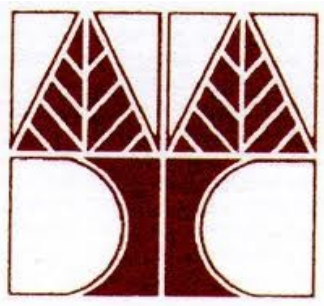
Ατομική Διπλωματική Εργασία

ΣΥΓΚΡΙΣΗ ΕΠΙΔΟΣΗΣ ΚΑΙ ΠΡΟΓΡΑΜΜΑΤΙΣΜΟΥ ΜΕ ΤΗΝ ΧΡΗΣΗ ΤΗΣ OPENACC ΚΑΙ ΤΗΣ CUDA

Ανδρέας Κωνσταντίνου

ΠΑΝΕΠΙΣΤΗΜΙΟ ΚΥΠΡΟΥ



ΤΜΗΜΑ ΠΛΗΡΟΦΟΡΙΚΗΣ

Μάιος 2013

ΠΑΝΕΠΙΣΤΗΜΙΟ ΚΥΠΡΟΥ

ΤΜΗΜΑ ΠΛΗΡΟΦΟΡΙΚΗΣ

Σύγκριση Επίδοσης και Προγραμματισμού με την χρήση της OpenACC και της CUDA

Ανδρέας Κωνσταντίνου

Επιβλέπων Καθηγητής
Pedro Trancoso

Η Ατομική Διπλωματική Εργασία υποβλήθηκε προς μερική εκπλήρωση των απαιτήσεων απόκτησης του πτυχίου Πληροφορικής του Τμήματος Πληροφορικής του Πανεπιστημίου Κύπρου

Μάιος 2013

Ευχαριστίες

Με την ολοκλήρωση της εργασίας αυτής θα ήθελα να εκφράσω τις ευχαριστίες μου στον επιβλέποντα καθηγητή μου κ. Pedro Trancoso για την ευκαιρία που μου έδωσε να ασχοληθώ με το συγκεκριμένο θέμα και για όλη την υποστήριξη και καθοδήγηση που μου παρείχε όλο το διάστημα που εργάστηκα για να διεκπεραιώσω την εργασία αυτή. Επίσης, θα ήθελα να εκφράσω την ευγνωμοσύνη μου στον διδακτορικό φοιτητή Ανδρέα Διαβαστό για τον χρόνο που αφιέρωσε για να με βοηθήσει όπου χρειάστηκα την βοήθεια του, καθώς και τον Post-Doc ερευνητή του Ινστιτούτου Μηχανικής Συστημάτων και Υπολογιστών Έρευνας και Ανάπτυξης της Λισσαβόνας(Instituto de Engenharia de Sistemas e Computadores Investigação e Desenvolvimento em Lisboa - INESC-ID Lisboa) κ. Frederico Pratas για το έτοιμο κώδικα της Cuda που μου έστειλε αλλά και τις χρήσιμες υποδείξεις του σε οποία προβλήματα αντιμετώπισα.

Περίληψη

Η αυξημένη τάση που υπάρχει στις μέρες μας να χρησιμοποιούνται οι επιταχυντές(accelerators) στο GPGPU προγραμματισμό είχε σαν αποτέλεσμα να αναπτυχθούν διεπαφές προγραμματισμού(APIs) χαμηλού επιπέδου(OpenCL, CUDA, etc.). Όμως ο προγραμματισμός με αυτές τις Διεπαφές Προγραμματισμού γίνεται αρκετά συχνά κουραστική, χρονοβόρα και άρα μη παραγωγική διαδικασία. Το πρότυπο OpenACC είναι ένα υψηλού επιπέδου προγραμματιστικό μοντέλο που χρησιμοποιεί οδηγίες μεταγλωττιστή για να καθορίσει βρόγχους και περιοχές κώδικα σε C, C++ και Fortan τις οποίες θα εκτελέσει ο επιταχυντής. Πήραμε τρία προγράμματα τα οποία προσπαθήσαμε να τρέξουμε παράλληλα και να επιτύχουμε όσο το δυνατό υψηλότερη επίδοση, χρησιμοποιώντας την OpenACC. Στην συνέχεια αφού τρέξαμε τα προγράμματα αυτά και με την χρήση της CUDA(χαμηλού επιπέδου API) συγκρίναμε τις επιδόσεις της OpenACC και της CUDA με την σειριακή εκτέλεση των προγραμμάτων. Πιο κάτω θα παρουσιάσουμε τα αποτελέσματα της δουλειάς αυτής χρησιμοποιώντας τις μετρήσεις που πήραμε. Για κάθε εκτέλεση του κάθε προγράμματος αυξάναμε το μέγεθος των αρχείων εισόδου που χρειαζόταν για να διαβάσει τα δεδομένα από μέσα.

Στόχος της εργασίας αυτής ήταν να έρθουμε σε επαφή με την καινούργια OpenACC και να συγκρίνουμε την επίδοση της με άλλη διεπαφή προγραμματισμού χαμηλότερου επιπέδου, όπως είναι η CUDA, έτσι ώστε να δούμε τα πλεονεκτήματα που μας προσφέρει η OpenACC. Τα τελικά αποτελέσματα στην επίδοση της OpenACC ήταν κοντά με τα αποτελέσματα στην επίδοση της Cuda. Για το πρώτο πρόγραμμα η επίδοση της OpenACC σε σχέση με την επίδοση της CUDA ήταν κατά μέσο όρο στο 58% για τα 7 αρχεία εισόδου. Για το δεύτερο πρόγραμμα ήταν κατά μέσο όρο στο 76% και για το τρίτο ήταν στο 81%. Από τα αποτελέσματα αυτά καταφέραμε να δείξουμε πως το μεγάλο πλεονέκτημα της OpenACC είναι πως μας δίνει πολύ καλά αποτελέσματα με μικρή προσπάθεια καταφέρνοντας να αξιοποιήσει με αποτελεσματικό τρόπο τις κάρτες γραφικών.

Περιεχόμενα

Κεφάλαιο 1

Εισαγωγή

1.1 Παράλληλη Επεξεργασία

Σύμφωνα με τους Almasi και Gottlieb [6] ο ορισμός ενός παράλληλου υπολογιστή είναι ο εξής: «Ένας παράλληλος υπολογιστής είναι μια συλλογή επεξεργαστικών στοιχείων τα οποία επικοινωνούν και συνεργάζονται μεταξύ τους, με απώτερο σκοπό την επίλυση μεγάλων προβλημάτων γρήγορα.». Από τον ορισμό αυτό προκύπτουν πολλά ερωτήματα. Πώς αυτά τα επεξεργαστικά στοιχεία θα επικοινωνούν μεταξύ τους, ποιος θα είναι ο ακριβής αριθμός τους, πως θα συνεργάζονται για να επιλύσουν ένα πρόβλημα, ποσό ποιο γρήγορα θα επιλύουν ένα πρόβλημα σε σχέση με την χρήση ενός απλού επεξεργαστικού στοιχείου. Επιπλέον ένα πολύ σημαντικό θέμα, κρίσιμης σημασίας, είναι πόσο εύκολα και με ποιο τρόπο ένας προγραμματιστής θα διαχειρίζεται τα επεξεργαστικά αυτά στοιχεία.

Η βασικότερη αρχή της παράλληλης επεξεργασίας είναι πως ως επί το πλείστον τα μεγάλα προβλήματα που είναι για να επιλυθούν μπορούν να διασπαστούν σε μικρότερα προβλήματα που μπορούν με την σειρά τους να επιλυθούν παράλληλα. Άρα κατά την διαδικασία της παράλληλης επεξεργασίας μπορούμε να πούμε πως συμβαίνουν την ίδια στιγμή πολλές λειτουργίες παράλληλα.

Στις μέρες μας οι εφαρμογές γίνονται μεγαλύτερες και δυσκολότερες ενώ ταυτόχρονα ο όγκος δεδομένων που πρέπει να δουλέψει πάνω του μια εφαρμογή έφτασε σε τεράστια νούμερα που ένας απλός επεξεργαστής αδυνατεί να ανταποκριθεί στους υπολογισμούς που πρέπει να γίνουν. Ακόμα και ο συνδυασμός μερικών επεξεργαστών, οι οποίοι είναι μάλιστα πολυπύρηνοι, δεν καταφέρνει να αντεπεξέλθει ώστε να επιλυθεί κάποιο μεγάλο και δύσκολο πρόβλημα γρήγορα. Αυτό έχει σαν αποτέλεσμα να γίνετε αναγκαία η χρήση συσκευών, όπως είναι οι κάρτες γραφικών, οι οποίες διαθέτουν εκατοντάδες πυρήνες. Η πρόκληση είναι τεράστια αφού αν αξιοποιηθούν με το καταλληλότερο τρόπο τέτοιου είδους συσκευές καταφέρνουμε να επιλύσουμε πολλά δύσκολα και μεγάλα προβλήματα.

1.2 Αρχιτεκτονικές και Μοντέλα Παράλληλου Προγραμματισμού

Υπάρχουν τρία βασικά είδη αρχιτεκτονικών παράλληλου προγραμματισμού. Πρώτο είδος έχουμε τους υπολογιστές μοιραζόμενης μνήμης ή Πολυεπεξεργαστές. Οι πολυεπεξεργαστές έχουν πρόσβαση σε ένα καθολικό χώρο φυσικών διευθύνσεων και εργάζονται ανεξάρτητα αλλά μοιράζονται τον ίδιο πόρο μνήμης. Οι αλλαγές σε περιοχές μνήμης που προκαλεί ο ένας επεξεργαστής είναι ορατές σε όλους τους επεξεργαστές.

Δεύτερο είδος αρχιτεκτονικών παράλληλου προγραμματισμού είναι οι Υπολογιστές Κατανεμημένης Μνήμης ή Πολύ-Υπολογιστές. Οι αρχιτεκτονικές κατανεμημένης μνήμης διαφέρουν μεταξύ τους, αλλά παρουσιάζουν ένα κοινό χαρακτηριστικό: κάθε επεξεργαστής έχει τοπική (ιδιωτική) μνήμη ενώ η επικοινωνία με απομακρυσμένη μνήμη (τοπική μνήμη των άλλων επεξεργαστών) επιτυγχάνεται μέσω δικτύου και συνήθως με τη ρητή συμμετοχή του απομακρυσμένου επεξεργαστή. Ο κάθε επεξεργαστής έχει τον ιδιωτικό του χώρο διευθύνσεων, έτσι δεν υπάρχει η έννοια του καθολικού χώρου διευθύνσεων, κοινού για όλους τους επεξεργαστές. Αφού ο κάθε επεξεργαστής έχει τη δική του τοπική μνήμη, πιθανές τροποποιήσεις σε θέσεις μνήμης ενός επεξεργαστή δεν επηρεάζουν τη μνήμη άλλων επεξεργαστών. Η κρυφή μνήμη λειτουργεί ακριβώς όπως και στα συμβατικά συστήματα και δεν τίθεται ζήτημα συνοχής της κρυφής μνήμης(Cache Cohernece).Όταν ένας επεξεργαστής

πρέπει να προσπελάσει δεδομένα απομακρυσμένης μνήμης, συνήθως ο προγραμματιστής πρέπει ρητά να ορίσει πότε και ποια δεδομένα θα προσπελαστούν. Παρόμοια ο συγχρονισμός μεταξύ παράλληλων εργασιών σε διαφορετικούς επεξεργαστές γίνεται ρητά από το πρόγραμμα. Οι τεχνολογίες δικτύου που χρησιμοποιούνται κυμαίνονται από απλό Ethernet μέχρι ειδικά υπερταχέα δίκτυα. Επίσης η επικοινωνία μπορεί να είναι καθολική ή σημειακή, δηλαδή μπορεί να έχουμε ομοιόμορφη και ανομοιόμορφη επικοινωνία.

Τρίτο είδος είναι οι Υβριδικοί Παράλληλοι Υπολογιστές. Οι μεγαλύτεροι και ταχύτεροι σύγχρονοι υπολογιστές συνδυάζουν τις αρχιτεκτονικές μοιραζόμενης και κατανεμημένης μνήμης. Τυπικά, στην απλούστερη εκδοχή έχουμε Πολυπύρηνους επεξεργαστές συνδεδεμένους μέσω ενός γρήγορου τοπικού δικτύου. Εναλλακτικά, ο κάθε κόμβος μπορεί να διαθέτει και GPUs ως συνεπεξεργαστές τύπου SIMD, επιτρέποντας μια σύνθετη αρχιτεκτονική που περιλαμβάνει Distributed Memory MIMD, Shared Memory MIMD και SIMD συστήματα στον ίδιο παράλληλο υπολογιστή.

Τα μοντέλα παράλληλου προγραμματισμού παρέχουν ένα επίπεδο αφαίρεσης μεταξύ των παράλληλων αλγορίθμων και των παράλληλων αρχιτεκτονικών, ώστε να είναι δυνατή η σχεδίαση και συγγραφή όσο το δυνατό πιο ευέλικτου και μεταφερτού κώδικα. Δεν φαίνεται ξεκάθαρα αλλά τα μοντέλα προγραμματισμού δεν συνδέονται άμεσα με συγκεκριμένες αρχιτεκτονικές. Το κάθε μοντέλο υλοποιείται ευκολότερα σε κάποιες αρχιτεκτονικές αλλά θεωρητικά το κάθε μοντέλο μπορεί να υλοποιηθεί σε οποιαδήποτε αρχιτεκτονική. Η χρήση ή επικράτηση κατά καιρούς κάποιων μοντέλων συχνά σχετίζεται με τη διαθέσιμη τεχνολογία και άλλες συγκυριακές επιλογές. Υπάρχουν αρκετά μοντέλα παράλληλου προγραμματισμού, όμως αυτά που έχουν επικρατήσει είναι το Μοντέλο Νημάτων και το Μοντέλο Μεταβίβασης Μηνυμάτων. Το Μοντέλο Νημάτων υλοποιείτε από την OpenMP με οδηγίες προς τον μεταγλωττιστή και το POSIX Threads με ένα σύνολο από βιβλιοθήκες. Το Μοντέλο Μεταβίβασης Μηνυμάτων υλοποιείτε από τη Διεπαφή Μεταβίβασης Μηνυμάτων (Message Passing Interface, MPI).

1.3 Διεπαφή Προγραμματισμού Εφαρμογών OpenACC

Η Διεπαφή Προγραμματισμού Εφαρμογών(Application Programming Interface, API) OpenACC περιγράφει μια συλλογή από οδηγίες μεταγλωττιστή (compiler directives) για να καθορίσει βρόγχους (loops) και περιοχές (regions) κώδικα σε C, C++ και Fortran τις οποίες θα εκφορτώσει σε ένα συνδεδεμένο επιταχυντή, παρέχοντας τη δυνατότητα μεταφοράς σε

λειτουργικά συστήματα, επεξεργαστές υποδοχής (host CPUs) και επιταχυντές. Οι περισσότερες OpenACC οδηγίες ισχύουν για το αμέσως επόμενο δομημένο block ή loop. Ένα δομημένο block είναι μια απλή δήλωση ή μια δήλωση ένωσης (C και C++) ή μια ακολουθία καταστάσεων (Fortran) με ένα ενιαίο σημείο εισόδου στην κορυφή και ένα ενιαίο σημείο εξόδου στο κάτω μέρος.

Το μοντέλο αυτό είναι εύκολα κατανοητό και παρέχει την δυνατότητα να παραλληλοποιήσουμε εφαρμογές με απλό τρόπο επιτυγχάνοντας αύξηση στην επίδοση βάζοντας ελάχιστο κόπο στις πλείστες περιπτώσεις. Με ελάχιστες πρόσθετες γραμμές κώδικα μπορούμε να εκμεταλλευτούμε τις τεράστιες δυνατότητες των καρτών γραφικών. Σε αντίθεση με άλλα παράλληλα προγραμματιστικά μοντέλα, όπως η CUDA και η OpenCL, η OpenACC είναι εύκολα κατανοητή και η εκμάθηση της είναι απλή ακόμα και από άπειρους σχετικά προγραμματιστές δίνοντας την δυνατότητα σε μικρο χρονικό διάστημα να αυξάνουμε την επίδοση ενός σειριακού προγράμματος χωρίς να προσθέτουμε εκατοντάδες επιπλέον γραμμές κώδικα και να διαφοροποιούμε την δομή ενός προγράμματος. Προς παρόν το νέο αυτό πρότυπο υποστηρίζετε μόνο από κάρτες γραφικών της Nvidia.

1.4 Αντικείμενο της Εργασίας

Στο μοντέλο προγραμματισμού της OpenACC ο προγραμματιστής είναι υπεύθυνος να διακρίνει τις περιοχές κωδικά μέσα σε ένα σειριακό πρόγραμμα έτσι ώστε χρησιμοποιώντας τις κατάλληλες οδηγίες προς τον μεταγλωττιστή να καταφέρει να παράξει το μέγιστο δυνατό παραλληλισμό αξιοποιώντας στο έπακρο την ή τις συσκευές που έχει στην διάθεση του(CPU, GPU). Αν και υπάρχουν ήδη διάφορα μοντέλα παράλληλου προγραμματισμού όπως είναι η CUDA και η OpenCL, που αξιοποιούν με εξαιρετικό τρόπο τις κάρτες γραφικών, εν τούτοις σε πολλές περιπτώσεις η προσπάθεια που καταβάλλετε για να αξιοποιηθούν είναι επίπονη και χρειάζονται άριστες γνώσεις της αρχιτεκτονικής των υπολογιστών. Με την καινοτομία που μας προσφέρει η καινούργια OpenACC, δηλαδή να μπορούμε να εκμεταλλευτούμε τις τεράστιες υπολογιστικές δυνατότητες μιας κάρτας γραφικών με απλή προσθήκη ενός η μερικών οδηγιών προς τον μεταγλωττιστή και χωρίς να κατέχουμε άριστες γνώσης της αρχιτεκτονικής ενός υπολογιστή, η παραλληλοποίηση ενός μεγάλου και δύσκολου σειριακού κώδικα γίνετε απλή και γρήγορη.

1.5 Στόχοι

Στόχος μας στην εργασία αυτή είναι να δείξουμε τις δυνατότητες της OpenACC η οποία μας προσφέρει μια απλότητα στον παράλληλο προγραμματισμό. Προσθέτοντας λοιπόν μερικές γραμμές κώδικα σε μεγάλες σειριακές εφαρμογές στοχεύουμε να επιτύχουμε πολύ καλές επιδόσεις που θα πλησιάζουν επιδόσεις που δίνει το μοντέλο της CUDA στο οποίο όμως για να πάρουμε τις μεγάλες επιδόσεις θα χρειαστεί να διαφοροποιήσουμε σε μεγάλο βαθμό το σειριακό πρόγραμμα και να προσθέσουμε εκατοντάδες γραμμές κώδικα.

Μέσα από την πιο πάνω προσπάθεια που θα κάνουμε θα καταγράψουμε τον χρόνο που θα χρειαστούμε για να παραλληλοποιήσουμε τον σειριακό κώδικα στοχεύοντας να δείξουμε πόσο γρήγορα μπορούμε να παραλληλοποιήσουμε ένα σειριακό πρόγραμμα με την χρήση του μοντέλου της OpenACC. Μια άλλη δυνατότητα της OpenACC που στοχεύουμε να δείξουμε είναι η φορητότητα. Η OpenACC δουλεύει τόσο με την χρήση κεντρικών μονάδων επεξεργασίας(CPUs) όσο και με την χρήση κάρτων γραφικών(GPUs) καθώς επίσης και με συνδυασμό των δύο παρέχοντας έτσι μια φορητότητα ως προς την συσκευή που μας βολεύει να χρησιμοποιήσουμε.

1.6 Οργάνωση και Μεθοδολογία

Στην εργασία αυτή ως πρώτο βήμα ασχοληθήκαμε με την μελέτη του εγχειριδίου της OpenACC, διάφορων διαλέξεων που δημοσίευσε η Nvidia καθώς και διάφορων επιστημονικών άρθρων που ασχολούνται με την μελέτη του καινούργιου αυτού μοντέλου. Στην συνέχεια πήραμε 3 διαφορετικά Queries, που θα εξηγήσουμε τον αλγόριθμο τους παρακάτω, τα οποία ήταν γραμμένα για να εκτελούνται σειριακά και εισαγάγατε τις απαραίτητες οδηγίες προς τον μεταγλωττιστή ώστε να καταφέρουμε να τα τρέξουμε παράλληλα και να πάρουμε τους χρόνους εκτέλεσης τους. Τρέξαμε το κάθε Query με 7 διαφορετικά μεγέθη αρχείων σαν είσοδο από όπου θα διάβαζε το πρόγραμμα τα δεδομένα. Ακολούθησε μια σύγκριση με τους χρόνους εκτέλεσης για τον σειριακό κώδικα. Για να αποδείξουμε τα πλεονεκτήματα της OpenACC πήραμε την παράλληλή έκδοση των Queries αυτών που γράφτηκαν από έμπειρο προγραμματιστή για το μοντέλο της CUDA, και τα τρέξαμε έτσι να ώστε να γίνει μια σύγκριση ανάμεσα σε δύο πρότυπα που αξιοποιήσουν τις κάρτες γραφικών της Nvidia με διαφορετικό τρόπο το κάθε ένα.

