτελε-

στούν αυτόνομα.

Ενδιαφέρον προκαλεί η ονομασία warp, την οποία η NVIDIA διάλεξε από την ύφανση και συγκεκριμένα από τον παραδοσιακό αργαλείο λόγω του τρόπου που περνούν παράλληλα τα νήματα της κλωστής από το μηχάνημα.

Ένα half-warp είναι το πρώτο ή δεύτερο μισό ενός warp, ενώ ένα quarter-warp είναι το πρώτο, δεύτερο, τρίτο ή τέταρτο τμήμα ενός warp.

Όταν δίνεται σε ένα SM μια ομάδα από νήματα (thread block) να εκτελέσει, αυτά χωρίζονται σε warps με διαδοχικά id (thread id) και δρομολογούνται από το warp scheduler , με το πρώτο warp να περιέχει το νήμα με thread 0.

Κάθε warp εκτελεί μια εντολή τη φορά, οπότε η μέγιστη αποδοτικότητα επιτυγχάνεται όταν και τα τριάντα δύο (32) νήματα του warp έχουν το ίδιο μονοπάτι εκτέλεσης (execution path). Σε αντίθετη περίπτωση όταν κάποιο ή κάποια από τα νήματα του warp παρεκκλίνουν λόγω διακλαδώσεων στα δεδομένα (data-dependent conditional branch) όπως στην περίπτωση ενός ισχυρισμού if , τότε το κάθε μονοπάτι από τις διακλαδώσεις εκτελείται σειριακά και μόνο όταν έχουν εξαντληθεί όλες οι διακλαδώσεις, τα νήματα του warp επιστρέφουν όλα στο ίδιο μονοπάτι.

Απόκλιση στο μονοπάτι εκτέλεσης λόγω διακλαδώσεων (branch divergence) μπορεί να συμβεί μόνο μέσα σε ένα warp και αυτό διότι κάθε warp εκτελείται αυτόνομα ανεξάρτητα από το τμήμα του κώδικα που εκτελεί.

Μια θεμελιώδης διαφορά της αρχιτεκτονικής *SIMT* σε σχέση με την προσέγγιση *SIMD (Single-Instruction,Multiple Data)* είναι οτι επιτρέπει στους προγραμματιστές να αναπτύσσουν παράλληλο λογισμικό αγνοώντας ουσιαστικά το τρόπο με τον οποίο δουλεύει η αρχιτεκτονική SIMT σε αντίθεση με την αρχιτεκτονική SIMD όπου ο προγραμματιστής πρέπει λάβει υπόψιν του το πλήθος των πράξεων

που μπορούν να εκτελεστούν με μια εντολή.

3.2.2 Παραλληλισμός στο επίπεδο του υλικού

Οι μετρητές προγράμματος, το αρχείο καταχωρητών καθώς και όλα τα στοιχεία εκτέλεσης (execution state) για κάθε warp που εκτελείται διατηρούνται στη κάρτα γραφικών (on-chip) οπότε το κόστος εναλλαγής των warps (context-switching) είναι μηδαμινό. Επομένως, κάθε φορά που δίνεται μια καινούργια εντολή, δρομολογείται στο warp που έχει έτοιμα νήματα για εκτέλεση της εντολής αυτής.

Ο αριθμός των ομάδων από νήματα (thread blocks) και των warps που μπορούν να εκτελούνται ταυτόχρονα καθορίζεται από τον αριθμό των καταχωρητών (registers) και το μέγεθος της κοινής μνήμης (shared memory) που χρησιμοποιούνται σε σχέση με τον αριθμό και την ποσότητα που είναι διαθέσιμα.

Ο συνολικός αριθμός από warps μέσα σε μια ομάδα νημάτων (thread block) W_{block} προκύπτει από την παρακάτω εξίσωση:

$$W_{block} = \lceil (\frac{T}{W_{size}}, 1) \rceil$$

όπου

- T, ο αριθμός των νημάτων,
- W_{size}, Το μέγεθος του warp, το οποίο ισούται με 32,
- $\lceil (x, y) \rceil$, ισούται με x, στρογγυλοποιημένο στο κοντινότερο πολλαπλάσιο του y.

Αντίστοιχα, ο αριθμός των καταχωρητών προκύπτει ως εξής:

Για συσκευές με υπολογιστική δυνατότητα (compute capability) 1.x:

$$R_{block} = \lceil \lceil W_{block}, G_w \rceil \times W_{size} \times R_k, G_T \rceil$$

Για συσκευές με υπολογιστική δυνατότητα (compute capability) 2.x:

$$R_{block} = \lceil R_k \times W_{size}, G_T \rceil \times W_{block}$$

όπου

- G_W, ο βαθμός λεπτομέρειας του warp (warp allocation granuality), που ισούται με 2,
- R_k, ο αριθμός των καταχωρητών που χρησιμοποιούνται από μια συνάρτηση (kernel),
- G_T, ο βαθμός λεπτομέρειας των νημάτων (thread allocation granuality), που ισούται με:
  - 256 για συσκευές με υπολογιστική ικανότητα 1.0,
  - 512 για συσκευές με υπολογιστική ικανότητα 1.2 και 1.3,
  - 64 για συσκευές με υπολογιστική ικανότητα 2.$\times$.

Τέλος, ο συνολικός αριθμός κοινόχρηστης μνήμης (shared memory) S_{block} που δεσμεύεται από μια ομάδα νημάτων (thread block) είναι:

$$S_{block} = \lceil S_k, G_S \rceil$$

όπου

- S_k, το μέγεθος κοινής μνήμης που χρησιμοποιείται από μια συνάρτηση (kernel),
- G_S, ο βαθμός λεπτομέρειας της κοινόχρηστης μνήμης (shared memory allocation granuality), που ισούται με:
  - 512 για συσκευές με υπολογιστική ικανότητα $1.\times$
  - 128 για συσκευές με υπολογιστική ικανότητα $2.\times$.

3.3 Το μοντέλο προγραμματισμού *CUDA*

Ένα πρόγραμμα γραμμένο για την τεχνολογία CUDA, αποτελείται από τον πηγαίο κώδικα που είναι γραμμένος για να εκτελείται στη Κεντρική Μονάδα Επεξεργασίας (CPU) και από το τμήμα του κώδικα, το οποίο είναι κατάλληλο για παράλληλη επεξεργασία, που εκτελείται στη κάρτα γραφικών. Λόγω αυτού του τρόπου εκτέλεσης ενός προγράμματος, η αρχιτεκτονική CUDA θεωρείται ετερόγενη αρχιτεκτονική.

Το τμήμα του πηγαίου κώδικα που τρέχει στο CPU, έχει ως πρωταρχικό σκοπό να αρχικοποιήσει και να εκχωρήσει τη μνήμη στη κάρτα γραφικών, να μεταφέρει τα δεδομένα προς επεξεργασία στη μνήμη της κάρτας γραφικών καθώς και να δρομολογήσει ένα kernel[3.3.1]. Η δρομολόγηση των kernels γίνεται ασύγχρονα. Όταν δρομολογηθεί ένας kernel, ο έλεγχος επιστρέφει στο CPU που συνεχίζει την εκτέλεση του υπόλοιπου προγράμματος εφόσον αυτό υπάρχει.

3.3.1 Kernels, μέγεθος Grid και μέγεθος Block

Τα τμήματα του κώδικα που εκτελούνται παράλληλα στη κάρτα γραφικών, ονομάζονται kernels. Πρόκειται για συναρτήσεις γραμμένες σε μια διευρυμένη

έκδοση της γλώσσας προγραμματισμού C. Ένας kernel εκτελείται ανά στιγμή στη κάρτα γραφικών, και πολλά νήματα cuda (cuda threads) εκτελούν αντίστοιχα ένα kernel.

Τα νήματα cuda διαφέρουν από τα ``παραδοσιακά`` νήματα που εκτελούνται στη Κεντρική Μονάδα Επεξεργασίας ως προ το ότι είναι πάρα πολύ ελαφρά, δεν εισάγουν καθυστέρηση για τη δημιουργία τους (creation overhead) και η εναλλαγή των νημάτων γίνεται στιγμιαία (instant switching). Επιπλέον, για να υπάρχει αποδοτικότητα, η τεχνολογία CUDA χρησιμοποιεί χιλιάδες νήματα, ενώ ένας πολυπύρηνος επεξεργαστής χρησιμοποιεί μόλις δεκάδες.

Ένας kernel εκτελείται από ένα μονοδιάστατο πίνακα (μια σειρά) από νήματα που όλα εκτελούν τον ίδιο κώδικα και το καθένα από αυτά έχει το δικό του αριθμητικό διαχωριστικό (thread ID) το οποίο και χρησιμοποιεί για να τον υπολογισμό διευθύνσεων μνήμης καθώς και για αποφάσεις σχετικές με διακλαδώσεις εκτέλεσης.

Το σύνολο των νημάτων που εκτελούν ένα kernel ομαδοποιούνται σε ένα δισδιάστατο πλέγμα από μπλοκ νημάτων. Το κάθε μπλοκ αποτελεί μια τρισδιάστατη δομή από νήματα. Με τον τρόπο αυτό το κάθε thread ID είναι μοναδικό για το κάθε νήμα που εκτελούν ένα kernel. Η αριθμοδότηση σε κάθε νήμα που ανήκει σε αυτή τη πολυδιάστατη δομή γίνεται μέσω κάποιων ειδικών μεταβλητών που είναι υλοποιημένες στο CUDA.

Το κάθε μπλοκ αποτελεί, όπως αναφέραμε προηγουμένως, μια δισδιάστατη δομή. Η διευθυνσιοδότηση σε αυτή τη δομή γίνεται μέσω της ειδικής μεταβλητής *blockIdx*. Συγκεκριμένα, με την μεταβλητή *blockIdx.x* για το x-άξονα και τη μεταβλητή *blockIdx.y* για τον y-άξονα.

Κάθε μπλοκ περιέχει και ορίζει μια τρισδιάστατη δομή νημάτων. Η μεταβλητή

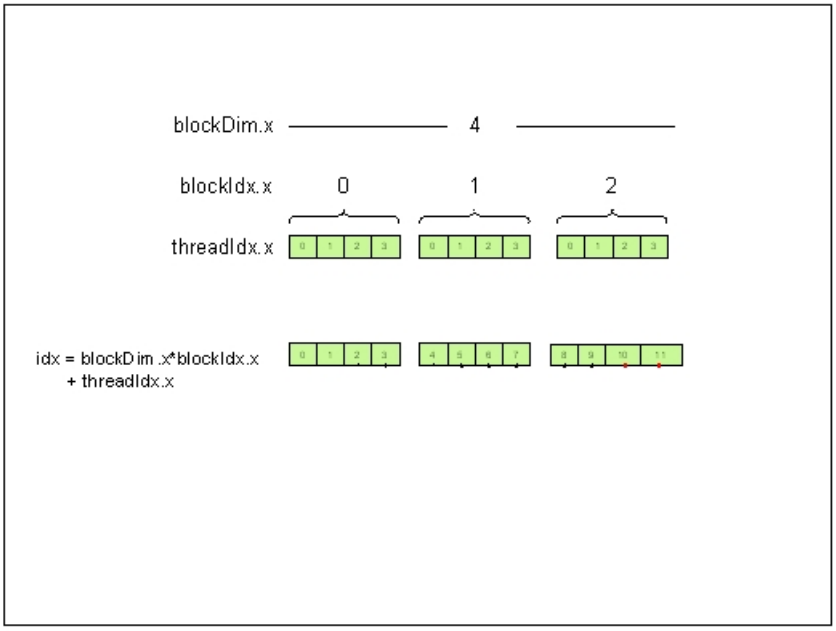
για την διευθυνσιοδότηση στη συγκεκριμένη δομή είναι η *threadIdx*. Συγκεκριμένα η *threadIdx.x* για τον x-άξονα, η *threadIdx.y* για τον y-άξονα και η *threadIdx.z* για τον z-άξονα.

Τέλος, υπάρχει και η μεταβλητή *threadDim.x* που είναι ο αριθμός των νημάτων που ανήκουν σε ένα μπλοκ.

Με τις τρείς (3) αυτές μεταβλητές μπορεί να οριστεί μοναδικά ένα νήμα. Για παράδειγμα για ένα πλέγμα αποτελούμενο από τρία μπλοκ των τεσσάρων νημάτων διαστάσεων $1 \times 1 \times 12$ έχουμε:

$$int\ idx = blockDim.x * blockIdx.x + threadIdx.x \tag{1.1}$$

όπου blockDim.x = 4, blockIdx.x = 0, 1, 2 και threadIdx.x = 0, 1, 2, 3



Σχήμα 12: Παράδειγμα αριθμοδότησης νημάτων

Οι διαστάσεις ενός kernel μπορούν να δωθούν ως ορίσματα κατά τη δρομολό-

γηση του με τον εξής τρόπο:

$$function\_name <<< gridSize, blockSize >>> (arg1, arg2, ..., argn)$$

Οι μεταβλητές *gridSize, blockSize* είναι τύπου dim3. Πρόκειται για μια δομή δεδομένων τύπου διανύσματος στη C, βασισμένο στη δομή uint3 και χρησιμοποιείται μόνο για τον ορισμό των διαστάσεων. [28]

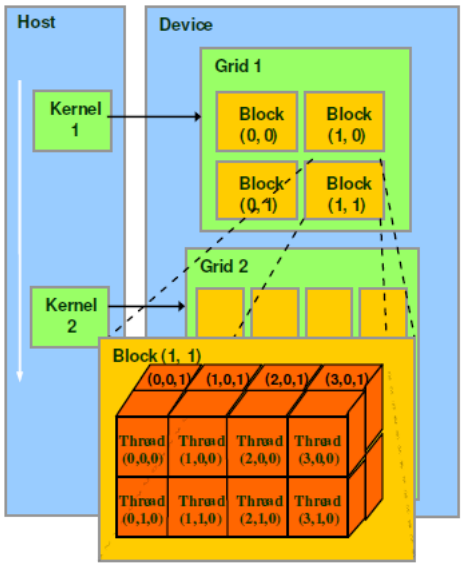
Ο ορισμός μιας συνάρτησης που θα εκτελεστεί από τα νήματα CUDA ως kernel ορίζεται με τον ακόλουθο τρόπο:

$$\_\_global\_\_ \ void \ function\_name <<< gridSize, blockSize >>> (arg1 \ , arg2 \ , \ ... \ , argn)$$

Τα νήματα ενός kernel που ανήκουν μόνο στο ίδιο μπλοκ μπορούν να συνεργάζονται και να επικοινωνούν μέσω της κοινής μνήμης (shared memory). Με το τρόπο αυτό αποφεύγονται υπολογισμοί αλλά και μειώνεται δραστικά η απαίτηση σε εύρος μνήμης.

3.3.2 Ιεραρχία μνήμης

Μέχρι τώρα έχει καταστεί σαφές οτι το CPU και το GPU το κάθε ένα έχει το δικό του χώρο μνήμης. Δουλεία του Host, όπως αναφέραμε προηγουμένως είναι να εκχωρήσει μνήμη στη κάρτα γραφικών. Αυτό γίνεται μέσω του API που προσφέρει το CUDA. Υπάρχουν αρκετές συναρτήσεις και αρτκετοί τρόποι εκχώρησης μνήμης. Ο πιο απλός τρόπος είναι μέσω μια τροποποιημένης έκδοσης της συνάρτησης *malloc* για CUDA. Η σύνταξη είναι ως εξής:



Σχήμα 13: Στο σχήμα αυτό φαίνεται ο τρόπος εκτέλεσης δύο kernels παράλληλα με το κώδικα που τρέχει στο host. Επίσης διακρίνεται και η δομή ενός πλέγματος.

$$cudaError\_t\ cudaMalloc(void\ **pointer,\ size\_t\ nbytes);$$

Η μεταφορά των δεδομένων από και προς τη κάρτα γραφικών γίνεται με τη συνάρτηση *cudaMemcpy* ως εξής:

$$cudaError\_t\ cudaMemcpy(void\ *dst,\ void\ *src,\ size\_t\ nbytes,$$
$$enum\ cudaMemcpyKind\ direction);$$

όπου η παράμετρος enum cudaMemcpyKind μπορεί να είναι:

- cudaMemcpyHostToDevice

  Για μεταφορά δεδομένων από το host σύστημα, προς τη κάρτα γραφικών.

- cudaMemcpyDeviceToHost

  Για μεταφορά δεδομένων από τη κάρτα γραφικών προς το host σύστημα.

- cudaMemcpyDeviceToDevice

  Για μεταφορά δεδομένων από τη κάρτα γραφικών προς την κάρτα γραφικών.

Εκτός από την εκχώρηση μνήμης με τον παραπάνω τρόπο, υπάρχει και η δυνατότητα να χρησιμοποιηθεί μνήμη "pinned``. Με τον τρόπο αυτό το υποσύστημα μνήμης του υπολογιστή γίνεται κοινό τόσο για το Host όσο και για το GPU και έτσι αποφεύγονται οι μετακινήσεις δεδομένων από και προς τη κάρτα γραφικών.

Το μειονέκτημα της μεθόδου αυτής είναι αφενός ότι καταναλώνει μέρος της μνήμης του συστήματος που μπορεί να αποβεί προβληματικό γιατί δεν υπάρχει κάποιος έλεγχος για την ποσότητα μνήμης που δεσμεύει, και αφετέρου η μνήμη του συστήματος είναι αισθητά πιο αργή από τη μνήμη της κάρτας γραφικών.

3.3.3 Shared Memory

Όπως έχουμε αναφέρει, το GPU έχει και κοινόχρηστη μνήμη (shared memory), συγκεκριμένα 16KB, η οποία είναι αρκετές τάξης μεγέθους πιο γρήγορη από τη μνήμη της κάρτας γραφικών (global memory). Για να το επιτύχει αυτό, η μνήμη είναι χωρισμένη σε ισόποσα τμήματα που ονομάζονται banks και μπορούν να προσπελαστούν ταυτόχρονα. Για παράδειγμα, αν υπάρχουν $\times$ αιτήματα για εγγραφή ή ανάγνωση, όπου το κάθε ένα είναι για μια διεύθυνση που βρίσκεται σε $\times$ διαφορετικά banks, τότε το εύρος ζώνης είναι $\times$ φορές μεγαλύτερο από το εύρος μιας ενιαίες μονάδας μνήμης.

Αντιθέτως, αν κάποιες από τις διευθύνσεις βρίσκονται στο ίδιο bank, τότε υπάρχει bank conflict και η πρόσβαση στη μνήμη γίνεται σειριακά. Όπως γίνεται κατανοητό, ο σωστός χειρισμός της συγκεκριμένης μνήμης είναι απαραίτητος ώστε να εξαλείφονται τα bank conflicts για να επιτυγχάνεται η μέγιστη απόδοση.

Η εκχώρηση μνήμης μέσα από ένα kernel γίνεται ως εξής:

$$\_\_shared\_\_\ data\_type\ smem\_array[\ nbytes\ ];$$

για παράδειγμα

$$\_\_shared\_\_\ int\ smem\_array[\ 128\ ];$$

Στην επόμενη ενότητα περιγράφεται το μαθηματικό μοντέλο για τη μερική λύση της διαφορικής εξίσωσης Black-Scholes και παρουσιάζεται η εφαρμογή την οποία επιλέξαμε για παραλληλοποίηση.

4 Black-Scholes Option Pricing

4.1 Γενικά

Η πρόβλεψη των τιμών ενός χαρτοφυλακίου αποτελεί ένα μεγάλο πρόβλημα των μηχανικών που απασχολούνται στο τομέα των χρηματοοικονομικών από τότε που οργανώθηκε η διαπραγμάτευση μετοχών, το 1973[2]. [31]

Υπάρχουν δίαφορα είδη μετοχών:

- Οι μετοχές με Δικαίωμα Αγοράς (call option)
- Οι μετοχές με Δικαίωμα Πώλησης (put option)

Η απλούστερη μορφή μετοχής είναι η Ευρωπαϊκή μετοχή (European call option). Ουσιαστικά, πρόκειται για μια μορφή δέσμευσης μεταξύ δύο πλευρών. Του εκδότη της μετοχής και του επενδυτή, όπου ο επενδυτής έχει το δικαίωμα αλλά οχι και την υποχρέωση, είτε να αγοράσει ένα υποκείμενο χρεόγραφο (Δικαίωμα Αγοράς) είτε να πουλήσει ένα υποκείμενο χρεόγραφο (Δικαίωμα Πώλησης) σε μια συγκεκριμένη τιμή εξάσκησης (strike price), στην ημερομηνία λήξης του χρεογράφου.

Πρόκειται για μια μορφή δέσμευσης, γιατί ενώ ο επενδυτής έχει το δικαίωμα της αγοράς ή πώλησης, ο εκδότης έχει την πιθανή υποχρέωση της πώλησης ή αγοράς εφόσον ο κομιστής του δικαιώματος επιλέξει την εξάσκηση αυτού.

