Ατομική Διπλωματική Εργασία

ΕΚΤΕΛΕΣΗ ΤΟΥ QUERY 6 ΤΟΥ BENCHMARK TPC-H ΣΕ ΚΑΡΤΑ ΓΡΑΦΙΚΩΝ, ΜΕ CUDA

Μαρία Λοϊζίδη

ΠΑΝΕΠΙΣΤΗΜΙΟ ΚΥΠΡΟΥ

ΤΜΗΜΑ ΠΛΗΡΟΦΟΡΙΚΗΣ

Δεκέμβριος 2009

ΠΑΝΕΠΙΣΤΗΜΙΟ ΚΥΠΡΟΥ

ΤΜΗΜΑ ΠΛΗΡΟΦΟΡΙΚΗΣ

ΕΚΤΕΛΕΣΗ ΤΟΥ QUERY 6 ΤΟΥ BENCHMARK TPC-H ΣΕ ΚΑΡΤΑ ΓΡΑΦΙΚΩΝ, ΜΕ CUDA

Μαρία Λοϊζίδη

Επιβλέπων Καθηγητής
Pedro Trancoso

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

Δεκέμβριος 2009

Ευχαριστίες

Στα πλαίσια ανάπτυξης και ολοκλήρωσης της διπλωματικής μου εργασίας ένα σημαντικό άτομο το οποίο με βοήθησε ιδιαίτερα με τις συμβουλές του και με τη βοήθεια του ήταν ο επιβλέπων καθηγητής Δρ. Pedro Trancoso, καθηγητής του τμήματος πληροφορικής του Πανεπιστημίου Κύπρου.

Ως επιβλέπων καθηγητής έχοντας πολλή εμπειρία και γνώση όσο αφορά την παράλληλη επεξεργασία, με βοήθησε να κατανοήσω κάποια σημαντικά θέματα όσο αφορά τον παράλληλο υπολογισμό και να μπορέσω να προχωρήσω με την εργασία μου. Τελικά πέραν της ολοκλήρωσης της διπλωματικής εργασίας, κατάφερα με την βοήθεια του Δρ. Pedro Trancoso να εμβαθύνω στην θεωρία του παράλληλου υπολογισμού, ενός θέματος πολύ σημαντικού στον τομέα της πληροφορικής στις μέρες μας. Ασχολήθηκα με πολύ ενδιαφέροντες αλγορίθμους και έμαθα την λειτουργία της γλώσσας προγραμματισμού CUDA.

Θα ήθελα επίσης να ευχαριστήσω το διδακτορικό φοιτητή Παναγιώτη Πετρίδη, που με βοήθησε σε κάποιες δυσκολίες που αντιμετώπισα κατά τη διάρκεια της εργασίας μου.

Περίληψη

Κατά την ανάπτυξη και ολοκλήρωση της ερευνητικής μου εργασίας ερευνήθηκαν διάφορα θέματα που σχετίζονται με τον παράλληλο υπολογισμό και πιο συγκεκριμένα μελετήθηκε η γλώσσα προγραμματισμού CUDA της NVIDIA και εκτελέστηκε ο αλγόριθμος βάσεων δεδομένων Q6 (Q6: Only a single scan operation) σύμφωνα με αυτό. Για το αλγόριθμο αυτό έγινε στην συνέχεια πειραματική αξιολόγηση ώστε να εξαχθούν διάφορα συμπεράσματα.

Για να εκμεταλλευτούμε τις τεράστιες υπολογιστικές δυνατότητες που προσφέρονται από την ανάπτυξη συστημάτων παράλληλης επεξεργασίας, επινοήθηκαν παράλληλοι αλγόριθμοι που να μπορούν να διαχειρίζονται ένα μεγάλο αριθμό επεξεργαστών και θα δουλεύουν παράλληλα για την γρήγορη επίλυση σύνθετων υπολογιστικών προβλημάτων. Για να το επιτύχουμε αυτό απαιτείται από μέρους των επεξεργαστών αποδοτική συνεργασία μεταξύ τους και προσεκτική διαχείριση των διαθέσιμων πόρων. Το πακέτο CUDA από την NVIDIA είναι το πιο διαδεδομένο για εφαρμογές GPGPU. Βασίζεται σε μια επεκταμένη έκδοση της γλώσσας C. Η CUDA παρέχει τη δυνατότητα της εκτέλεσης κάποιων υπολογισμών παράλληλα στο GPU με τη χρήση Kernel Function. Έχει συναρτήσεις που μεταφέρουν το περιεχόμενο μιας μεταβλητής του CPU σε μια μεταβλητή του GPU και το αντίθετο.