Η εκδοσή των Queries σε CUDA που τα πήραμε έτοιμα είναι μια εκδοσή με τις μέγιστες δυνατές βελτιστοποιήσεις. Σαν δεύτερο στάδιο προσθέσαμε βελτιστοποιήσεις του κώδικα για την OpenACC εκδοσή των Queries. Ξανασυγκρίναμε την OpenACC με την CUDA και καταλήξαμε σε επιδόσεις πού αφορούσαν την καλύτερη προσπάθεια και για τις δύο εκδόσεις. Για να δικαιολογήσουμε στην κάθε περίπτωση την καλύτερη επίδοση που παρατηρητών στην CUDA αφαιρέσαμε κάποιες συναρτήσεις που χρησιμοποιεί η CUDA και δεν μπορούμε κατά αντίστοιχο τρόπο να τις χρησιμοποιήσουμε στην OpenACC. Αφαιρέσαμε δηλαδή κάποια πλεονεκτήματα της CUDA μειώνοντας έτσι την επίδοσή της και συγκρίναμε ξανά τις δυο εκδόσεις και καταλήξαμε στα τελικά μας συμπεράσματα. Σε κάθε μια από τις 3 φάσεις που περάσαμε συλλέξαμε τους χρόνους εκτέλεσης και παρουσιάσαμε με γραφικές παραστάσεις τις επιταχύνσεις που πήραμε για OpenACC και CUDA σε σχέση με την σειριακή εκτέλεση καθώς και το ποσοστό τοις εκατόν της επίδοσης της OpenACC σε σχέση με την CUDA.

1.7 Δομή της Εργασίας

Η εργασία αυτή χωρίζεται σε 6 κεφάλαια μέσα από τα οποία περιγράφεται το αντικείμενο με το οποίο ασχοληθήκαμε και παρουσιάζονται τα αποτελέσματα και τα συμπεράσματα που εξάγαμε μέσα από την δουλειά αυτή. Στο Κεφάλαιο 2 από τις διάφορες πήγες που αναφέρονται στην βιβλιογραφία μελετήσαμε σχετικές δουλειές που έχουν γίνει για το θέμα με το οποίο ασχολούμαστε και τις περιγράφουμε. Στο Κεφάλαιο 3 γίνεται μια εισαγωγή στις κάρτες γραφικών, αναλύοντας το τι είναι μια κάρτα γραφικών, την ιστορία της καθώς και την αρχιτεκτονική Fermi. Επίσης γίνεται και μια αναφορά στον GPGPU προγραμματισμό. Στο Κεφάλαιο 4 κάνουμε μια εισαγωγή στο μοντέλο προγραμματισμού της OpenACC. Αναφέρεται το μοντέλο εκτέλεσης, το μοντέλο μνήμης, διάφορα constructs και runtime routines του προτύπου αυτού. Επίσης στο 4ο Κεφάλαιο θα δούμε μέσα από ένα απλό παράδειγμα πως χρησιμοποιούμε την OpenACC. Στο Κεφάλαιο 5 αναλύονται οι τρεις εφαρμογές που παραλληλοποιήσαμε με την χρήση της OpenACC παρουσιάζοντας σε γραφικές παραστάσεις τα αποτελέσματα που βγάλαμε για κάθε φάση που περάσαμε για να φτάσουμε στο τελικό μας αποτέλεσμα. Στο 6ο και τελευταίο Κεφάλαιο αναφέρονται τα συμπεράσματα που βγάλαμε μέσα από την εργασία αυτή καθώς και μελλοντική εργασία που μπορεί να συμπληρώσει την εργασία αυτή.

Κεφάλαιο 2

Ανάλυση Σχετικών Εργασιών

Για να καταφέρουμε να ολοκληρώσουμε την εργασία αυτή και να πετύχουνε τους στόχους που έχουμε θέσει εξ αρχής έπρεπε πρώτα να μελετήσουμε κάποια άρθρα που ασχολούνται με το θέμα το οποίο ασχολούμαστε και εμείς. Έτσι επιλέξαμε συγκεκριμένα άρθρα τα οποία μας έδωσαν μια πρώτη εικόνα για την δουλειά που έχει γίνει μέχρι τώρα στο θέμα αυτό και πήραμε μια ιδέα για τον τρόπο που θα εργαστούμε για να αποπερατώσουμε το έργο μας. Τα άρθρα που διαβάσαμε ασχολούνται με τις πρώτες γνώσεις που πρέπει να πάρει ένας νέος χρήστης της OpenACC[9-11] και με συγκρίσεις του μοντέλου της OpenACC με άλλα μοντέλα[1-4] και καταγράφονται στην βιβλιογραφία που βρίσκεται στο τελευταίο τμήμα της εργασίας αυτής.

Το μοντέλο της OpenACC είναι ένα μοντέλο παράλληλου προγραμματισμού όπου κάθε διεργασία έχει πολλά νήματα. Τα νήματα αυτά εκτελούν το ρόλο μιας διεργασίας το καθένα και περιέχουν ροές εντολών που έχουν κοινό χώρο διευθύνσεων. Έχουν επίσης ιδιωτική στοίβα κλήσεων και κατάσταση επεξεργαστή. Τα νήματα αυτά δημιουργούνται σειριακά από ια διεργασία και εκτελούνται παράλληλα. Τερματίζονται από το κυρίως πρόγραμμα. Προγραμματιστικά το μοντέλο αυτό υλοποιείτε με οδηγίες προς τον μεταγλωττιστή και συναρτήσεις βιβλιοθηκών. Δύο από τις βασικότερες οδηγίες προς τον μεταγλωττιστή που χρησιμοποιούνται στην OpenACC είναι το parallel construct και το kernels construct[9]. Το kernels construct προέρχεται από το PGI Accelerator μοντέλο. Το εσωτερικό loop στο kernels construct μετατρέπετε από το μεταγλωττιστή σε παράλληλους kernels που τρέχουν αποτελεσματικά στην GPU. Υπάρχουν τρία διαφορετικά βήματα σε αυτή την διαδικασία που ακολουθούνται. Το πρώτο βήμα είναι να προσδιοριστούν τα loops που μπορούν να

εκτελεστούν παράλληλα. Το δεύτερο βήμα είναι να γίνει map ο αφηρημένος loop παραλληλισμός σε συγκεκριμένο παραλληλισμό υλικού(hardware). Για μια Nvidia CUDA GPU αυτό σημαίνει να γίνει mapping ένα παράλληλο loop σε grid-level παραλληλισμό(blockidx) ή thread-level παραλληλισμό(threadidx). Με OpenACC όρους αυτό σημαίνει ο gang παραλληλισμός σε grid-level παραλληλισμό και ο vector παραλληλισμός σε thread-level παραλληλισμό. Ο μεταγλωττιστής μπορεί να χαρτογραφήσει(mapping) ένα loop σε πολλαπλά επίπεδα παραλληλισμού με την χρήση strip-mining. Τέλος στο τρίτο βήμα ο μεταγλωττιστής πρέπει να παράξει και να βελτιστοποιήσει το πραγματικό κώδικα για να εφαρμόσει το επιλεγμένο χαρτογραφημένο παραλληλισμό. Το parallel construct προέρχεται από το OpenMP parallel construct. Ένα OpenMP parallel construct δημιουργεί ένα αριθμό από παράλληλα νήματα που αμέσως ξεκινούν να εκτελούν το σώμα του parallel construct. Όταν ένα νήμα φτάσει στο work-sharing loop αυτό το νήμα θα εκτελέσει ένα υποσύνολο των επαναλήψεων του loop. Στο τέλος του loop τα νήματα συγχρονίζονται με ένα barrier εκτός και αν ο χρήστης ρητώς δηλώσει ότι δεν απαιτείτε barrier. Κατά τον ίδιο τρόπο το OpenACC parallel construct δημιουργεί ένα αριθμό parallel gangs που αμέσως αρχίζουν την εκτέλεση του σώματος του construct. Όταν ένα gang φτάσει σε ένα work-sharing loop αυτό το gang θα εκτελέσει ένα υποσύνολο των επαναλήψεων του loop. Μια διαφορά ανάμεσα στο OpenACC parallel construct και στο OpenMP parallel construct είναι ότι στο OpenACC δεν υπάρχει barrier στο τέλος ενός work-sharing loop.

Κάθε διεπαφή προγραμματισμού χρειάζεται και τον κατάλληλο μεταγλωττιστή για να δουλέψει σωστά. Στην OpenACC χρησιμοποιούνται οι PGI μεταγλωττιστές[10]. Το μοντέλο προγραμματισμού PGI είναι παρόμοιο με το μοντέλο προγραμματισμού της OpenACC, αν και υπάρχουν σημαντικές διαφορές μεταξύ τους αφού το OpenACC μοντέλο υποστηρίζει κάποια πράγματα που δεν υποστηρίζει το PGI μοντέλο. Το OpenACC μοντέλο υποστηρίζει το parallel construct, το present data clause, asychronous data movement τόσο καλά όπως asychronous computation και υποστηρίζει ακόμα τρία επίπεδα παραλληλισμού(gang, worker, vectror). Η στοχευμένη αρχιτεκτονική του μοντέλου είναι μια συλλογή στοιχείων επεξεργασίας(PEs), όπου κάθε PE μπορεί να εκτελέσει vector εντολές. Για μια Nvidia GPU, τα PEs μπορούν να γίνουν map στους Streaming Multiprocessors, και η vector διάσπαση μπορεί να γίνει map σε ένα warp. Η gang διάσπαση μπορεί να γίνει map μέσα στα PEs, οι workers μέσα στην πολυνηματική διάσπαση μέσα σε ένα PE, και η vector διάσπαση σε vector εντολές. Δεν υπάρχει υποστήριξη για οποιοδήποτε συγχρονισμό ανάμεσα σε gangs, δεδομένου ότι οι workers θα πρέπει να εκτελεστούν από το ίδιο PE, και θα μοιράζονται τους πόρους(όπως data caches), που θα κάνουν την πρόσβαση στα κοινά δεδομένα πιο αποτελεσματικά.

Ο Nvidia Fermi GPU είναι ο πρώτος στόχος των PGI compilers [11]. Οι Nvidia GPUs έχουν περισσότερους από 16 multiprocessors. Ο κάθε multiprocessor έχει δύο SIMD units, κάθε unit με 16 παράλληλα thread processors όπου κάθε processor τρέχει συγχρονισμένα με τους υπόλοιπους. Κάθε GPU έχει την δική του μνήμη που είναι μεγαλύτερη από 6GB σήμερα. Αν και υπάρχει η CUDA και η OpenCL που μπορούν αποτελεσματικά να εκμεταλλευτούν αυτή την αρχιτεκτονική με την OpenACC μπορούμε ευκολότερα και σε λιγότερο χρόνο να εξοικειωθούμε μαζί της. Επίσης αγνοώντας της οδηγίες προς τον μεταγλωττιστή ένα πρόγραμμα μπορεί να τρέξει και στην CPU πράγμα το οποίο δεν γίνετε στην CUDA και στην OpenCL.

Στοχεύοντας να πάρουμε μια εικόνα για το πως μπορούμε να διεκπεραιώσουμε την εργασία μας μελετήσαμε διάφορα άρθρα που κάνουν σύγκριση του προγραμματιστικού μοντέλου της OpenACC με άλλα μοντέλα καθώς και άρθρα που ασχολούνται με τον συνδυασμό της χρήσης του μοντέλου της OpenACC με άλλα μοντέλα. Αν και τα επιστημονικά άρθρα που ασχολούνται με το θέμα της OpenACC ήταν περιορισμένα, αφού η OpenACC είναι πολύ καινούργια και πολύ ερευνητές τώρα βρίσκονται στο στάδιο μελέτης του προτύπου αυτού, καταφέραμε να πάρουμε μια ιδέα για τον τρόπο με τον οποίο προσεγγίζετε η OpenACC στην επιστημονική κοινότητα.

Συνδυάζοντας την καινούργια OpenACC με την OpenMP[1] μια ομάδα από επιστήμονες κατάφεραν να εκτελέσουν μια εφαρμογή σε μικρότερους χρόνους αξιοποιώντας τα πλεονεκτήματα που δίνουν και τα δυο αυτά μοντέλα παράλληλου προγραμματισμού. Οι επιστήμονες αυτοί κατάφεραν με την χρήση της OpenACC να μετατρέψουν την εφαρμογή τους από ένα level παραλληλισμό σε 2 level παραλληλισμού πετυχαίνοντας έτσι τα καλύτερα αποτελέσματα αξιοποιώντας με καλύτερο τρόπο τους επεξεργαστές και τις κάρτες γραφικών. Το μόνο που έκαναν για να πάρουν καλύτερα αποτελέσματα ήταν να αλλάξουν μερικές λέξεις σε προγράμματα που παραλληλοποιήθηκαν με την χρήση της OpenMP ώστε να εκτελεστούν με την χρήση της OpenACC. Οι οδηγίες μεταγλωττιστή της OpenACC παρέχουν ένα μηχανισμό για να μείνει μια εφαρμογή γραμμένη σε υψηλού επιπέδου γλώσσα(OpenMP), ένα χαρακτηριστικό πολύ σημαντικό για να μεταφερθούν πολλά λογισμικά που μας έμειναν σαν “κληρονομιά” σε πολυπύρηνους επεξεργαστές και συσκευές επιτάχυνσης(GPUs).

Αν και ο συνδυασμός της OpenACC με άλλα μοντέλα όπως την OpenMP είναι πολύ χρήσιμος εν τούτοις για να καταλάβουμε την μεγάλη αξία του προτύπου της OpenACC

χρειάστηκε να δούμε μερικές συγκρίσεις με πρότυπα όπως είναι η OpenCL και η CUDA τα οποία αξιοποιούν σε εξαιρετικά μεγάλο βαθμό τις κάρτες γραφικών. Μια ομάδα από ερευνητές στο Πανεπιστήμιο Aachen της Γερμανίας είχαν μια πρώτη εμπειρία με το καινούργιο μοντέλο της OpenACC συγκρίνοντας το με το μοντέλο της OpenCL[2] για δύο real-world προσομοιώσεις κώδικα από τους τομείς της μηχανικής και της ιατρικής. Η προσομοίωση κώδικα από τον τομέα της ιατρικής ήταν πιο περίπλοκη από αυτή του τομέα της μηχανικής. Οι συγκρίσεις που έγιναν λοιπόν έδειξαν ότι με λίγη προγραμματιστική προσπάθεια, που έγινε από αυτή την ομάδα ερευνητών, η επίδοση της μηχανικής προσομοίωσης με την χρήση της OpenACC έδωσε 80% της καλύτερης προσπάθειας που έγινε με την OpenCL, ενώ για την πιο περίπλοκη ιατρική προσομοίωση η επίδοση με την χρήση της OpenACC ήταν μόνο στο 40% της καλύτερης προσπάθειας που έγινε με την OpenCL. Για να πάρουν τα αποτελέσματα αυτά χρειάστηκε να γράψουν πολύ περισσότερες γραμμές κώδικα στην περίπτωση που χρησιμοποίησαν την OpenCL, ενώ με την χρήση της OpenACC χρειάστηκε να προσθέσουν/αλλάξουν ελάχιστες γραμμές κώδικα. Έτσι κατέληξαν στο συμπέρασμα πως με την χρήση του μοντέλου της OpenACC μπορούμε να πάρουμε πολύ καλά αποτελέσματα με πολύ μικρή προσπάθεια. Με μια πρώτη ματιά μπορούμε να πούμε ότι το 40% της επίδοσης που δίνει η περίπλοκη ιατρική προσομοίωση φαίνεται οδυνηρό αποτέλεσμα. Όμως σύμφωνα με τους ερευνητές αυτή η απώλεια της επίδοσης οφείλετε κυρίως στην έλλειψη της ικανότητας να αξιοποιηθεί η τοπική μνήμη της GPU και είναι στο χέρι τους να εφαρμόσουν global sychronization για να αποτρέψουν race contitions. Έτσι πιστεύουν πως με την εισαγωγή επιπλέον οδηγιών προς τον μεταγλωττιστή θα αυξηθεί πολύ περισσότερο η επίδοση για την περίπλοκη ιατρική προσομοίωση.

Ερευνητές παραλληλοποίησαν την εφαρμογή του Matrix Multiplication(MxM), και τρείς άλλες εφαρμογές(thermal simulation(HS), LU Decomposition(LUD), nonlinear global optimization method for DNA sequence alignments(NW)) με την χρήση των directive based μοντέλων OpenACC, hiCuda, OpenMP και PGI Accelerator[3]. Συγκρίνοντας αυτά τα μοντέλα παρατήρησαν πως για την εφαρμογή MxM όπου πρόκειτε για floating point επίδοση η OpenACC έδωσε καλύτερη επίδοση από την hiCuda και την OpenMP αλλά όχι και από το PGI Accelerator. Για την HS εφαρμογή παρατήρησαν πως η OpenACC και η hiCuda δίνουν περιπου τα ίδια αποτελέσματα για πιο μεγάλα δεδομένα εισόδου που είναι κατα πολύ καλύτερα σε συγκριση με τα αλλα δυο μοντέλα(OpenMP, PGI), ενω για μικρα δεδομένα εισόδου η OpenACC δινεί αρκετα καλύτερα απότελεσματα από την hiCuda και PGI αλλά όχι και από την OpenMP η οποία για μικρά δεδομένα εισόδου μπορεί να φτάσει σχεδόν και τις επιδόσεις της CUDA. Παρόμοια αποτελέσματα προέκυψαν και για τις άλλες δύο εφαρμογές(LUD, NW).

Το OpenACC API προς το παρόν έχει σχεδιαστεί για να δουλεύει μόνο με κάρτες γραφικών της Nvidia. Υπάρχουν διαφορετικά είδη αρχιτεκτονικών της Nvidia(Fermi, Tesla, Kepler) με τις οποίες μπορεί η OpenACC να δουλέψει. Η κάθε αρχιτεκτονική έχει τα πλεονεκτήματα και τα μειονεκτήματα της. Χρησιμοποιώντας τις αρχιτεκτονικές αυτές πάνω σε διαφορετικά είδη εφαρμογών(Black-Scholes, Monde Carlo, Bonds, Repo) ερευνητές του πανεπιστημίου της Delaware στις Η.Π.Α βρήκαν ότι η κάθε εφαρμογή δουλεύει καλύτερα με συγκεκριμένη αρχιτεκτονική[4]. Οι ερευνητές αυτοί έτρεξαν όλες τις εφαρμογές με την χρήση της OpenACC πάνω σε τέσσερις διαφορετικούς GPUs(C1060(Tesla), C2050(Fermi), GTX670(GK104 Kepler), K20(GK110 Kepler)) και παρατήρησαν ότι η C2050, που έχει τους λιγότερους cores, είχε καλύτερα αποτελέσματα από την GTX670 γαι τρείς εφαρμογές(Black-Scholes, Bonds, Repo). Μια πιθανή εξήγηση γαι αυτό είναι ότι η GTX670 δεν είναι βελτιστοποιημένη για GPGPU προγραμματισμό όπως την C2050. Κατέληξαν λοιπόν στο συμπέρασμα ότι όταν κάποιος προσπαθεί, χρησιμοποιώντας το OpenACC API, να παραλληλοποιήσει μια εφαρμογή και έχοντας στην διάθεση του κάρτες γραφικών διαφορετικής αρχιτεκτονικής πρέπει να είναι σε θέση να ξέρει ποια αρχιτεκτονική θα του φέρει τα καλύτερα αποτελέσματα.