Αντιθέτως, οι Αμερικάνικες μετοχές άρουν το χρονικό περιορισμό και μπορούν να διαπραγματευτούν ανά πάσα χρονική στιγμή μέχρι την καταληκτική τους ημερομηνία, συμπεριλαμβανομένης αυτής. [32], [33]

[2]Η οργανωμένη διαπραγμάτευση μετοχών, με τη μορφή που γνωρίζουμε σήμερα, ξεκίνησε το 1973 με την ίδρυση του πρώτου χρηματιστηρίου στη πολιτεία του Σικάγο *(C.B.O.E. - Chicago Board Options Exchange*)

Το κέρδος για μια μετοχή με Δικαίωμα Αγοράς προκύπτει από την διαφορά της τιμής του χρεογράφου την ημέρα λήξης του και της τιμής εξάσκησης, μείον το τίμημα που είχε πληρωθεί για την αγορά των δικαιωμάτων του. Αντιθέτως, για μια μετοχή με Δικαίωμα Πώλησης το κέρδος προκύπτει από τη διαφορά της τιμής εξάσκησης και της τιμής του χρεογράφου την ημέρα λήξης, μείον το αντίτιμο που είχε δοθεί για την αγορά του.

4.2 Black-Scholes equation

Από τα παραπάνω, γίνεται σαφές οτι η τιμή μιας μετοχής επηρεάζεται από παράγοντες, όπως η τρέχουσα τιμή ενός χρεόγραφου, η ημερομηνία λήξης αυτού καθώς και η τιμή εξάσκησης.

Άλλος σημαντικός παράγοντας που επηρεάζει την τιμή μιας μετοχής είναι το Επιτόκιο Μηδενικού Κινδύνου το οποίο μπορεί να επιτευχθεί επενδύοντας σε οικονομικά προϊόντα που δεν ενσωματώνουν ρίσκο. Παρόλο που ακίνδυνη επένδυση υπάρχει μόνο στη θεωρία, μια ασφαλής επένδυση θεωρούνται τα κρατικά ομόλογα καθώς η πιθανότητα να πτωχεύσει μια χώρα είναι πραγματικά πολύ μικρή, εκτός και αν κάποιος έχει επενδύσει στην Ελλάδα.

Στο επιτόκιο αυτό r, το κέρδος μετά από τη πάροδο ενός χρονικού διαστήματος T (συνήθως αρκετά χρόνια) για μια επένδυση ενός ποσού P, είναι $P \times e^{rT}$.

Το μαθηματικό μοντέλο που χρησιμοποιείται κατα κόρον και περιγράφει καλύτερα την συμπεριφορά της τιμής μιας μετοχής είναι το μοντέλο Black-Scholes για πρόβλεψη τιμών μετοχών. Με την υπόθεση της μη Εξισορροπητική Κερδοσκοπίας (arbitage) [3] η διαφορική εξίσωση Black-Scholes είναι:

[3] Το arbitage είναι η ταυτόχρονη αγορά και πώληση της ίδιας ή παρόμοιας επένδυσης σε δύο διαφορετικές αγορές και σε δύο διαφορετικές τιμές οι οποίες μπορούν να οδηγήσουν σε κέρδη χωρίς την ανάληψη κινδύνου

$$rS = \frac{\partial S}{\partial t} + rx\frac{\partial S}{\partial t} + \frac{1}{2}\sigma^2 x^2 \frac{\partial S}{\partial x^2} \tag{1.2}$$

με

$$S := S(t,x)\ ,\ \ t\epsilon[0,T]\ ,\ \ x\epsilon[0,X_{max}] \tag{1.3}$$

όπου:

- S, η τιμή της μετοχής,
- r, το επιτόκιο μηδενικού κινδύνου,
- σ, η διακύμανση της τιμής της μετοχής,
- x, το πλήθος των μετοχών (πλήθος χρεωγράφων),
- t, ο χρόνος μέχρι την ημερομηνία λήξης, με $t\epsilon[0,T]$, όπου T η ημερομηνία λήξης.

Οι συνοριακές συνθήκες για την δευτεροβάθμια παραβολική μερική διαφορική εξίσωση 1.2 είναι:

$$S(T,x) = max\{x-K,0\}, \tag{1.3a}$$

$$S(t,0) = 0, \tag{1.3b}$$

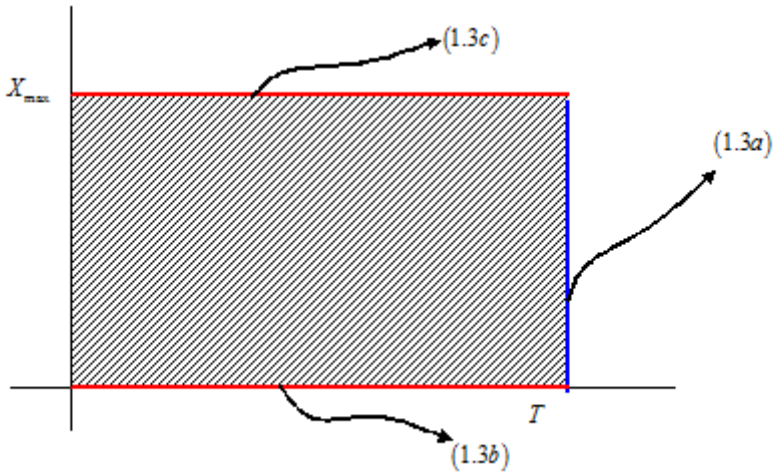
$$\frac{\partial^2 S}{\partial x^2}|_{(t,X_{max})} = 0 \tag{1.3c}$$

όπου r, σ, K πραγματικές και γνήσια θετικές σταθερές και $S(T,x) = max\{x-K,0\}$ είναι η τιμή της μετοχής την ημερομηνία λήξης.

Η εξίσωση (1.2) δεν μπορεί να λυθεί αναλυτικά για όλες τις περιπτώσεις μετοχών. Μπορεί όμως να προσεγγιστεί η λύση της με αλγεβρικές μεθόδους πεπε-

ρασμένων διαφορών. Με τη μέθοδο των πεπερασμένων διαφορών υπολογίζεται προσεγγιστικά η τιμή της άγνωστης συνάρτησης σε ένα σύνολο διακριτών σημείων. Οι μερικές παράγωγοι που προκύπτουν από την εξίσωση (1.2) αντικαθίστονται από πεπερασμένες διαφορές που προκύπτουν σε ανάλυση σειρών Taylor στα διακριτά σημεία που μας ενδιαφέρουν.

Το χωρίο στο οποίο θέλουμε να επιλύσουμε το πρόβλημα μας, που περιγράφεται από τις εξισώσεις (1.4) φαίνεται στο παρακάτω σχήμα:



Σχήμα 14: Το διακριτό πλέγμα επίλυσης του προβλήματος.

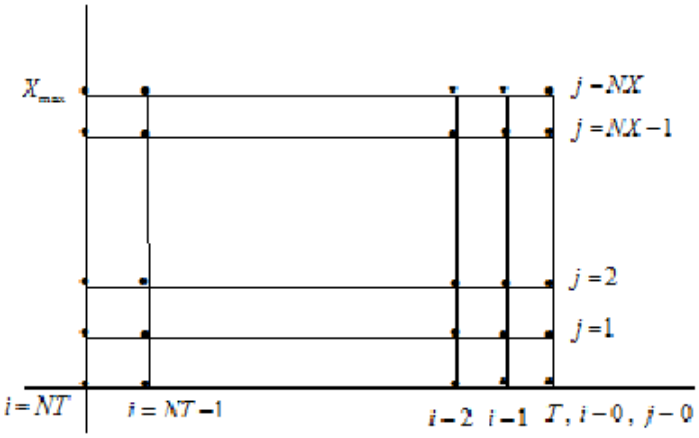
Η έμμεση μέθοδος των πεπερασμένων διαφορών που επιλέχθηκε είναι η μέθοδος Crank-Nicholson. Ο λόγος που χρησιμοποιείται έμμεση μεθοδολογία είναι ότι έχουμε μεγαλύτερη ευστάθεια κατά την αριθμητική επίλυση του προβλήματος. Πέρα από την ευστάθεια υπάρχει και μεγάλη ακρίβεια με τη συγκεκριμένη μεθοδολογία, γεγονός που επιτρέπει να χρησιμοποιήσουμε και μεγαλύτερα χρονικά διαστήματα:

$$\frac{\partial S}{\partial t} \approx \frac{S(x, t + \delta t) - S(x, t)}{\delta t} \tag{1.5}$$

4.3 Crank-Nicholson Scheme

Η συγκεκριμένη μέθοδος είναι μια επαναληπτική μέθοδος πεπερασμένων διαφορών που υπολογίζει τη τιμή της υπό εξέτασης συνάρτησης σε διακριτά σημεία. Ο τομέας στον οποίο επιλύουμε περιγράφεται από τις εξισώσεις (1.4) και φαίνεται στο Σχήμα 14.

Διακριτοποιούμε το χωρίο μας, χρησιμοποιώντας *(NX + 1)* ισαπέχοντες χωρικούς κόμβους και *(NT + 1)* ισαπέχοντες χρονικούς κόμβους όπως φαίνεται και στο σχήμα που ακολουθεί. Χρησιμοποιούμε το συμβολισμό ***i*** για τους χωρικούς κόμβους και το συμβολισμό ***j*** για τους χρονικούς κόμβους.



Σχήμα 15: Οι κόμβοι επίλυσης i, j

Δεδομένης της συνοριακής συνθήκης (1.3a) θέλουμε να ισχύει $i = 0$ για $t = T$ καθότι μέσω αυτής γνωρίζουμε τις τιμές της συνάρτησης για την τελική χρονική

στιγμή $t = T$ και θέλουμε να προσδιορίσουμε αριθμητικά τις τιμές της για προηγούμενες χρονικές στιγμές. Επιπλέον ισχύει $i = NT$ για $t = 0$, ενώ για τον τυχαίο κόμβο i έχουμε:

$$t_i = T - i \times \delta t, \text{ όπου } \delta t = \frac{T}{NT}, \ i = 0, 1, \ \dots, NT \tag{1.6}$$

Για τους χωρικούς κόμβους, ισχύει $j = 0$ όταν $x = 0$ και $j = NX$ για $x = X_{max}$, ενώ για τον τυχαίο κόμβο j έχουμε:

$$x_j = j \times \delta x, \text{ όπου } \delta x = \frac{X_{max}}{NX}, j = 0, 1, \ \dots, NX \tag{1.7}$$

Μέσω της αριθμητικής επίλυσης, όπως έχουμε αναφέρει, υπολογίζουμε προσεγγιστικά τις τιμές της συνάρτησης $S(t, x)$ στους κόμβους του πλέγματος στο σχήμα 15 και ειδικότερα των τιμών $S(t_i, x_j) \ \forall \ i\epsilon[0, NX], \ j\epsilon[0, NT]$.

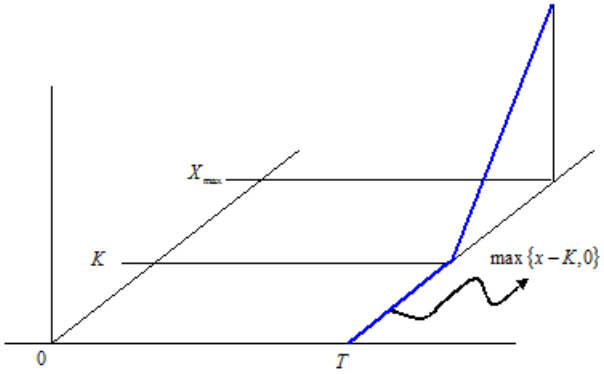
Συμβολίζουμε με

$$S_{i,j} := S(t_i, x_j) \tag{1.8}$$

την τιμή της συνάρτησης $S(\cdot)$ στον κόμβο i, j.

Λόγω των συνοριακών συνθηκών (1.3a) και (1.3b), οι τιμές $S_{0,j}, j\epsilon[0, NX]$ και $S_{i,0}, i\epsilon[0, NT]$, είναι εξαρχής γνωστές. Η τιμή $S_{0,0}$ συναληθεύεται και από τις δύο συνοριακές συνθήκες. Η συνοριακή συνθήκη (1.3a) φαίνεται στο σχήμα που ακολουθεί.

Γνωρίζουμε από τη (1.3a), ότι τη χρονική στιγμή $t = T = t_0$, τις τιμές $S_{0,j}$. Με την έμμεση μέθοδο πεπερασμένων διαφορών που χρησιμοποιούμε και λαμβάνοντας υπόψιν τις συνοριακές (1.3b), (1.3c) μπορούμε να υπολογίσουμε τις τιμές $S(\cdot)$, δηλαδή τις τιμές της την αμέσως προηγούμενη χρονική στιγμή $t = T - \delta t$.



Σχήμα 16: Η συνοριακή συνθήκη (1.3a)

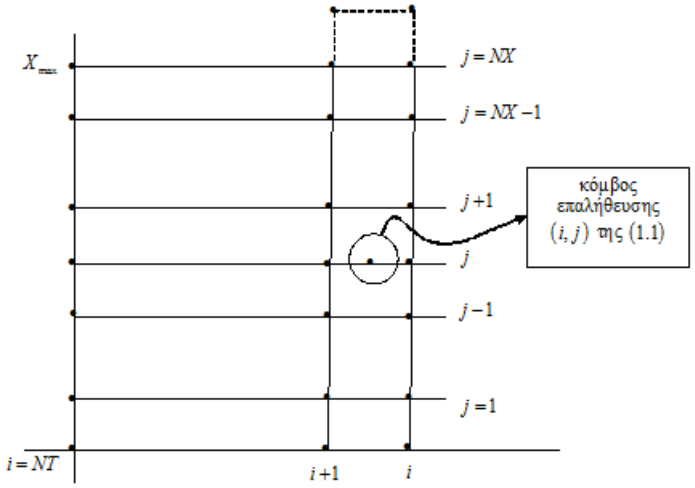
Με τον ίδιο τρόπο, αν γνωρίζουμε τις τιμές $S_{i,j}, j\epsilon[0, NX]$ για κάποια χρονική στιγμή $i = 1, 2, \ \ldots\ , NT-1$, μπορούμε να υπολογίσουμε τις τιμές $S_{i+1,j}, j\epsilon[0, NX]$, δηλαδή τις τιμές την αμέσως προηγούμενη χρονική στιγμή $t = T - (i + 1)\delta t$.

Στην έμμεση μεθοδολογία των πεπερασμένων διαφορών *Crank-Nicholson*, επαληθεύουμε την εξίσωση (1.2) σε κόμβους του χωρίου στο οποίο ορίσαμε το πρόβλημα, με προσεγγίσεις της συνάρτησης $S(\cdot)$ καθώς και των μερικών παραγώγων αυτής εκφραζόμενες ως διαφορές των τιμών της σε γειτονικούς κόμβους. Οι κόμβοι στους οποίους προσεγγίζουμε την συνάρτηση **δεν** είναι οι κόμβοι του πλέγματος που ορίζουν οι εξισώσεις (1.5), (1.6). Είναι όλα τα μέσα γειτονικών κόμβων που αντιστοιχούν στον ίδιο χωρικό κόμβο, εκτός από αυτά που αντιστοιχούν στο χωρικό κόμβο $j = 0$, δηλαδή τα σημεία

$$\left(\frac{t_i + t_{i+1}}{2}, x_j\right), \ i\epsilon[0, NT-1], \ j\epsilon[1, NX] \tag{1.9}$$

Ως **κόμβος επαλήθευσης** (i, j) (Σχήμα 17) ορίζουμε το σημείο $\left(\frac{t_i+t_{i+1}}{2}, x_j\right)$

και χρησιμοποιούμε το συμβολισμό $\tilde{S}_{i,j}$, $\frac{\partial \tilde{S}_{i,j}}{\partial x}$ κοκ για να εκφράσουμε τις προσεγγιστικές τιμές της συνάρτησης $S(\cdot)$ καθώς και των παραγώγων της στον εν λόγω κόμβο.



Σχήμα 17: Κόμβος επαλήθευσης (i, j) της 1.2

Οι προσεγγίσεις της συνάρτησης $S(\cdot)$ καθώς και των μερικών παραγώγων της στον κόμβο επαλήθευσης (i, j) δίνονται από τις ακόλουθες σχέσεις:

$$\tilde{S}_{i,j} := \frac{S_{i,j} + S_{i+1,j}}{2} , \tag{1.9a}$$

$$\frac{\partial \tilde{S_{i,j}}}{\partial t} := -\frac{S_{i+1,j} - S_{i,j}}{\delta t} , \tag{1.9b}$$

$$\frac{\partial \tilde{S}_{i,j}}{\partial x} := \frac{S_{i+1,j+1} - S_{i+1,j-1} + S_{i,j+1} - S_{i,j-1}}{4\delta x} , \tag{1.9c}$$

$$\frac{\partial \tilde{S}^2_{i,j}}{\partial x^2} := \frac{S_{i+1,j+1} - 2S_{i+1,j} + S_{i+1,j-1} + S_{i,j+1} - 2S_{i,j} + S_{i,j-1}}{2\delta x^2} . \tag{1.9d}$$

Υποθέτοντας οτι οι εκφράσεις της $S(\cdot)$ και των μερικών παραγώγων της όπως δίνονται από τις εξισώσεις (1.10) επαληθεύουν την (1.2) στο κόμβο επαλήθευσης

(i, j), εξάγουμε ότι:

$$r\tilde{S}_{i,j} = -\frac{\partial \tilde{S}_{i,j}}{\partial t} + rx\frac{\partial \tilde{S}_{i,j}}{\partial t} + \frac{1}{2}\sigma^2 x^2 \frac{\partial \tilde{S}_{i,j}}{\partial x^2}\ , \tag{1.11}$$

ή ισοδύναμα

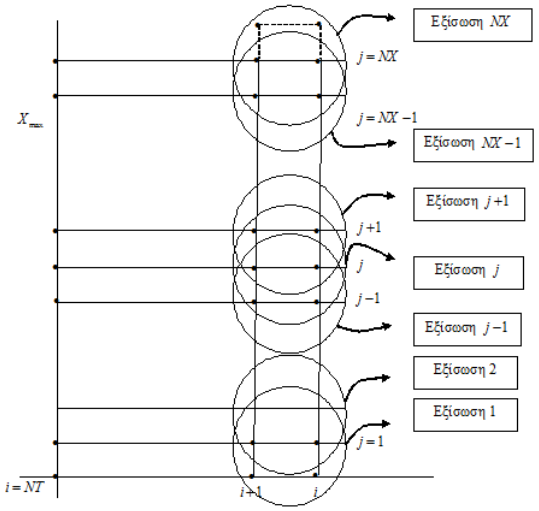
$$\begin{aligned} r \cdot \frac{S_{i,j} + S_{i+1,j}}{2} = &- \frac{S_{i+1,j} - S_{i,j}}{\delta t} \\ &+ r \cdot x \cdot \frac{S_{i+1,j+1} - S_{i+1,j-1} + S_{i,j+1} - S_{i,j-1}}{4\delta x} \\ &+ \frac{1}{2} \cdot \sigma^2 \cdot j^2 \cdot x^2 \cdot \frac{S_{i+1,j+1} - 2S_{i+1,j} + S_{i+1,j-1}}{2\delta x^2} \\ &+ \frac{1}{2} \cdot \sigma^2 \cdot j^2 \cdot x^2 \cdot \frac{S_{i,j+1} - 2S_{i,j} + S_{i,j-1}}{2\delta x^2} \end{aligned} \tag{1.11}$$

Λαμβάνοντας υπόψιν την εξίσωση (1.6), δηλαδή $x = j \times \delta x$, η εξίσωση (1.11) γράφεται:

$$\begin{aligned} r \cdot \frac{S_{i,j} + S_{i+1,j}}{2} = &- \frac{S_{i+1,j} - S_{i,j}}{\delta t} \\ &+ r \cdot j \cdot \cancel{\delta x} \frac{S_{i+1,j+1} - S_{i+1,j-1} + S_{i,j+1} - S_{i,j-1}}{4\cancel{\delta x}} \\ &+ \frac{1}{2} \cdot \sigma^2 \cdot j^2 \cdot j^2 \cdot \cancel{\delta x^2} \cdot \frac{S_{i+1,j+1} - 2S_{i+1,j} + S_{i+1,j-1}}{2\cancel{\delta x^2}} \\ &+ \frac{1}{2} \cdot \sigma^2 \cdot j^2 \cdot j^2 \cdot \cancel{\delta x^2} \cdot \frac{S_{i,j+1} - 2S_{i,j} + S_{i,j-1}}{2\cancel{\delta x^2}} \end{aligned} \tag{1.12}$$

Στον πίνακα που ακολουθεί φαίνονται οι εξισώσεις που επιλύονται σε κάθε χρονικό βήμα.