Μετά την μελέτη της CUDA ακολουθεί η υλοποίηση του αλγόριθμου Q6 (Q6: Only a single scan operation) και η πειραματική αξιολόγηση του που οδήγησε σε διάφορα συμπεράσματα όσο αφορά την συμπεριφορά του αλγόριθμου αλλά και της γλώσσας προγραμματισμού CUDA.

Περιεχόμενα

Κεφάλαιο **1** **Εισαγωγή…..... 1**

1.1 Κίνητρο διπλωματικής εργασίας 2

1.2 Στόχος διπλωματικής εργασίας 3

1.3 Μεθοδολογία που ακολουθήθηκε 3

1.4 Οργάνωση Κειμένου 5

Κεφάλαιο **2 Παραλληλισμός……...…….……... 6**

2.1 GPUs και GPU σε CUDA με TPC-H Queries 6

2.2 Παράλληλος υπολογισμός και εφαρμογές του (GPGPU),σε CUDA 9

2.3 Χρήση της πλατφόρμας CUDA με GPU GeForce 8800 GTX 14

Κεφάλαιο **3 Περιγραφή γλώσσας προγραμματισμού CUDA - Compute Unified Device Architecture…… 20**

3.1 Διαδικασίες βάσεων δεδομένων που χρησιμοποιούν το GPU 19

3.2 Γλώσσα προγραμματισμού CUDA 21

3.3 Επεξήγηση προγραμματιστικών εντολών σε CUDA 23

3.4 Περιγραφή και υλοποίηση συνάρτησης που παρουσιάζει το παραλληλισμό στο GPU σε CUDA 30

Κεφάλαιο **4 Περιγραφή του αλγορίθμου Query 6 (Q6)…….. 35**

4.1 Περιγραφή αλγορίθμου 36

4.2 Ψευδοκώδικας αλγορίθμου 36

4.3 Περιγραφή κώδικα αλγορίθμου Q6 37

4.4 Αποτελέσματα αλγορίθμου Q6 σε C++ και CUDA 41

Κεφάλαιο **5 Συμπεράσματα……..... 48**

5.1 Συμπεράσματα 48

5.2 Μελλοντική εργασία 49

ΒΙΒΛΙΟΓΡΑΦΙΑ..... 50

Παράρτημα Α..... Α-1

Κεφάλαιο 1

Εισαγωγή

1.1 Κίνητρο διπλωματικής εργασίας
2
1.2 Στόχος διπλωματικής εργασίας
3
1.3 Μεθοδολογία που ακολουθήθηκε
3
1.4 Οργάνωση Κειμένου
5

Σκοπός του Κεφαλαίου 1 είναι να παρουσιάσει τους λόγους που οδήγησαν σε αυτή την μελέτη και την διαδικασία που ακολουθήθηκε για την ολοκλήρωση της. Στο πρώτο και δεύτερο Υποκεφάλαιο 1.1, 1.2 παρουσιάζεται το κίνητρο και ο στόχος της διπλωματικής εργασίας. Δηλαδή αναφέρεται το θέμα της εργασίας και γίνεται μια περιγραφή του πακέτου της γλώσσας προγραμματισμού CUDA της NVIDIA. Τονίζεται ο στόχος της διπλωματικής εργασίας και η χρησιμότητα γενικά του παράλληλου υπολογισμού. Στην συνέχεια στο Υποκεφάλαιο 1.3 παρουσιάζονται τα βήματα που ακολουθήθηκαν για την αποπεράτωση της εργασίας. Με λίγα λόγια παρουσιάζεται η μεθοδολογία που ακολουθήθηκε και η λεπτομερής ανάλυση των βημάτων, με περιγραφή όλων όσων έγιναν κατά την διάρκεια ολοκλήρωσης της εργασίας. Τέλος στο Υποκεφάλαιο 1.4 παρουσιάζεται το τι θα ακολουθήσει στην συνέχεια του εγγράφου.