Κεφάλαιο 3

Εισαγωγή στις Κάρτες Γραφικών

3.1 Τί είναι οι Κάρτες Γραφικών

Μια μονάδα επεξεργασίας γραφικών (GPU) είναι ένας ειδικός βελτιστοποιημένος παράλληλος επεξεργαστής για την επιτάχυνση γραφικών υπολογισμών. Η GPU έχει σχεδιαστεί ειδικά για να εκτελεί τους πολλούς floating-point υπολογισμούς που είναι απαραίτητοι για την απόδοση 3D γραφικών. Οι μοντέρνοι GPU επεξεργαστές είναι μαζικά παράλληλοι και πλήρως προγραμματιζόμενοι. Η παράλληλη floating-point υπολογιστική ισχύς που βρέθηκε στον σύγχρονο GPU είναι κατά πολύ υψηλότερη από ότι σε μια CPU. Ένα GPU μπορούμε να τον βρούμε σε ένα ευρύ φάσμα συστημάτων, από προσωπικούς υπολογιστές και φορητούς υπολογιστές σε κινητά τηλέφωνα και υπερυπολογιστές. Με την παράλληλη δομή που διαθέτουν οι GPUs , εφαρμόζουν ένα αριθμό από 2D και 3D αρχέγονη γραφική επεξεργασία στο υλικό(hardware), καθιστώντας τους πολύ πιο γρήγορους από ότι μια CPU γενικής χρήσης σε αυτές τις πράξεις.

3.2 Η ιστορία των Κάρτων Γραφικών

Αρχικά οι Μονάδες Επεξεργασίας Γραφικών σχεδιάστηκαν για τον υπολογισμό των πολύπλοκων γεωμετρικών και μαθηματικών πράξεων που είναι αναγκαίοι για την απόδοση των γραφικών. Κύριος στόχος των GPUs ήταν η μετατροπή των συντεταγμένων από τον 3Δ χώρο στον 2Δ χώρο της οθόνης (graphics pipeline) και η λιγότερη επιβάρυνση της CPU.

Είναι τόσο παλιές όσο σχεδόν η εμφάνιση των πρώτων προσωπικών υπολογιστών. Την αρχή έκανε η IBM το 1981 όταν το IBM PC που κυκλοφόρησε διάθετε κάρτα γραφικών. Η κάρτα αυτή ήταν ένα κύκλωμα (κάρτα) που προέβαλε μόνο κείμενο σε πράσινο ή λευκό χρώμα με μαύρο υπόβαθρο και ονομαζόταν Monochrome Display Adapter(MDA). Λιγά χρόνια μετά(1984) η IBM κυκλοφορεί το IBM Professional Graphics Controller(PGA) που προσφέρει υψηλότερη ανάλυση και βάθος χρώματος υποστηρίζοντας ανάλυση ως και 640x480 με 256 χρώματα και ρυθμό ανανέωσης 60 Hertz. Από τα μέσα της δεκαετίας του 1990 μέχρι και σήμερα υπάρχει μια ραγδαία εξέλιξη. Μια απο τις μεγαλύτερες εταιρίες στο τομέα των κάρτων γραφικών είναι η Nvidia η οποία το 1999 εισήγαγε το όρο GPU για πρώτη φορά. Το 2001 από το fixed function pipeline πάμε στο programmable pipeline(GeForce 3). Η αρχή του GPU computing γίνετε το 2003 μέσω του DirectX 9. Δηλαδή πλέον οι GPUs δεν χρησιμοποιούνται μόνο για την απόδοση των γραφικών αλλά χρησιμοποιούνται και γα γενικού σκοπού υπολογισμούς(General-purpose graphics processing unit - GPGPU).

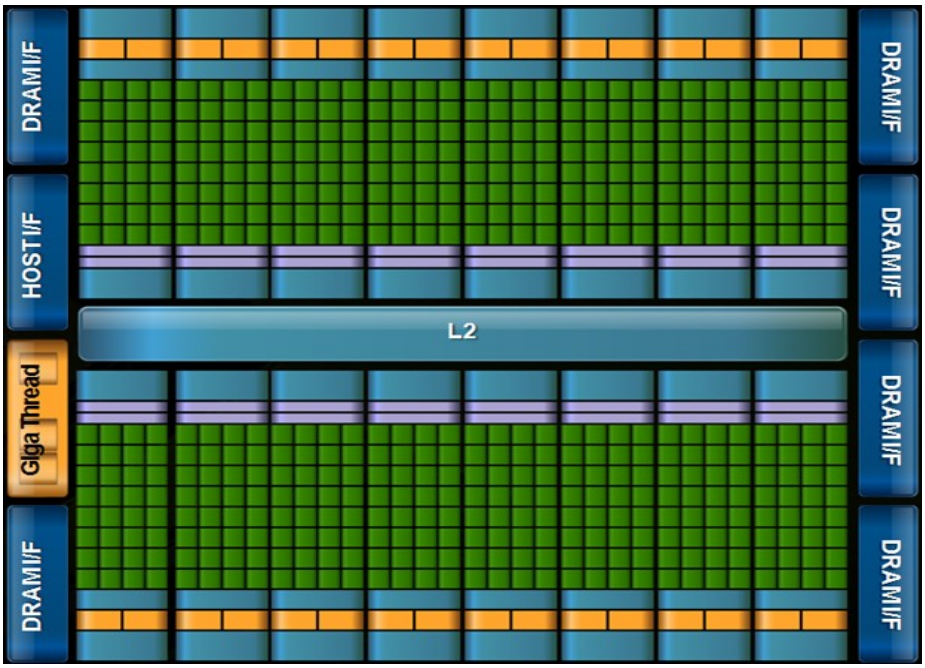
Το 2006 με την νέα σειρά της Nvidia GeForce 8 αντιμετωπίζετε η GPU ως ένα σύστημα παράλληλων επεξεργαστών. Επίσης το 2006 είναι και η αφετηρία της αρχιτεκτονικής CUDA(Compute Unified Device Architecture).

3.3 Nvidia Fermi

Ο GPU σχεδιασμός στοχεύει σε εφαρμογές με πολλαπλά νήματα που κυριαρχούνται από μακρύς σειρές από υπολογιστικές οδηγίες. Η εξέλιξη της τεχνολογίας στον σχεδιασμό GPU αντιπροσωπεύετε από την Nvidia με την αρχιτεκτονική επόμενης γενιάς CUDA, που ονομάζεται Fermi. Το Σχήμα 3.1 δείχνει ένα υψηλού επιπέδου μπλοκ διάγραμμα του πρώτου Fermi chip.

Στην πλατφόρμα λογισμικού Nvidia CUDA τα υπολογιστικά στοιχεία των αλγορίθμων είναι γνωστά σαν kernels. Μια εφαρμογή ή συνάρτηση βιβλιοθήκης μπορεί να αποτελείτε από 1 ή περισσότερους kernels. Όταν συνταχθούν οι kernels αποτελούνται από πολλά

νήματα(threads) που εκτελούν το ίδιο πρόγραμμα παράλληλα. Ένα νήμα είναι σαν επανάληψη ενός βρόγχου. Πολλά νήματα διαχωρίζονται σε threads blocks που περιέχουν μέχρι και 1536 νήματα. Όλα αυτά τα νήματα στο thread block θα τρέξουν σε ένα Streaming Multiprocessor(SM).



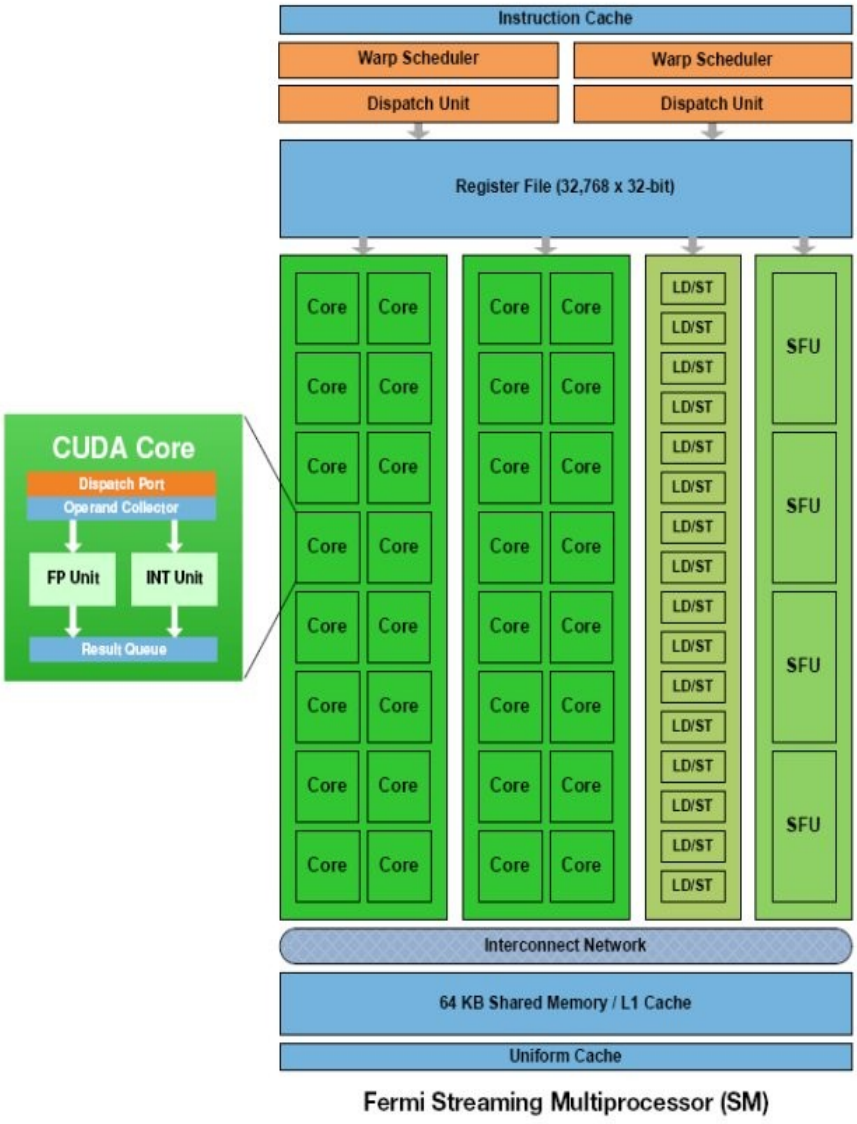
Σχήμα 3.1 : Block Diagram της Αρχιτεκτονικής Fermi (Πηγή:Nvidia)

Τα threads blocks διαιρούνται σε warps των 32 νημάτων. Το warp είναι η θεμελιώδης μονάδα αποστολής μέσα σε ένα SM. Στην Fermi, δύο warps από διαφορετικά threads blocks(ακόμα και διαφορετικούς kernels) μπορούν να εκδοθούν και να εκτελεστούν ταυτόχρονα αυξάνοντας την χρησιμοποίηση του υλικού και την ενεργειακή απόδοση.

Τα threads blocks οργανώνονται σε grids, τα οποία το καθένα θα εκτελέσει ένα μοναδικό kernel. Ανά πάσα στιγμή, ολόκληρη η συσκευή Fermi είναι διαθέσιμη σε μια και μόνο εφαρμογή. Η Fermi υποστηρίζει την ταυτόχρονη εκτέλεση πολλαπλών kernels, και από την ίδια εφαρμογή, όπου κάθε kernel διανέμεται σε ένα ή περισσότερους SMs στην συσκευή. Αυτό έχει σαν αποτέλεσμα να αποτρέπεται η μη χρησιμοποίηση μέρους της συσκευής.

Η εναλλαγή από την μία εφαρμογή σε άλλη είναι 20 φορές γρηγορότερη στον Fermi από προηγούμενης γενιάς GPUs. Αυτή η εναλλαγή διαχειρίζεται από το chip-level GigaThread hardware Thread Sheduler, ο οποίος διαχειρίζεται 1536 παράλληλα ενεργά νήματα για κάθε SM στους 16 kernels.

Οι Streaming Multiprocessor της Fermi περιλαμβάνουν 32 cores, όπου ο καθένας μπορεί να εκτελέσει πράξεις κινητής υποδιαστολής(floating point) και ακέραιες πράξεις(integer) μαζί με 16 load-store μονάδες για πράξεις της μνήμης, τέσσερις μονάδες ειδικών λειτουργιών, και 64K τοπικής SRAM διαιρεμένη σε cache και τοπική μνήμη. Στο σχήμα 3.2 παρουσιάζεται η δομή ενός SM. Σε κάθε SM, οι πυρήνες διαιρούνται σε δύο execution blocks των δεκαέξι πυρήνων το καθένα. Μαζί με την ομάδα των δεκαέξι load-store μονάδων και των τεσσάρων SFUs(Special Function Units) υπάρχουν τέσσερα μπλόκς εκτέλεσης ανά SM. Στην εργασία μας θα χρησιμοποιήσουμε GPUs με Fermi αρχιτεκτονική για να τρέξουμε τις εφαρμογές μας.



Σχήμα 3.2 : Δομή του Streaming Multiprocessor(Πηγή:Nvidia)

3.4 Γενικής Χρήσης Μονάδα Επεξεργασίας Γραφικών(GPGPU)

Το υλικό γραφικών όπως το όνομα του υποδηλώνει, άρχισε να ισχύει με την εικόνα και τους γραφικούς υπολογισμούς στο μυαλό, αλλά το νεότερο πιο προγραμματιζόμενο υλικό είναι σε θέση να εκμεταλλευτεί πιο γενικούς υπολογισμούς.

Μεγάλο μέρος των υπολογισμών που ασχολούνται με τις εικόνες που φαίνονται στην οθόνη δεν έχουν άμεση σχέση με τις εικόνες, αλλά με πιο γενικά μαθηματικά χαρακτηριστικά. Είναι αυτά τα πιο γενικά χαρακτηριστικά μαζί με το stream processing μοντέλο υπολογισμού που επιτρέπει στην GPU να είναι κατάλληλη για γρήγορους, αποτελεσματικούς, μη γραφικούς υπολογισμούς. Τα στάδια του 3Δ γραφικού pipeline που χρησιμοποιούνται από την GPU έχουν εξελιχθεί από μία εφαρμογή λογισμικού σε graphics coprocessor της κύριας CPU, σε ένα προγραμματιζόμενο γραφικό pipeline και τα στάδια γίνονται όλο και πιο προγραμματιζόμενα και το πιο σημαντικό πιο γενικά. Καθώς αυτές οι προγραμματιζόμενες φάσεις γίνονται πιο γενικής φύσεως η GPU μπορεί να θεωρηθεί ως μια γενικής χρήσης stream processor. Αν κι οι CPUs παρέχουν υψηλότερη απόδοση για διαδοχικό κώδικα γίνεται όλο και πιο δύσκολο απλά να προσθέσουν τρανζίστορ για να βελτιώσουν τις επιδόσεις αυτού του τύπου κώδικα. Αντιθέτως, δεδομένου ότι οι προγραμματιζόμενες GPUs έχουν βελτιστοποιηθεί για υψηλότερους παράλληλους vertex και fragment shading κώδικες, οι GPUs μπορούν να χρησιμοποιήσουν επιπλέον τρανζίστορ πολύ πιο αποτελεσματικά τα οποία επιτρέπουν στις GPUs να έχουν ένα υψηλότερο ποσοστό επίδοσης, καθώς η τεχνολογία ημιαγωγών βελτιώνεται. Επειδή οι GPUs έχουν εξελιχθεί σε πολύ αποτελεσματικούς και ισχυρούς υπολογιστικούς μεταποιητές, οι ερευνητές έχουν μελετήσει τρόπους με τους οποίους η δύναμη της GPUs θα μπορούσε να αξιοποιηθεί για την επίλυση προβλημάτων πιο γενικού σκοπού, εκτός από τα γραφικά. Στην εργασία μας θα αξιοποιήσουμε τις GPUs για να επιλύσουμε προβλήματα γενικού σκοπού πιο γρήγορα.

Κεφάλαιο 4

Εισαγωγή στην OpenACC

4.1 Μοντέλο Εκτέλεσης

Το μοντέλο εκτέλεσης, που υποτελή στόχο των OpenACC API ενεργοποιημένων μεταγλωττιστών, είναι host κατευθυνόμενη εκτέλεση με μια συνδεδεμένη συσκευή-επιταχυντή, όπως είναι ένας GPU. Το μεγαλύτερο μέρος της εφαρμογής του χρήστη εκτελείτε στο host. Η συσκευή εκτελεί παράλληλες περιοχές, οι οποίες τυπικά περιέχουν καταμερισμένα loops, ή kernels περιοχές, οι οποίες τυπικά περιέχουν ένα ή περισσότερα loops τα οποία θα εκτελεστούν σαν kernels. Ακόμα και σε περιοχές στοχευμένες από τον επιταχυντή, ο host πρέπει να ενορχηστρώσει την εκτέλεση δεσμεύοντας μνήμη στον επιταχυντή, αστικοποιώντας τα δεδομένα μεταφοράς, στέλνοντας τον κώδικα στον επιταχυντή, περνώντας παραμέτρους στην παράλληλη περιοχή, βάζοντας σε ουρά τον

κώδικα της συσκευής, περιμένοντας για την ολοκλήρωση, μεταφέροντας τα αποτελέσματα πίσω στον host και αποδεσμεύοντας την μνήμη. Στις περισσότερες περιπτώσεις ο host μπορεί να βάλει σε μια ουρά μια σειρά από διεργασίες για να εκτελεστούν στην συσκευή η μια μετά την άλλη.

Οι πιο πρόσφατοι επιταχυντές υποστηρίζουν δύο ή τρία επίπεδα παραλληλισμού. Οι περισσότεροι επιταχυντές υποστηρίζουν coarse-grain παραλληλισμό, στον οποίο έχουμε πλήρως παράλληλη εκτέλεση στις μονάδες εκτέλεσης. Μπορεί να υπάρχει περιορισμένη υποστήριξη για συγχρονισμό σε coarse-grain παράλληλες διεργασίες. Πολλοί επιταχυντές υποστηρίζουν επίσης fine-grain παραλληλισμό που συχνά εφαρμόζεται ως πολλαπλά νήματα εκτέλεσης σε μια ενιαία μονάδα εκτέλεσης, τα οποία συνήθως ενεργοποιούνται γρήγορα στην μονάδα εκτέλεσης για να ανεχθεί μεγάλες καθυστερήσεις μνήμης σε διεργασίες. Τέλος, οι περισσότεροι επιταχυντές επίσης υποστηρίζουν SIMD ή vector διεργασίες μέσα σε κάθε μονάδα εκτέλεσης. Το μοντέλο εκτέλεσης από την πλευρά της συσκευής εκθέτει αυτά τα πολλαπλά επίπεδα παραλληλισμού και ο προγραμματιστής πρέπει να κατανοήσει τις διαφορές μεταξύ τους. Για παράδειγμα ένα πλήρως παράλληλο loop και ένα loop που είναι διανυσματικό αλλά απαιτεί συγχρονισμό μεταξύ των δηλώσεων. Ένα πλήρως παράλληλο loop μπορεί να προγραμματιστεί για coarse-grain παράλληλη εκτέλεση. Loops με εξαρτήσεις πρέπει είτε να χωριστούν για να επιτρέπουν coarse-grain παράλληλη εκτέλεση ή να προγραμματιστούν να για να εκτελεστούν σε μια ενιαία μονάδα εκτέλεσης χρησιμοποιώντας fine-grain παραλληλισμό, vector παραλληλισμό, ή διαδοχικά.

4.2 Μοντέλο Μνήμης

Η πιο σημαντική διαφορά μεταξύ ενός host-only προγράμματος και ενός host+accelerator προγράμματος είναι ότι η μνήμη στο επιταχυντής μπορεί να είναι εντελώς ξεχωριστή από τη μνήμη του host. Αυτή είναι και η περίπτωση με τους περισσότερους πρόσφατους επιταχυντές, για παράδειγμα. Στην περίπτωση αυτή ο host μπορεί να μην είναι σε θέση να διαβάσει ή να γράψει τη μνήμη της συσκευής άμεσα διότι δεν αντιστοιχίζεται στον εικονικό χώρο μνήμης του host. Όλες οι μεταφορές δεδομένων από την μνήμη του host στην μνήμη της συσκευής θα πρέπει να εκτελούνται από τις runtime κλήσεις βιβλιοθήκης του host που ρητά μετακινούν τα δεδομένα μεταξύ των ξεχωριστών μνήμων, χρησιμοποιώντας συνήθως άμεσης προσπέλασης μνήμης (DMA). Ομοίως, δεν είναι σωστό να αναλάβει ο επιταχυντής να διαβάσει ή να γράψει τη μνήμη του host, αν και αυτό υποστηρίζεται από ορισμένες συσκευές επιταχυντή.