Από την εξίσωση (1.12) φαίνεται ότι η αρχική μας εξίσωση έχει πλέον μετατραπεί σε ένα σύστημα γραμμικών εξισώσεων, με αγνώστους τις τιμές της $S(\cdot)$ τα



Σχήμα 18: Κόμβοι επίλυσης ανά χρονικό βήμα

χρονικά διαστήματα $(i+1)$. Γράφοντας, την (1.12) σε μια πιο συμπαγή μορφή, έχουμε:

$$a_j \cdot S_{i+1,j-1} + b_j \cdot S_{i+1,j} + c_j \cdot S_{i+1,j+1} = e_{i,j} \tag{1.12}$$

με

$$e_{i,j} = d_{0,j} \cdot S_{i,j-1} + d_{1,j} \cdot S_{i,j} + d_{2,j} \cdot S_{i,j+1} \tag{1.13}$$

όπου

- $a_j = \frac{1}{4}\left(rj - \sigma^2 j^2\right)$
- $b_j = \frac{1}{\delta t} + \frac{1}{2}\left(\sigma^2 j^2 + r\right)$
- $c_j = -\frac{1}{4}\left(rj + \sigma^2 j^2\right)$
- $d_{0,j} = -a_j$

- $d_{1,j} = -\frac{1}{\delta t} - \frac{1}{2}\left(\sigma^2 j^2 + r\right)$

- $d_{2,j} = -c_j$

Σε κάθε χρονικό βήμα, λύνεται το σύστημα των γραμμικών εξισώσεων προκειμένου να βρεθεί η τιμή του $S(\cdot)$. Η κεντρική διαγώνιος του συστήματος παραμένει η ίδια ενώ το δεξί μέρος του συστήματος ($e_{i,j}$) υπολογίζεται κάθε φορά από τις τιμές του προηγούμενου χρονικού βήματος.

Το τρισδιαγώνιο σύστημα που προκύπτει είναι:

$$\begin{pmatrix} b_0 & c_0 & 0 & \cdots & 0 \\ a_1 & b_1 & c_1 & \cdots & 0 \\ \vdots & \ddots & \ddots & \ddots & \vdots \\ 0 & \cdots & a_{N-2} & b_{N-2} & C_{N-2} \\ 0 & \cdots & 0 & a_{N-1} & b_{N-1} \end{pmatrix} \times \begin{pmatrix} S_0 \\ S_1 \\ \vdots \\ S_{N-2} \\ S_{N-1} \end{pmatrix} = \begin{pmatrix} e_0 \\ e_1 \\ \vdots \\ e_{N-2} \\ e_{N-1} \end{pmatrix} \tag{1.14}$$

4.3.1 LU - Decomposition

Ένας τρόπος επίλυσης του τρισδιαγώνιου συστήματος είναι να χρησιμοποιηθεί η μέθοδος *LU-decomposition*. Πρόκειται για ένα απλό και γρήγορο αλγόριθμο, όπου ο χρόνος επίλυσης είναι γραμμικός προς το μέγεθος του χωρίου επίλυσης.

Ο αλγόριθμος έχει τρία μέρη, τον υπολογισμό των άνω και κάτω δισδιαγώνιων πινάκων L και U όπου $L \cdot U = A$, μια εμπρόσθια φάση κατά την οποία σαρώνει και λύνει το κάτω δισδιαγώνιο σύστημα ώστε $L \cdot z = y$ και μια φάση που σαρώνει και λύνει με φορά προς τα πίσω το άνω διαγώνιο σύστημα για να λύσει το σύστημα $U \cdot x = z$. Στο τέλος, η λύση ικανοποιεί την συνθήκη $A \cdot x = LU \cdot x = L \cdot z = y$. [34]

Την συγκεκριμένη μέθοδο την έχουμε χρησιμοποιήσει ως σημείο αναφοράς ως προς το χρόνο ολοκλήρωσης της στη Κεντρική Μονάδα Επεξεργασίας.

4.3.2 Odd-Even Reduction

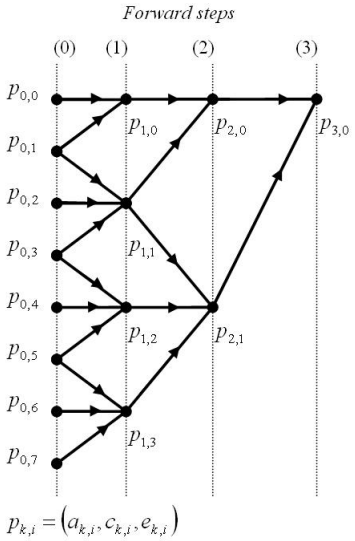
Πρόκειται για μια επαναληπτική μέθοδο επίλυσης του τρισδιαγώνιου συστήματος, την οποία και χρησιμοποιήσαμε στην υλοποίηση μας για την αρχιτεκτονική CUDA. Παρόλο που η συγκεκριμένη μέθοδος απαιτεί περισσότερους υπολογισμούς για την επίλυση του συστήματος από τη μέθοδο LU, μπορεί να παραλληλοποιηθεί και περιμένουμε τα αποτελέσματα μας να είναι αρκετά καλύτερα σε σχέση με τη μέθοδο LU.

Η μέθοδος αυτή χωρίζεται σε δύο φάσεις, την προς τα εμπρός φάση *(forward-phase)* και την προς τα πίσω φάση*(backward-phase)*. Στην πρώτη φάση, σε κάθε επανάληψη διαγράφονται από το σύστημα οι άγνωστοι που βρίσκονται στις ζυγές στήλες με αποτέλεσμα να μένει ένα σύστημα μισό από το αρχικό μόνο με τους μονούς αγνώστους. Η διαδικασία αυτή συνεχίζεται μέχρι να απομείνει ένα μοναδιαίο σύστημα με ένα άγνωστο.

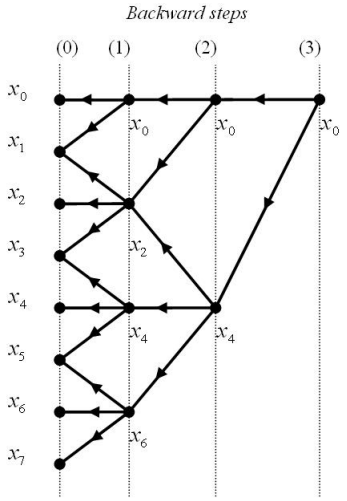
Στην προς τα πίσω φάση υπολογίζονται οι ζυγοί άγνωστοι που είχαν εξαλειφθεί αρχίζοντας από ένα μοναδιαίο σύστημα και καταλήγοντας σε ένα σύστημα ισοδύναμο με το αρχικό, στο οποίο έχουν υπολογιστεί πλέον όλοι οι άγνωστοι. [35]

Στην υλοποίηση που ακολουθήσαμε, κανονικοποιήσαμε την κύρια διαγώνιο ώστε να αποφευχθεί η υπερχείλιση που προκύπτει από τους πολλούς πολλαπλασιασμούς που απαιτούνται προκειμένου να υπολογιστεί το καινούργιο σύστημα που προκύπτει σε κάθε βήμα της μεθόδου.

Στα σχήματα που ακολουθούν φαίνονται τα δύο στάδια της μεθόδου odd-even.



Σχήμα 19: Forward-phase



Σχήμα 20: Backward-phase

Βασική προϋπόθεση για τη συγκεκριμένη μέθοδο είναι οτι το χωρίο που επιλέγουμε πρέπει να είναι δύναμη του δύο.

Στο Κεφάλαιο που ακολουθεί περιγράφεται η υλοποίηση που ακολουθήθηκε καθώς και τα διάφορα στάδια βελτιστοποιήσης.

5 Υλοποίηση

Κύριως σκοπός του συγκεκριμένου Κεφαλαίου είναι να περιγράψει και να επεξηγήσει στον αναγνώστη την υλοποίηση που ακολουθήσαμε σε όλα τα στάδια εξέλιξης της συγκεκριμένης εργασίας.

Περιγράφεται η υλοποίηση της εφαρμογής μας τόσο για την x86 Intel αρχιτεκτονική, όσο και για την αρχιτεκτονική CUDA. Αναφέρονται τα διάφορα στάδια βελτιστοποίησης καθώς και ο τρόπος που χρησιμοποιήθηκαν και διαμοιράστηκαν τα δεδομένα προκειμένου να επιτύχουμε την μέγιστη απόδοση.

Για την εξέλιξη της εφαρμογής εργαστήκαμε σε δύο πλατφόρμες υλικού, προκειμένου αφενός να έχουμε μια ευρύτερη άποψη για τις μετρήσεις μας και αφετέρου να δούμε πως ο ίδιος πηγαίος κώδικας με τις ίδιες βελτιστοποιήσεις, αποδίδει σε διαφορετικές γενιές καρτών γραφικών.

5.1 Υλικό

Συγκεκριμένα, ως βασικός σταθμός εργασίας επιλέχθηκε ένα σύστημα με τη κάρτα γραφικών *Geforce 9600GT*, επεξεργαστική μονάδα *Intel Core 2 Duo (μοντέλο P8600)* χρονισμένη στα 2.40GHz με 3072KB κρυφής μνήμης (cache) και μέγεθος μνήμης 3093588KB τεχνολογίας DDR2.

Πέρα, του συστήματος που περιγράψαμε χρησιμοποιήσαμε και έναν υπολογιστή με επεξεργαστή *AMD Athlon x2 4400+* με μέγεθος κρυφής μνήμης 1024KB και χωρητικότητα 1026048KB στο υποσύστημα μνήμης τεχνολογία DDR1 [4] [5]. Ως κάρτα γραφικών χρησιμοποιήθηκε η *Geforce 260 GTX*.

[4] Επειδή η τεχνολογία του υποσυστήματος μνήμης είναι πιο παλιά από την σημερινή, υπερχρονίστηκε ελαφρά μαζί με τον επεξεργαστή προκειμένου να αυξηθεί το εύρος και να μειωθεί ο χρόνος απόκρισης.

[5] Υποσύστημα μνήμης: 1024MB@440MHz, 1T, 2 CAS Latency.

Και για τις δύο κάρτες γραφικών χρησιμοποιήθηκαν οι ίδιες εκδόσεις οδηγού (driver), CUDA και μεταγλωττιστή καθώς και η ίδια διανομή Λειτουργικού Συστήματος.

Οι προδιαγραφές των καρτών γραφικών συνοψίζονται στους παρακάτω πίνακες:

	Geforce 9600GT	*Geforce 260 GTX*
Compute Capability	1.1	1.3
Global Memory	512MB	897MB
Shared Memory	16384 bytes	16384 bytes
Multiprocessors	4	27
Cores	32	216
Clock rate	1.25GHz	1.24GHz
Maximum Threads per Block	512	512

Πίνακας 2: Οι προδιαγραφές των GPU

Η κάρτα γραφικών με το ολοκληρωμένο κύκλωμα *Geforce 260 GTX* αποτελεί την πιο εξελιγμένη GPU που είχαμε στη διάθεση μας προκειμένου να ολοκληρώσουμε την εργασία μας.

5.2 Εφαρμογή

Όπως έχουμε αναφέρει και στην υποενότητα 4.3, για την αριθμητική επίλυση του προβλήματος μας, χρησιμοποιούμε τις μεθόδους *Odd-Even Reduction* και *LU-Decomposition*.

Η μέθοδος *LU-Decomposition* είναι μια σειριακή μέθοδος και δεν παραλληλοποιείται.

Η μέθοδος *Odd-Even Reduction* αποτελείται από τρία βασικά βήματα:

1. Τον υπολογισμού του δεξιού μέρους $e_{i,j}$,

συνάρτηση rhs(right_hand_side),

2. Την προς τα εμπρός φάση (forward phase), όπου το αρχικό σύστημα διαιρείται στο μισό μέχρι να φτάσουμε σε ένα μοναδιαίο,
   συνάρτηση forward_non_recursive_plain,

3. Την προς τα πίσω φάση (backward phase), όπου από το μοναδιαίο σύστημα φτάνουμε στο αρχικό υπολογίζοντας του όρους που είχαμε εξαλείψει κατά την μείωση του αρχικού συστήματος,
   συνάρτση backward_non_recursive_plain.

Τα τρία αυτά σημεία είναι που επιλέξαμε να παραλληλοποιήσουμε και να εκτελέσουμε στη κάρτα γραφικών. Η επιλογή έγινε μετά από profiling μεσώ του Intel VTune για το σύστημα με τον Intel επεξεργαστή και μέσω GProf της GNU για το σύστημα με τον AMD επεξεργαστή [36] [38].

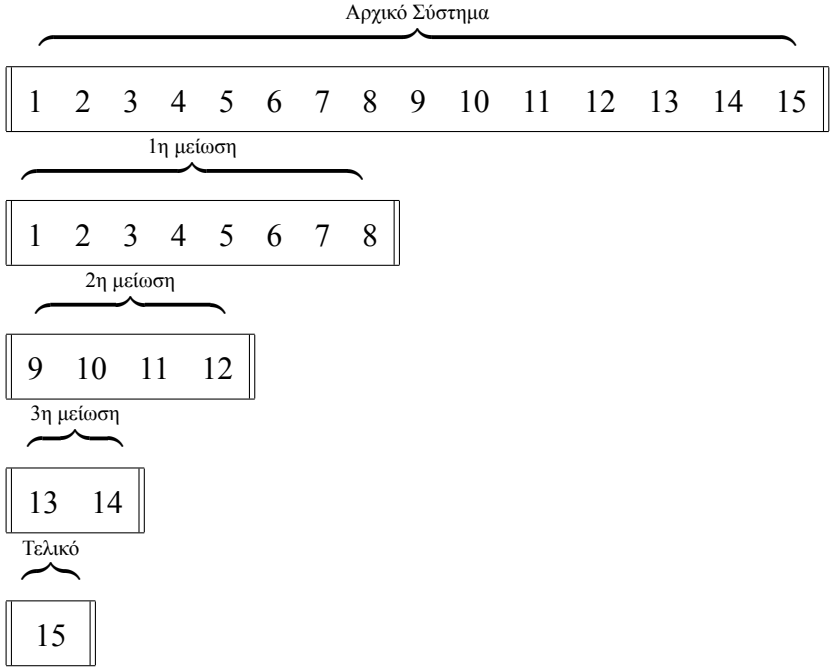
Το υπόλοιπο μέρος της εφαρμογής αποτελείται από συναρτήσεις αρχικοποίησης και ανάθεσης τιμών για τη μέθοδο *Crank-Nicholson*, δέσμευση και διαγραφή μνήμης τόσο για τη κάρτα γραφικών όσο και για το Host, χειρισμό λαθών καθώς και έλεγχο των kernels.

5.2.1 Αποθήκευση των δεδομένων στη μνήμη

Το σύστημα εξισώσεων που προκύπτει κατά τη μείωση του αρχικού συστήματος στο μισό υπολογίζεται από τις τιμές του δεύτερου. Η διαδικασία αυτή συνεχίζεται μέχρι να φτάσουμε σε ένα μοναδιαίο σύστημα. Εάν για παράδειγμα, έχουμε ένα χωρίο διαστάσεων 16×32 $(NX = 16,\ NT = 32)$, τότε η μείωση του συστήματος γίνεται ώς εξής:

Από το αρχικό σύστημα, τα δεκαέξι (16) στοιχεία στην συγκεκριμένη περίπτωση προκύπτει ένα σύστημα 8×32 από το οποίο προκύπτει στη συνέχεια ένα 4×32 κ.ο.κ.

Η αποθήκευση των δεδομένων στη μνήμη για τα διαδοχικά στάδια της μείωσης γίνεται σειριακά. Δηλαδή, για το χωρίο του παραδείγματος μας, τα δεδομένα από τη πρώτη μείωση καταλαμβάνουν τις οκτώ (8) πρώτες θέσεις, από τη δεύτερη τις επόμενες τέσσερις κ.ο.κ μέχρι να φτάσουμε στο τελικό μοναδιαίο σύστημα εξισώσεων. Η διαδικασία αυτή οπτικοποιείται στο σχήμα που ακολουθεί.



5.3 Profiling

Όπως αναφέρθηκε προηγουμένως, πριν τη μεταφορά μέρους της εφαρμογής στη κάρτα γραφικών, μετρήσαμε την απόδοση αυτής για την x86 αρχιτεκτονική στα δύο συστήματα που χρησιμοποιούμε. Μέσω των εφαρμογών *Intel VTune Amplifier για Linux* και *GNU Profiler* καταγράψαμε σε ποιες συναρτήσεις η εφαρμογή μας καταναλώνει τον περισσότερο χρόνο. Στους πίνακες που ακολουθούν

φαίνονται οι μετρήσεις από τα δύο συστήματα για δύο διαφορετικά χωρία, μεγέθους 8192×16384 και 16384×32768. Στους πίνακες αυτούς αποτυπώνεται ο συνολικός χρόνος εκτέλεσης της εφαρμογής καθώς και κάθε συνάρτησης ξεχωριστά καθώς και το ποσοστό επί τοις εκατό που εκτελείται η κάθε μια στον επεξεργαστή.

Επίσης, έχουμε συμπεριλάβει και τη μέθοδο *LU-Decomposition* την οποία έχουμε ως σημείο αναφοράς. Δεν θα ασχοληθούμε άλλο με την συγκεκριμένη μέθοδο στη συνέχεια της εργασίας μας.

	Intel Core 2 Duo		*AMD Athlon x2*	
Process	*Time %*	*Time(sec)*	*Time*	*Time(sec)*
Forward_phase	65.3%	5.749s	55.99%	9.02s
Backward_phase	17.9%	1.654s	20.79%	3.35s
Rhs	16.3%	1.451s	23.22%	3.74s
Total	99.5%	8.858s	100%	16.11s
LU-Decomposition	99.6%	127.714s	100%	8.90s

Πίνακας 3: Χωρίο μεγέθους 8192×16384

	Intel Core 2 Duo		*AMD Athlon x2*	
Process	*Time %*	*Time(sec)*	*Time*	*Time(sec)*
Forward_phase	65.0%	22.060s	55.28%	34.95s
Backward_phase	17.9%	6.341s	21.12%	13.35s
Rhs	16.7%	6.159s	23.60%	14.92s
Total	99.6%	34.547s	100%	63.22s
LU-Decomposition	99.6%	153.823s	100%	36.25s

Πίνακας 4: Χωρίο μεγέθους 16384×32768

Από τα παραπάνω προκύπτει οτι όλο τον υπολογιστικό χρόνο η εφαρμογή τον εξαντλεί σε αυτές τις τρείς συναρτήσεις και ιδίως στην προς τα εμπρός φάση

της μεθόδου Odd-Even. Αυτές οι τρείς συναρτήσεις είναι που θα παραλληλοποιήσουμε στο CUDA.

5.4 CUDA Kernels

Όπως έχει αναφερθεί και στο Κεφάλαιο 3, τα νήματα CUDA έχουν μια μοναδική διευθυνσιοδότηση το καθένα, και δρομολογούνται για εκτέλεση ομαδικά σε blocks. Έτσι, στην περίπτωση που έχουμε ένα χωρίο μεγέθους 8192×16384, θα έχουμε νήματα αριθμημένα από το 0 εώς το 8191 . Η αριθμοδότηση γίνεται ως εξής:

Αρχικά επιλέγουμε τον αριθμό των νημάτων που θέλουμε να έχουμε ανά block. Υποστηρίζονται στο CUDA μέχρι 768 νήματα. Έπειτα από δοκιμές και μετρήσεις καταλήξαμε ότι ο βέλτιστος αριθμός για τη δική μας εφαρμογή είναι 128 και 256 νήματα. Επομένως για ένα χωρίο διάστασης 8192 στο άξονα-x και για 128 νήματα έχουμε $8192/128 = 64$ blocks.

Όπως έχουμε αναφέρει κάθε νήμα του Grid παίρνει το διαχωριστικό του (Thread ID) από την εξίσωση (1.1). Επομένως, μπορούμε να φανταστούμε ότι έχουμε ένα πίνακα νημάτων, όπως φαίνεται στο παρακάτω σχήμα.

0	1	2	...	...	...	8190	8191

Πίνακας 5: Array of Threads

Για την μεταφορά της εφαρμογής στη κάρτα γραφικών χρησιμοποιήσαμε τέσσερις kernels. Ένα kernel που υπολογίζει το δεξιό μέρος της εξίσωσης, ένα για τη προς τα πίσω φάση επίλυσης του συστήματος και δύο kernels για την προς τα εμπρός φάση, έναν που υπολογίζει το σύστημα που προκύπτει από την πρώτη

μείωση του αρχικού συστήματος στο μισό, δηλαδή σε ένα σύστημα 4095×16384 στην περίπτωση του παραδείγματος μας, και έναν που υπολογίζει τις υπόλοιπες μειώσεις μέχρι να φτάσουμε στο τελικό μοναδιαίο σύστημα.

Ο λόγος που έγινε αυτή η επιλογή είναι γιατί όπως έχουμε αναφέρει κάθε καινούργιο μειωμένο σύστημα προκύπτει από τις τιμές του προηγούμενου. Κάθε CUDA block εκτελείται σε ένα Streaming Multiprocessor (SM) και τα blocks δρομολογούνται για εκτέλεση στο πρώτο διαθέσιμο SM. Επομένως, μπορούμε να πούμε οτι τα νήματα CUDA εκτελούνται παράλληλα με αποτέλεσμα τα νήματα να επιχειρούν να διαβάσουν θέσεις μνήμης για να υπολογίσουν το σύστημα στην μείωση έστω X, από τις διευθύνσεις μνήμης που θα έπρεπε να είχε γράψει η μείωση (X-1) προτού αυτό γίνει.