1.1 Κίνητρο διπλωματικής εργασίας

Ο παράλληλος υπολογισμός θεωρείται ότι είναι πολύ σημαντικός και επίσης ότι αποτελεί το μέλλον της επεξεργασίας σε υπολογιστικά συστήματα. Κάποιοι λόγοι που τον κάνουν όχι μόνο να είναι σημαντικός, αλλά και απαραίτητος, είναι το γεγονός ότι σήμερα δημιουργούνται πολύπλοκοι και μεγάλοι υπολογισμοί, που ένας σειριακός υπολογιστής δεν μπορεί να τους διεκπεραιώσει. Επιπρόσθετα, το κόστος των διάφορων παράλληλων συστημάτων έχει πλέον μειωθεί αρκετά, οπόταν η χρήση του παράλληλου υπολογισμού είναι αρκετά συμφέρουσα στις μέρες μας [2].

Παρατηρούμε ότι κάποια παράλληλα μοντέλα χρησιμοποιούν τεχνικές διαμοιρασμού μνήμης (shared-memory techniques) και κάποια άλλα τεχνικές μετάδοσης δεδομένων (message passing). Κάποια εκμεταλλεύονται την τοπικότητα των διχτύων, ενώ κάποια άλλα όχι. Ως αποτέλεσμα της πιο πάνω κατάστασης, κάθε παράλληλος κώδικας που γράφεται, γράφεται σύμφωνα με τον υπολογιστή στον οποίο θα τρέξει. Συνεπώς όταν επιθυμούμε η συγκεκριμένη εφαρμογή να τρέξει σε διαφορετικό παράλληλο υπολογιστή, απαιτείται μια θεμελιώδης αναπροσαρμογή του κώδικα.

Για να κάνουμε την παράλληλη επεξεργασία πιο εύχρηστη, πρέπει να δημιουργηθεί ένα αρκετά απλό μοντέλο παράλληλου υπολογισμού, ώστε να μπορούν να το χρησιμοποιούν εύκολα οι προγραμματιστές και να γράφουν κώδικες. Επίσης πρέπει να είναι τέτοιο, ώστε να μπορούν εύκολα οι αρχιτέκτονες υλικού υπολογιστών (hardware) να σχεδιάζουν αποδοτικό υλικό, που να υποστηρίζει κώδικες γραμμένους σε αυτό το μοντέλο. Τέλος πρέπει να μπορεί να βοηθά στην ανάπτυξη και ανάλυση αλγόριθμων.

Το πακέτο CUDA της NVIDIA είναι ένα υψηλά υποσχόμενο μοντέλο για παράλληλο υπολογισμό.

1.2 Στόχος διπλωματικής εργασίας

Για να εκμεταλλευτούμε τις τεράστιες υπολογιστικές δυνατότητες που προσφέρονται από την ανάπτυξη συστημάτων παράλληλης επεξεργασίας, επινοήθηκαν παράλληλοι αλγόριθμοι που να μπορούν να διαχειρίζονται ένα μεγάλο αριθμό επεξεργαστών και θα δουλεύουν παράλληλα για την γρήγορη επίλυση σύνθετων υπολογιστικών προβλημάτων. Για να το επιτύχουμε αυτό απαιτείται από μέρους των επεξεργαστών αποδοτική συνεργασία μεταξύ τους και προσεκτική διαχείριση των διαθέσιμων πόρων.
Στα πλαίσια αυτής της διπλωματικής εργασίας, μελετήθηκε το πακέτο CUDA που είναι μια γλώσσα προγραμματισμού που προσφέρει παραλληλισμό. Αυτή η γλώσσα διαθέτει συναρτήσεις για την μεταφορά δεδομένων από το CPU στο GPU και αντίθετα και παρατηρείται ότι η χρήση του GPU οδηγεί σε αύξηση της απόδοσης. Ο παραλληλισμός στα Kernel Functions χρησιμοποιεί τα threads. Κάθε ένα από τα threads εκτελεί ένα υπολογισμό παράλληλα με τα υπόλοιπα και έτσι αποφεύγεται η σειριακή εκτέλεση των λειτουργιών. Αυτό βοηθάει επίσης στην απόδοση. Είναι μια γλώσσα που προσφέρει σε υψηλό βαθμό το παραλληλισμό και τα οποία θα αναλυθούν στη συνέχεια.