Η έννοια της ξεχωριστής μνήμης, του host και της συσκευής επιτάχυνσης, είναι πολύ εμφανής στις χαμηλού επιπέδου γλώσσες προγραμματισμού επιταχυντή όπως η CUDA και η OpenCL, στις οποίες τα δεδομένα μεταφοράς μεταξύ των μνήμων μπορεί να κυριαρχήσει ο κώδικας του χρήστη. Στο OpenACC μοντέλο, τα δεδομένα μεταφοράς μεταξύ των μνημών είναι σιωπηρή διαδικασία και ελέγχεται από τον μεταγλωττιστή, με βάση τις οδηγίες από τον προγραμματιστή. Ωστόσο, ο προγραμματιστής πρέπει να γνωρίζει ενδεχομένως τις ξεχωριστές μνήμες για πολλούς λόγους. Πρώτα γιατί το εύρος ζώνης της μνήμης ανάμεσα στην μνήμη του host και της συσκευής καθορίζει και το επίπεδο της έντασης του υπολογισμού που απαιτείτε για την αποτελεσματική επιτάχυνση μιας δεδομένης περιοχής κώδικα και δεύτερον γιατί το περιορισμένο μέγεθος μνήμης της συσκευής μπορεί να απαγορεύει την εκφόρτωση των περιοχών κώδικα που λειτουργούν σε πολύ μεγάλες ποσότητες δεδομένων.

Από την πλευρά του επιταχυντή, μερικοί επιταχυντές (όπως οι τρέχουσες GPUs) εφαρμόζουν ένα αδύναμο μοντέλο μνήμης. Ειδικότερα, δεν υποστηρίζουν τη συνοχή μνήμης μεταξύ των εργασιών που εκτελούνται από διαφορετικές μονάδες εκτέλεσης, ακόμα και στην ίδια μονάδα εκτέλεσης η συνοχή της μνήμης εξασφαλίζεται μόνο όταν οι λειτουργίες μνήμης χωρίζονται από ρητό εμπόδιο. Διαφορετικά, αν μια λειτουργία ενημέρωση μια θέση μνήμης και μια άλλη διαβάζει την ίδια θέση, ή δύο λειτουργίες αποθηκεύουν μια τιμή στην ίδια θέση, το υλικό δεν μπορεί να εγγυηθεί το ίδιο αποτέλεσμα για κάθε εκτέλεση. Ενώ ένας μεταγλωττιστής μπορεί να εντοπίσει ορισμένα πιθανά σφάλματα αυτού του είδους, είναι ωστόσο δυνατό να γράψει μια επιταχυνόμενη παράλληλη ή kernels περιοχή που παράγει ασυνεπή αριθμητικά αποτελέσματα.

Μερικοί τρέχων επιταχυντές έχουν μια διαχειρίσιμη λογισμικά cache, ορισμένοι έχουν hardware διαχειρίσιμες caches, και οι περισσότεροι έχουν hardware caches που μπορούν να χρησιμοποιηθούν σε ορισμένες περιπτώσεις και περιορίζονται μόνο σε ανάγνωση δεδομένων. Σε χαμηλό επίπεδο μοντέλα προγραμματισμού, όπως CUDA και OpenCL, αυτές οι caches είναι διαχειρίσιμες από τον προγραμματιστή. Στο OpenACC μοντέλο, αυτές οι caches διαχειρίζονται από τον μεταγλωττιστή με hints από τον προγραμματιστή με την μορφή οδηγιών.

4.3 Εντολές(directives)

Στην C και την C++ οι OpenACC οδηγίες προσδιορίζονται χρησιμοποιώντας τον μηχανισμό #pragma που παρέχεται από την γλώσσα. Οι μεταγλωττιστές τυπικά αγνοούν τις OpenACC οδηγίες εάν η υποστήριξη είναι απενεργοποιημένη ή δεν προβλέπεται. Η σύνταξη των οδηγιών είναι ως ακολούθως:

#pragma acc directive-name [clause [[,] clause]…] new-line

Κάθε οδηγία ξεκινά με #pragma acc. Μια OpenACC οδηγία εφαρμόζεται για την αμέσως ακολουθούμενη δήλωση, δομημένου μπλοκ ή βρόγχου. Μόνο μία οδηγία-όνομα μπορεί να καθοριστεί ανά οδηγία. Η σειρά με την οποία εμφανίζονται τα clauses δεν είναι σημαντική, και τα clauses μπορούν να επαναληφθούν, εκτός αν ορίζονται διαφορετικά. Μερικά clauses έχουν μια λίστα από παραμέτρους. Η λίστα με τα ονόματα των μεταβλητών, των πινάκων η των υποπίνακων είναι διαχωρισμένη με κόμμα. Μια εφαρμογή OpenACC λειτουργεί σαν να υπάρχουν εσωτερικές μεταβλητές ελέγχου (internal control variables - ICVs) οι οποίες ελέγχουν την συμπεριφορά του προγράμματος. Αυτές οι ICVs αρχικοποιούνται από την εφαρμογή και μπορεί να τους δοθεί τιμή μέσω των μεταβλητών περιβάλλοντος και μέσω κλήσεων σε OpenACC API ρουτίνες. Το πρόγραμμα μπορεί να ανακτήσει τις τιμές μέσω των OpenACC API ρουτινών. Οι μεταβλητές αυτές είναι: (i) acc-device-type-var, ελέγχει το είδος της συσκευής επιταχυντή που χρησιμοποιείτε, (ii) acc-device-num-var, ελέγχει ποια συσκευή επιταχυντή του επιλεγόμενου τύπου χρησιμοποιείτε.

Οι αρχικές τιμές ορίζονται κατά την υλοποίηση. Μέτα που θα οριστούν οι αρχικές τιμές, αλλά πριν από οποιοδήποτε OpenACC construct ή API ρουτίνα θα εκτελεστεί, οι μεταβλητές όλων των μεταβλητών περιβάλλοντος που ορίστηκαν από τον χρήστη διαβάζονται και οι συναφείς ICVs τροποποιούνται αναλόγως. Με από αυτό το σημείο, οι ICVs μπορούν να τροποποιηθούν είτε από το πρόγραμμα είτε εξωτερικά. Τα clauses στα OpenACC constructs δεν μπορούν να τροποποιήσουν τις ICVs τιμές.

4.4 Parallel Construct

Το construct αυτό εχει την εξής σύνταξη:

#pragma acc parallel [clause [[,] clause]…] new-line

structured block

Όταν το πρόγραμμα συναντήσει ένα parallel construct, gangs από workers δημιουργούνται για να εκτελέσουν μια επιταχυνόμενη παράλληλη περιοχή. Όταν δημιουργηθούν τα gangs, ο αριθμός των gangs και των workers σε κάθε gang παραμένει σταθερός σε όλη την διάρκεια της παράλληλης περιοχής. Ένας worker σε κάθε gang ξεκινά να εκτελεί τον κώδικα σε ένα δομημένο μπλοκ του construct. Αν το async clause δεν υπάρχει, τότε υπάρχει ένα σιωπηρό εμπόδιο(barrier) στο τέλος της επιταχυνόμενης παράλληλης περιοχής, και το κύριος πρόγραμμα θα περιμένει μέχρις ότου όλα τα gangs θα τελειώσουν την εκτέλεση.

Υπάρχουν κάποιοι περιορισμοί στο parallel construct. Οι OpenACC parallel περιοχές μπορεί να μην μπορούν να περιέχουν άλλες parallel η kernels περιοχές, το πρόγραμμα μπορεί να μην διακλαδώνεται μέσα ή έξω από ένα OpenACC parallel construct και το πρόγραμμα δεν πρέπει να εξαρτάται από μια σειρά αξιολογήσεων των clauses, ή από οποιεσδήποτε παρενέργειες των αξιολογήσεων.

4.5 Kernels Construct

Το construct αυτό έχει την εξής σύνταξη:
#pragma acc kernels [clause [[,] clause]…] new-line
structured block

Ο μεταγλωττιστής θα σπάσει το κώδικα μιας kernels περιοχής σε μια ακολουθία από accelerator kernels. Τυπικά κάθε εσωτερικό loop θα είναι ένας ξεχωριστός kernel. Όταν το πρόγραμμα συνάντηση ένα kernels construct, θα ξεκινήσει την ακολουθία των πυρήνων σε σειρά στην συσκευή. Ο αριθμός και η διαμόρφωση των gangs από workers και το μέγεθος του vector(νήματος) μπορεί να είναι διαφορετικός σε κάθε kernel.

Αν το async clause δεν υπάρχει, τότε υπάρχει ένα σιωπηρό εμπόδιο(barrier) στο τέλος της kernels περιοχής, και το κυρίως πρόγραμμα θα περιμένει μέχρις ότου όλοι οι kernels θα τελειώσουν την εκτέλεση.

Υπάρχουν κάποιοι περιορισμοί στο kernels construct. Οι OpenACC kernels περιοχές μπορεί να μην μπορούν να περιέχουν άλλες parallel η kernels περιοχές, το πρόγραμμα μπορεί να μην διακλαδώνεται μέσα ή έξω από ένα OpenACC kernels construct και το πρόγραμμα δεν

πρέπει να εξαρτάται από μια σειρά αξιολογήσεων των clauses, ή από οποιεσδήποτε παρενέργειες των αξιολογήσεων.

4.6 Data Construct

Το construct αυτό εχει την εξής σύνταξη:
#pragma acc data [clause [[,] clause]…] new-line
structured block

Το data construct καθορίζει πίνακες και υποπίνακες για να δεσμευτούν στην μνήμη της συσκευής κατά την διάρκεια της περιοχής, εάν τα δεδομένα πρέπει να αντιγραφούν από την μνήμη του host στην μνήμη της συσκευής κατά την είσοδο στην περιοχή, και αντιγράφονται από την μνήμη της συσκευής στην μνήμη του host κατά την έξοδο από την περιοχή.

4.7 Loop Construct

Το construct αυτό έχει την εξής σύνταξη:
#pragma acc loop [clause [[,] clause]…] new-line
for loop

Το construct αυτό εφαρμόζεται σε ένα loop που πρέπει να ακολουθήσει αμέσως την εν λόγω οδηγία. Η loop οδηγία μπορεί να περιγράψει τι είδους παραλληλισμό θα χρησιμοποιηθεί για την εκτέλεση του βρόγχου και να δηλώσει ιδιωτικές μεταβλητές, πίνακες και πράξεις reduction. Ορισμένα clauses δεν είναι έγκυρα στο parallel construct και ορισμένα δεν είναι έγκυρα στο kernels construct. Σε μια parallel περιοχή, η οδηγία loop χωρίς gang, worker ή vector clauses επιτρέπουν στην εφαρμογή αυτόματα να διαλέξει αν θα εκτελέσει το loop σε gangs, workers μέσα σε ένα gang, ή αν θα το εκτελέσει σαν vector πράξεις. Η εφαρμογή επίσης μπορεί να επιλέξει να χρησιμοποιήσει vector πράξεις για να εκτελέσει κάθε loop χωρίς το loop directive, χρησιμοποιώντας κλασσική αυτόματη διανυσματοποίηση.

4.8 Runtime Library Routines

Στο OpenACC API υπάρχουν αρκετές Library routines οι οποίες μας επιτρέπουν να αλλάξουμε κατά το χρόνο εκτέλεσης να αλλάξουμε τις παραμέτρους που χρησιμοποιεί το μοντέλο. Μέσω των ρουτινών αυτών μπορούμε να αρχικοποιήσουμε το περιβάλλον εκτέλεσης, να διαφοροποιήσουμε την συσκευή επιτάχυνσης που θα χρησιμοποιηθεί, να πάρουμε το τύπο της συσκευής επιτάχυνσης, να δεσμεύσουμε μνήμη στην συσκευή και

γενικά να μπορούμε να κάνουμε το προγραμματισμό πιο ευέλικτο. Μας δίνουν αρκετή ελευθερία που χρειάζεται για να πάρουμε τα καλύτερα αποτελέσματα.

4.9 Τοποθέτηση εντολών

Όλες αυτές οι οδηγίες και ρουτίνες που είδαμε στα προηγούμενα υποκεφάλαια για να αξιοποιήσουν σωστά τις συσκευές που έχουν στην διάθεση τους και να εκμεταλλευτούν στο έπακρον τις δυνατότητες τους πρέπει να τοποθετηθούν η κάθε μια εκεί που χρειάζεται. Για παράδειγμα το parallel construct και το kernels construct μπορούν να χρησιμοποιηθούν για να παραλληλοποιήσουν μια ίδια περιοχή κώδικα αλλά θα αποδώσουν διαφορετικά ανάλογα με το είδος κώδικα που καλούνται να παραλληλοποιήσουν. Στο Σχήμα 4.1 παρατίθεται ο κώδικα ενός προγράμματος που χρησιμοποιεί το OpenACC API για να εκτελέσει παράλληλα τις ένα εκατομμύριο επαναλήψεις ενός βρόγχου. Χρησιμοποιώντας την οδηγία προς τον μεταγλωττιστή πριν ακριβώς από το βρόγχο δίνουμε την εντολή στο μεταγλωττιστή να σπάσει το κώδικα που περικλείετε μέσα στο βρόγχο σε accelerator kernels. Κάθε εσωτερική επανάληψη του βρόγχου θα είναι ένας kernel. Επίσης βλέπουμε και την χρησιμοποίησή του data construct το οποίο εδώ χρησιμοποιείτε για να αντιγραφούν οι πίνακες x και y στην συσκευή επιτάχυνσης και να μεταφερθούν πίσω στον host μετά το τέλος της παράλληλης περιοχής.

```
#include <math.h>
#include <string.h>
#include <openacc.h>

void saxpy(int n, float a, float *x, float *y)
{
        #pragma acc kernels
        for (int i = 0; i < n; ++i)
                y[i] = a*x[i] + y[i];
}

int main(int argc, char** argv)
{
        int N = 1000000;
        float x[N];
       float y[N];
        x = (float*)malloc(sizeof(float)*N);

        y = (float*)malloc(sizeof(float)*N);
        // Perform SAXPY on 1000000 elements

        #pragma acc data copy(x,y)
        {
                saxpy(N, 2.0, x, y);
        }

}
```

Σχήμα 4.1 : Παράδειγμα κώδικα που χρησιμοποιεί το OpenACC API(Πηγη: NVidia)

Είναι εμφανές από το παραπάνω παράδειγμα πως το OpenACC API είναι απλό και εύκολο στην χρήση αφού το μόνο που χρειάστηκε να κάνουμε είναι να προσθέσουμε δύο οδηγίες προς το μεταγλωττιστή και την βιβλιοθήκη openacc.h ώστε να καταφέρουμε να κάνουμε το πρόγραμμα να τρέξει παράλληλα σε πολύ μικρότερο χρόνο από μια σειριακή εκτέλεση. Στο Σχήμα 4.2 φαίνεται ο αντίστοιχος κώδικας για να τρέξει το πρόγραμμα αυτό με την χρήση της CUDA. Είναι ξεκάθαρο πως με την χρήση της CUDA χρειάστηκε να αλλάξουμε την συνάρτηση που εκτελεί τις επαναλήψεις καθώς και να προσθέσουμε περισσότερες γραμμές κώδικα από συναρτήσεις της CUDA ώστε να δεσμεύσουμε μνήμη στη συσκευή επιτάχυνσης.

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <iostream>
#include <fstream>

using namespace std;

__global__
void saxpy(int n, float a, float *x, float *y)
{
  int i = blockIdx.x*blockDim.x + threadIdx.x;
  if (i < n) y[i] = a*x[i] + y[i];
}

int main(int argc, char *argv[]){

        int N = 1000000;
        float *x;
        float *y;
        float *d_x;
        float *d_y;

        // Allocate input vectors x and y in host memory
        x = (float*)malloc(sizeof(float)*N);

        y = (float*)malloc(sizeof(float)*N);
        d_x = (float*)malloc(sizeof(float)*N);

        d_y = (float*)malloc(sizeof(float)*N);

        cudaMemcpy(x, d_x, N, cudaMemcpyHostToDevice);
        cudaMemcpy(y, d_y, N, cudaMemcpyHostToDevice);

        // Perform SAXPY on 1000000 elements
        saxpy<<<4096,256>>>(N, 2.0, x, y);

        cudaMemcpy(d_y, y, N, cudaMemcpyDeviceToHost);

}
```

Σχήμα 4.2 : Παράδειγμα κώδικα που χρησιμοποιεί τη CUDA(Πηγη: NVidia)

Κεφάλαιο 5

Αξιολόγηση

5.1 Διαδικασία Αξιολόγησης

Μελετώντας το μοντέλο προγραμματισμού της OpenACC είμαστε τώρα σε θέση να διαφοροποιήσουμε το σειριακό κώδικα μιας εφαρμογής ώστε να την παραλληλοποιήσουμε με την χρήση της OpenACC και τρέχοντας τον να καταφέρουμε να πετύχουμε υψηλή βελτίωση των επιδόσεων φτάνοντας επιδόσεις που μπορεί να μας δώσει μια αντίστοιχη παραλληλοποίηση με την CUDA.

Πήραμε λοιπόν τρεις εφαρμογές που εκτελούν 3 διαφορετικά Queries βάσεων δεδομένων και τα τρέξαμε σειριακά συλλέγοντας τους χρόνους εκτέλεσης τους για το κομμάτι υπολογισμού μόνο. Στην συνέχεια πήραμε την υλοποίηση των Queries αυτών σε CUDA, όπως τα υλοποίησε ένας έμπειρος προγραμματιστής CUDA, για την καλύτερη προσπάθεια και τρέξαμε τα Queries σε CUDA και συλλέξαμε τους χρόνους εκτέλεσης για το κομμάτι του

υπολογισμού μόνο. Να σημειωθεί ότι για το κάθε Query δίναμε κάποια αρχεία βάσεων δεδομένων σαν είσοδο για να παραχθούν κάποιοι πίνακες από αυτά τα αρχεία οι οποίοι πίνακες είναι απαραίτητοι στο κομμάτι υπολογισμού του τελικού αποτελέσματος. Το κάθε Query το εκτελέσαμε για 7 διαφορετικά μεγέθη αρχείων εισόδου αυξάνοντας κάθε φορά το μέγεθος των αρχείων.

Στην συνέχεια ακολούθησε η διαδικασία παραλληλοποίησης των Queries χρησιμοποιώντας το OpenACC πρότυπο. Η διαδικασία αυτή περιελάμβανε 3 διαφορετικές φάσεις. Στην πρώτη φάση απλά προσθέσαμε τις κατάλληλες OpenACC οδηγίες προς τον μεταγλωττιστή και εκτελέσαμε τα Queries για τα διαφορετικά μεγέθη αρχείων εισόδου συλλέγοντας τους χρόνους εκτέλεσης. Στην δεύτερη φάση προσθέσαμε το καλύτερο loop unrolling που μπορούσαμε στους βρόγχους που παραλληλοποιήσαμε συλλέγοντας πάλι τους χρόνους εκτέλεσης. Και τέλος στην τρίτη φάση προσθέσαμε code optimization όπου μπορούσαμε και συλλέξαμε τους χρόνους εκτέλεσης. Και στις τρεις φάσεις που ακολουθήσαμε οι χρόνοι που πήραμε ήταν για το κομμάτι του υπολογισμού μόνον. Επίσης σε όλες τις εκτελέσεις για OpenACC εκτελέσαμε 3 φορές το κάθε Query για κάθε αρχεία εισόδου. Η πρώτη φορά ήταν για να εκτελεστεί ο παράλληλος κώδικας με χρήση μόνο της CPU, η δεύτερη φορά με την χρήση μόνο της GPU και η τρίτη φορά με την χρήση της CPU και GPU μαζί.

Τελευταία διαδικασία που ακολουθήσαμε ήταν να αφαιρέσουμε τα διάφορα optimizations που γίνονται στον κώδικα της CUDA και δεν μπορούν να εφαρμοστούν κατα ανάλογο τρόπο στην εφαρμογή της OpenACC και έτσι να βρούμε και τους λόγους της καλύτερης επίδοσης που παρουσίαζε η CUDA. Αφαιρώντας τα διαφορά optimizations από το κώδικα της CUDA εκτελέσαμε ξανά τα Queries και πήραμε τους νέους χρόνους εκτέλεσης. Έχοντας τώρα όλα τα αποτελέσματα συγκεντρωμένα βγάλαμε τις γραφικές παραστάσεις για την βελτίωση της επίδοσης με CUDA και OpenACC σε σχέση με την σειριακή εκτέλεση καθώς και την επίδοση της OpenACC σε σχέση με την επίδοση της CUDA για όλα τα Queries και όλα τα μεγέθη αρχείων εισόδου.

5.2 Πειραματική Διάταξη

Τα τρία διαφορετικά Queries που χρησιμοποιήσαμε για να εκτελέσουμε την εργασία αυτή είναι το Query 3, το Query 6 και το Query 12. Τα Queries αυτά εκτελέστηκαν σε μηχανή που διέθετε συσκευές επιτάχυνσης(GPUS) οι οποίες είναι απαραίτητες για την παραλληλοποίηση με CUDA και OpenACC. Τα μεγέθη των αρχείων εισόδου που χρησιμοποιήθηκαν

παρουσιάζονται στον πίνακα 5.1. Στο πίνακα 5.2 φαίνονται τα χαρακτηριστικά της μηχανής που εκτελέσαμε την εργασία μας. Για κάθε μέγεθος εισόδου τρέξαμε 3 φορές το κάθε Query αλλάζωντας την συσκευή επιτάχυνσης που χρησημοποιούσαμε για να τρέξει παράλληλα ο κώδικας μας. Την πρώτη φορά η συσκευή επιτάχυνσης ήταν μόνο η CPU, την δευτερη φορά ήταν μονο η GPU και την τρίτη φορά η CPU και η GPU συνεργάζονταν για να τρέξουν μαζί παράλληλα το κώδικα μας

	lineitem.tbl	**orders.tbl**	**customer.tbl**
1	60175	15000	1500
2	120515	30000	3000
3	299814	75000	7500
4	600572	150000	15000
5	1199969	300000	30000
6	2999671	750000	75000
7	6001215	1500000	150000

Πίνακας 5.1 : Μεγέθη αρχείων εισόδου σε αριθμό γραμμών

Hardware	**Software**
CPU: Intel(R) Core(TM)i7 950 @ 3.07GHz	CUDA + OpenCL v5.0
RAM: (2x) Corsair XMS DDR3 3x2GB @ 2GHz	CC 4.6.2, OpenMP
MB: ASUS P6T SE	Intel Parallel Studio XE 2013(C/C++ and Fortran)
GPU0: (MSI) NVIDIA GeForce GTX 580	PGI Fortran/C/C++/OpenACC 12.6
GPU1: (EVGA) NVIDIA GeForce GTX 580	SPEC CPU 2006

Πίνακας 5.2 : Χαρακτηριστικά Μηχανής που εκτελέσαμε την εργασία μας

Τα τρία αυτά Queries προέρχονται από το TPC-H Benchmark[5]. Είναι ένα benchmark υποστήριξης λήψης αποφάσεων και αποτελείτε από μια οικογένεια επιχειρησιακού προσανατολισμού ad-hoc queries και ταυτόχρονων τροποποιήσεων δεδομένων. Τα queries και τα δεδομένα που βρίσκονται στην βάση δεδομένων έχουν επιλεγεί έτσι ώστε να έχουν ευρύ βιομηχανικό ενδιαφέρον. Το benchmark αυτό απεικονίζει τα συστήματα υποστήριξης αποφάσεων που εξετάζουν μεγάλους όγκους δεδομένων, εκτελούν queries με υψηλό βαθμό πολυπλοκότητας, και δίνουν απαντήσεις σε κρίσιμα επιχειρηματικά ερωτήματα.

Ακολουθεί πιο κάτω μια σύντομη περιγραφή των τριών Queries:

– **Query 3**

Στο Query αυτό ψάχνουμε για τον αριθμό των πελατών που παράγγειλαν μια οικοδομή μετά το 1980 και την πήραν πριν το 1995. Σαν είσοδο στο πρόγραμμα αυτό δίνουμε τρία αρχεία βάσεων δεδομένων. Το ένα αρχείο περιέχει όλα τα lineitems, το δεύτερο αρχείο περιέχει όλες τις παραγγελίες και το τρίτο όλους τους πελάτες . Στο σχήμα 5.1(α) παρουσιάζεται η SQL μορφή του Query αυτού και στο σχήμα 5.1(β) ο αντίστοιχος C κώδικας.