Στην περίπτωση του παραδείγματος μας για τη δεύτερη μείωση του συστήματος σε 2048 έχουμε:

1	2	3	4	···	···	8190	8191

Πίνακας 6: Αρχικό Σύστημα

Διευθύνσεις μνήμης

1	2	···	···	4095
1,2	3,4	···	···	8190,8191

Πίνακας 7: 1η μείωση

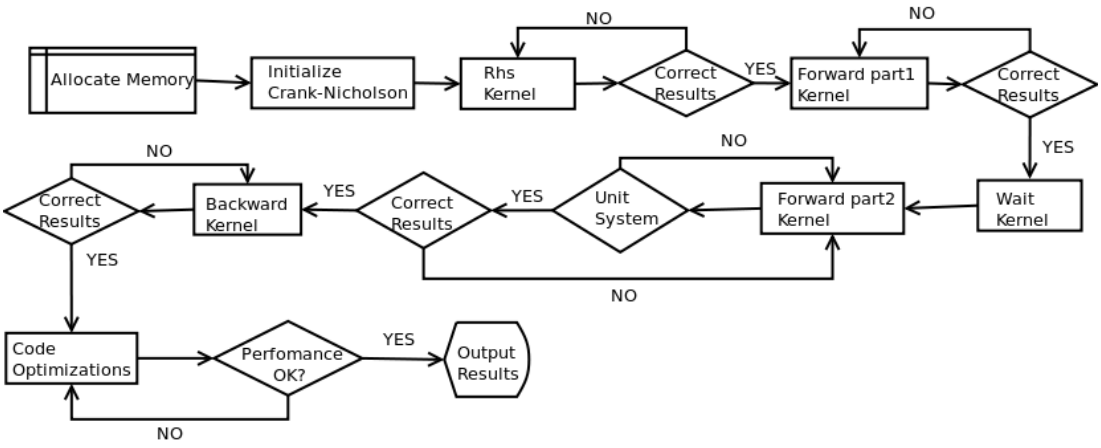
Στους πίνακες φαίνεται ποιες διευθύνσεις μνήμης διαβάζονται ώστε να προκύψει κάθε φορά το καινούργιο μειωμένο σύστημα. Στο σημείο αυτό να υπενθυμίσουμε ότι η αποθήκευση όλων των συστημάτων που προκύπτουν γίνεται σε ένα ενιαίο πίνακα και όχι σε πολλαπλούς. Η αποθήκευση γίνεται διαδοχικά, δηλαδή

Διευθύνσεις μνήμης			
1	···	···	2047
1,2	···	···	4094,4095

Πίνακας 8: 2η μείωση

το πρώτο σύστημα αποθηκεύεται στις πρώτες θέσεις, το δεύτερο στις επόμενες κοκ.

Στη συνέχεια ακολουθεί ένα διάγραμμα ροής στο οποίο απεικονίζονται όλα τα στάδια κατά τη διάρκεια ανάπτυξης της εφαρμογής μας.



Σχήμα 21: Διάγραμμα ροής

Στο σημείο αυτό να σημειώσουμε οτι τα αποτελέσματα που ακολουθούν στη συνέχεια του κειμένου σχετικά με την απόδοση της εφαρμογής μας, πάρθηκαν από το profiler της NVIDIA και είναι ο μέσος όρος που προκύπτει έπειτα από πέντε (5) μετρήσεις.

Στη συνέχεια ακολουθεί μια περιγραφή του κάθε kernel του συστήματος μας καθώς και η απόδοση του χωρίς καμία βελτιστοποίηση.

5.4.1 Right Hand Side (rhs) Kernel

Η προσέγγιση που ακολουθήθηκε στο συγκεκριμένο kernel είναι πολύ άμεση και απλή. Το κάθε διαχωριστικό των νημάτων (Thread ID) προκύπτει από τον τύπο 1.1, έχοντας ουσιαστικά τόσα νήματα όσο είναι και το μέγεθος του χωρίου. Το κάθε νήμα υπόλογίζει στη συνέχεια το δεξιό μέρος του συστήματος στη θέση που αντιστοιχεί στο διαχωριστικό του με βάση τον τύπο που προβλέπει η εξίσωση 1.13.

Στη συνέχεια ακολουθούν οι πίνακες στους οποίους παρουσιάζεται η απόδοση της συγκεκριμένης υλοποίησης στις κάρτες γραφικών που χρησιμοποιήσαμε.

	Geforce 9600GT		*Geforce 260 GTX*	
rhs kernel	*128 Threads*	*256 Threads*	*128 Threads*	*256 Threads*
GPU Time %	11.17%	11.14%	9.78%	8.98%
GPU Time (sec)	1.258	1.264s	0.125s	0.126s
Occupancy	100%	100%	100%	100%
CPU Time	1.451s	1.451s	3,742s	3,742s
Speedup	$\sim 1.15\times$	$\sim 1.15\times$	$\sim 29.94\times$	$\sim 29.70\times$

Πίνακας 9: Χωρίο μεγέθους 8192×16384

5.4.2 Forward Phase part 1 Kernel

Ο kernel αυτός είναι υπεύθυνος για την πρώτη μείωση του αρχικού συστήματος στο μισό. Στην έκδοση χωρίς καμία βελτιστοποίηση, τα νήματα του kernel χωρίζονται σε αυτά με μονά διαχωριστικά και σε αυτά με ζυγά. Στη συνέχεια τα μισά πρώτα νήματα, δηλαδή αυτά από 0 εώς 4095 τοποθετούν στις αντίστοιχες θέσεις του πίνακα τις εξισώσεις που αντιστοιχούν σε αυτά τα διακριτά σημεία του πλέγματος από τα στοιχεία που προκύπτουν από το αρχικό σύστημα, δηλαδή από

το 0 εώς το 8191.

Για να έχουμε ένα πιο ολοκληρωμένο μέτρο σύγκρισης για την εκτέλεση του προς τα εμπρός βήματος στην κάρτα γραφικών, θα παρουσιάσουμε την απόδοση του συγκεκριμένου kernel αθροιστικά μαζί με αυτή του επόμενου, καθώς έχουμε χωρίσει το προς τα εμπρός βήμα σε δύο kernels.

5.4.3 Forward Phase part 2 Kernel

Η λογική της συγκεκριμένης υλοποίησης είναι παρόμοια με τον προηγούμενο kernel (forward phase part1). Η μεγάλη διαφορά είναι οτι μετά από κάθε μείωση το σύστημα μειώνεται στο μισό και κατα συνέπεια και τα νήματα που εκτελούνται ταυτόχρονα. Για παράδειγμα στην τρίτη μείωση που το σύστημα είναι 2048×16384, τα ενεργά νήματα είναι αντίστοιχα 2048.

Στη συνέχεια παρουσιάζουμε αθροιστικά την απόδοση των δύο kernels (part1 και part2) και τα συγκρίνουμε με το χρόνο εκτέλεσης στους Κεντρικούς Επεξεργαστές των συστημάτων μας.

	Geforce 9600GT		*Geforce 260 GTX*	
Forward Phase Part1 & Part2	*128 Threads*	*256 Threads*	*128 Threads*	*256 Threads*
GPU Time %	24.3%, 25.4%	24.2%, 25.7%	14.6%, 45.7%	13.3%, 47.7%
GPU Time (sec)	5.600	5.664s	0.772s	0.850s
Occupancy	100% , 66.7%	100%, 66.7%	100%	100%
CPU Time	5.749s	5.749s	9.020s	9.020s
Speedup	$\sim 1.03\times$	$\sim 1.03\times$	$\sim 11.68\times$	$\sim 10.60\times$

Πίνακας 10: Χωρίο μεγέθους 8192×16384

5.4.4 Backward Phase Kernel

Ο συγκεκριμένος kernel, έχει σαν είσοδο τον αριθμό των συνολικών μειώσεων που έχουν γίνει από τους προηγούμενους, και υπολογίζει τις λύσεις, σε κάθε χρονικό βήμα, των αγνώστων που είχαν εξαλειφθεί από το προς τα εμπρός βήμα εως ότου φτάσει σε ένα σύστημα μεγέθους ίσο με το αρχικό, όπου όλες οι εξισώσεις είναι πλέον λυμένες.

Αξίζει να σημειώσουμε ότι λόγω του τρόπου λειτουργίας του συγκεκριμένου αλγορίθμου, η μεταφορά της συγκεκριμένης μεθόδου στη κάρτα γραφικών δεν είναι η βέλτιστη καθώς προκειμένου να υπολογιστεί το αρχικό σύστημα τα νήματα διαβάζουν δεδομένα από διάσπαρτες θέσεις μνήμης από όλο το μήκος του πίνακα με αποτέλεσμα να μην μπορούμε να διαβάσουμε μαζικά ολόκληρα τμήματα μνήμης από διαδοχικά νήματα, χάνοντας έτσι σε απόδοση (no memory coalescing).

Παρόλα αυτά τα αποτελέσματα, αν και όχι βέλτιστα, είναι άκρως ικανοποιητικά, ειδικά στη κάρτα *Geforce 260 GTX* όπου έχει αλλάξει η αρχιτεκτονική και έχουν χαλαρώσει οι συνθήκες που απαιτούνται προκειμένου να επιτευχθεί memory coalescing.

Στον πίνακα που ακολουθεί συνοψίζονται οι επιδόσεις που επιτύχαμε στα δύο συστήματα μας:

5.5 Βελτιστοποιήσεις

Στη ενότητα αυτή παρουσιάζονται οι βελτιστοποιήσεις που έγιναν στο επίπεδο του πηγαίου κώδικα καθώς και το κέρδος σε απόδοση τόσο για τους επιμέρους kernels όσο και συνολικά για το χρόνο εκτέλεσης.

	Geforce 9600GT		*Geforce 260 GTX*	
Backward Phase kernel	*128 Threads*	*256 Threads*	*128 Threads*	*256 Threads*
GPU Time %	39,1%	38,9%	9.8%	, 9.0%
GPU Time (sec)	4,400s	4.416s	0.383s	0.429s
Occupancy	83,3%	66,7%	100%	100%
CPU Time	1.654s	1.654s	3.350s	3.350s
Speedup	$\sim -2.66\times$	$\sim -2.67\times$	$\sim 8.75\times$	$\sim 7.81\times$

Πίνακας 11: Χωρίο μεγέθους 8192×16384

5.5.1 Βελτιστοποιήσεις κατά την μεταγλώττιση

Η πρώτη βελτιστοποίηση που χρησιμοποιήσαμε δεν ήταν στον πηγαίο κώδικα αλλά κατά την μεταγλώττιση της εφαρμογής. Συγκεκριμένα κατά την μεταγλώττιση με τον nvcc χρησιμοποιήσαμε την εντολή *--use_fast_math*.

Παρόλο που η NVIDIA δεν δίνει ακριβώς το κέρδος σε απόδοση με τη συγκεκριμένη εντολή, ισχύει ότι με την συγκεκριμένη εντολή οι μαθηματικές πράξεις γίνονται στις Ειδικές Μονάδες (Special Functions Units - SFUs) της κάρτας γραφικών και οι πράξεις για να εκτελεστούν καταναλώνουν δεκάξι (16) με τριάντα δύο (32) κύκλους ανά warp, ενώ χωρίς τη συγκεκριμένη εντολή καταναλώνουν εκατοντάδες.

5.5.2 Shared Memory

Η πρώτη βελτιστοποίηση που εφαρμόσαμε στον πηγαίο κώδικα ήταν η χρήση κοινόχρηστης μνήμης. Τα αποτελέσματα ήταν ικανοποιητικά από το κέρδος σε απόδοση. Ακολουθεί μια περιγραφή για κάθε kernel ξεχωριστά καθώς και το κερδος σε απόδοση σε σύγκριση με το χρόνο εκτέλεσης του καθενός στο CPU.

5.5.3 Right Hand Side (rhs) Kernel

Χρησιμοποιήσαμε shared memory ίση σε μέγεθος με το πλήθος των νημάτων. Συγκεκριμένα, η μνήμη δουλεύει ως ένα είδος κρυφής μνήμης (cache) όπου νήματα που ανήκουν στο ίδιο block τοποθετούν διαδοχικά τα δεδομένα που διαβάζουν από τη μνήμη στη Shared Memory. Έπειτα, τα νήματα παίρνουν τα δεδομένα και υπολογίζουν το δεξιό μέλος της εξίσωσης.

Το κέρδος σε απόδοση έγκειται στην μεγάλη ταχύτητα της συγκεκριμένης μνήμης. Επειδή είναι πάνω στο ολοκληρωμένο κύκλωμα (on-chip) δεν υπάρχουν σχεδόν καθόλου χρονικές καθυστερήσεις για μεταφορά δεδομένων, ιδίως στην περίπτωση που εξαλείφονται τα bank conflicts, καθώς όλα τα νήματα του block εργάζονται παράλληλα.

Η απόδοση της συγκεκριμένης υλοποίησης φαίνεται στον παρακάτω πίνακα:

	Geforce 9600GT		*Geforce 260 GTX*	
rhs kernel	*128 Threads*	*256 Threads*	*128 Threads*	*256 Threads*
GPU Time %	7,9%	8,3%	14,0%	15.7%
GPU Time (sec)	0,567s	0,599s	0.133s	0.165s
Occupancy	66,7%	66,7%	50%	50%
CPU Time	1,451s	1,451s	3,742s	3,742s
Speedup	$\sim 2.56\times$	$\sim 2.42\times$	$\sim 28,13\times$	$\sim 22,68\times$

Πίνακας 12: Χωρίο μεγέθους 8192×16384

5.5.4 Forward Phase part 1 Kernel

Πάλι η λογική που χρησιμοποιήσαμε Shared Memory είναι όμοια με τον προηγούμενο kernel. Όπως και προηγουμένως τα αποτελέσματα από τη συγκεκριμένη βελτιστοποίηση θα παρουσιαστούν μαζί με τον επόμενο CUDA kernel ώστε να έχουμε ένα πιο στοχευμένο μέτρο σύγκρισης σε σχέση με την x86 υλοποίηση.

5.5.5 Forward Phase part 2 Kernel

Τα αποτελέσματα από τη χρήση Shared Memory από όλο το εμπρός βήμα είναι άκρως εντυπωσιακά. Με τη χρήση της συγκεκριμένης μνήμης καταφέραμε να επιτύχουμε για την προς τα εμπρός φάση του αλγορίθμου μείωση μεγαλύτερη του 50% στο χρόνο εκτέλεσης των δύο kernels. Ακολουθούν οι ακριβείς μετρήσεις.

	Geforce 9600GT		*Geforce 260 GTX*	
Forward Phase Part1 & Part2	*128 Threads*	*256 Threads*	*128 Threads*	*256 Threads*
GPU Time %	13.1%, 17.4%	13.3%, 17.5%	14.5%, 30.1%	13.1%, 30.6
GPU Time (sec)	2.186	2.219s	0.431s	0.460s
Occupancy	100% , 83.3%	100%, 66.7%	100%	100%
CPU Time	5.749s	5.749s	9.020s	9.020s
Speedup	$\sim 2.63\times$	$\sim 2.60\times$	$\sim 20.93\times$	$\sim 19.61\times$

Πίνακας 13: Χωρίο μεγέθους 8192×16384

5.5.6 Backward Phase Kernel

Λόγω του περιορισμού που εισάγει ο συγκεκριμένος αλγόριθμος και περιγράψαμε προηγουμένως δεν χρησιμοποιήσαμε Shared Memory για τον kernel αυτό. Οι προσπάθειες που κάναμε κατέληγαν σε αποτελέσματα χειρότερα από το αρχικό καθώς είχαμε πάρα πολλά bank conflicts που λόγω αλγοριθμικού σχεδιασμού δεν ήταν δυνατό να επιλυθούν.

5.5.7 Texture Memory

Στο στάδιο αυτό χρησιμοποιήσαμε μόνο Texture Memory για να βελτιώσουμε την απόδοση της εφαρμογής. Οι λόγοι που αποφασίσαμε να χρησιμοποιήσουμε Texture Memory ήταν πολλαπλοί.

Μπορούμε να φανταστούμε τη μνήμη Texture σαν μια διεπαφή υλικού μεταξύ του GPU και της κύριας μνήμης της κάρτας γραφικών, την οποία ο προγραμματιστής μπορεί να συνδέσει με διάφορα τμήματα της κύριας μνήμης.

Χρησιμοποιείται ως ένα είδος buffer για την κύρια μνήμη της κάρτας και δουλεύει με τον ίδιο τρόπο που δουλεύει και για τα γραφικά.

Δεν θέτει περιορισμούς στον τρόπο προσπέλασης και είναι ιδανική όταν υπάρχει μια σχετική τοπικότητα των δεδομένων (spatial locality), καθώς δεν άρει τους περιορισμούς για memory coalescing. Επομένως, γίνεται σαφές οτι είναι ιδανική για την περίπτωση τόσο της μείωσης του συστήματος, όσο και για τη φάση ανάκτησης και επίλυσης του αρχικού καθώς τα δεδομένα διαβάζονται από γειτονικές αλλά όχι διαδοχικές θέσεις μνήμης.

Επιπλέον, η διευθυνσιοδότηση της μνήμης γίνεται μέσω των ειδικών μονάδων της συγκεκριμένης μνήμης (Texture Units) αποφορτίζοντας έτσι ακόμη παραπάνω το GPU.

Στους πίνακες που ακολουθούν παρουσιάζονται οι χρονικές επίδοσεις του κάθε kernel ξεχωριστά για χωρίο μεγέθους 8192×16384.

	Geforce 9600GT		*Geforce 260 GTX*	
rhs kernel	*128 Threads*	*256 Threads*	*128 Threads*	*256 Threads*
GPU Time %	7,45%	7.63%	11,05%	11,78%
GPU Time (sec)	0.498s	0,521s	0.132s	0.157s
Occupancy	66,7%	66,7%	50%	50%
CPU Time	1,451s	1,451s	3,742s	3,742s
Speedup	$\sim 2.9\times$	$\sim 2.78\times$	$\sim 28.35\times$	$\sim 23,83\times$

Πίνακας 14: Rhs kernel

	Geforce 9600GT		*Geforce 260 GTX*	
Forward Phase Part1 & Part2	*128 Threads*	*256 Threads*	*128 Threads*	*256 Threads*
GPU Time %	9.2%, 17.5%	9.2%, 18.5%	4.8%, 42.1%	13.4%, 42.7%
GPU Time (sec)	1.789s	1.888s	0.560s	0.605s
Occupancy	100% , 83.3%	100%, 66.7%	100%	100%
CPU Time	5.749s	5.749s	9.020s	9.020s
Speedup	$\sim 3.21\times$	$\sim 3.04\times$	$\sim 16.11\times$	$\sim 14.91\times$

Πίνακας 15: Forward Phase Kernel

	Geforce 9600GT		*Geforce 260 GTX*	
Backward Phase kernel	*128 Threads*	*256 Threads*	*128 Threads*	*256 Threads*
GPU Time %	65.08%	64.7%	31.9%	32.1%
GPU Time (sec)	4,403s	4.419s	0.382s	0.427s
Occupancy	83,3%	66,7%	100%	100%
CPU Time	1.654s	1.654s	3.350s	3.350s
Speedup	$\sim -2.66\times$	$\sim -2.67\times$	$\sim 8.77\times$	$\sim 7.85\times$

Πίνακας 16: Backward Phase Kernel

Στην επόμενη ενότητα γίνεται επεξήγηση κάποιων αλγοριθμικών μικρο βελτιώσεων σε σχέση με τον τρόπο που τα νήματα προσπελάσουν την μνήμη και παρουσιάζεται η απόδοση του βέλτιστου πηγαίου κώδικα τόσο για κάθε kernel ξεχωριστά όσο και για το σύνολο της εφαρμογής.

5.6 Απόδοση συστήματος

Βασικός στόχος της συγκεκριμένης εργασίας, πέρα από την επίτευξή όσον το δυνατόν μεγαλύτερης απόδοσης είναι και η αξιοπιστία των αποτελεσμάτων 5.7. Στο πλαίσιο αυτό, προκειμένου να γίνει σωστή μεταφορά του αλγορίθμου από την

Intel x86 αρχιτεκτονική στη κάρτα γραφικών έπρεπε να ξεπεραστεί ένα σημαντικό εμπόδιο.

Όπως έχουμε εξηγήσει κάθε μειωμένο σύστημα προκύπτει από το προηγούμενο και είναι ακριβώς το μισό. Η δυσκολία που συναντήσαμε ήταν όταν το σύστημα ήταν ίσο με το μέγεθος ενός CUDA block, δηλαδή όταν προέκυπτε σύστημα μεγέθους 128 ή 256.

Όταν το σύστημα έχει το συγκεκριμένο μέγεθος, με βάση τον αλγόριθμο, τα νήματα του ίδιου block πρέπει να γράψουν στις διευθύνσεις μνήμης του πίνακα, τις εξισώσεις που προκύπτουν για το σύστημα αυτού του μεγέθους αλλά παράλληλα να διαβάσουν τις ίδιες διευθύνσεις για την δημιουργία του συστήματος που θα προκύψει από τη μείωση που ακολουθεί.