1.3 Μεθοδολογία που ακολουθήθηκε

Στα πλαίσια ολοκλήρωσης της ερευνητικής μου εργασίας απέκτησα πολλές χρήσιμες εμπειρίες και μου δόθηκε η δυνατότητα να εμβαθύνω σε θέματα παράλληλων αλγόριθμων και συγκεκριμένα στην χρήση της γλώσσας προγραμματισμού CUDA της NVIDIA και να αντιληφθώ την πραγματική αξία της παράλληλης επεξεργασίας. Επίσης εξάσκησα και βελτίωσα τις ικανότητες μου στον προγραμματισμό αλγόριθμων και εξοικειώθηκα με τον τρόπο διεξαγωγής της πειραματικής αξιολόγησης αλγόριθμων.

Για την αποπεράτωση της ερευνητικής μου εργασίας χρειάστηκε να γίνουν 4 βασικά βήματα :

1) Μελέτη επιστημονικών άρθρων που ασχολούνται με GPGPU εφαρμογές και τη γλώσσα προγραμματισμού CUDA της NVIDIA.
2) Υλοποίηση και εκτέλεση αλγόριθμου σε CUDA (TPC-H Benchmarks Query6).
3) Πειραματική αξιολόγηση αυτών.
4) Δημιουργία εγγράφου της ατομικής διπλωματικής μου εργασίας

Στην **πρώτη φάση** μελετήθηκε το πακέτο CUDA που είναι το πιο διαδεδομένο για εφαρμογές GPGPU. Για τη μελέτη αυτή χρησιμοποιήθηκαν διάφορα επιστημονικά έγγραφα τα οποία εντοπίστηκαν μέσω αναζήτησης στο διαδίκτυο.

Στην **δεύτερη φάση** υλοποιήθηκε ο αλγόριθμος αυτός σύμφωνα με το query 6 χρησιμοποιώντας την γλώσσα προγραμματισμού CUDA. Ο αλγόριθμος αυτός υπήρχε υλοποιημένος και σε C++, τον οποίο χρησιμοποίησα για να κάνω μετά την πειραματική αξιολόγηση και σύγκριση των δυο. Ο κώδικας υλοποίησης του αλγόριθμου στην γλώσσα προγραμματισμού CUDA βρίσκετε στο Παράρτημα Α.

Στην **τρίτη φάση** έγινε η πειραματική αξιολόγηση των αλγόριθμων που υλοποιήθηκαν, αρχικά έγιναν μετρήσεις για την απόδοση των αλγορίθμων που υλοποιήθηκαν τόσο σε C++ όσο και σε CUDA για διάφορα μεγέθη του αρχείου (μέγεθος δεδομένων). Στη συνέχεια έγιναν μετρήσεις για την αλλαγή του μεγέθους του block size κατά πόσο επηρεάζει την απόδοση σε CUDA. Για αυτές τις μετρήσεις χρησιμοποιήθηκε ένας ηλεκτρονικός υπολογιστή του Πανεπιστημίου Κύπρου ο οποίος διέθετε κάρτα γραφικών NVIDIA και συγκεκριμένα GeForce 8800 GTS. Για τα πιο πάνω και έγινε μια σύγκριση τόσο των υλοποιήσεων σε CUDA με C++ όσο και μεταξύ τους σε CUDA για διαφορετικές κάρτες γραφικών.

Και τέλος στη **τέταρτη φάση** έγινε η συγγραφή του εγγράφου αυτού για την παρουσίαση των δυνατοτήτων της γλώσσας προγραμματισμού CUDA της NVIDIA και των αποτελεσμάτων που πήρα από κάποιους αλγορίθμους.