```
select
        l_orderkey ,
from
        customer, orders, lineitem
where
        c_mkt segment = ‘ [BUILDING] ‘
        and  c_custkey = o_custkey
        and  l_orderkey = o_orderkey
        and  o_orderdate < date ‘ [1980] ‘
        and  l_shipdate > date ‘ [1995] ‘ ;
```

Σχήμα 5.1(α) : Απλοποιημένη μορφή του TPC-H Q3

```
for(int i = start; i < end; i=i+1)
{
        curr_ord1 = lineitem[i][0];
        if(lineitem[i][3] > 1980)
        {
            for(int j = 0; j < ORDERS; j=j+1)
                if((curr_ord1 == orders[j][0]) && (orders[j][2] < 1995))
                  for(int k = 0; k < CUSTOMER; k++)
                      if((customer[k][0] == orders[j][1]) && (customer[k][1] == 2))
                          sum++;
        }
}
```

Σχήμα 5.1(β) : C κώδικας του TPC-H Q3 για το κομμάτι υπολογισμού

- Query 6

Στο Query αυτό ψάχνουμε για τον αριθμό των items που στάλθηκαν το 1994 με έκπτωση από 0.00 μέχρι 0.02 και έχουν ποσότητα μικρότερη από 24. Επίσης υπολογίζουμε ένα άθροισμα προσθέτοντας κάθε φορά που βρίσκουμε ένα items με τις

προαναφερθέντες προϋποθέσεις το γινόμενο του χρόνου αποστολής και της τιμής του item. Σαν είσοδο στο πρόγραμμα αυτό δίνουμε ένα αρχείο βάσεων δεδομένων. Το αρχείο αυτό περιέχει όλα τα lineitems. Στο σχήμα 5.2(α) παρουσιάζεται η SQL μορφή του Query αυτού και στο σχήμα 5.2(β) ο αντίστοιχος C κώδικας. .

```
select
        sum( l_extendedprice * l_discount )
        as revenue
from
        lineitem
where
        l_shipdate >= date ‘ [1994] ‘
        and l_shipdate < date ‘ [1995] ‘
        + interval ‘1 ‘ year
        and l_orderkey = o_orderkey
        and l_discount between
        [0.01] - 0.01 and + 0.01
        and l_quantity < [24] ;
```

Σχήμα 5.2(α) :Απλοποιημένη μορφή του TPC-H Q6

```
for(i = start; i < end; i=i+1)
{
        if((line_item[i][2]>=1994)&&(line_item[i][2]<1995)&&((line_item[i]
        [1]>(DISCOUNT-0.01)) && (line_item[i][1]<(DISCOUNT+0.01))) &&
        (line_item[i][3]<24))
        {
                sum+=line_item[i][0]*line_item[i][2];
                result_count++;
         }
}
```

Σχήμα 5.2(β) : C κώδικας του TPC-H Q6 για το κομμάτι υπολογισμού

- Query 12

Στο Query αυτό ψάχνουμε για τον αριθμό των παραγγελιών που παραλείφθηκαν το έτος 1994. Σαν είσοδο στο πρόγραμμα αυτό δίνουμε δύο αρχεία βάσεων δεδομένων. Το ένα αρχείο περιέχει όλα τα lineitems και το δεύτερο αρχείο περιέχει όλες τις παραγγελίες. Στο σχήμα 5.3(α) παρουσιάζεται η SQL μορφή του Query αυτού και στο σχήμα 5.3(β) ο αντίστοιχος C κώδικας. .

```
select
        sum( case when
                o_orderpriority = '1-URGENT' or
                o_orderpriority = '2-HIGH'
                then 1 else 0 end )
        as high_line_count
from
        orders , lineitem
where
        o_orderkey = l_orderkey
        and l_shipmode in
        ( ' [ SHIPMODE1 ] ' , ' [SHIPMODE2 ] ' )
        and l_commitdate < l_receiptdate
        and l_shipdate < l_commitdate
        and l_receiptdate >= date ' [1994] '
        and l_receiptdate < date ' [1995] '
        + interval '1 ' year ;
```

Σχήμα 5.3(α) :Απλοποιημένη μορφή του TPC-H Q12

```
for(int j =0; j < LINEITEM; j=j+1)
{
        for(int i = 0; i < ORDERS; i=i+1)
        {
                if((lineitem[j][4] >= 1994) && (lineitem[j][4] < 1995))
                        if (lineitem[j][0] == orders[i][0])
                                sum++;
        }
}
```

Σχήμα 5.3(β) : C κώδικας του TPC-H Q12 για το κομμάτι υπολογισμού

5.3 Παρουσίαση και Ανάλυση Αποτελεσμάτων

Αφού τρέξαμε παραλληλοποιήσαμε τα Queries πήραμε τα αποτελέσματα και τους χρόνους εκτέλεσης. Αρχικά για κάθε εκτέλεση εξετάσαμε αν το τελικό αποτέλεσμα ήταν σωστό συγκρίνοντας τα αποτελέσματα με τα αποτελέσματα που μας έδωσε η σειριακή εκτέλεση των Queries. Εδώ θα δείξουμε για το κάθε Query τις 3 φάσεις που περάσαμε για να καταλήξουμε στο τελικό μας αποτέλεσμα δείχνοντας κάθε φορά τις αλλαγές που κάναμε στον κώδικα. Θα δείξουμε επιπρόσθετα και τα αποτελέσματα αφαιρώντας κάποιες runtime routines της CUDA σαν την φάση 4. Επίσης για κάθε φάση του κάθε Query θα αναλύσουμε και θα παρουσιάσουμε τις γραφικές παραστάσεις που πήραμε για την βελτίωση της επίδοσης(speed up) σε σχέση με τη σειριακή εκτέλεση καθώς και τη γραφική παράσταση για την βελτίωση της επίδοσης της OpenACC σε σχέση με την βελτίωση της επίδοσης της CUDA.

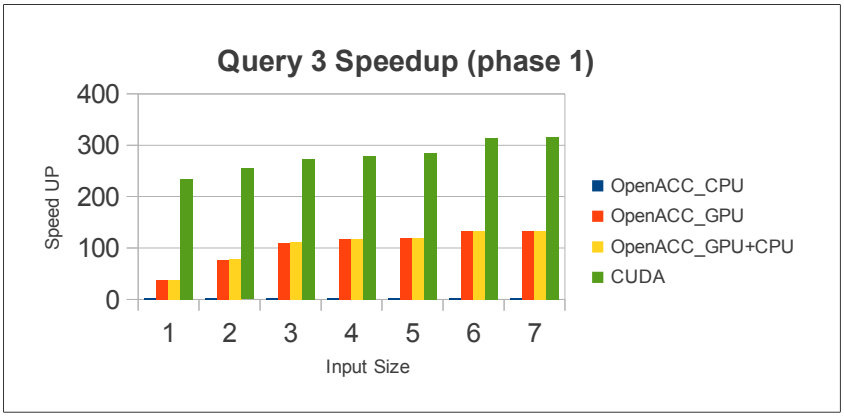
5.3.1 Query 3

- Φάση 1

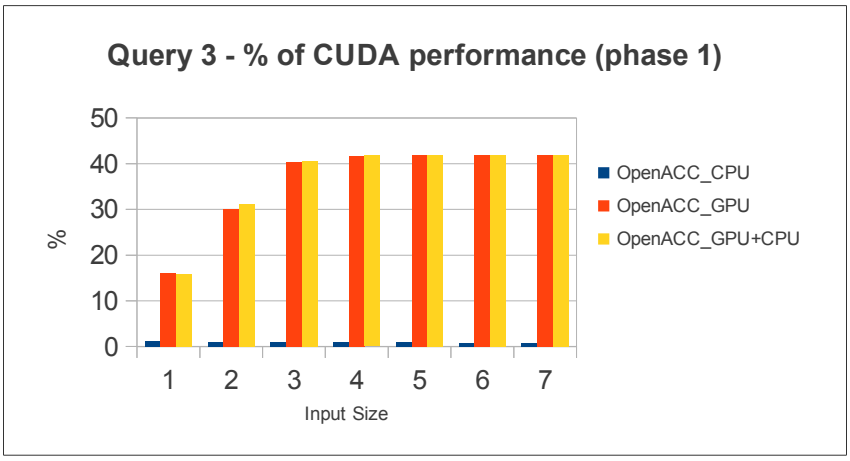
Στη φάση αυτή απλά προσθέσαμε μια οδηγία προς το μεταγλωττιστή. Δηλαδή προσθέσαμε 3 γραμμές κώδικα(προσθήκη βιβλιοθήκης OpenACC, αρχικοποίηση περιβάλλοντος OpenACC, οδηγία προς το μεταγλωττιστή για παράλληλη εκτέλεση περιοχής κώδικα). Στο Σχήμα 5.4(α) φαίνονται οι προσθήκες που κάναμε στο C κώδικα για να τρέξουμε το Query με το OpenACC API. Στο Σχήμα 5.4(β) παρουσιάζονται οι βελτιώσεις στην επίδοση της OpenACC(CPU, GPU, CPU+GPU) και της καλύτερης προσπάθειας για εκτέλεση σε CUDA για τα όλα τα μεγέθη αρχείων εισόδου που αναφέραμε στο προηγούμενο υποκεφάλαιο. Επίσης στο Σχήμα 5.4(γ) παρουσιάζονται οι βελτιώσεις στην επίδοση της OpenACC(CPU, GPU, CPU+GPU) σε σχέση με τις βελτιώσεις στην επίδοση της CUDA για τα όλα τα μεγέθη αρχείων εισόδου. Όπως βλέπουμε από τις δύο αυτές γραφικές παραστάσεις η βελτίωση της επίδοσης(speed up) αυξάνεται όσο μεγαλύτερα είναι τα αρχεία εισόδου. Αυτό οφείλετε στο γεγονός πως αυξάνεται κάθε φορά το υπολογιστικό μέρος της εφαρμογής και άρα αυξάνεται η χρήση της GPU. Διακρίνουμε επίσης ότι χρησιμοποιώντας σαν συσκευή επιτάχυνσης την CPU η βελτίωση της επίδοσης μειώνετε και σε κάθε περίπτωση κυμαίνεται σε πολύ χαμηλά επίπεδα 2.2-2.7 . Η speedup που παίρνουμε σε σχέση με την speed up της καλύτερης προσπάθειας με CUDA βρίσκεται γύρω στο 40% . Δεδομένου της πολύ μικρής προσπάθειας και χρόνου που καταβάλαμε(προσθήκη μόνο τριών γραμμών κώδικα) τα αποτελέσματα μας αφήνουν αρκετά ικανοποιημένους.

```
#include <openacc.h>
....
acc_init(acc_device_nvidia);
....
#pragma acc kernels loop independent collapse(1) reduction(+:sum)
for(int i = start; i < end; i=i+1){
        curr_ord1 = lineitem[i][0];
        if(lineitem[i][3] > 1980) {
            for(int j = 0; j < ORDERS; j=j+1)
                if((curr_ord1 == orders[j][0]) && (orders[j][2] < 1995))
                  for(int k = 0; k < CUSTOMER; k++)
                      if((customer[k][0] == orders[j][1]) && (customer[k][1] == 2))
                          sum++;
        }
}
```

Σχήμα 5.4(α) : Κώδικας C Φάσης 1 στο Query 3



Σχήμα 5.4(β) : Επίδοση Φάσης 1 στο Query 3



Σχήμα 5.4(γ) : Επί τοις εκατόν ποσοστιαία επίδοση της OpenACC σε σχέση με την επίδοση της CUDA για την Φάση 1 του Query 3

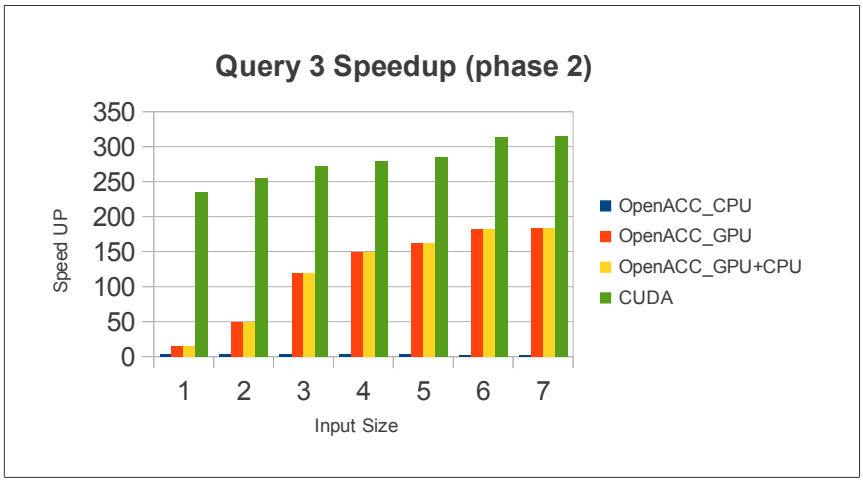
- Φάση 2

Στην φάση αυτή προσθέσαμε επιπλέον το καλύτερο δυνατό loop unrolling που μπορούσαμε για να πάρουμε καλύτερα αποτελέσματα. Το loop unrolling είναι η τεχνική που εφαρμόζεται σε βρόγχους για να αυξηθεί η ταχύτητα ενός προγράμματος με το να μειωθούν οι εντολές ελέγχου και αριθμητικής αύξησης του μετρητή σε κάθε τέλος μιας επανάληψης. Έτσι προσθέτοντας προστίθενται παρόμοιες εντολές στο εσωτερικό ενός βρόγχου και ο μετρητής αυξάνετε κατά τον αριθμό των παρόμοιων εντολών που εκτελούνται μέσα στο βρόγχο. Στην φάση αυτή προσθέσαμε loop unrolling τάξης 4. Στο Σχήμα 5.5(α) φαίνονται οι προσθήκες που κάναμε στο C κώδικα για να τρέξουμε το Query με το OpenACC API. Στο Σχήμα 5.5(α) παρουσιάζονται οι βελτιώσεις στην επίδοση της OpenACC(CPU, GPU, CPU+GPU) και της καλύτερης προσπάθειας για εκτέλεση σε CUDA για τα όλα τα μεγέθη αρχείων εισόδου .

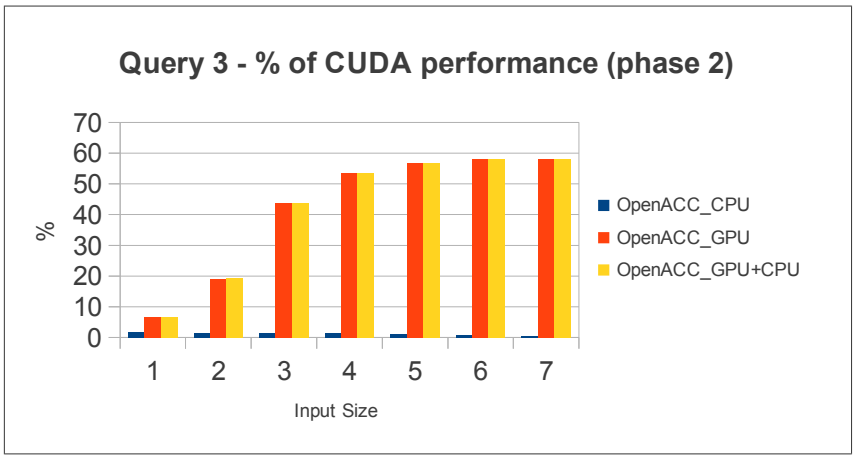
Επίσης στο Σχήμα 5.5(β) παρουσιάζονται οι βελτιώσεις στην επίδοση της OpenACC(CPU, GPU, CPU+GPU) σε σχέση με τις βελτιώσεις στην επίδοση της CUDA για τα όλα τα μεγέθη αρχείων εισόδου. Όπως βλέπουμε από τις δύο αυτές γραφικές παραστάσεις τα αποτελέσματα είναι σαφώς καλύτερα για τα μεγαλύτερα αρχεία εισόδου(3-7). Αυτό οφείλετε στο γεγονός πως αποφεύγουμε πάρα πολλούς ελέγχους που γίνονται για κάθε καινούργια επανάληψη του πρώτου εσωτερικού βρόχου. Διακρίνουμε τώρα επίσης ότι έχουμε μια αύξηση της speed up χρησιμοποιώντας μόνο την CPU(μέχρι 3.75). Στην φάση αυτή μας βοήθησε το loop unrolling να πάρουμε μια speed up κοντά στο 60% σε σχέση με την speed up για την καλύτερη προσπάθεια με CUDA. Και σε αυτή τη φάση η προσπάθεια που καταβάλαμε σε χρόνο είναι πολύ μικρότερη από ότι η προσπάθεια που καταβάλετε από ένα έμπειρο χρήστη της CUDA για να πάρει τα καλύτερα αποτελέσματα που μπορεί.