Όπως αρχίζει να γίνεται σαφές, το πρόβλημα προκύπτει όταν τα νήματα πρέπει να πάρουν δεδομένα από διευθύνσεις μνήμης που δεν έχουν γραφτεί ακόμη.

Η λύση που δόθηκε ήταν να μοιράσουμε το τελευταίο block σε δύο τμήματα. Το αρχικό μισό block υπολογίζει το σύστημα στη μείωση που βρίσκεται κάθε φορά, έστω X, ενώ το άνω μισό παραμένει ανενεργό στην πρώτη φάση και στη συνέχεια αφού συγχρονιστούν τα νήματα υπολογίζει το επόμενο σύστημα, έστω (X+1). Με τον τρόπο αύτο παρόλο που υπάρχει κάποιο μικρό κόστος σε απόδοση διασφαλίζεται η ακεραιότητα και ορθότητα των δεδομένων.

```
if(idx > N - blockDim.x){
  while( loop>=2 ){
    if( (idx > offset) && ( idx < offset + (loop>>1) ) )
        compute_system();
  }
}
/*Upper Half of the block*/
```

```
if(threadIdx.x >=blockDim.x/2)
  if( ( threadIdx.x > 0 ) && ( threadIdx.x < (blockDim.x-1) ) )
      compute_next_system();
```

Στη συνέχεια τις ενότητας παρατίθενται οι μετρήσεις των βέλτιστων kernels καθώς και ολόκληρης της εφαρμογής. Έπίσης, έχουμε συμπεριλάβει και την μέθοδο *LU-Decomposition* που εκτελείται στον επεξεργαστή ως σημείο αναφοράς.

Επειδή το ζητούμενο είναι η επίδειξη της απόδοσης ενός μαζικά παράλληλου συστήματος αλλά και του ετερόγενου μοντέλου προγραμματισμού, στους πίνακες έχουμε συμπεριλάβει την απόδοση του συστήματος τόσο μέχρι το σύστημα να γίνει μοναδιαίο αλλά και μέχρι να μειωθεί στο $128/256 \times 16384$. Ο λόγος που έγινε αυτό είναι γιατί σε τέτοιες τάξεις μεγέθους χωρίου δεν περιμένουμε μεγάλη διαφορά στην απόδοση καθώς τα ενεργά νήματα είναι πολύ λίγα.

Οι μετρήσεις έγιναν για χωρίο της τάξης 8192×16384 αλλά και 16384×32768 προκειμένου να δούμε την κλιμάκωση της απόδοσης σε σχέση με την Κεντρική Επεξεργαστική Μονάδα.

Αρχικά παρουσιάζουμε την απόδοση των επιμέρους βέλτιστων kernels για τα χωρία που προαναφέραμε.

	Geforce 9600GT		*Geforce 260 GTX*	
rhs kernel	*128 Threads*	*256 Threads*	*128 Threads*	*256 Threads*
GPU Time %	12.4%	12.5%	13.3%	12.5%
GPU Time (sec)	0.509s	0,513s	0.134s	0.135s
Occupancy	100%	100%	100%	100%
CPU Time	1,451s	1,451s	3,742s	3,742s
Speedup	$\sim 2.85\times$	$\sim 2.83\times$	$\sim 27.93\times$	$\sim 27.72\times$

Πίνακας 17: Optimal Rhs Kernel - Grid 8192×16384

	Geforce 9600GT		*Geforce 260 GTX*	
Forward Phase Part1 & Part2	*128 Threads*	*256 Threads*	*128 Threads*	*256 Threads*
GPU Time %	14.5%, 15.9%	14.5%, 15.8%	4.8%, 42.1%	13.4%, 42.7%
GPU Time (sec)	1.249s	1.244s	0.502s	0.5180s
Occupancy	66.7% , 66.7%	66.7% , 66.7%	50%, 66.7%	50%, 50%
CPU Time	5.749s	5.749s	9.020s	9.020s
Speedup	$\sim 4.06\times$	$\sim 4.62\times$	$\sim 17.97\times$	$\sim 17.41\times$

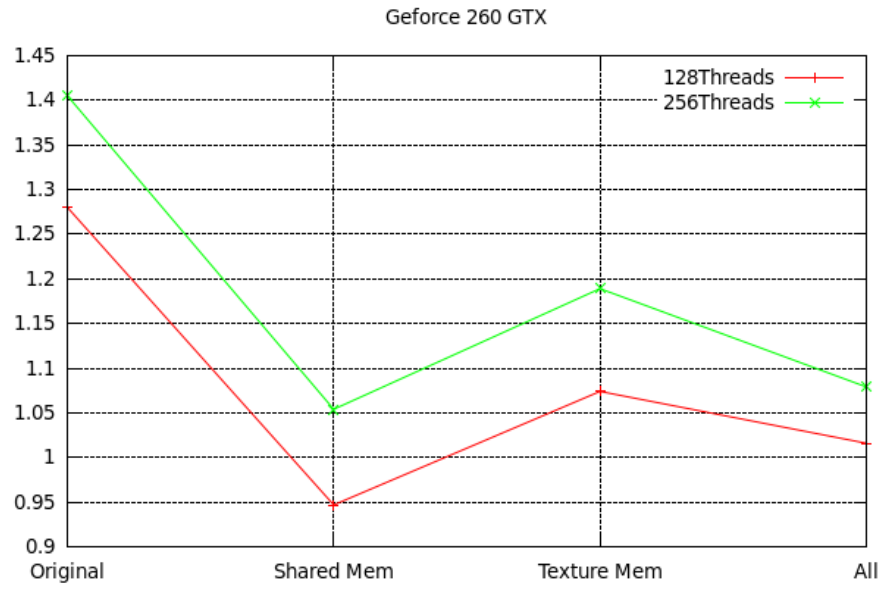
Πίνακας 18: Optimal Forward Phase Kernel - 8192×16384

	Geforce 9600GT		*Geforce 260 GTX*	
Backward Phase kernel	*128 Threads*	*256 Threads*	*128 Threads*	*256 Threads*
GPU Time %	57.2%	57.3%	37.5%	39.4%
GPU Time (sec)	2.349s	2.356s	0.382s	0.426s
Occupancy	83,3%	66,7%	100%	100%
CPU Time	1.654s	1.654s	3.350s	3.350s
Speedup	$\sim -1.42\times$	$\sim -1.42\times$	$\sim 8.77\times$	$\sim 7.86\times$

Πίνακας 19: Optimal Backward Phase Kernel - 8192×16384

Στις γραφικές παραστάσεις που ακολουθούν απεικονίζεται το κέρδος σε απόδοση για τα διάφορα στάδια.

Σχήμα 22: Κλιμάκωση απόδοσης στη κάρτα γραφικών *Geforce 9600 GT*



Σχήμα 23: Κλιμάκωση απόδοσης στη κάρτα γραφικών *Geforce 260 GTX*

Στη συνέχεια ακολουθούν οι μετρήσεις για χωρίο 16384×32768.

	Geforce 9600GT		*Geforce 260 GTX*	
rhs kernel	*128 Threads*	*256 Threads*	*128 Threads*	*256 Threads*
GPU Time %	11.2%	9,0%	13.3%	12.5%
GPU Time (sec)	1,615s	1,624s	0.347s	0.361s
Occupancy	100%	100%	100%	100%
CPU Time	6,159s	6,159s	14,920s	14,920s
Speedup	$\sim 3.81\times$	$\sim 3.79\times$	$\sim 42.30\times$	$\sim 41.33\times$

Πίνακας 20: Optimal Rhs Kernel - Grid 16384×32768

	Geforce 9600GT		*Geforce 260 GTX*	
Forward Phase Part1 & Part2	*128 Threads*	*256 Threads*	*128 Threads*	*256 Threads*
GPU Time %	12.4%, 14.2%	10.3%, 15.6%	40.8%, 42.1%	13.4%, 42.7%
GPU Time (sec)	3.837s	4.718s	1.218s	1.265s
Occupancy	66.7% , 66.7%	66.7% , 33.3%	50%, 50%	50%, 62.5%
CPU Time	22.060s	22.060s	34.950s	34.950s
Speedup	$\sim 5.75\times$	$\sim 4.68\times$	$\sim 28.69\times$	$\sim 27.63\times$

Πίνακας 21: Optimal Forward Phase Kernel - 16384×32768

	Geforce 9600GT		*Geforce 260 GTX*	
Backward Phase kernel	*128 Threads*	*256 Threads*	*128 Threads*	*256 Threads*
GPU Time %	62.19%	64.69%	39.3%	40.45%
GPU Time (sec)	8.97s	1.162s	1.015s	1.105s
Occupancy	83,3%	66,7%	100%	100%
CPU Time	6.341s	6.341s	14.920s	14.920s
Speedup	$\sim -1.41\times$	$\sim 5.45\times$	$\sim 14.70\times$	$\sim 13.50\times$

Πίνακας 22: Optimal Backward Phase Kernel - 16384×32768

Έπειτα παρουσιάζουμε τα αποτελέσματα έχοντας σταματήσει την εκτέλεση

του αλγορίθμου χωρίς να υπολογίσει τις τελευταίες μειώσεις. Δηλαδή, όταν το σύστημα είναι ίσο με το μέγεθος του block.

	Geforce 9600GT		*Geforce 260 GTX*	
rhs kernel	*128 Threads*	*256 Threads*	*128 Threads*	*256 Threads*
GPU Time %	10.6%	10,2%	16.7%	16.9%
GPU Time (sec)	0.263s	0,515s	0.131s	0.133s
Occupancy	100%	100%	100%	100%
CPU Time	1,451s	1,451s	3,742s	3,742s
Speedup	$\sim 5.52\times$	$\sim 2.82\times$	$\sim 28.56\times$	$\sim 28.14\times$

Πίνακας 23: Optimal Rhs Kernel - Grid 8192×16384

	Geforce 9600GT		*Geforce 260 GTX*	
Forward Phase Part1 & Part2	*128 Threads*	*256 Threads*	*128 Threads*	*256 Threads*
GPU Time %	12,0%, 16.5%	11.8%, 17.8%	21.3%, 25.6%	23%, 25.4%
GPU Time (sec)	0,703s	1.493s	0.368s	0.382s
Occupancy	66.7% , 50%	66.7% , 33,3%	50%, 62,5%	50%, 50%
CPU Time	5.749s	5.749s	9.020s	9.020s
Speedup	$\sim 8.18\times$	$\sim 3.85\times$	$\sim 24.51\times$	$\sim 23.61\times$

Πίνακας 24: Optimal Forward Phase Kernel - 8192×16384

	Geforce 9600GT		*Geforce 260 GTX*	
Backward Phase kernel	*128 Threads*	*256 Threads*	*128 Threads*	*256 Threads*
GPU Time %	60,9%	60,2%	36.6%	34.6%
GPU Time (sec)	1,506s	3,034s	0.286s	0.274s
Occupancy	83,3%	66,7%	100%	100%
CPU Time	1.654s	1.654s	3.350s	3.350s
Speedup	$\sim 1.1\times$	$\sim -1.83\times$	$\sim 11.71\times$	$\sim 12.23\times$

Πίνακας 25: Optimal Backward Phase Kernel - 8192×16384

και για χωρίο 16384×32768 έχουμε:

	Geforce 9600GT		*Geforce 260 GTX*	
rhs kernel	*128 Threads*	*256 Threads*	*128 Threads*	*256 Threads*
GPU Time %	9.4%	11.1%	17.3%	12.5%
GPU Time (sec)	1,633s	1,620s	0.349s	0.364s
Occupancy	100%	100%	100%	100%
CPU Time	6,159s	6,159s	14,920s	14,920s
Speedup	$\sim 3.77\times$	$\sim 3.80\times$	$\sim 42.75\times$	$\sim 40.99\times$

Πίνακας 26: Optimal Rhs Kernel - Grid 16384×32768

	Geforce 9600GT		*Geforce 260 GTX*	
Forward Phase Part1 & Part2	*128 Threads*	*256 Threads*	*128 Threads*	*256 Threads*
GPU Time %	10.7%, 14.0%	12.8%, 14.1%	22.3%, 24.7%	13.4%, 42.7%
GPU Time (sec)	4.285s	3.918s	0.953s	1.001s
Occupancy	66.7% , 50%	66.7% , 66.7%	50%, 66.7%	50%, 50%
CPU Time	22.060s	22.060s	34.950s	34.950s
Speedup	$\sim 5.15\times$	$\sim 5.63\times$	$\sim 36.67\times$	$\sim 34.91\times$

Πίνακας 27: Optimal Forward Phase Kernel - 16384×32768

	Geforce 9600GT		*Geforce 260 GTX*	
Backward Phase kernel	*128 Threads*	*256 Threads*	*128 Threads*	*256 Threads*
GPU Time %	65.9%	61.9%	35.7%	40.45%
GPU Time (sec)	1.144s	9.009s	0.723s	0.700s
Occupancy	83,3%	66,7%	100%	100%
CPU Time	6.341s	6.341s	14.920s	14.920s
Speedup	$\sim 5.54\times$	$\sim -1.42\times$	$\sim 20.63\times$	$\sim 21.30\times$

Πίνακας 28: Optimal Backward Phase Kernel - 16384×32768

Αυτό που παρατηρούμε και ήταν αναμενόμενο είναι οτι η κάρτα *Geforce 260 GTX* έχει παντού πολύ καλύτερους χρόνους σε σχέση με τη κάρτα *Geforce 9600GT*.

Βασικός παράγοντας σε αυτό, πέρα από την παραπάνω ισχύ και την πιο γρήγορη μνήμη της κάρτας είναι η βελτιωμένη αρχιτεκτονική της.

Συγκεκριμένα, ο τρόπος προσπέλασης της μνήμης από τα νήματα. Αυτό παρατηρείται πολύ έντονα ειδικά στην προς τα πίσω φάση όπου τα νήματα προσπελάσουν γειτονικές αλλά όχι διαδοχικές θέσεις μνήμης.

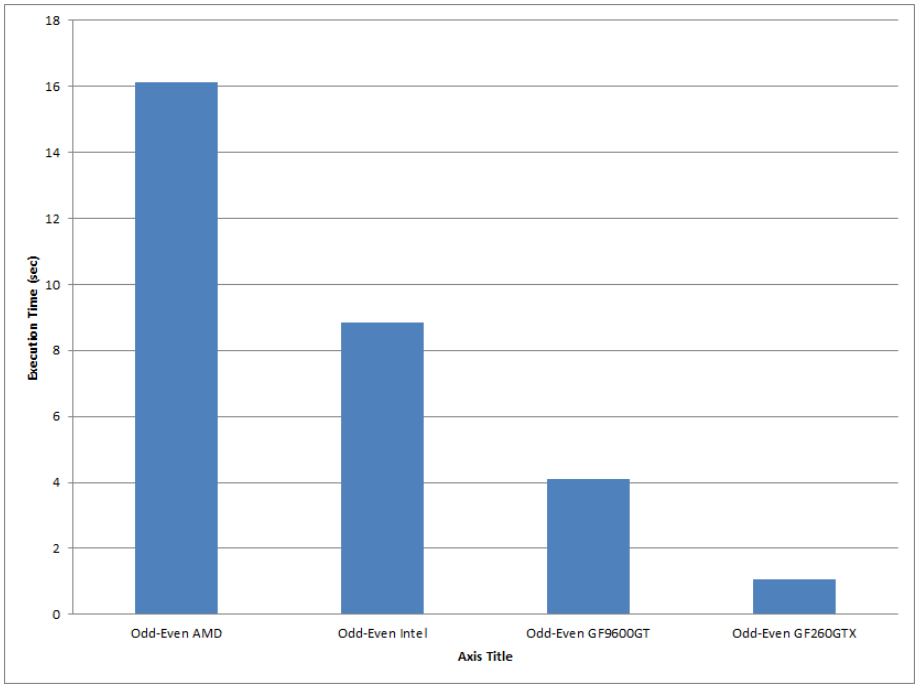
Στην συνέχεια παραθέτουμε την συνολική απόδοση τις εφαρμογής σε σχέση με τις επεξεργαστικές μονάδες. Στις μετρήσεις έχουμε συμπεριλάβει και την μέθοδο *LU-Decomposition*.

	Grid 8192×16384		*Grid* 16384×32768	
Execution Time	*128 Threads*	*256 Threads*	*128 Threads*	*256 Threads*
CPU Time(Intel) - Odd-Even (sec)	8.858s		34.547s	
CPU Time(Intel) - LU (sec)	39.495s		132.169s	
CPU Time(AMD) - Odd-Even (sec)	16.110s		63.220s	
CPU Time(AMD) - LU (sec)	8.900s		36.25s	
GPU 9600GT (sec)	4.107s	4.113s	7.062s	14.54s
GPU 260GTX (sec)	1.016s	1.079s	2.250s	2.065s

Πίνακας 29: Αθροιστικός χρόνος εκτέλεσης των μεθόδων

Στο σημείο αυτό να τονίσουμε ότι οι παραπάνω χρόνοι έχουν προκύψει αθροιστικά από τα αποτελέσματα που είχαμε στη διάθεση μας μέσω των profilers. **Δεν** είναι οι συνολικοί χρόνοι εκτέλεσης. Ο λόγος που παραθέτουμε έτσι τα δεδομένα είναι για να έχουμε αρχικά μια ακριβή εικόνα της αύξησης της απόδοσης στις συγκεκριμένες μεθόδους.

Στο επόμενο γράφημα φαίνεται ο χρόνος εκτέλεσης των συναρτήσεων που υλοποιούν τη μέθοδο *Odd-Even*.



Σχήμα 24: Κλιμάκωση απόδοσης στη κάρτα γραφικών *Geforce 9600 GT*

Ο συνολικός χρόνος εκτέλεσης της εφαρμογής για κάθε αρχιτεκτονική πάρθηκε με την εντολή *time* του Λειτουργικού Συστήματος. Για την x86 Intel αρχιτεκτονική θα μπορούσαμε να είχαμε χρησιμοποιήσει και το VTune, όμως το αποφύγαμε για λόγους ομοιομορφίας των αποτελεσμάτων.

Ο συνολικός χρόνος εκτέλεσης σε κάθε αρχιτεκτονική απεικονίζεται στη συνέχεια.

Αυτό που παρατηρούμε είναι οτι η τελική επιτάχυνση στο συνολικό χρόνο εκτέλεσης της εφαρμογής είναι πολύ λιγότερη σε σχέση με την επιτάχυνση στο επίπεδο των συναρτήσεων. Η συνολική επιτάχυνση φαίνεται στους πίνακες που

	Grid 8192×16384		*Grid* 16384×32768	
Execution Time	*128 Threads*	*256 Threads*	*128 Threads*	*256 Threads*
CPU Time(Intel) - Odd-Even (sec)	9.366s		37.465s	
CPU Time(Intel) - LU (sec)	42.704s		153.823s	
CPU Time(AMD) - Odd-Even (sec)	16.118s		63.315s	
CPU Time(AMD) - LU (sec)	8.928s		36.414s	
GPU 9600GT (sec)	5.018s	4.784s	16.425s	16.565s
GPU 260GTX (sec)	2.214s	2.223s	3.231s	3.260s

Πίνακας 30: Συνολικός χρόνος εκτέλεσης

ακολουθούν.

Η μείωση στην απόδοση οφείλεται σε δίαφορους παράγοντες. Καθυστέρηση υπάρχει από τον πηγαίο κώδικα που εκτελείται στον επεξεργαστή και είναι υπεύθυνος για τη δέσμευση και αποδέσμευση μνήμης, το χειρισμό λαθών και τη δρομολόγηση των kernels.

Επιπλέον, σημαντικός χρόνος σπαταλείται κατά τη μεταφορά των δεδομένων από και προς τη κάρτα γραφικών, χρόνος κατα τον οποίο δεν έκτελειται καμία συνάρτηση. Τέλος, ένας ακόμη παράγοντας καθυστέρησης είναι και η δημιουργία, καταστροφή και διευθυνσιοδότηση της μνήμης texture.

	128 Threads	*256 Threads*
Speedup 9600GT vs Intel	$1.87\times$	$1.96\times$
Speedup 260GTX vs AMD	$7.28\times$	$7.25\times$

Πίνακας 31: Συνολική επιτάχυνση για χωρίο 8192×16384

	128 Threads	*256 Threads*
Speedup 9600GT vs Intel	$2.28\times$	$2.26\times$
Speedup 260GTX vs AMD	$19.60\times$	$19.42\times$

Πίνακας 32: Συνολική επιτάχυνση για χωρίο 16384×32768

5.7 Επαλήθευση των αποτελεσμάτων

Το τελικό βήμα κατά την ανάπτυξη της εφαρμογής μας είναι η επαλήθευση των αποτελεσμάτων. Παρόλο που σε όλη τη διάρκεια της ανάπτυξης ελέγχαμε την ορθότητα των πράξεων κατά την εκτέλεση των μεθόδων, στο τέλος έπρεπε να ελεγχθεί το τελικό αποτέλεσμα.