```
#include <openacc.h>
....
acc_init(acc_device_nvidia);
....
#pragma acc kernels loop independent collapse(1) reduction(+:sum)
for(int i = start; i < end; i=i+1)
{
    curr_ord1 = lineitem[i][0];
     if(lineitem[i][3] > 1980)
    {
         for(int j = 0; j < ORDERS; j=j+4)
         {
            if((curr_ord1 == orders[j][0]) && (orders[j][2] < 1995))
               for(int k = 0; k < CUSTOMER; k++)
                  if((customer[k][0] == orders[j][1]) && (customer[k][1] == 2))
                        sum++;
            if((curr_ord1 == orders[j+1][0]) && (orders[j+1][2] < 1995))
               for(int k = 0; k < CUSTOMER; k++)
                  if((customer[k][0] == orders[j+1][1]) && (customer[k][1] == 2))
                        sum++;
            if((curr_ord1 == orders[j+2][0]) && (orders[j+2][2] < 1995))
               for(int k = 0; k < CUSTOMER; k++)
                  if((customer[k][0] == orders[j+2][1]) && (customer[k][1] == 2))
                        sum++;
             if((curr_ord1 == orders[j+3][0]) && (orders[j+3][2] < 1995))
               for(int k = 0; k < CUSTOMER; k++)
                  if((customer[k][0] == orders[j+3][1]) && (customer[k][1] == 2))
                        sum++;
         }
    }
}
```

Σχήμα 5.5(α) : Κώδικας C Φάσης 2 στο Query 3



Σχήμα 5.5(β) : Επίδοση Φάσης 2 στο Query 3



Σχήμα 5.5(γ) : Επί τοις εκατόν ποσοστιαία επίδοση της OpenACC σε σχέση με την επίδοση της CUDA για την Φάση 2 του Query 3

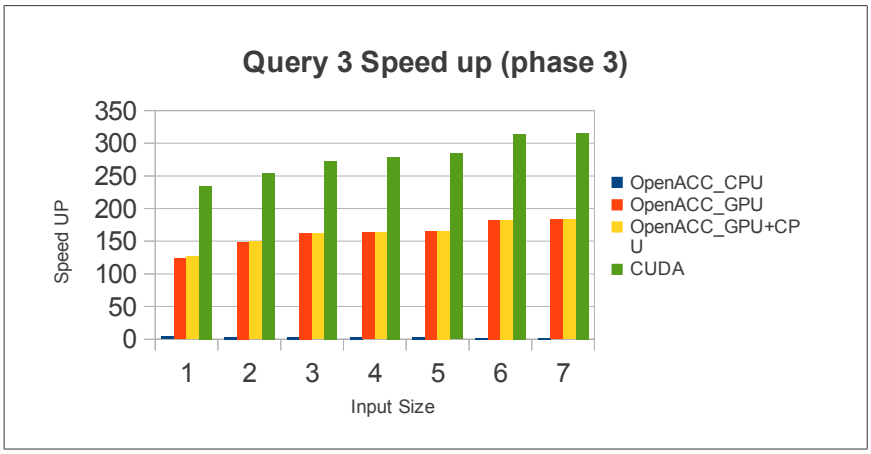
- Φάση 3

Στην φάση αυτή προσθέσαμε επίσης κάποιες βελτιώσεις στον κώδικα. Οι βελτιώσεις αυτές αφορούσαν τους πίνακες που μεταφέρονται στην συσκευή επιτάχυνσης(GPU). Μετατρέψαμε τους δισδιάστατους πίνακες σε μονοδιάστατους με τις απαραίτητες διαστάσεις μόνο. Στο Σχήμα 5.6(α) φαίνονται οι προσθήκες που κάναμε στο C κώδικα για να τρέξουμε το Query με το OpenACC API. Στο Σχήμα 5.6(β) παρουσιάζονται οι βελτιώσεις στην επίδοση της OpenACC(CPU, GPU, CPU+GPU) και της καλύτερης προσπάθειας για εκτέλεση σε CUDA για τα όλα τα μεγέθη αρχείων εισόδου . Επίσης στο Σχήμα 5.6(γ) παρουσιάζονται οι

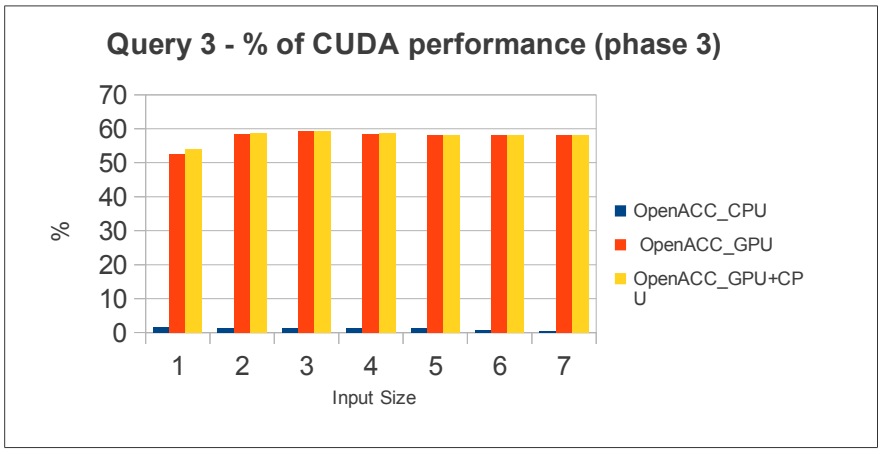
βελτιώσεις στην επίδοση της OpenACC(CPU, GPU, CPU+GPU) σε σχέση με τις βελτιώσεις στην επίδοση της CUDA για τα όλα τα μεγέθη αρχείων εισόδου. Όπως βλέπουμε από τις δύο αυτές γραφικές παραστάσεις έχουμε πολύ καλύτερα αποτελέσματα για σχετικά μικρότερα αρχεία εισόδου καταφέρνοντας να έχουμε speed up κοντά στο 60% για όλα τα αρχεία εισόδου. Η βελτιώσεις που παρατηρούνται σε σχέση με την προηγούμενη φάση οφείλονται στο γεγονός πως οι πίνακες που αντιγράφονται στην GPU για τους υπολογισμούς βρίσκονται σε συνεχόμενη μνήμη και είναι μικρότεροι σε σχέση με τους δισδιάστατους πίνακες που δεν βρίσκονται ούτε σε συνεχόμενη μνήμη.

```
.....
LINEITEM = count;
float lineitem_c[count*2];
for(i=0; i<count;i++){
    lineitem_c[i*2]=lineitem[i][0];
    lineitem_c[i*2+1]=lineitem[i][3];
}
.....
ORDERS = count2;
float orders_c[count2*4];
for(i2=0; i2<count2;i2++){
    orders_c[i2*4]=orders[i2][0];
    orders_c[i2*4+1]=orders[i2][1];
    orders_c[i2*4+2]=orders[i2][2];
    orders_c[i2*4+3]=orders[i2][3];
}
.....
CUSTOMER = count3;
float customer_c[count3*2];
for(i3=0; i3<count3;i3++){
    customer_c[i3*2]=customer[i3][0];
    customer_c[i3*2+1]=customer[i3][1];
}
....
#pragma acc kernels loop independent collapse(1) reduction(+:sum)
for(i = start; i < end; i++)
{
  ......
  ......
}
```

Σχήμα 5.6(α) : Κώδικας C Φάσης 3 στο Query 3



Σχήμα 5.6(β) : Επίδοση Φάσης 3 στο Query 3

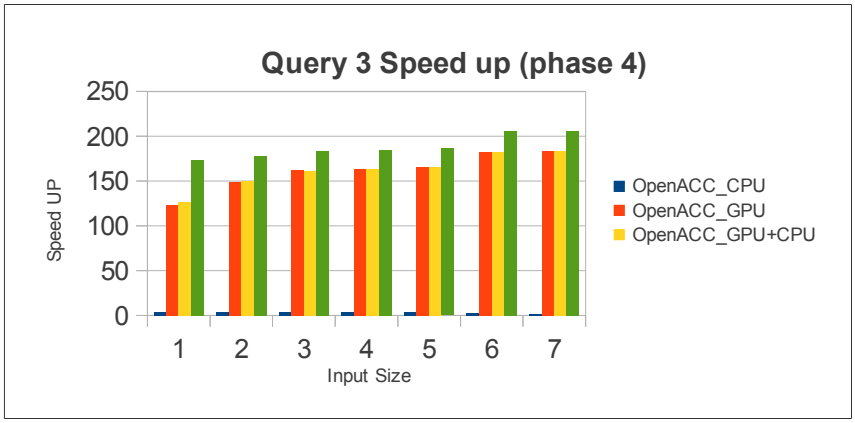


Σχήμα 5.6(γ) : Επί τοις εκατόν ποσοστιαία επίδοση της OpenACC σε σχέση με την επίδοση της CUDA για την Φάση 3 του Query 3

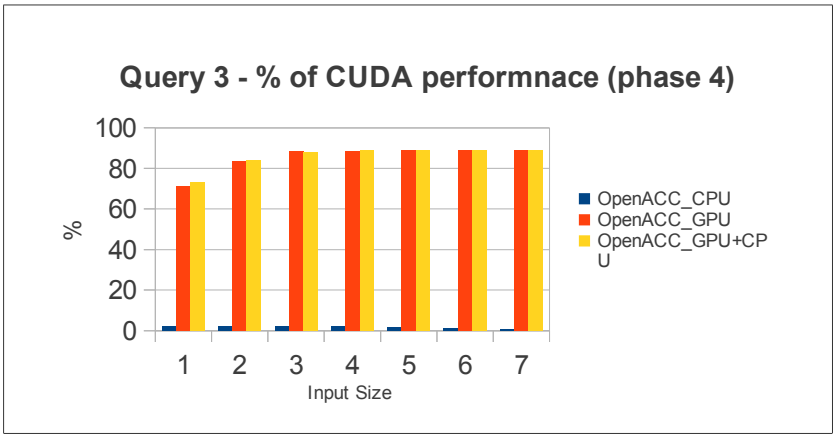
- Φάση 4

Στην φάση αυτή αφαιρέσαμε από τον κώδικα της CUDA κάποιες runtime ρουτίνες με στόχο να βρούμε τους λόγους της καλύτερης επίδοσης που παρουσιάζει η CUDA. Αυτές οι runtime ρουτίνες που χρησιμοποιούνται στη CUDA υλοποιούν με βέλτιστο τρόπο το loop unrolling καθώς έχουν την δυνατότητα να αξιοποιούν με πολύ καλύτερο τρόπο τους καταχωρητές της GPU. Αφαιρέσαμε λοιπόν τις ρουτίνες και υλοποιήσαμε το loop unrolling με την ίδια λογική που εφαρμόζεται και στην υλοποίηση μας με την OpenACC. Στο Σχήμα 5.7(α) παρουσιάζονται οι βελτιώσεις στην επίδοση της OpenACC(CPU, GPU, CPU+GPU) και της

καλύτερης προσπάθειας για εκτέλεση σε CUDA για τα όλα τα μεγέθη αρχείων εισόδου . Επίσης στο Σχήμα 5.7(β) παρουσιάζονται οι βελτιώσεις στην επίδοση της OpenACC(CPU, GPU, CPU+GPU) σε σχέση με τις βελτιώσεις στην επίδοση της CUDA για τα όλα τα μεγέθη αρχείων εισόδου. Όπως βλέπουμε σε αυτές τις δύο γραφικές παραστάσεις αφαιρώντας κάποιες runtime ρουτίνες που χρησιμοποιεί η CUDA και δεν μπορούν να χρησιμοποιηθούν κατά αντίστοιχο τρόπο στην OpenACC η βελτίωση της επίδοσης της OpenACC σε σχέση με την CUDA είναι περίπου κοντά στο 90% . Η καλύτερη speed up που συνεχίζει να παρουσιάζει ακόμα η CUDA οφείλετε στο γεγονός πως μπορεί η CUDA κατά την διάρκεια της εκτέλεσης θα προσαρμόσει το mapping στην GPU ανάλογα με το μέγεθος του προβλήματος ενώ με την OpenACC αυτό δεν μπορεί να γίνει αφού πρέπει πριν τη εκτέλεση να δοθούν στο μεταγλωττιστή οι οδηγίες για το πως θα κάνει το mapping και δεν είναι δυνατόν να γνωρίζουμε το μέγεθος του προβλήματος πριν από την εκτέλεση του κώδικα.



Σχήμα 5.7(α) : Επίδοση Φάσης 4 στο Query 3



Σχήμα 5.7(β) : Επί τοις εκατόν ποσοστιαία επίδοση της OpenACC σε σχέση με την επίδοση της CUDA για την Φάση 4 του Query 3

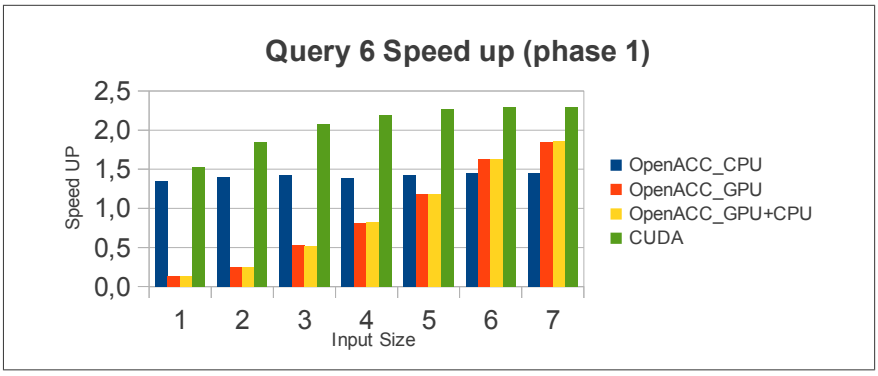
5.3.2 Query 6

- Φάση 1

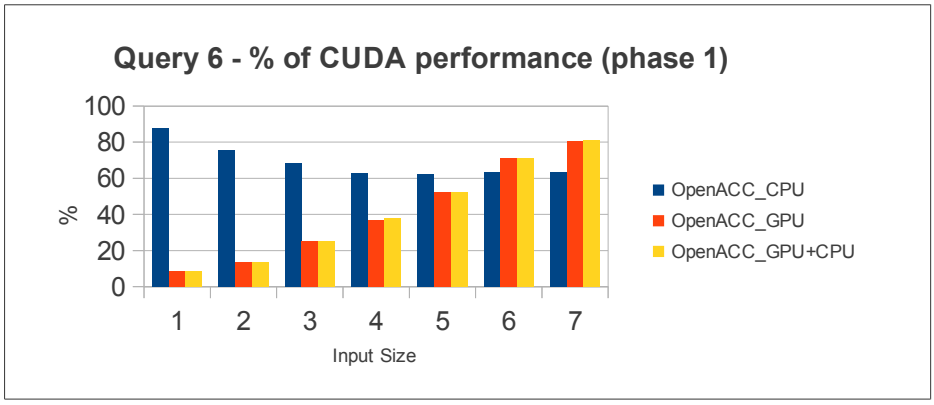
Στη φάση αυτή απλά προσθέσαμε μια οδηγία προς το μεταγλωττιστή. Στο Σχήμα 5.8(α) φαίνονται οι προσθήκες που κάναμε στο C κώδικα για να τρέξουμε το Query με το OpenACC API. Στο Σχήμα 5.8(β) παρουσιάζονται οι βελτιώσεις στην επίδοση της OpenACC(CPU, GPU, CPU+GPU) και της καλύτερης προσπάθειας για εκτέλεση σε CUDA για τα όλα τα μεγέθη αρχείων εισόδου που αναφέραμε στο προηγούμενο υποκεφάλαιο. Επίσης στο Σχήμα 5.8(γ) παρουσιάζονται οι βελτιώσεις στην επίδοση της OpenACC(CPU, GPU, CPU+GPU) σε σχέση με τις βελτιώσεις στην επίδοση της CUDA για τα όλα τα μεγέθη αρχείων εισόδου. Όπως βλέπουμε από τις δύο αυτές γραφικές παραστάσεις υπάρχει μια σταθερή καλή βελτίωση της επίδοσης χρησιμοποιώντας την CPU. Αυτό οφείλετε στο γεγονός πως το υπολογιστικό κομμάτι του Query αυτού είναι πολύ μικρό και η μεταφορά των δεδομένων στην GPU στοιχίζει περισσότερο από ότι ο υπολογισμός. Έτσι για αρχεία εισόδου 1-5 που είναι σχετικά πολύ μικρού υπολογιστικού μεγέθους για την εφαρμογή είναι προτιμότερη η χρήση CPU που δεν απαιτεί οποιαδήποτε μεταφορά δεδομένων αφού τα δεδομένα βρίσκονται είδη σε αυτήν. Για τα αρχεία εισόδου 6 και 7 που αυξάνεται και το υπολογιστικό κομμάτι η χρήση GPU ή GPU+CPU δίνει καλύτερα αποτελέσματα από την χρήση μόνο CPU. Οι επιδόσεις που παίρνουμε είναι εξαιρετικές αφού με προσθήκη μόνο μιας οδηγίας παίρνουμε 65%-87% της καλύτερης προσπάθειας με CUDA.

```
#include <openacc.h>
....
acc_init(acc_device_nvidia);
....
#pragma acc parallel loop num_gangs(256), vector_length(128) reduction(+:sum) reduction(+:result_count)
for(i = start; i < end; i=i+1)
{
        if((line_item[i][2]>=DATE1)&&(line_item[i][2]<DATE2)&&((line_item[i][1]>(DISCOUNT-0.01))&&(line_item[i][1]<(DISCOUNT+0.01)))&&(line_item[i][3]<QUANTITY))
        {
                sum+=line_item[i][0]*line_item[i][2];
                result_count++;
        }
}
```

Σχήμα 5.8(α) : Κώδικας C Φάσης 1 στο Query 6



Σχήμα 5.8(β) : Επίδοση Φάσης 1 στο Query 6



Σχήμα 5.8(γ) : Επί τοις εκατόν ποσοστιαία επίδοση της OpenACC σε σχέση με την επίδοση της CUDA για την Φάση 1 του Query 6

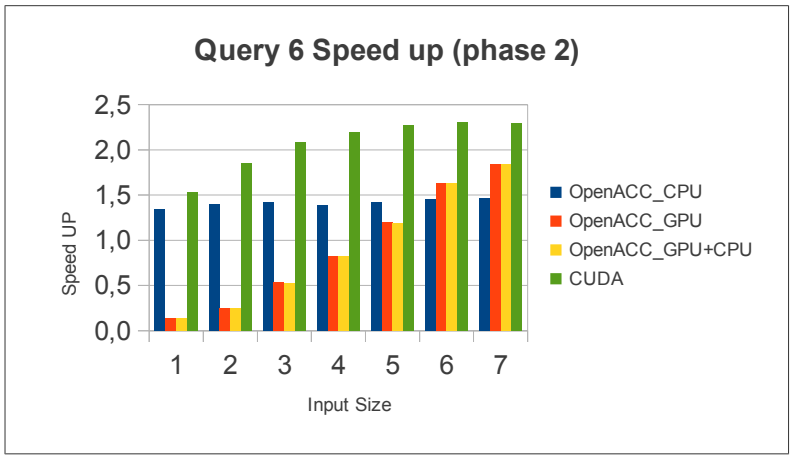
- Φάση 2

Στην φάση αυτή προσθέσαμε επιπλέον το καλύτερο δυνατό loop unrolling που μπορούσαμε για να πάρουμε καλύτερα αποτελέσματα για τις περιπτώσεις των GPU και GPU+CPU εκτελέσεων. Για την περίπτωση της CPU εκτέλεσης δεν εφαρμόσαμε loop unrolling διότι παίρναμε χειρότερα αποτελέσματα. Στο Σχήμα 5.9(α) φαίνονται οι προσθήκες που κάναμε στο C κώδικα για να τρέξουμε το Query με το OpenACC API. Στο Σχήμα 5.9(β) παρουσιάζονται οι βελτιώσεις στην επίδοση της OpenACC(CPU, GPU, CPU+GPU) και της καλύτερης προσπάθειας για εκτέλεση σε CUDA για τα όλα τα μεγέθη αρχείων εισόδου που αναφέραμε στο προηγούμενο υποκεφάλαιο. Επίσης στο Σχήμα 5.9(γ) παρουσιάζονται οι βελτιώσεις στην επίδοση της OpenACC(CPU, GPU, CPU+GPU) σε σχέση με τις βελτιώσεις στην επίδοση της CUDA για τα όλα τα μεγέθη αρχείων εισόδου. Όπως βλέπουμε από τις γραφικές παραστάσεις δεν καταφέραμε να πετύχουμε κάτι θετικό από το loop unrolling που

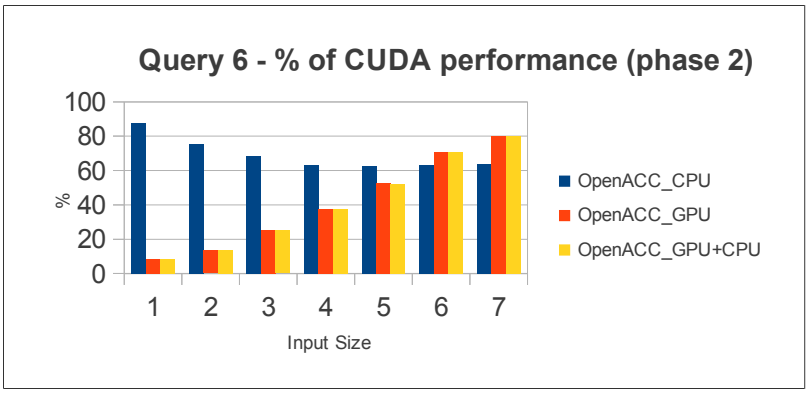
εφαρμόσαμε. Αυτό οφείλετε στο γεγονός πως το υπολογιστικό κομμάτι του Query 6 δεν είναι μεγάλο και έτσι οι πράξεις που αποφεύγονται με ένα loop unrolling 2 είναι πολύ περιορισμένες.

```
#include <openacc.h>
....
acc_init(acc_device_nvidia);
....
#pragma acc parallel loop num_gangs(256), vector_length(128) reduction(+:sum)
reduction(+:result_count)
for(i = start; i < end; i=i+2)
{
   if((line_item[i][2]>=DATE1) && (line_item[i][2]<DATE2) && ((line_item[i][1]>(DISCOUNT-0.01))
                    && (line_item[i][1]<(DISCOUNT+0.01))) && (line_item[i][3]<QUANTITY))
   {
          sum+=line_item[i][0]*line_item[i][2];
          result_count++;
   }
          if((line_item[i+1][2]>=DATE1) && (line_item[i+1][2]<DATE2) && ((line_item[i+1]
[1]>(DISCOUNT-0.01))&&(line_item[i+1][1]<(DISCOUNT+0.01)))&&(line_item[i+1]
[3]<QUANTITY))
   {
          sum+=line_item[i+1][0]*line_item[i+1][2];
          result_count++;
   }
}
```

Σχήμα 5.9(α) : Κώδικας C Φάσης 2 στο Query 6



Σχήμα 5.9(β) : Επίδοση Φάσης 2 στο Query 6



Σχήμα 5.9(γ) : Επί τοις εκατόν ποσοστιαία επίδοση της OpenACC σε σχέση με την επίδοση της CUDA για την Φάση 2 του Query 6

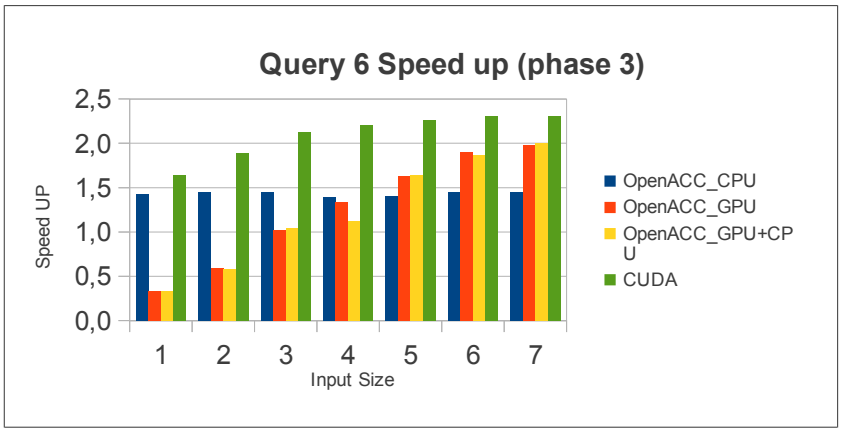
- Φάση 3

Στην φάση αυτή προσθέσαμε επίσης κάποιες βελτιώσεις στον κώδικα. Οι βελτιώσεις αυτές αφορούσαν τον πίνακα που μεταφερόταν στην συσκευή επιτάχυνσης(GPU). Μετατρέψαμε τον δισδιάστατο πίνακα σε μονοδιάστατο με τις απαραίτητες διαστάσεις μόνο. Οι απαραίτητες διαστάσεις στην περίπτωση αυτή ήταν όλες οι διαστάσεις του πίνακα, αλλά αυτή η μετατροπή πάλι είναι θετική διότι μεταφέρονται τα δεδομένα συνεχόμενα από την μνήμη. Στο Σχήμα 5.10(α) φαίνονται οι προσθήκες που κάναμε στο C κώδικα για να τρέξουμε το Query με το OpenACC API. Στο Σχήμα 5.10(β) παρουσιάζονται οι βελτιώσεις στην επίδοση της OpenACC(CPU, GPU, CPU+GPU) και της καλύτερης προσπάθειας για εκτέλεση σε CUDA για τα όλα τα μεγέθη αρχείων εισόδου που αναφέραμε στο προηγούμενο υποκεφάλαιο. Επίσης στο Σχήμα 5.10(γ) παρουσιάζονται οι βελτιώσεις στην επίδοση της OpenACC(CPU, GPU, CPU+GPU) σε σχέση με τις βελτιώσεις στην επίδοση της CUDA για τα όλα τα μεγέθη αρχείων εισόδου. Όπως βλέπουμε από τις δυο αυτές γραφικές παραστάσεις καταφέραμε να αυξήσουμε τις βελτιώσεις στην επίδοση χρησιμοποιώντας GPU ή GPU+CPU περίπου 5%-10% για όλα τα μεγέθη αρχείων εισόδου. Αυτό οφείλετε στο γεγονός πως τα δεδομένα στους πίνακες βρίσκονται τώρα συνεχόμενα στην μνήμη και μεταφέρονται ταχύτερα στην GPU.

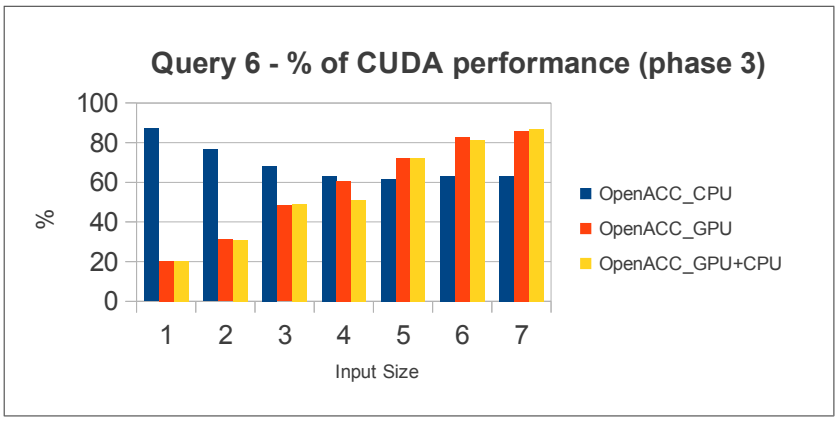
```
#include <openacc.h>
....
acc_init(acc_device_nvidia);
.....
float line_item_c[count*4];
for(i=0; i<count;i++){
        line_item_c[i*4]=line_item[i][0]; line_item_c[i*4+1]=line_item[i][1];
        line_item_c[i*4+1]=line_item[i][2]; line_item_c[i*4+3]=line_item[i][3];
        line_item_c[i*4+2]=line_item[i][2]; line_item_c[i*4+3]=line_item[i][3];
        line_item_c[i*4+3]=line_item[i][2]; line_item_c[i*4+3]=line_item[i][3];
}
.....
#pragma acc parallel loop num_gangs(256), vector_length(128) reduction(+:sum) reduction(+:result_count)
for(i = start; i < end; i=i+2){

        ......
}
```

Σχήμα 5.10(α) : Κώδικας C Φάσης 3 στο Query 6



Σχήμα 5.10(β) : Επίδοση Φάσης 3 στο Query 6



Σχήμα 5.10(γ) : Επί τοις εκατόν ποσοστιαία επίδοση της OpenACC σε σχέση με την επίδοση της CUDA για την Φάση 3 του Query 6

- Φάση 4

Στην φάση αυτή προσπαθήσαμε να αφαιρέσουμε από τον κώδικα της CUDA κάποιες runtime ρουτίνες με στόχο να βρούμε τους λόγους της καλύτερης επίδοσης που παρουσιάζει η CUDA. Όμως στο Query αυτό δεν υπήρχε κώδικας της CUDA που ήταν προς αλλαγή και έτσι τα αποτελέσματα παρέμειναν τα ίδια με την προηγούμενη φάση.

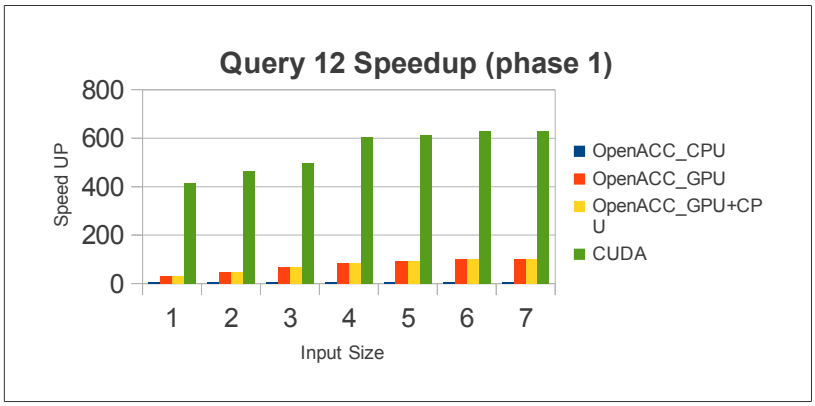
5.3.2 Query 12

- Φάση 1

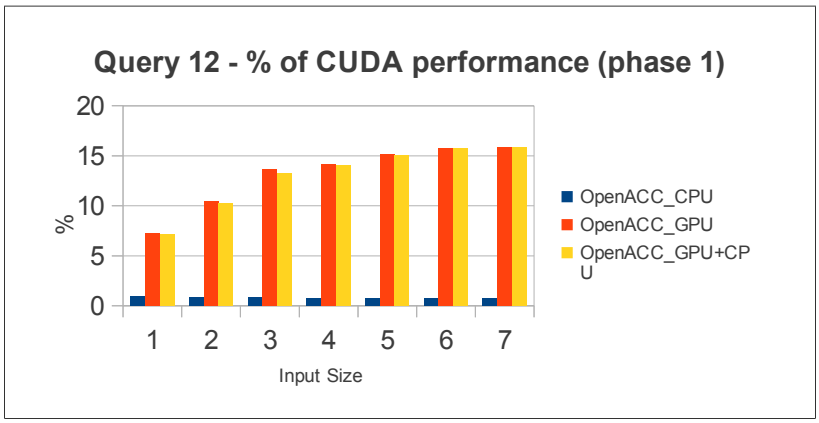
Στη φάση αυτή απλά προσθέσαμε μια οδηγία προς το μεταγλωττιστή. Σχήμα 5.11(α) φαίνονται οι προσθήκες που κάναμε στο C κώδικα για να τρέξουμε το Query με το OpenACC API. Στο Σχήμα 5.11(β) παρουσιάζονται οι βελτιώσεις στην επίδοση της OpenACC(CPU, GPU, CPU+GPU) και της καλύτερης προσπάθειας για εκτέλεση σε CUDA για τα όλα τα μεγέθη αρχείων εισόδου . Επίσης στο Σχήμα 5.11(γ) παρουσιάζονται οι βελτιώσεις στην επίδοση της OpenACC(CPU, GPU, CPU+GPU) σε σχέση με τις βελτιώσεις στην επίδοση της CUDA για τα όλα τα μεγέθη αρχείων εισόδου. Στις δύο αυτές γραφικές παραστάσεις βλέπουμε πως παρόλο που παίρνουμε πολύ καλό speedup σε σχέση με την σειριακή εκτέλεση εντούτοις η speed up αυτή είναι πολύ χαμηλή σε σχέση με αυτήν που παίρνουμε στην καλύτερη προσπάθεια με CUDA. Παρατηρούμε επίσης ότι η speedup αυξάνεται όσο μεγαλύτερα αρχεία εισόδου έχουμε και αυτό οφείλετε στο γεγονός πως η χρήση της GPU γίνεται σε μεγαλύτερο βαθμό αφού το κομμάτι υπολογισμού γίνεται ολοένα και μεγαλύτερο.

```
#include <openacc.h>
....
acc_init(acc_device_nvidia);
....
#pragma acc parallel loop num_gangs(1024), vector_length(512) reduction(+:sum)
for(int j =0; j < LINEITEM; j=j+1){
        for(int i = 0; i < ORDERS; i=i+1)
        {
                if((lineitem[j][4] >= 1994) && (lineitem[j][4] < 1995))
                        if (lineitem[j][0] == orders[i][0])
                                sum++;
        }
}
```

Σχήμα 5.11(α) : Κώδικας C Φάσης 1 στο Query 12



Σχήμα 5.11(β) : Επίδοση Φάσης 1 στο Query 12



Σχήμα 5.11(γ) : Επί τοις εκατόν ποσοστιαία επίδοση της OpenACC σε σχέση με την επίδοση της CUDA για την Φάση 1 του Query 12

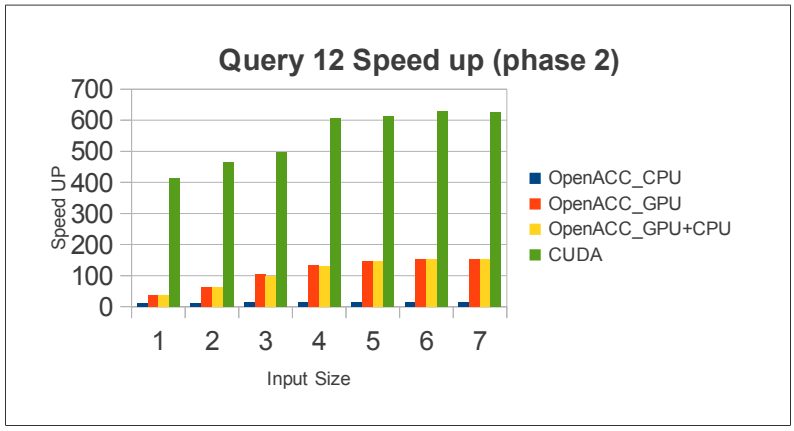
- Φάση 2

Στην φάση αυτή προσθέσαμε επιπλέον το καλύτερο δυνατό loop unrolling που μπορούσαμε για να πάρουμε καλύτερα αποτελέσματα. Σχήμα 5.12(α) φαίνονται οι προσθήκες που κάναμε στο C κώδικα για να τρέξουμε το Query με το OpenACC API. Στο Σχήμα 5.12(β) παρουσιάζονται οι βελτιώσεις στην επίδοση της OpenACC(CPU, GPU, CPU+GPU) και της καλύτερης προσπάθειας για εκτέλεση σε CUDA για τα όλα τα μεγέθη αρχείων εισόδου . Επίσης στο Σχήμα 5.12(γ) παρουσιάζονται οι βελτιώσεις στην επίδοση της OpenACC(CPU, GPU, CPU+GPU) σε σχέση με τις βελτιώσεις στην επίδοση της CUDA για τα όλα τα μεγέθη αρχείων εισόδου. Εδώ βλέπουμε πως η χρήση του loop unrolling μας βοήθησε να αυξήσουμε την speed up αλλά σε σύγκριση και πάλι με την CUDA βρίσκεται σε χαμηλά επίπεδα.

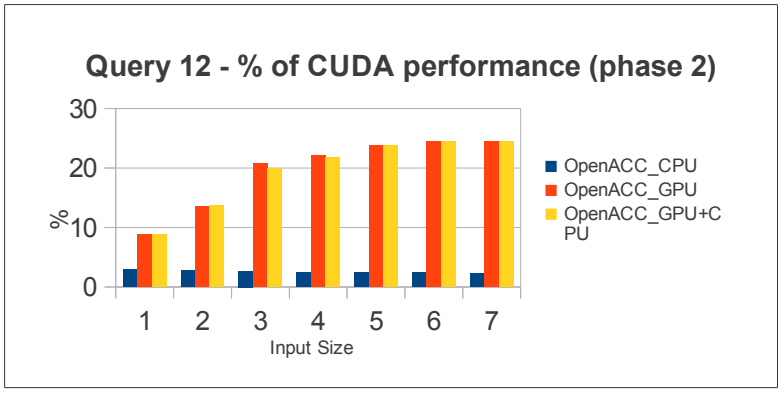
Ωστόσο η προσπάθεια για να πάρουμε τα αποτελέσματα αυτά είναι πολύ μικρότερη στην OpenACC σε σχέση με την CUDA.

```
#include <openacc.h>
....
acc_init(acc_device_nvidia);
....
#pragma acc parallel loop num_gangs(1024), vector_length(512) reduction(+:sum)
for(int j =0; j < LINEITEM; j=j+1)
{
        for(int i = 0; i < ORDERS; i=i+8)
        {
                if((lineitem[j][4] >= 1994) && (lineitem[j][4] < 1995))
                {
                        if (lineitem[j][0] == orders[i][0])
                                sum++;
                        if (lineitem[j][0] == orders[i+1][0])
                                sum++;
                        if (lineitem[j][0] == orders[i+2][0])
                                sum++;
                        if (lineitem[j][0] == orders[i+3][0])
                                sum++;
                        if (lineitem[j][0] == orders[i+4][0])
                                sum++;
                        if (lineitem[j][0] == orders[i+5][0])
                                sum++;
                        if (lineitem[j][0] == orders[i+6][0])
                                sum++;
                        if (lineitem[j][0] == orders[i+7][0])
                                sum++;
                }
        }
}
```

Σχήμα 5.12(α) : Κώδικας C Φάσης 2 στο Query 12



Σχήμα 5.12(β) : Επίδοση Φάσης 2 στο Query 12



Σχήμα 5.12(γ) : Επί τοις εκατόν ποσοστιαία επίδοση της OpenACC σε σχέση με την επίδοση της CUDA για την Φάση 2 του Query 12

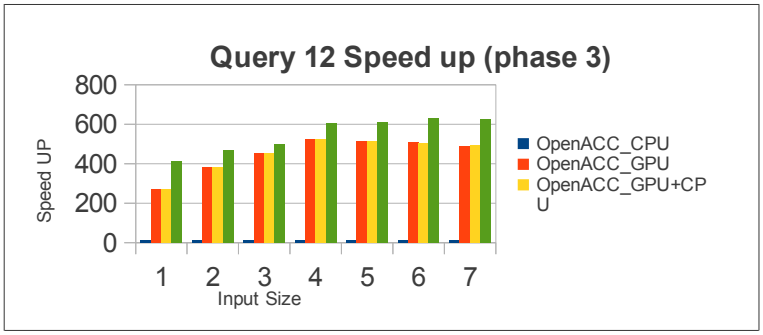
- Φάση 3

Στην φάση αυτή προσθέσαμε επίσης κάποιες βελτιώσεις στον κώδικα. Οι βελτιώσεις αυτές αφορούσαν τους πίνακες που μεταφέρονται στην συσκευή επιτάχυνσης(GPU). Μετατρέψαμε τους δισδιάστατους πίνακες σε μονοδιάστατους με τις απαραίτητες διαστάσεις μόνο. Σχήμα 5.12(α) φαίνονται οι προσθήκες που κάναμε στο C κώδικα για να τρέξουμε το Query με το OpenACC API. Στο Σχήμα 5.12(β) παρουσιάζονται οι βελτιώσεις στην επίδοση της OpenACC(CPU, GPU, CPU+GPU) και της καλύτερης προσπάθειας για εκτέλεση σε CUDA για τα όλα τα μεγέθη αρχείων εισόδου . Επίσης στο Σχήμα 5.12(γ) παρουσιάζονται οι βελτιώσεις στην επίδοση της OpenACC(CPU, GPU, CPU+GPU) σε σχέση με τις βελτιώσεις στην επίδοση της CUDA για τα όλα τα μεγέθη αρχείων εισόδου. Στην φάση αυτή βλέπουμε την εκτόξευση την βελτίωσης στην επίδοση σε πολύ ψηλά επίπεδα καταφέρνοντας να φτάσουμε κατά μέσο όρο γύρω στο 80%-85% της επίδοσης της καλύτερης προσπάθειας της CUDA. Η τεράστια αυτή θετική αλλαγή στην φάση αυτή οφείλετε στο ότι μετατρέψαμε τους πίνακες από δισδιάστατους σε μονοδιάστατους μόνο των απαραίτητων διαστάσεων καταφέρνοντας να μειώσουμε το κόστος μεταφοράς των δεδομένων στην κάρτα γραφικών καθώς και το κόστος διαβάσματος κάθε φορά από ένα δισδιάστατο πίνακα σε σχέση με το κόστος από το να διαβάσουμε δεδομένα από ένα συνεχόμενο μονοδιάστατο πίνακα. Αυτό διότι για τους δισδιάστατους πίνακες πρέπει να κάνουμε πρόσβαση στην μνήμη δύο φορές αντί μια φορά.

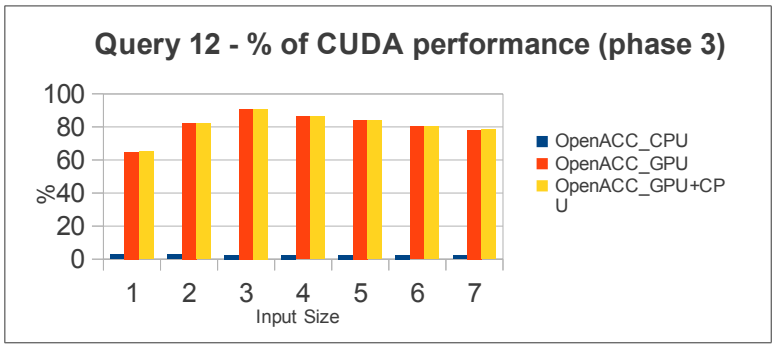
```
#include <openacc.h>
....
acc_init(acc_device_nvidia);
.....
LINEITEM=count;
float lineitem_c[count*2];
for(i=0; i<count;i++)
{
        lineitem_c[i*2]=lineitem[i][0];
        lineitem_c[i*2+1]=lineitem[i][4];
}
.....
ORDERS=count2;
float orders_c[count2];
for(i=0; i<count2;i++)
        orders_c[i]=orders[i][0];
....