Αφού μετρήσαμε την απόδοση της κάθε μεθόδου αλλά και του συνολικού χρόνου εκτέλεσης τυπώσαμε την έξοδο της τελικής μεθόδου και συγκρίναμε τα αποτελέσματα με την x86 υλοποίηση όπου και εξακριβώσαμε την ορθότητα των αποτελεσμάτων μας.

6 Συμπεράσματα και Μελλοντικές επεκτάσεις

Στο Κεφάλαιο αυτό παρουσιάζονται τα συμπεράσματα που αποκομίσαμε από την ολοκλήρωση της συγκεκριμένης εργασίας και παρατίθενται κάποιες σκέψεις σχετικά με το τρόπο που θα μπορούσαμε να την βελτιώσουμε.

6.1 Συμπεράσματα

Στην εργασία αυτή είχαμε ως βασικό στόχο να εξετάσουμε μαζικά παράλληλα συστήματα. Όπως φαίνεται από την έκταση του κειμένου μεγάλη έμφαση δόθηκε στην τεχνολογία CUDA. Ο μερικός επανασχεδιασμός των αλγορίθμων που χρησιμοποιήσαμε στέφθηκε με επιτυχία παρόλο που αυτό έγινε με μεγάλο κόστος. Σημαντικά προβλήματα προέκυψαν κατά την διαδικασία ανάπτυξης τόσο με τα εργαλεία που χρησιμοποιήσαμε όσο και στην προσπάθεια να επιτύχουμε την μεγαλύτερη δυνατή απόδοση.

Είναι δεδομένο οτι τα πολυπύρηνα συστήματα αυξάνουν κατά πολλές τάξεις μεγέθους την απόδοση, αλλά παράλληλα αυξάνουν και την πολυπλοκότητα για την ανάπτυξη εφαρμογών. Όσον αφορά την τεχνολογία CUDA καθώς και την δυσκολία προγραμματισμού μπορούμε να πούμε οτι είναι μια πολλά υποσχόμενη τεχνολογία που αποφορτίζει τον προγραμματιστή από την υποχρέωση να έχει βαθιά γνώση για την αρχιτεκτονική που προγραμματίζει. Επιπλέον, όσο εξελίσσεται η συγκεκριμένη τεχνολογία τόσο πιο πολλές δυνατότητες παρέχει στον προγραμματιστή και όπως είδαμε από την δική μας υλοποίηση, από γενιά σε γενιά οι κάρτες γραφικών γίνονται όλο και πιο ισχυρές, προσφέροντας στους χρήστες επεξεργαστική ισχύ εκατοντάδων GFlops στον οικιακό τους υπολογιστή.

6.2 Μελλοντικές Επεκτάσεις

Προκειμένου να αυξηθεί περαιτέρω η απόδοση, ενδεχομένως να απαιτείται μια διαφορετική προσέγγιση στην υλοποίηση CUDA. Πιο συγκεκριμένα, ίσως ο αλγοριθμικός επανασχεδιασμός της προς τα πίσω μεθόδου να απέδιδε μεγαλύτερη απόδοση καθώς ο τρόπος προσπέλασης των δεδομένων στη μνήμη απέχει από το να χαρακτηριστεί βέλτιστος. Όταν το occupancy σε κάποιους kernels είναι 50% , πιθανόν να μην είναι ικανό να ``κρύψει'' την καθυστέρηση που εισάγει η μνήμη.

Επιπλέον, μια ακόμη πιθανή βελτίωση είναι να τροποποιήσουμε την εφαρμογή ώστε να εκτελείται σε διάταξη SLI, triple SLI και quad SLI. Παρόλο, που αυτό εισάγει αρκετή πολυπλοκότητα στο τρόπο που θα δια μοιραστούν και θα συγχρονιστούν τα δεδομένα στις κάρτες γραφικών, υπάρχει μεγάλη πιθανότητα να αυξηθεί σημαντικά η απόδοση.

Τέλος, μια πιθανή ακόμη βελτίωση είναι να χρησιμοποιήσουμε ``pinned'' μνήμη. Με τον τρόπο αυτό εξ αλείφουμε την απαίτηση για μεταφορά των δεδομένων από και προς τη κάρτα γραφικών. Απαραίτητη προϋπόθεση όμως για αυτό είναι η ύπαρξη ενός συστήματος με ταχύτατη μνήμη μεγάλης χωρητικότητας καθώς σε διαφορετική περίπτωση η απόδοση θα είναι χειρότερη.

7 Παράρτημα

7.1 Λογισμικό

Σε όλη τη διάρκεια της εργασίας χρησιμοποιήθηκε ως επί το πλείστον Ελεύθερο Λογισμικό / Λογισμικό Ανοιχτού Κώδικα.

Ως Λειτουργικό Σύστημα χρησιμοποιήθηκε Ubuntu Linux έκδοση Lucid Lynx 10.04. Για τις γραφικές παραστάσεις χρησιμοποιήθηκε το πρόγραμμα της GNU, gnuplot ενώ το κείμενο γράφτηκε σε Latex και ως επεξεργαστής κειμένου επιλέχθηκε το πρόγραμμα από το γραφικό περιβάλλον του KDE, Kile.

Ένας από τους profilers που χρησιμοποιήθηκε είναι ο Gprof, της GNU ενώ το ιστόγραμμα έγινε στη σουίτα γραφείου LibreOffice.

Κατα την ενασχόληση μας με το Cell Processor χρησιμοποιήσαμε τα Λειτουργικά Συστήματα Fedora Linux, Ubuntu Linux, και Yellow Dog Linux για το Playstaion.

Από μη Ελεύθερο Λογισμικό χρησιμοποιήθηκαν οι οδηγοί γραφικών της NVIDIA, καθώς και τα CUDA Toolkit, CUDA SDK έκδοσης 3.0 Τέλος, χρησιμοποιήσαμε και το λογισμικό της IBM, IBM Cell Simulator.

7.2 Άδεια χρήσης

Το λογισμικό που αναπτύχθηκε και παρατίθεται στην ενότητα 7.3 είναι Ελεύθερο Λογισμικό, μπορεί να διανεμηθεί και/ή να τροποποιηθεί σύμφωνα με τους όρους της Γενικής Άδειας Χρήσης GNU (GNU GPL), όπως δημοσιεύεται από το Ίδρυμα Ελεύθερου Λογισμικού, της έκδοσης 3[16].

7.3 Cell Hello Wolrd

Στην ενότητα αυτή παρατίθεται ένα κλασικό παράδειγμα ανάπτυξης λογισμικού για την αρχιτεκτονική του Cell/B.E.

Listing 1: spu-hello.c

```
#include <stdio.h>

int
main (unsigned long long speid, addr64 argp, addr64 envp)
{
  printf ("Hello␣Wolrd!!␣from␣SPU:␣%llx\n", speid);
  return 0;
}
```

Listing 2: ppu-hello.c

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <libspe2.h>

#define N 4

struct thread_args
{
  struct spe_context *spe;
  void *argp;
  void *envp;
}

void
my_spe_thread (struct thread_args *arg)
{
  unsigned int flags = 0;
  unsigned int entry = SPE_DEFAULT_ENTRY;

  spe_context_run (arg->spe, &entry, flags, arg->argp, arg->envp, NULL);
  pthread_exit (NULL);
}

int
main ()
```

```
{
  pthread_t pts[N];
  spe_context_ptr_t spe[N];
  struct thread_args t_args[N];
  int value[N];
  int i;

  spe_program_handle_t *program;
  program = spe_image_open ("spu_hello");

  for (i = 0; i < N; i++)
    {
      spe[i] = spe_context_create (0, NULL);
      spe_program_load (spe[i], program);

      t_args[i].spe = spe[i];
      t_args[i].argp = &value[i];
      t_args[i].envp = NULL;
      pthread_create (&pts[i], NULL, &my_spe_thread, &t_args[i]);
    }

  for (i = 0; i < N; i++)
    pthread_join (pts[i], NULL);

  spe_image_close (program);

  for (i = 0; i < N; i++)
    spe_context_destroy (spe[i]);

  return 0;
}
```

7.4 Ο πηγαίος κώδικας της εφαρμογής

Στην ενότητα αυτή παρατίθεται ο πηγαίος κώδικας της εφαρμογής.

Listing 3: main.cu

```
#include <stdio.h>
#include <stdlib.h>
#include "tridiag_methods.h"

void
checkCudaError (const char *string)
{
  cudaError_t error = cudaGetLastError ();
  if (cudaSuccess != error)
    {
      fprintf (stderr, "␣Cuda␣error:␣%s:␣%s.\n", string,
               cudaGetErrorString (error));
      exit (EXIT_FAILURE);
    }
}


int
main (int argc, char *argv[])
{
  /*Dimensions of the grid */
//    int NT = 16384;
//    int NX = 8192;
  int NT = 32768;
  int NX = 16384;

  float S = 100.f;
  float X = 5.f;
  float T = 0.25;
  float r = 0.8;
  float v = 0.03;


  /*memory allocation on host for the arrays
   *needed by the differentials equations*/
  float *a, *b, *c, *d0, *d1, *d2, *u;
  size_t memsize = (NX + 1) * sizeof (float);
  //float *f=(float*)malloc(memsize);
  a = (float *) malloc (memsize);
  if (!a)
    perror ("Malloc␣-␣structure␣a");
```

```
b = (float *) malloc (memsize);
if (!b)
  perror ("Malloc - structure b");

c = (float *) malloc (memsize);
if (!c)
  perror ("Malloc - structure c");

d0 = (float *) malloc (memsize);
if (!d0)
  perror ("Malloc - structure d0");

d1 = (float *) malloc (memsize);
if (!d1)
  perror ("Malloc - structure d1");

d2 = (float *) malloc (memsize);
if (!d2)
  perror ("Malloc - structure d2");

u = (float *) malloc (memsize);
if (!u)
  perror ("Malloc - structure u");


PDEfiniteDiff_crank_nicholson_init (a, b, c, d0, d1, d2, u, S, X, T, r,
                                    v, NT, NX);

PDEfiniteDiff_crank_nicholson_new_bound_cond_no_init (a, b, c, d0, d1, d2,
                                                      u, NT, NX);


/* Output the results */
/*   int ctr=0;
   for( ctr=0; ctr<NX ; ctr++ )
   {
   printf("U[%d] = %f\n",ctr,u[ctr]);
   }
 */

/*Free memory on host */
free (a);
free (b);
free (c);
free (d0);
free (d1);
free (d2);
free (u);
```

```
    return EXIT_SUCCESS;
}
```

Listing 4: crank-nicholson.cu

```
#include <math.h>
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>
#include "tridiag_methods.h"

#define SHARED_MEM_SIZE 128
#define NUM_THREADS 128
#define GRID blockDim.x * gridDim.x

//TODO: ERROR CHECKING AND DOCUMENTATION


/*Texture Memory Reference Pointers*/
texture < float, 1, cudaReadModeElementType > texref_a;
texture < float, 1, cudaReadModeElementType > texref_c;
texture < float, 1, cudaReadModeElementType > texref_d0;
texture < float, 1, cudaReadModeElementType > texref_d1;
texture < float, 1, cudaReadModeElementType > texref_d2;
texture < float, 1, cudaReadModeElementType > texref_u;
texture < float, 1, cudaReadModeElementType > texref_e;
texture < float, 1, cudaReadModeElementType > texref_a_even;
texture < float, 1, cudaReadModeElementType > texref_c_even;
texture < float, 1, cudaReadModeElementType > texref_e_even;

float
phi_T (float s, float K)
{
  if (s > K)
    return s - K;
  else
    return 0;
}

/* boundary conditions: 2nd order derivatives at space boundaries are zero */
void
PDEfiniteDiff_crank_nicholson_init (float a[], float b[], float c[],
                                    float d0[], float d1[], float d2[],
                                    float u[], float S, float X, float T,
                                    float r, float v, int NT, int NX)
{

  float dx = S / ((float) NX), dt = T / ((float) NT);


  int j;
  for (j = 0; j < NX; j++)
```

```
{
  int k = j + 1;

  /*boundary condition*/
  u[j] = phi_T (j * dx, X);

  /*Crank Nicholson*/
  float temp1 = r * k / 2;
  float temp2 = v * v * k * k / 2;

  /*multiplied by (-2)*/
  a[j] = temp2 - temp1;

  b[j] = -2 / dt - 2 * temp2 - 2 * r;
  c[j] = temp2 + temp1;

  d1[j] = -2 / dt + 2 * temp2;

  /*initial condition
  like NX-1*/
  if (j == 0)
    {
      b[j] = b[j] + 2 * a[j];
      c[j] = c[j] - a[j];
      d1[j] = d1[j] - 2 * a[j];
    }

  /*initial upper bound
  Fn=2Fn-1 - Fn-2 because  the
  second derivative == 0*/
  if (j == NX - 1)
    {
      a[j] = a[j] - c[j];
      b[j] = b[j] + 2 * c[j];
      d1[j] = d1[j] - 2 * c[j];
    }
  /*
     The main diagonal of the tridiagonal system is normalized -
     each element of the other two diagonals and the right hand vector is divided
     element of the non-normalized main diagonal!
   */

  a[j] = a[j] / b[j];
  c[j] = c[j] / b[j];


  if (j == 0)
    d0[j] = 0.0;
  else
```

```
        /*due to symmetry and
         * because of  NX-1*/
        d0[j] = -a[j];

      d1[j] = d1[j] / b[j];
      if (j == NX - 1)
        d2[j] = 0.0;
      else
        /*due to symmetry*/
        d2[j] = -c[j];

      b[j] = 1.0;

      /* end of normalization of the main diagonal */
    }

  u[NX] = phi_T (NX * dx, X);

  a[0] = 0.0;
  /*
     convention of the serial tridiag_solver function ! !
     (1 st line of the tridiagonal system matrix is:[b[0]
     c[0] 0...0] (a[0] = 0))
   */
  c[NX - 1] = 0.0;
  /*
     convention of the serial tridiag_solver function ! !
     (1 st line of the tridiagonal system matrix is:[0...0 a[NX - 1]
     b[NX - 1]] (c[NX - 1] =0))}
   */


  __global__ void
    rhs (float *d_d0, float *d_d1, float *d_d2, float *d_u, float *d_e,
         int NX)
  {


    /*Thread Index - Each thread has a unique Id */
    int idx = blockIdx.x * blockDim.x + threadIdx.x;

    /*
       Version 1
       the for loop code its translated in cuda code
       like if(idx<NX)
       output[idx]=input[idx];
       This if statement gives branch divergence
       so the entire process is serilized.With
       the code below there is no divergence
```

```
 */

/*
    if (idx > 0 && idx < NX - 1)
    d_e[idx] =
    d_d0[idx] * d_u[idx] + d_d1[idx] * d_u[idx + 1] +
    d_d2[idx] * d_u[idx + 2];
    }
 */

/*Version 2.1 - Shared Memory */
/*this is done in order to achieve coalescing */
/*
    __shared__ float s_d0[SHARED_MEM_SIZE];
    __shared__ float s_d1[SHARED_MEM_SIZE];
    __shared__ float s_d2[SHARED_MEM_SIZE];
    __shared__ float s_u[SHARED_MEM_SIZE];
    __shared__ float s_u_plus_one[SHARED_MEM_SIZE];
    __shared__ float s_u_plus_two[SHARED_MEM_SIZE];

    s_d0[threadIdx.x] = d_d0[idx];
    s_d1[threadIdx.x] = d_d1[idx];
    s_d2[threadIdx.x] = d_d2[idx];
    s_u[threadIdx.x] = d_u[idx];

    if ((idx > 0) && (idx <= NX))
    s_u_plus_one[threadIdx.x] = d_u[idx];

    if ((idx > 1) && (idx <= NX + 1))
    s_u_plus_two[threadIdx.x] = d_u[idx];


    //Used to avoid RAW/WAR/WAW hazards
    __syncthreads ();

    if (idx < NX)
    {
    d_e[idx] =
    s_d0[threadIdx.x] * s_u[threadIdx.x] +
    s_d1[threadIdx.x] * s_u_plus_one[threadIdx.x] +
    s_d2[threadIdx.x] * s_u_plus_one[threadIdx.x];
    }
    __syncthreads ();
    }
 */

/*Version 2.2 -
    Shared Memory with multiple loads *
    /__shared__ float s_d0_u[SHARED_MEM_SIZE];
```

```
    __shared__ float s_d1_u[SHARED_MEM_SIZE];
    __shared__ float s_d2_u[SHARED_MEM_SIZE];

    s_d0_u[threadIdx.x] = d_d0[idx] * d_u[idx];
    s_d1_u[threadIdx.x] = d_d1[idx] * d_u[idx + 1];
    s_d2_u[threadIdx.x] = d_d2[idx] * d_u[idx + 2];

    //Used to avoid RAW/WAR/WAW hazards
    __syncthreads ();

    if (idx < NX)
    {
    d_e[idx] =
    s_d0_u[threadIdx.x] + s_d1_u[threadIdx.x] + s_d2_u[threadIdx.x];
    }
    __syncthreads ();

    /*Version 3.1 - Use of texture and shared Memory */
/*
    __shared__ float s_d0[SHARED_MEM_SIZE];
    __shared__ float s_d1[SHARED_MEM_SIZE];
    __shared__ float s_d2[SHARED_MEM_SIZE];
    __shared__ float s_u[SHARED_MEM_SIZE];
    __shared__ float s_u_plus_one[SHARED_MEM_SIZE];
    __shared__ float s_u_plus_two[SHARED_MEM_SIZE];

    s_d0[threadIdx.x] = tex1Dfetch (texref_d0, idx);
    s_d1[threadIdx.x] = tex1Dfetch (texref_d1, idx);
    s_d2[threadIdx.x] = tex1Dfetch (texref_d2, idx);
    s_u[threadIdx.x] = tex1Dfetch (texref_u, idx);
    s_u_plus_one[threadIdx.x] = tex1Dfetch (texref_u, idx + 1);
    s_u_plus_two[threadIdx.x] = tex1Dfetch (texref_u, idx + 2);

    //Used to avoid RAW/WAR/WAW hazards
    __syncthreads ();

    if (idx < NX)
    {
    d_e[idx] =
    s_d0[threadIdx.x] * s_u[threadIdx.x] +
    s_d1[threadIdx.x] * s_u_plus_one[threadIdx.x] +
    s_d2[threadIdx.x] * s_u_plus_one[threadIdx.x];

    }
    __syncthreads ();
    }

 */
/*Version 3.2 - Texture and shared memory - multiple loads */
```

```
  //FASTER OF ALL
  __shared__ float s_d0_u[SHARED_MEM_SIZE];
  __shared__ float s_d1_u[SHARED_MEM_SIZE];
  __shared__ float s_d2_u[SHARED_MEM_SIZE];

  s_d0_u[threadIdx.x] =
    tex1Dfetch (texref_d0, idx) * tex1Dfetch (texref_u, idx);
  s_d1_u[threadIdx.x] =
    tex1Dfetch (texref_d1, idx) * tex1Dfetch (texref_u, idx + 1);
  s_d2_u[threadIdx.x] =
    tex1Dfetch (texref_d2, idx) * tex1Dfetch (texref_u, idx + 2);

  //Used to avoid RAW/WAR/WAW hazards
  __syncthreads ();

  if (idx < NX)
    {
      d_e[idx] =
        s_d0_u[threadIdx.x] + s_d1_u[threadIdx.x] + s_d2_u[threadIdx.x];
    }

  __syncthreads ();

}




__global__ void
  tridiag_forward_non_recursive_plain_part1 (float *d_a, float *d_b,
                                             float *d_c, float *d_e,
                                             float *a_even, float *b_even,
                                             float *c_even, float *e_even,
                                             int N)
{
  /*Thread Indexes */
  int idx = blockDim.x * blockIdx.x + threadIdx.x;

  /*Creating odd and even thread Indexes
     from thread#2 and above */
  int even = idx * 2;
  int odd = (idx * 2) + 1;

  __shared__ float s_a[SHARED_MEM_SIZE];
  __shared__ float s_c[SHARED_MEM_SIZE];
  __shared__ float s_e_even[SHARED_MEM_SIZE];
```