#pragma acc parallel loop num_gangs(1024), vector_length(512) reduction(+:sum)
for(int j =0; j < LINEITEM; j=j+1)
{
        .......
}
```

Σχήμα 5.13(α) : Κώδικας C Φάσης 3 στο Query 12



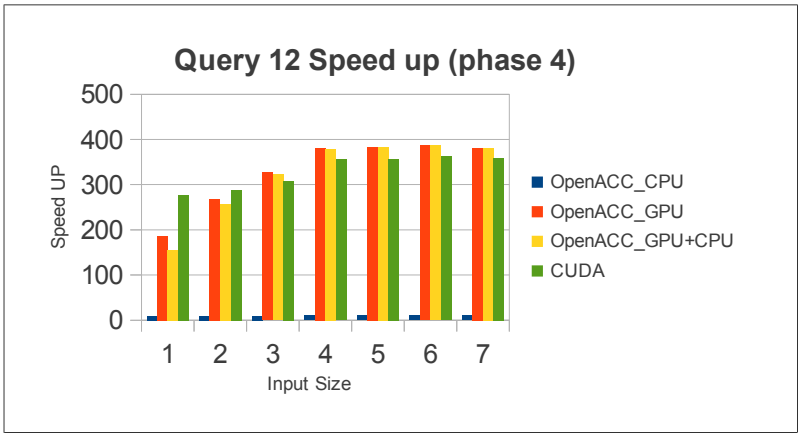
Σχήμα 5.13(β) : Επίδοση Φάσης 3 στο Query 12



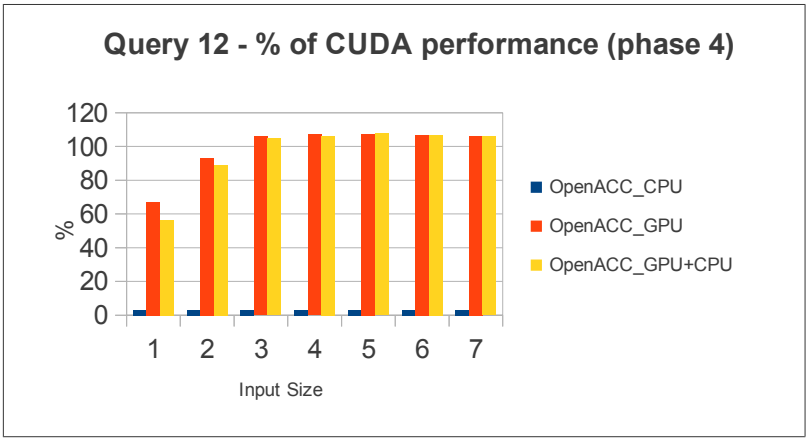
Σχήμα 5.13(γ) : Επί τοις εκατόν ποσοστιαία επίδοση της OpenACC σε σχέση με την επίδοση της CUDA για την Φάση 3 του Query 12

- Φάση 4

Στην φάση αυτή αφαιρέσαμε από τον κώδικα της CUDA κάποιες runtime ρουτίνες με στόχο να βρούμε τους λόγους της καλύτερης επίδοσης που παρουσιάζει η CUDA. Επίσης εφαρμόσαμε και στις 2 περιπτώσεις το ίδιο loop unrolling 4.Στο Σχήμα 5.14(α) παρουσιάζονται οι βελτιώσεις στην επίδοση της OpenACC(CPU, GPU, CPU+GPU) και της εκτέλεση σε CUDA για την φάση αυτή. Στο Σχήμα 5.14(β) παρουσιάζονται οι βελτιώσεις στην επίδοση της OpenACC(CPU, GPU, CPU+GPU) σε σχέση με τις βελτιώσεις στην επίδοση της CUDA για την φάση αυτή. Όπως βλέπουμε σε αυτές τις δύο γραφικές παραστάσεις αφαιρώντας κάποιες runtime ρουτίνες που χρησιμοποιεί η CUDA και δεν μπορούν να χρησιμοποιηθούν κατά αντίστοιχο τρόπο στην OpenACC η βελτίωση της επίδοσης της OpenACC σε σχέση με την CUDA είναι περίπου στα ίδια επίπεδα με την CUDA. Μόνο για τα αρχεία εισόδου μεγέθους 1 έχουμε επίδοση σχετικά μικρότερη από την CUDA(67%) και αυτό οφείλετε στο γεγονός ότι το mapping που γίνεται δεν είναι το πλέον αποδοτικότερο ώστε να αξιοποιείτε στο μέγιστο η GPU.



Σχήμα 5.14(α) : Επίδοση Φάσης 4 στο Query 12

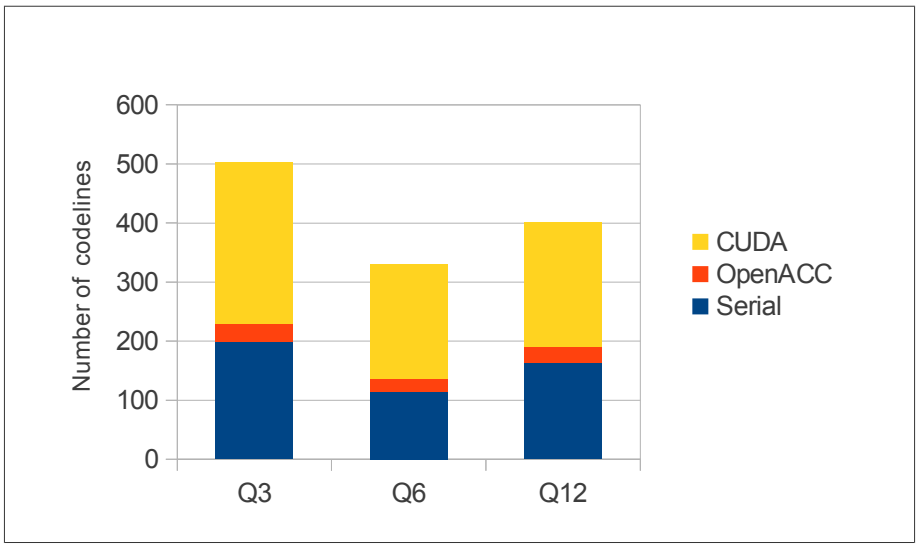


Σχήμα 5.14(β) : Επί τοις εκατόν ποσοστιαία επίδοση της OpenACC σε σχέση με την επίδοση της CUDA για την Φάση 4 του Query 12

5.4 Σύγκριση OpenACC με CUDA

Ένας από τους στόχους αυτής αυτής της εργασίας ήταν να συγκρίνουμε την OpenACC με την CUDA όσον αφορά την επίδοση που μπορούμε να πάρουμε χρησιμοποιώντας αυτά τα δύο πρότυπα παράλληλου προγραμματισμού αλλά και την προγραμματιστική προσπάθεια που χρειάζεται να βάλουμε ώστε να πάρουμε τις επιδόσεις αυτές. Για να ύπαρξη σύγκριση της προσπάθειας που καταβάλαμε θα συγκρίνουμε τον αριθμό γραμμών κώδικα καθώς και τον χρόνο που μας πήρε να ολοκληρώσουμε την κάθε φάση. Όσον αφορά τον κώδικα της CUDA και την προσπάθεια που καταβλήθηκε τα στοιχεία τα πήραμε από τον έμπειρο προγραμματιστή που υλοποίησε την καλύτερη προσπάθεια με CUDA. Στον Σχήμα 5.15 παρουσιάζεται ο αριθμός γραμμών κώδικα καθώς και ο χρόνος της κάθε υλοποίησης.

Βλέποντας τα στοιχεία του πίνακα αυτού καθώς και τις προηγούμενες γραφικές παραστάσεις της επίδοσης που μας δίνει η OpenACC σε σχέση με την CUDA μπορούμε εύκολα να καταλάβουμε πως χρησιμοποιώντας την OpenACC για να παραλληλοποιήσουμε μια εφαρμογή είναι πολύ εύκολος τρόπος, μη χρονοβόρος και δίνει πολύ καλά αποτελέσματα. Για το Query 3 για παράδειγμα προσθέσαμε 31 γραμμές κώδικα για να πάρουμε το καλύτερο αποτέλεσμα μη έχοντας προηγούμενη εμπειρία της OpenACC. Ενώ για την CUDA ο κ. Frederico, ο οποίος είναι έμπειρος προγραμματιστής CUDA, χρειάστηκε να γράψει 305 γραμμές περισσότερο κώδικα από τον σειριακό κώδικα του Query αυτού.



Σχήμα 5.15: Αριθμός γραμμών κώδικα για κάθε καλύτερη προσπάθεια με OpenACC και CUDA για τα 3 Queries

Κεφάλαιο 6

Συμπεράσματα

6.1 Συμπεράσματα

Στην εργασία μας αυτή είχαμε σαν στόχο να δείξουμε τις δυνατότητες που μας προσφέρει το μοντέλο της OpenACC. Δείξαμε ότι με πολύ απλό τρόπο προσθέτοντας λίγες μόνο γραμμές κώδικα σε σειριακές εφαρμογές μπορούμε να πάρουμε επιδόσεις που είναι πολύ κοντά στις επιδόσεις που δίνει μια υλοποίηση των ίδιων σειριακών εφαρμογών από έμπειρο προγραμματιστή της CUDA. Ενώ εμείς δεν διαθέταμε πείρα στην παραλληλοποίηση εφαρμογών με τη χρήση της OpenACC εντούτοις καταφέραμε να πάρουμε πολύ καλές επιδόσεις σε πολύ λιγότερο χρόνο και με λιγότερη προσπάθεια. Επίσης καταφέραμε να δείξουμε ότι η OpenACC είναι φορητή δίνοντας μας την ευχέρεια να αξιοποιήσουμε ότι συσκευή(CPU, GPU, etc.) έχουμε στην διάθεσή μας. Στο Query 6 όπου το υπολογιστικό κομμάτι της εφαρμογής ήταν σχετικά μικρό μας βόλευε καλύτερα η χρήση της CPU αντί της GPU για τα μικρά αρχεία εισόδου ενώ για τα μεγάλα αρχεία εισόδου του Query αυτού μας έδινε καλύτερα αποτελέσματα η χρήση της GPU ή η χρήση της GPU και της CPU μαζί. Στα άλλα δύο Query(Query3, Query12) όπου το υπολογιστικό κομμάτι των εφαρμογών ήταν πολύ μεγάλο η χρήση της CPU μόνης της δεν ήταν ιδιαίτερα αποδοτική ενώ η χρήση της GPU ήταν εξαιρετικά αποδοτική.

Ένα ακόμα σημαντικό συμπέρασμα που βγαίνει μέσα από την δουλεία μας είναι η επιλογή του είδους των οδηγιών(directive) που θα δώσουμε στο μεταγλωττιστή πριν από το παράλληλο κομμάτι της κάθε εφαρμογής. Σε εφαρμογές όπου ο αριθμός των επαναλήψεων ενός βρόγχου θα είναι σταθερός και δεν υπάρχουν συνθήκες ελέγχου πριν από ένα εσωτερικό βρόγχο η κατάλληλη OpenACC οδηγία προς τον μεταγλωττιστή είναι το parallel construct. Αυτό γιατί ο μεταγλωττιστής ξέρει πριν την εκτέλεση ότι σίγουρα όλες οι επαναλήψεις θα εκτελεστούν και αφού η λογική λειτουργίας αυτού του construct είναι πως όταν δημιουργηθούν τα gangs, ο αριθμός των gangs και των workers σε κάθε gang παραμένει σταθερός σε όλη την διάρκεια της παράλληλης περιοχής έτσι η χρήση του parallel construct είναι καταλληλότερη σε τέτοιου είδους παράλληλες περιοχές κώδικα. Στο Query 6 και στο Query 12 χρησιμοποιήσαμε το parallel construct διότι ο αριθμός των ήταν σταθερός. Αντίθετα σε εφαρμογές όπου ο αριθμός των επαναλήψεων δεν είναι σταθερός και εξαρτάται από συνθήκες πριν από βρόγχους, όπως συνέβαινε στο Query 3, η κατάλληλη OpenACC οδηγία προς τον μεταγλωττιστή είναι το kernels construct. Η λογική του kernels construct που το κάνει καταλληλότερο σε τέτοιου είδους παράλληλες περιοχές κώδικα είναι πως όταν το πρόγραμμα συνάντηση ένα kernels construct, θα ξεκινήσει την ακολουθία των πυρήνων σε σειρά στην συσκευή. Ο αριθμός και η διαμόρφωση των gangs από workers και το μέγεθος του vector(νήματος) μπορεί να είναι διαφορετικός σε κάθε kernel.

Συμπεραίνουμε επίσης πως ή αύξηση της επίδοσης με την χρήση μιας GPU συσκευής μπορεί να φτάσει μέχρι ενός σημείου. Αυτό οφείλετε στο γεγονός πως ο αριθμός πυρήνων μιας GPU είναι σταθερός και αν αξιοποιείτε πλήρως από ένα σημείο και μετά δεν θα μας επιφέρει περεταίρο αύξηση στην επίδοση. Αυτό συμβαίνει ακριβώς και στις εφαρμογές με τις οποίες δουλέψαμε εμείς. Ενώ για τα αρχεία εισόδου μεγέθους από 1 μέχρι 4 αυξάνεται με μεγάλο ρυθμό η βελτίωση της επίδοσης στις περιπτώσεις των αρχείων εισόδου μεγέθους 5, 6, 7 η βελτίωση της επίδοσης είναι πολύ μικρή ενώ και σε μερικές περιπτώσεις για τα αρχεία εισόδου μεγέθους 7 η βελτίωση της επίδοσης μειώνετε ή παραμένει η ίδια σε σχέση με τα αρχεία εισόδου 6 ή και σε σχέση με πιο μικρά αρχεία εισόδου.

6.2 Μελλοντική Εργασία

Από τα αποτελέσματα που καταφέραμε να πάρουμε από αυτή τη δουλειά είμαστε πολύ ικανοποιημένοι γιατί καταφέραμε να πετύχουμε τους στόχους που θέσαμε στα αρχικά στάδια της εργασίας. Καταφέραμε να πάρουμε μια πρώτη γεύση με το νέο αυτό μοντέλο

προγραμματισμού, που ονομάζεται OpenACC, και να δείξουμε τα μεγάλα πλεονεκτήματα που μπορεί να μας δώσει η χρήση της OpenACC σε σχέση με πιο χαμηλού επιπέδου μοντέλα παράλληλου προγραμματισμού, όπως είναι η CUDA. Η μελέτη όμως αυτή δεν σταματά εδώ αφού στοχεύουμε στο προσεχές μέλλον να συγκρίνουμε την OpenACC και με άλλου είδους μοντέλα όπως την OpenCL, που μοιάζει πολύ με την CUDA αλλά είναι πιο φορητή αφού μπορεί να αξιοποιηθεί από διαφορετικές GPUs και όχι μόνο αυτές της NVIDIA, και την OpenMP η οποία είναι directive based πρότυπο προγραμματισμού αλλά αξιοποιείτε κυρίως από τις CPUs.

Στοχεύουμε επίσης να τρέξουμε τις εφαρμογές αυτές σε διαφορετική GPU αρχιτεκτονική της NVIDIA που είναι η Kepler αρχιτεκτονική. Να υπενθυμίσουμε ότι σε αυτή την εργασία χρησιμοποιήσαμε GPU με Fermi αρχιτεκτονική(GTX 580). Ένας Streaming Multiprocessor στην Kepler αρχιτεκτονική διαθέτει πολύ περισσότερους πυρήνες από ένα της Fermi αρχιτεκτονικής. Αναμένουμε λοιπόν να πάρουμε πολύ μικρότερους χρόνους με την χρήση της Kepler αρχιτεκτονικής.

Μια άλλη ιδέα που μπορεί να μελετηθεί για να δούμε κατά πόσο μπορεί να δουλέψει είναι να διασπάσουμε το υπολογιστικό κομμάτι της κάθε μιας από τις δύο εφαρμογές που δουλέψαμε(Query3, Query 12), το οποίο είναι πολύ μεγάλο, και αφού η μηχανή στην οποία δουλεύουμε διαθέτει 2 GPUs να αξιοποιήσουμε και τις δύο περνώντας πολύ καλύτερους χρόνους.

Βιβλιογραφία

ΑΡΘΡΑ ΑΠΌ ΔΗΜΟΣΙΕΥΜΕΝΑ ΠΡΑΚΤΙΚΑ ΕΠΙΣΤΗΜΟΝΙΚΏΝ ΣΥΝΕΔΡΙΩΝ

[1] Levesque, John M., Ramanan Sankaran, and Ray Grout. "Hybridizing S3D into an exascale application using OpenACC: an approach for moving to multi-petaflops and beyond." Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis. IEEE Computer Society Press, 2012.

[2] Wienke, Sandra, et al. "OpenACC—First Experiences with Real-World Applications." Euro-Par 2012 Parallel Processing. Springer Berlin Heidelberg, 2012. 859-870.

[3] Reyes, Ruymán, et al. "A Comparative Study of OpenACC Implementations." XXIII Jornadas de Paralelismo (2012).

[4] Grauer-Gray, Scott, et al. "Accelerating Financial Applications on the GPU." (2013).

[5] James Kirk and Montgomery Scott, Starfleet Academy. “Exploiting Scalable Many-Core Architectures for Decision Support System Workloads: Optimizing TPC-H Queries on the Intel SCC”.

ΒΙΒΛΙΑ

[6] Singh J. P., Culler D., and Gupta A.. Parallel Computer Architecture: A Hardware/Software Approach. Morgan Kaufmann Publishers, 1998.

ΕΓΧΕΙΡΙΔΙΑ

[7] OpenACC.1.0_0
http://www.openacc.org/sites/default/files/OpenACC.1.0_0.pdf

[8] TCP-H version 2.15.0
http://www.tpc.org/tpch/spec/tpch2.15.0.pdf

ΗΛΕΚΤΡΟΝΙΚΕΣ ΠΗΓΕΣ

[9] Michael Wolfe, PGI Compiler Engineer. "OpenACC Kernels and Parallel Constructs", August 2012
http://www.pgroup.com/lit/articles/insider/v4n2a1.htm

[10] Michael Wolfe, PGI Compiler Engineer. "The PGI Accelerator Compilers with OpenACC", March 2012
http://www.pgroup.com/lit/articles/insider/v4n1a1.htm

[11] Michael Wolfe, PGI Compiler Engineer. "OpenACC Features in PGI Accelerator Fortran Compilers—Part 1", March 2012
http://www.pgroup.com/lit/articles/insider/v4n1a1a.htm

[12] Programming GPUs with OpenACC (Part 1 of 3)
http://developer.download.nvidia.com/GTC/PDF/GTC2012/PresentationPDF/S0517AMonday-Programming-GPUs-OpenACC.pdf

[13] Programming GPUs with OpenACC (Part 2 of 3)
http://developer.download.nvidia.com/GTC/PDF/GTC2012/PresentationPDF/S0517BMonday-Programming-GPUs-OpenACC.pdf

[14] Programming GPUs with OpenACC (Part 3 of 3)
http://developer.download.nvidia.com/GTC/PDF/GTC2012/PresentationPDF/S0517C-Monday-Programming-GPUs-OpenACC.pdf

[15] Official Website of Caps-Entreprise
http://www.caps-entreprise.com/

[16] Official Website of Cray: The Supercomputer Company
http://www.cray.com/Home.aspx

[17] Official Website of The Portland Group Inc
http://www.pgroup.com

[18] Official Website of Nvidia
http://www.nvidia.co.uk/page/home.html

[19] PGI Accelerator™ compilers
http://www.pgroup.com/resources/accel.htm