```
    __shared__ float s_e_odd[SHARED_MEM_SIZE];
    __shared__ float s_c_c[SHARED_MEM_SIZE];
    __shared__ float s_a_a[SHARED_MEM_SIZE];
//    __shared__ float s_e[ SHARED_MEM_SIZE ];
    float div;
    /* Version 1.0 - Plain Implementation */
    /*This initializes the tridiagonal system on the upper side */
    /*
       if (N >= 2)
       {
       //This if is equivalent to idx=0
       // ex a_even[idx]==a_even[0]
       if (idx < 1)
       {
       div = 1.f / (d_a[idx] * d_c[idx] - 1.f);
       a_even[idx] = 0.f;
       b_even[idx] = 1.f;
       c_even[idx] = d_c[idx] * d_c[idx + 1] * div;
       e_even[idx] = (d_c[idx] * d_e[idx + 1] - d_e[idx]) * div;
       }
       //Start from threadId 2 up to 4095
       //or the half of the system
       if ((idx > 1) && (idx < N / 2))
       {
       div =
       1.f / (1.f - d_c[even - 1] * d_a[even] - d_c[even] * d_a[odd]);
       a_even[idx] = -d_a[even - 1] * d_a[even] * div;
       b_even[idx] = 1.f;
       c_even[idx] = -d_c[even] * d_c[odd] * div;
       e_even[idx] =
       (d_e[even] - d_e[even - 1] * d_a[even] -
       d_c[even] * d_e[odd]) * div;
       }
       }
     */
    /*Version 1.1 - Plain Implementation with texture memory */
    /*
       if (N >= 2)
       {
       //This if is equivalent to idx=0
       //  ex a_even[idx]==a_even[0]
       if (idx < 1)
       {
       div =
       1.f / (tex1Dfetch (texref_a, idx) * tex1Dfetch (texref_c, idx) -
       1.f);
       a_even[idx] = 0.f;
       b_even[idx] = 1.f;
       c_even[idx] =
```

```
    tex1Dfetch (texref_c , idx) * tex1Dfetch (texref_c ,
    idx + 1) * div;
    //e_even[idx] = d_c[idx]*d_e[idx+1] - d_e[idx]*div;
    e_even[idx] =
    (tex1Dfetch (texref_c , idx) * tex1Dfetch (texref_e , idx + 1) -
    tex1Dfetch (texref_e , idx)) * div;
    }

    //Start from threadId 2
    if ((idx > 1) && (idx < N / 2))
    {
    div =
    1.f / (1.f -
    tex1Dfetch (texref_c , even - 1) * tex1Dfetch (texref_a ,
    even) -
    tex1Dfetch (texref_c , even) * tex1Dfetch (texref_a ,
    odd));
    a_even[idx] =
    -tex1Dfetch (texref_a , even - 1) * tex1Dfetch (texref_a ,
    even) * div;
    b_even[idx] = 1.f;
    c_even[idx] =
    tex1Dfetch (texref_c , even) * tex1Dfetch (texref_c , odd) * div;
    //e_even[idx_half]=(d_e[idx] - d_e[idx-1]*d_a[idx] - d_c[idx]*d_e[idx+1]*div);
    e_even[idx] =
    (tex1Dfetch (texref_e , even) -
    tex1Dfetch (texref_e , even - 1) * tex1Dfetch (texref_a ,
    even) -
    tex1Dfetch (texref_c , even) * tex1Dfetch (texref_e ,
    odd)) * div;
    }
    }
 */
//Version 2.1 - Shared Memory Implementation
/*
    if (N >= 2)
    {
    if (idx < (N / 2))
    {
    s_a[threadIdx.x] = d_a[even];
    s_c[threadIdx.x] = d_c[even];
    s_e[threadIdx.x] = d_e[even];
    //This if is equivalent to idx=0
    // ex a_even[idx]==a_even[0]
    if (idx < 1)
    {
    div = 1.f / (s_a[threadIdx.x] * s_c[threadIdx.x] - 1.f);
    a_even[idx] = 0.f;
    b_even[idx] = 1.f;
```

```
    c_even[idx] = s_c[threadIdx.x] * s_c[threadIdx.x] * div;
    e_even[idx] =
    (s_c[threadIdx.x] * d_e[threadIdx.x] -
    s_e[threadIdx.x]) * div;
    }
    else
    {
    //Start from threadId 2
    div =
    1.f / (1.f - s_c[threadIdx.x - 1] * s_a[threadIdx.x] -
    s_c[threadIdx.x] * s_a[threadIdx.x + 1]);
    a_even[idx] = -s_a[threadIdx.x - 1] * s_a[threadIdx.x] * div;
    b_even[idx] = 1.f;
    c_even[idx] = -s_c[threadIdx.x] * s_c[threadIdx.x + 1] * div;
    e_even[idx] =
    (s_e[threadIdx.x] -
    s_e[threadIdx.x - 1] * s_a[threadIdx.x] -
    s_c[threadIdx.x] * s_e[threadIdx.x + 1]) * div;
    }
    }
    }
 */
//Version 2.2 - Shared Memory and texture memory
if (N >= 2)
  {
    if (idx < (N / 2))
      {
        s_a[threadIdx.x] = tex1Dfetch (texref_a, even);
        s_c[threadIdx.x] = tex1Dfetch (texref_c, even);
        s_e_even[threadIdx.x] = tex1Dfetch (texref_e, even);
        s_e_odd[threadIdx.x] = tex1Dfetch (texref_e, odd);
        s_c_c[threadIdx.x] =
          tex1Dfetch (texref_c, even) * tex1Dfetch (texref_c, odd);
        s_a_a[threadIdx.x] =
          tex1Dfetch (texref_a, even) * tex1Dfetch (texref_a, even - 1);
        __syncthreads ();
        //s_e[threadIdx.x]=d_e[idx];
        //This if is equivalent to idx=0
        // ex a_even[idx]==a_even[0]
        if (idx < 1)
          {
            div = 1.f / (s_a[threadIdx.x] * s_c[threadIdx.x] - 1.f);
            a_even[idx] = 0.f;
            b_even[idx] = 1.f;
            c_even[idx] = s_c_c[threadIdx.x] * div;
            e_even[idx] =
              (s_c[threadIdx.x] * s_e_odd[threadIdx.x] -
               s_e_even[threadIdx.x]) * div;
          }
```

```
            else
              {
                div =
                  1.f / (1.f - s_c[threadIdx.x] * s_a[threadIdx.x] -
                         s_c[threadIdx.x] * s_a[threadIdx.x]);
                a_even[idx] = -s_a_a[threadIdx.x] * div;
                b_even[idx] = 1.f;
                c_even[idx] = -s_c_c[threadIdx.x] * div;
                e_even[idx] =
                  (s_e_even[threadIdx.x] -
                   s_e_even[threadIdx.x] * s_a[threadIdx.x] -
                   s_c[threadIdx.x] * s_e_odd[threadIdx.x]) * div;
              }
            __syncthreads ();
          }
      }
}



__global__ void
  tridiag_forward_non_recursive_plain_part2 (float *d_a, float *d_b,
                                             float *d_c, float *d_e,
                                             float *a_even, float *b_even,
                                             float *c_even, float *e_even,
                                             float *x, int N,
                                             int *d_start, int *d_end,
                                             int *d_step)
{
  int idx = (blockDim.x * blockIdx.x) + threadIdx.x;
  int start, end, k;
  int offset = __umul24 (gridDim.x, blockDim.x) / 2;
  int loop = offset;
  float div;
  k = 0;
  start = 0;
  end = offset;
  int even = idx * 2;
  int odd = (idx * 2) - 1;
  __shared__ float s_a_even[SHARED_MEM_SIZE];
  __shared__ float s_e_even[SHARED_MEM_SIZE];
  __shared__ float s_c_even[SHARED_MEM_SIZE];
  __shared__ float s_a_a_even[SHARED_MEM_SIZE];
  __shared__ float s_c_c_even[SHARED_MEM_SIZE];
  /*Version 1 - Plain Implementation */
  /*
     if (idx >= N / 2)
     {
```

```
    while (loop >= 2)
    {
    if (idx == offset)
    {
    div = 1.f / (a_even[odd - GRID] * c_even[even - GRID] - 1.f);
    a_even[idx] = 0.f;
    b_even[idx] = 1.f;
    c_even[idx] = a_even[even - GRID] * c_even[odd - GRID] * div;
    e_even[idx] =
    (c_even[even - GRID] * e_even[odd - GRID] -
    e_even[even - GRID]) * div;
    }

    if ((idx > offset) && (idx < offset + (loop >> 1)))
    {
    div =
    1.f / (1.f - c_even[even - GRID - 1] * a_even[even - GRID] -
    c_even[odd - GRID] * a_even[odd - GRID]);
    a_even[idx] =
    -a_even[even - GRID - 1] * a_even[even - GRID] * div;
    b_even[idx] = 1.f;
    c_even[idx] = -c_even[even - GRID] * c_even[odd - GRID] * div;
    e_even[idx] =
    (e_even[even - GRID] -
    e_even[even - GRID - 1] * a_even[even - GRID] -
    c_even[even - GRID] * e_even[odd - GRID]) * div;
    }
    start = end;
    offset = offset + (loop >> 1);
    end = offset;
    loop = loop >> 1;
    k++;
    __syncthreads ();
    }
    }
 */
/*Version 2 - Texture Memory */
/*
    if (idx >= N / 2)
    {
    while (loop >= 2)
    {
    if (idx == offset)
    {
    div =
    1.f / (tex1Dfetch (texref_a_even, odd - GRID) *
    tex1Dfetch (texref_c_even, even - GRID) - 1.f);
    a_even[idx] = 0.f;
    b_even[idx] = 1.f;
```

```
c_even[idx] =
tex1Dfetch (texref_a_even,
even - GRID) * tex1Dfetch (texref_c_even,
odd - GRID) * div;
e_even[idx] =
(tex1Dfetch (texref_c_even, even - GRID) *
tex1Dfetch (texref_e_even,
odd - GRID) - tex1Dfetch (texref_e_even,
even - GRID)) * div;
}

if ((idx > offset) && (idx < offset + (loop >> 1)))
{
div =
1.f / (1.f -
tex1Dfetch (texref_c_even,
even - GRID -
1) * tex1Dfetch (texref_a_even,
even - GRID) -
tex1Dfetch (texref_c_even,
odd - GRID) * tex1Dfetch (texref_a_even,
odd - GRID));
a_even[idx] =
-tex1Dfetch (texref_a_even,
even - GRID - 1) * tex1Dfetch (texref_a_even,
even -
GRID) * div;
b_even[idx] = 1.f;
c_even[idx] =
-tex1Dfetch (texref_c_even,
even - GRID) * tex1Dfetch (texref_c_even,
odd - GRID) * div;
e_even[idx] =
(tex1Dfetch (texref_e_even, even - GRID) -
tex1Dfetch (texref_e_even,
even - GRID - 1) * tex1Dfetch (texref_a_even,
even - GRID) -
tex1Dfetch (texref_c_even,
even - GRID) * tex1Dfetch (texref_e_even,
odd - GRID)) * div;
}
start = end;
offset = offset + (loop >> 1);
end = offset;
loop = loop >> 1;
k++;
__syncthreads ();
}
}
```

```
 */
/* Version 3 – Shared Memory */
/*
    if (idx >= N / 2)
    {
    s_a_even[threadIdx.x] = a_even[even – GRID];
    s_c_even[threadIdx.x] = c_even[even – GRID];
    s_e_even[threadIdx.x] = e_even[even – GRID];
    while (loop >= 2)
    {
    if (idx == offset)
    {
    div =
    1.f / (s_a_even[threadIdx.x + 1] * s_c_even[threadIdx.x] –
    1.f);
    a_even[idx] = 0.f;
    b_even[idx] = 1.f;
    c_even[idx] =
    s_a_even[threadIdx.x] * s_c_even[threadIdx.x + 1] * div;
    e_even[idx] =
    (s_c_even[threadIdx.x] * s_e_even[threadIdx.x + 1] –
    s_e_even[threadIdx.x]) * div;
    }

    if ((idx > offset) && (idx < offset + (loop >> 1)))
    {
    div =
    1.f / (1.f –
    s_c_even[threadIdx.x – 1] * s_a_even[threadIdx.x] –
    s_c_even[threadIdx.x + 1] * s_a_even[threadIdx.x +
    1]);
    a_even[idx] =
    –s_a_even[threadIdx.x – 1] * s_a_even[threadIdx.x] * div;
    b_even[idx] = 1.f;
    c_even[idx] =
    –s_c_even[threadIdx.x] * s_c_even[threadIdx.x + 1] * div;
    e_even[idx] =
    (s_e_even[threadIdx.x] –
    s_e_even[threadIdx.x – 1] * s_a_even[threadIdx.x] –
    s_c_even[threadIdx.x] * s_e_even[threadIdx.x + 1]) * div;
    }
    start = end;
    offset = offset + (loop >> 1);
    end = offset;
    loop = loop >> 1;
    k++;
    __syncthreads ();
    }
    }
```

```
 */
/*Version 4 - Shared and Texture Memory */
/*
    if (idx >= N / 2)
    {
    s_a_a_even[threadIdx.x] =
    -tex1Dfetch (texref_a_even,
    even - GRID) * tex1Dfetch (texref_a_even,
    even - GRID - 1);
    s_c_c_even[threadIdx.x] =
    tex1Dfetch (texref_c_even, even - GRID) * tex1Dfetch (texref_c_even,
    odd - GRID);
    s_a_even[threadIdx.x] = tex1Dfetch (texref_a_even, even - GRID);
    s_c_even[threadIdx.x] = tex1Dfetch (texref_c_even, even - GRID);
    s_e_even[threadIdx.x] = tex1Dfetch (texref_e_even, even - GRID);
    //              s_e_even_minusone[threadIdx.x] = tex1Dfetch(texref_e,even - GRI
    __syncthreads ();
    while (loop >= 2)
    {
    if (idx == offset)
    {
    div =
    1.f / (s_a_even[threadIdx.x] * s_c_even[threadIdx.x] - 1.f);
    a_even[idx] = 0.f;
    b_even[idx] = 1.f;
    c_even[idx] =
    s_a_even[threadIdx.x] * s_c_even[threadIdx.x] * div;
    e_even[idx] =
    (s_c_even[threadIdx.x] * s_e_even[threadIdx.x] -
    s_e_even[threadIdx.x]) * div;
    }

    if ((idx > offset) && (idx < offset + (loop >> 1)))
    {
    div =
    1.f / (1.f -
    s_c_even[threadIdx.x - 1] * s_a_even[threadIdx.x] -
    s_c_even[threadIdx.x] * s_a_even[threadIdx.x]);
    a_even[idx] = s_a_a_even[threadIdx.x] * div;
    b_even[idx] = 1.f;
    c_even[idx] = -s_c_c_even[threadIdx.x] * div;
    e_even[idx] =
    (s_e_even[threadIdx.x] -
    s_e_even[threadIdx.x - 1] * s_a_even[threadIdx.x] -
    s_c_even[threadIdx.x] * s_e_even[threadIdx.x + 1]) * div;
    }
    start = end;
    offset = offset + (loop >> 1);
    end = offset;
```

```
    loop = loop >> 1;
    k++;
    __syncthreads ();
    }
    }
 */
/*OPTIMAL CODE */
if (idx >= N / 2)
  {
    s_a_a_even[threadIdx.x] =
      -tex1Dfetch (texref_a_even,
                   even - GRID) * tex1Dfetch (texref_a_even,
                                              even - GRID - 1);
    s_c_c_even[threadIdx.x] =
      tex1Dfetch (texref_c_even, even - GRID) * tex1Dfetch (texref_c_even,
                                                            odd - GRID);
    s_a_even[threadIdx.x] = tex1Dfetch (texref_a_even, even - GRID);
    s_c_even[threadIdx.x] = tex1Dfetch (texref_c_even, even - GRID);
    s_e_even[threadIdx.x] = tex1Dfetch (texref_e_even, even - GRID);
    __syncthreads ();
    while (loop > blockDim.x)
      {
        /*
           if (loop == 512)
           printf ("found the desired spot with loop = %d\n", loop);
         */

        if (idx == offset)
          {
            div =
              1.f / (s_a_even[threadIdx.x] * s_c_even[threadIdx.x] - 1.f);
            a_even[idx] = 0.f;
            b_even[idx] = 1.f;
            c_even[idx] =
              s_a_even[threadIdx.x] * s_c_even[threadIdx.x] * div;
            e_even[idx] =
              (s_c_even[threadIdx.x] * s_e_even[threadIdx.x] -
               s_e_even[threadIdx.x]) * div;
          }

        if ((idx > offset) && (idx < offset + (loop >> 1)))
          {
            div =
              1.f / (1.f -
                     s_c_even[threadIdx.x - 1] * s_a_even[threadIdx.x] -
                     s_c_even[threadIdx.x] * s_a_even[threadIdx.x + 1]);
            a_even[idx] = s_a_a_even[threadIdx.x] * div;
            b_even[idx] = 1.f;
            c_even[idx] = -s_c_c_even[threadIdx.x] * div;
```

```
            e_even[idx] =
              (s_e_even[threadIdx.x] -
               s_e_even[threadIdx.x - 1] * s_a_even[threadIdx.x] -
               s_c_even[threadIdx.x] * s_e_even[threadIdx.x + 1]) * div;
          }
        start = end;
        offset = offset + (loop >> 1);
        end = offset;
        loop = loop >> 1;
        k++;
        __syncthreads ();
      }
  }

//For the last 128 elements
if (idx > N - blockDim.x)
  {
    while (loop >= 2)
      {
        if ((idx > offset) && (idx < offset + (loop >> 1)))
          {
            if (threadIdx.x < (blockDim.x / 2))
              {
                if (threadIdx.x == 0)
                  {
                    div =
                      1.f / (1.f -
                             s_c_even[blockDim.x -
                                      1] * s_a_even[threadIdx.x] -
                             s_c_even[threadIdx.x] *
                             s_a_even[threadIdx.x + 1]);
                    a_even[idx] = s_a_a_even[threadIdx.x] * div;
                    b_even[idx] = 1.f;
                    c_even[idx] = -s_c_c_even[threadIdx.x] * div;
                    e_even[idx] =
                      (s_e_even[threadIdx.x] -
                       s_e_even[threadIdx.x - 1] * s_a_even[threadIdx.x] -
                       s_c_even[threadIdx.x] * s_e_even[threadIdx.x +
                                                        1]) * div;
                  }

                if ((threadIdx.x > 0) && (threadIdx.x < (blockDim.x - 1)))
                  {
                    div =
                      1.f / (1.f -
                             s_c_even[threadIdx.x -
                                      1] * s_a_even[threadIdx.x] -
                             s_c_even[threadIdx.x] *
                             s_a_even[threadIdx.x + 1]);
```

```
            a_even[idx] = s_a_a_even[threadIdx.x] * div;
            b_even[idx] = 1.f;
            c_even[idx] = -s_c_c_even[threadIdx.x] * div;
            e_even[idx] =
              (s_e_even[threadIdx.x] -
               s_e_even[threadIdx.x - 1] * s_a_even[threadIdx.x] -
               s_c_even[threadIdx.x] * s_e_even[threadIdx.x +
                                                1]) * div;
          }

        if (threadIdx.x == (blockDim.x - 1))
          {
            div =
              1.f / (1.f -
                     s_c_even[threadIdx.x -
                              1] * s_a_even[threadIdx.x] -
                     s_c_even[threadIdx.x] * s_a_even[0]);
            a_even[idx] = s_a_a_even[threadIdx.x] * div;
            b_even[idx] = 1.f;
            c_even[idx] = -s_c_c_even[threadIdx.x] * div;
            e_even[idx] =
              (s_e_even[threadIdx.x] -
               s_e_even[threadIdx.x - 1] * s_a_even[threadIdx.x] -
               s_c_even[threadIdx.x] * s_e_even[threadIdx.x +
                                                1]) * div;
          }
      }

    /*
       The Upper Half of the block
       This is done because we want to read
       addresses that they have not been written
       yet, so we have to load from the source
       in the same time the first half of the block
       reads them
     */
    if (threadIdx.x >= blockDim.x / 2)
      {

        s_a_a_even[threadIdx.x] =
          s_a_a_even[threadIdx.x - blockDim.x / 2];
        s_c_c_even[threadIdx.x] =
          s_c_c_even[threadIdx.x - blockDim.x / 2];
        s_e_even[threadIdx.x] =
          s_e_even[threadIdx.x - blockDim.x / 2];
        s_a_even[threadIdx.x] =
          s_a_even[threadIdx.x - blockDim.x / 2];
        s_c_even[threadIdx.x] =
          s_c_even[threadIdx.x - blockDim.x / 2];
```

```
if (threadIdx.x == 0)
  {
    div =
      1.f / (1.f -
             s_c_even[blockDim.x -
                      1] * s_a_even[threadIdx.x] -
             s_c_even[threadIdx.x] *
             s_a_even[threadIdx.x + 1]);
    a_even[idx] = s_a_a_even[threadIdx.x] * div;
    b_even[idx] = 1.f;
    c_even[idx] = -s_c_c_even[threadIdx.x] * div;
    e_even[idx] =
      (s_e_even[threadIdx.x] -
       s_e_even[threadIdx.x - 1] * s_a_even[threadIdx.x] -
       s_c_even[threadIdx.x] * s_e_even[threadIdx.x +
                                        1]) * div;
  }

if ((threadIdx.x > 0) && (threadIdx.x < (blockDim.x - 1)))
  {
    div =
      1.f / (1.f -
             s_c_even[threadIdx.x -
                      1] * s_a_even[threadIdx.x] -
             s_c_even[threadIdx.x] *
             s_a_even[threadIdx.x + 1]);
    a_even[idx] = s_a_a_even[threadIdx.x] * div;
    b_even[idx] = 1.f;
    c_even[idx] = -s_c_c_even[threadIdx.x] * div;
    e_even[idx] =
      (s_e_even[threadIdx.x] -
       s_e_even[threadIdx.x - 1] * s_a_even[threadIdx.x] -
       s_c_even[threadIdx.x] * s_e_even[threadIdx.x +
                                        1]) * div;
  }

if (threadIdx.x == (blockDim.x - 1))
  {
    div =
      1.f / (1.f -
             s_c_even[threadIdx.x -
                      1] * s_a_even[threadIdx.x] -
             s_c_even[threadIdx.x] * s_a_even[0]);
    a_even[idx] = s_a_a_even[threadIdx.x] * div;
    b_even[idx] = 1.f;
    c_even[idx] = -s_c_c_even[threadIdx.x] * div;
    e_even[idx] =
      (s_e_even[threadIdx.x] -
       s_e_even[threadIdx.x - 1] * s_a_even[threadIdx.x] -
```

```
                                s_c_even[threadIdx.x] * s_e_even[threadIdx.x +
                                                                 1]) * div;
                        }
                    }
                }
              start = end;
              offset = offset + (loop >> 1);
              end = offset;
              loop = loop >> 1;
              k++;
            }
        }

      /*
         Write the data for the
         reconsruction of the initial
         array
         The index N is used because
         we wan only one thread to
         write the data - saves a lot
         of time
       */
      if (idx == (N - 3))
        {
          *d_start = start;
        }
      else if (idx == (N - 2))
        {
          *d_end = end;
        }
      else if (idx == (N - 1))
        {
          *d_step = k;
        }
      __syncthreads ();
    }




    __global__ void
      tridiag_backward_non_recursive_plain (float *d_a, float *d_b,
                                            float *d_c, float *d_e,
                                            float *a_even,
                                            float *b_even,
                                            float *c_even,
                                            float *e_even, float *x,
```

```
                                        int *d_start, int *d_end,
                                        int *d_step)
{

  /*Thread Index */
  unsigned int idx = blockDim.x * blockIdx.x + threadIdx.x;
  int odd = (idx * 2) + 1;
  int even = idx * 2;
  unsigned int z = 1;
  unsigned int z_k = 1;
  int j;
  int start, end, step;
  int id;
  int N = 2;
  start = *d_start;
  end = *d_end;
  step = *d_step;
  /* Version 1.0 - Plain Implementation */
  /*
     if (idx == 0)
     {
     x[idx] = e_even[start];
     }

     for (; step >= 1; step--)
     {
     if (N > 2)
     {
     z_k = z << step;
     if ((idx > N) && (idx <= (2 * N - 2)))
     {
     if (idx & 1)
     {
     id = idx - N;
     j = start + id;
     x[id * z_k] =
     e_even[j] - a_even[j] * x[(id - 1) * z_k] -
     c_even[j] * x[(id - 1) * z_k];
     }
     }
     }

     if (idx == (N - 1))
     {
     if (((N - 1) & 1))
     {
     x[(N - 1) * z_k] =
     e_even[start + N - 1] - a_even[start + N -
     1] * x[(N - 2) * z_k];
```

```
    }
    }

    N = N << 1;
    end = start;
    start = end - N;
    __syncthreads ();
    }


    __syncthreads ();
    if (idx < (GRID / 2) - 1)
    {
    x[odd] = d_e[odd] - d_a[odd] * x[even] - d_c[odd] * x[odd + 1];
    }

    //this is the last thread
    //  saves time
    if (idx == N - 1)
    {
    if (((N - 1) & 1))
    {
    x[N - 1] = d_e[N - 1] - d_a[N - 1] * x[N - 2];
    }
    }
  */
/*Version 1.1 - Texture Memory */
if (idx == 0)
  {
    x[idx] = tex1Dfetch (texref_e_even, start);
  }

for (; step >= 1; step--)
  {
    if (N > 2)
      {
        z_k = z << step;
        if ((idx > N) && (idx <= (2 * N - 2)))
          {
            if (idx & 1)
              {
                id = idx - N;
                j = start + id;
                x[id * z_k] =
                  tex1Dfetch (texref_e_even,
                              j) - tex1Dfetch (texref_a_even,
                                               j) *
                  tex1Dfetch (texref_u,
                              (id - 1) * z_k) -
```

```
                    tex1Dfetch (texref_c_even, j) * tex1Dfetch (texref_u,
                                                                (id -
                                                                 1) * z_k);
                }
            }
        }

      if (idx == (N - 1))
        {
          if (((N - 1) & 1))
            {
              x[(N - 1) * z_k] =
                tex1Dfetch (texref_e_even,
                            start + N - 1) -
                tex1Dfetch (texref_a_even,
                            start + N - 1) * tex1Dfetch (texref_u,
                                                         (N - 2) * z_k);
            }
        }

      N = N << 1;
      end = start;
      start = end - N;
      __syncthreads ();
    }


__syncthreads ();
if (idx < (GRID / 2) - 1)
  {
    x[odd] =
      tex1Dfetch (texref_e, odd) - tex1Dfetch (texref_a,
                                               odd) *
      tex1Dfetch (texref_u, even) - tex1Dfetch (texref_c,
                                                odd) *
      tex1Dfetch (texref_u, odd + 1);
  }

//this is the last thread
//  saves time
if (idx == N - 1)
  {
    if (((N - 1) & 1))
      {
        x[N - 1] =
          tex1Dfetch (texref_e, N - 1) - tex1Dfetch (texref_a,
                                                     N -
                                                     1) *
          tex1Dfetch (texref_u, N - 2);
```

```
            }
        }
    }



/* boundary conditions: 2nd order derivatives at space boundaries are zero */
    void
      PDEfiniteDiff_crank_nicholson_new_bound_cond_no_init (float a[],
                                                            float b[],
                                                            float c[],
                                                            float d0[],
                                                            float d1[],
                                                            float d2[],
                                                            float u[],
                                                            int NT, int NX)
    {


      /*
         Declaration and memory allocation
         of the arrays that will work on the device
         its the same as in "main".
         SYMVASI.oti yparxei sto host tha exei k to
         idio onoma sto device me ena d mprosta.exairesei
         ta 4 arrays me to even
       */
      float *d_a, *d_b, *d_c;
      float *d_d0, *d_d1, *d_d2;
      float *d_u;
      //empty array tha gemisei apo rhs
      float *d_e;
      float *a_even;
      float *b_even;
      float *c_even;
      float *e_even;
      /*
         data unit in which the number of the
         iterations of the tridiagonal system
         are stored
       */
      int *d_start, *d_end, *d_step;
      cudaDeviceProp deviceProp;
#if CUDART_VERSION < 2020
#error "This␣CUDART␣versoin␣does␣not␣support␣mapped␣memory"
#endif
      //Get Device Properties and verify that device 0
      //support mapped memory
```

```
cudaGetDeviceProperties (&deviceProp, 0);
if (!deviceProp.canMapHostMemory)
  {
    fprintf (stderr, "Device␣%d␣cannot␣map␣host␣memory\n", 0);
    exit (EXIT_FAILURE);
  }

//Set device flags for mapping host memory
cudaSetDeviceFlags (cudaDeviceMapHost);
//Paged  Memory to pass around to the kernels
//allocate host mapped arrays
cudaHostAlloc ((void **) &h_start, sizeof (int), cudaHostAllocMapped);
cudaHostAlloc ((void **) &h_end, sizeof (int), cudaHostAllocMapped);
cudaHostAlloc ((void **) &h_step, sizeof (int), cudaHostAllocMapped);
//Get the Device Pointers
cudaHostGetDevicePointer ((void **) &start, (void *) h_start, 0);
cudaHostGetDevicePointer ((void **) &end, (void *) h_end, 0);
cudaHostGetDevicePointer ((void **) &step, (void *) h_step, 0);
/*Memory allocation on the device */
size_t d_memsize = (NX + 1) * sizeof (float);
//TODO catch errors
/*Pageable Memory Section */
cudaMalloc ((void **) &d_a, d_memsize);
cudaMalloc ((void **) &d_b, d_memsize);
cudaMalloc ((void **) &d_c, d_memsize);
cudaMalloc ((void **) &d_d0, d_memsize);
cudaMalloc ((void **) &d_d1, d_memsize);
cudaMalloc ((void **) &d_d2, d_memsize);
cudaMalloc ((void **) &d_u, d_memsize);
cudaMalloc ((void **) &d_start, sizeof (int));
cudaMalloc ((void **) &d_end, sizeof (int));
cudaMalloc ((void **) &d_step, sizeof (int));
//empty array allocation only
cudaMalloc ((void **) &d_e, d_memsize);
cudaMalloc ((void **) &a_even, NX * sizeof (float));
cudaMalloc ((void **) &b_even, NX * sizeof (float));
cudaMalloc ((void **) &c_even, NX * sizeof (float));
cudaMalloc ((void **) &e_even, NX * sizeof (float));
//Copy data from host to device
//TODO catch errors
cudaMemcpy (d_a, a, d_memsize, cudaMemcpyHostToDevice);
cudaMemcpy (d_b, b, d_memsize, cudaMemcpyHostToDevice);
cudaMemcpy (d_c, c, d_memsize, cudaMemcpyHostToDevice);
cudaMemcpy (d_d0, d0, d_memsize, cudaMemcpyHostToDevice);
cudaMemcpy (d_d1, d1, d_memsize, cudaMemcpyHostToDevice);
cudaMemcpy (d_d2, d2, d_memsize, cudaMemcpyHostToDevice);
cudaMemcpy (d_u, u, d_memsize, cudaMemcpyHostToDevice);
/*
   This is for the use of Page-locked Memory
```

```
    empty array allocation only
 */
/*
    cudaMalloc ((void **) &d_a, d_memsize);
    cudaMalloc ((void **) &d_b, d_memsize);
    cudaMalloc ((void **) &d_c, d_memsize);
    cudaMalloc ((void **) &d_d0, d_memsize);
    cudaMalloc ((void **) &d_d1, d_memsize);
    cudaMalloc ((void **) &d_d2, d_memsize);
    cudaMalloc ((void **) &d_u, d_memsize);
    cudaMalloc ((void **) &d_e, d_memsize);
    cudaMalloc ((void **) &a_even, NX * sizeof (float));
    cudaMalloc ((void **) &b_even, NX * sizeof (float));
    cudaMalloc ((void **) &c_even, NX * sizeof (float));
    cudaMalloc ((void **) &e_even, NX * sizeof (float));
    //pinned memory transfers
    cudaMemcpyAsync (d_a, a, d_memsize, cudaMemcpyHostToDevice, 0);
    cudaMemcpyAsync (d_b, b, d_memsize, cudaMemcpyHostToDevice, 0);
    cudaMemcpyAsync (d_c, c, d_memsize, cudaMemcpyHostToDevice, 0);
    cudaMemcpyAsync (d_d0, d0, d_memsize, cudaMemcpyHostToDevice, 0);
    cudaMemcpyAsync (d_d1, d1, d_memsize, cudaMemcpyHostToDevice, 0);
    cudaMemcpyAsync (d_d2, d2, d_memsize, cudaMemcpyHostToDevice, 0);
    cudaMemcpyAsync (d_u, u, d_memsize, cudaMemcpyHostToDevice, 0);
 */
/*Bind Memory to a texture pointer */
cudaBindTexture (0, texref_a, d_a, d_memsize);
cudaBindTexture (0, texref_c, d_c, d_memsize);
cudaBindTexture (0, texref_e, d_e, d_memsize);
cudaBindTexture (0, texref_d0, d_d0, d_memsize);
cudaBindTexture (0, texref_d1, d_d1, d_memsize);
cudaBindTexture (0, texref_d2, d_d2, d_memsize);
cudaBindTexture (0, texref_u, d_u, d_memsize);
cudaBindTexture (0, texref_a_even, a_even, NX * sizeof (float));
cudaBindTexture (0, texref_c_even, c_even, NX * sizeof (float));
cudaBindTexture (0, texref_e_even, e_even, NX * sizeof (float));
cudaThreadSynchronize ();
/*Grid parameters */
int numBlocks = NX / NUM_THREADS;
dim3 dimGrid (numBlocks);
dim3 dimBlock (NUM_THREADS);
/*
    cudaThreadSynchronise
    Because CUDA kernel launch is asynchronous, and returns immediately,
    this function can be used in order to make sure all kernel launches
    are synchronised. This is rather nifty when we have a number of kernels
    in a for loop
 */
int i;
for (i = 0; i < NT; i++)
```

```
    {
      rhs <<< dimGrid, dimBlock >>> (d_d0, d_d1, d_d2, d_u, d_e, NX);
      /*
         d_u + 1 =
         the address of the second element of the array,
         equal to & d_u[1]
       */
      tridiag_forward_non_recursive_plain_part1 <<<
        dimGrid, dimBlock >>> (d_a, d_b, d_c, d_e, a_even, b_even, c_even,
                               e_even, NX);
      tridiag_forward_non_recursive_plain_part2 <<< dimGrid,
        dimBlock >>> (d_a, d_b, d_c, d_e, a_even, b_even, c_even,
                      e_even, d_u + 1, NX, d_start, d_end, d_step);
      tridiag_backward_non_recursive_plain <<< dimGrid,
        dimBlock >>> (d_a, d_b, d_c, d_e, a_even, b_even, c_even,
                      e_even, d_u + 1, d_start, d_end, d_step);
    }

  /*Unbind texture from memory */
  cudaUnbindTexture (texref_a);
  cudaUnbindTexture (texref_c);
  cudaUnbindTexture (texref_e);
  cudaUnbindTexture (texref_d0);
  cudaUnbindTexture (texref_d1);
  cudaUnbindTexture (texref_d2);
  cudaUnbindTexture (texref_u);
  cudaUnbindTexture (texref_a_even);
  cudaUnbindTexture (texref_c_even);
  cudaUnbindTexture (texref_e_even);
  /*Free memory from the device */
  cudaFree (d_a);
  cudaFree (d_b);
  cudaFree (d_c);
  cudaFree (d_d0);
  cudaFree (d_d1);
  cudaFree (d_d2);
  cudaFree (d_u);
  cudaFree (d_e);
  cudaFree (a_even);
  cudaFree (b_even);
  cudaFree (c_even);
  cudaFree (e_even);
  cudaFree (d_start);
  cudaFree (d_end);
  cudaFree (d_step);
/*
       cudaFreeHost(h_start);
       cudaFreeHost(h_end);
       cudaFreeHost(h_step);
```

```
*/
  }
```

Listing 5: Makefile

```
# ******************************************************************************
# Project: tridiag_parallel_test_delay
# SubTree: ./src
# FileName: Makefile
# Author: p106r
# ******************************************************************************
# Makefile for the src directory
# ******************************************************************************

#SHELL=/bin/dash
#SHELL=/bin/bash

CC = nvcc
EMU = -deviceemu
CCC = $(CC)  $(CFLAGS) -I$(INCLUDE)
#CFLAGS = -02 -g -pg -g3
CFLAGS = -use_fast_math -g
CCC_EMU = $(CC) $(EMU) $(CFLAGS)  -I$(INCLUDE)

INCLUDE = $(HOME)/cuda-code/lib/
LINKOBJ  = main.o tridiag_methods.o crank_nicholson.o
SOURCES = $(LINKOBJ:.o=.cu)
BIN  = /home/p106r/cuda-code/bin/
EXEFILE  = cuda_tridiag_parallel_test_delay
SRC_PATH = /home/p106r/cuda-code/src/


#LIBS =  -L"C:/Dev-Cpp/lib"  -lgmon -pg  -g3
#INCS =  -I"C:/Dev-Cpp/include"
#CXXINCS =  -I"C:/Dev-Cpp/lib/gcc/mingw32/3.4.2/include"  -I"C:/Dev-Cpp/include/c++/3.
-I"C:/Dev-Cpp/include/c++/3.4.2/mingw32"  -I"C:/Dev-Cpp/include/c++/3.4.2"
-I"C:/Dev-Cpp/include"

build: $(LINKOBJ)
        $(CCC) $(LINKOBJ) -o $(EXEFILE)
        mv $(EXEFILE) $(BIN)


all: build

debug:
        nvcc -ptx -I/home/p106r/cuda-code/lib crank_nicholson.cu tridiag_methods.cu ma

emu: $(SOURCES)
        $(CCC_EMU) $(SOURCES) -o emulation

clean:
```

```
	rm -f $(LINKOBJ) $(BIN)/$(EXEFILE)) *.BAK *.bak *~ debug emulation *.ptx
	rm $(BIN)/$(EXEFILE)


main.o: main.cu
	$(CCC) -c $(SRC_PATH)main.cu -o main.o

tridiag_methods.o: tridiag_methods.cu
	$(CCC) -c $(SRC_PATH)tridiag_methods.cu -o tridiag_methods.o

crank_nicholson.o: crank_nicholson.cu
	$(CCC) -c $(SRC_PATH)crank_nicholson.cu -o crank_nicholson.o
```

Αναφορές

[1] Sony. *Playstation*

http://gr.playstation.com

[2] IBM. *BladeCenter QS22*

http://www-03.ibm.com/systems/bladecenter/hardware/servers/qs22/

[3] Matthew Scarpino, *Programming The Cell Processor, for Games, Graphics and Computation*, Prentice Hall, 2008

[4] Wikipedia, *Moore's Law*

http://en.wikipedia.org/wiki/Moore's_law

[5] David Culler and Jaswinder Pal Singh with Anoop Gupta, *Parallel Computer Architecture, A Hardware / Software Approach*

[6] Cell/B.E. *Resource Center and multicore Accelaration Zone*

http://www.ibm.com/developerworks/power/index.html

[7] Sony Computer Entertainment US Research & Development

http://www.research.scea.com/

[8] Cell Architecture Explained Version2

http://www.blachford.info/computer/Cell/Cell0_v2.html

[9] The Cell project at IBM Research

http://www.research.ibm.com/cell/

[10] Gentoo, *Gentoo Linux*

`http://www.gentoo.org/`

[11] OpenSuse, *OpenSuse Linux*

`http://www.openSUSE.org`

[12] Ubuntu, *Ubuntu Linux*

`http://www.ubuntu.com/`

[13] Yellow Dog, *Yellow Dog Linux*

`http://www.yellowdoglinux.com/`

[14] Fedora Project, *Fedora Project Linux*

`http://fedoraproject.org/`

[15] Graig I. Watson ,Michael D. Garris ,Elham Tabassi ,Charles L. Wilson ,R. Michael McCabe ,Stanley Janet *User's Guide to NIST Fingerpring Image Software 2 (NFIS2)*, National Institute of Standards and Technology

[16] GNU General Public License

`http://www.gnu.org/licenses/gpl.html`

[17] GNU Lesser General Public License

`http://www.gnu.org/licenses/lgpl.html`

[18] IBM International License Agreement for Early Release of Programs

`http://www-03.ibm.com/software/sla/sladb.nsf/blalookup/4en?opendocument&la_select`

[19] The OpenGL

`http://www.opengl.org/`

[20] The Direct3D

`http://en.wikipedia.org/wiki/Microsoft_Direct3D`

[21] Dr. Dobbs, High Performance Computing Article

`http://drdobbs.com`

[22] The PCI Express Bus

`http://www.interfacebus.com/PCI-Express-Bus-PCIe-Description.html`

[23] The PCI Express Bus, National Instruments

`http://zone.ni.com/devzone/cda/tut/p/id/3767`

[24] NVIDIA CUDA C Programming Guide

`http://developer.nvidia.com/cuda-toolkit-40rc2#Linux`

[25] NVIDIA CUDA C Best Practices Guide

`http://developer.nvidia.com/cuda-toolkit-40rc2#Linux`

[26] NVIDIA CUDA Visual Profiler

`http://developer.nvidia.com/cuda-toolkit-40rc2#Linux`

[27] NVIDIA CUDA Reference Manual

`http://developer.nvidia.com/cuda-toolkit-40rc2#Linux`

[28] NVIDIA CUDA Programming Guide, *Appendix B.3.2*

`http://developer.nvidia.com/cuda-toolkit-40rc2#Linux`

[29] Georgios Chatzivretas, Ioannis Papaefstathiou, *A Reconfigurable Architecture for Optin Pricing using the Crank-Nicholson Sceme*

[30] Bernard Mawah, *Option Pricing with Transaction Costs and a non-linear Black-Scholes equation*

[31] C.B.O.E, History of C.B.O.E.
`http://www.cboe.com/aboutcboe/History.aspx`

[32] Paul Wilmott, Sam Howison, Jeff Dewynne *Mathematics of Financial Derivatives*

[33] Graig Kolb, Matt Pharr *Option Pricing on the GPU, Gems 2*

[34] Harold S. Stone, *Parallel Tridiagonal Equation Solvers*

[35] Garry H. Rodrigue, Niel K. Madsen and Jack I. Karush
Odd-Even Reduction for Banded Linear Equations

[36] Intel VTune, *VTune Amplifier XE*
`http://software.intel.com/en-us/articles/intel-vtune-amplifier-xe/`

[37] Intel VTune, *VTune User guide*
`http://software.intel.com/sites/products/documentation/hpc/atom/application/vtune_user_guide.pdf`

[38] GNU, *GNU Profiler*
`http://www.cs.utah.edu/dept/old/texinfo/as/gprof.html`