1.4 Οργάνωση Κειμένου

Στην συνέχεια ακολουθεί το Κεφάλαιο 2 που γίνεται η παρουσίαση του παραλληλισμού, GPU και GPGPU και παρουσίαση αποτελεσμάτων από άλλες έρευνες που έγιναν. Στο Κεφάλαιο 3 γίνεται παρουσίαση της προγραμματιστικής γλώσσας CUDA και των δυνατοτήτων της. Θα γίνει αναφορά σε διάφορες εντολές που μπορούν να υλοποιήσουν αλγορίθμους που οδηγούν στο παραλληλισμό. Στη συνέχεια αυτού του κεφαλαίου θα γίνει παρουσίαση κάποιας συνάρτησης που εκτελείται στο GPU με παραλληλισμό. Έπειτα στο κεφάλαιο 4 θα γίνει παρουσίαση αυτής της επερώτησης βάσεων δεδομένων (Q6) όπου θα γίνει περιγραφή του κώδικα (Q6) που έχω υλοποιήσει σε CUDA και θα παρουσιαστούν κάποια αποτελέσματα εκτέλεσης της σε C++ και CUDA και στη συνέχεια σύγκριση των αποτελεσμάτων αυτών. Στο τελευταίο κεφάλαιο θα γίνει παρουσίαση των γενικών συμπερασμάτων που έβγαλα από αυτή τη διπλωματική εργασία.

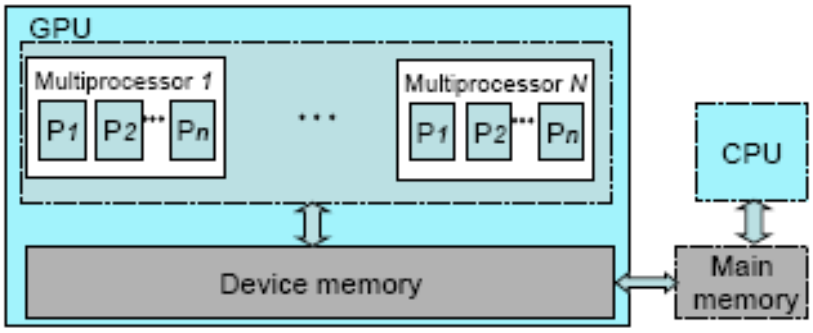
Κεφάλαιο 2

Παραλληλισμός

2.1 GPUs και GPU σε CUDA με TPC-H Queries
6
2.2 Παράλληλος υπολογισμός και εφαρμογές του (GPGPU), σε CUDA
9
2.3 Χρήση της πλατφόρμας CUDA με GPU GeForce 8800 GTX
14

2.1 GPUs και GPU σε CUDA με TPC-H Queries

Οι Μονάδες Γραφικής Επεξεργασίας – Graphics Processing Units (GPUs) – παραδοσιακά χρησιμοποιούνται για γραφικές εφαρμογές κυρίως σε παιχνίδια υπολογιστών. Η αρχιτεκτονική των GPUs διαφέρει σημαντικά από την αντίστοιχη των CPUs και ειδικότερα των νεότερων multi-core CPUs. Κατά κανόνα τα GPUs αποτελούνται από εκατοντάδες επεξεργαστές SIMD (Single Instruction Multiple Data) οι οποίοι προσφέρουν τη δυνατότητα εκτέλεσης παράλληλων λειτουργιών. Αντίθετα ο αριθμός των επεξεργαστών ακόμη και των πιο εξελεγμένων multi-core CPUs είναι δραματικά μικρότερος (πρόσφατα κυκλοφόρησαν ευρέως στην αγορά οι quad-core CPUs ενώ 8 ή 16 επεξεργαστές είναι τυπικό για ένα server). Επιπρόσθετα τα transistor των GPUs αφοσιώνονται στις υπολογιστικές μονάδες αντί στα caches στην περίπτωση των CPUs, με τα caches των GPUs να είναι κατά κανόνα 10 φορές μικρότερα από τα αντίστοιχα του CPU. Στο Σχήμα 2.1 παρουσιάζεται ένα παράδειγμα αρχιτεκτονικού μοντέλου GPU. Παρουσιάζεται πως το GPU με τους πολλαπλούς επεξεργαστές δρα ως συν-επεξεργαστής για το CPU.

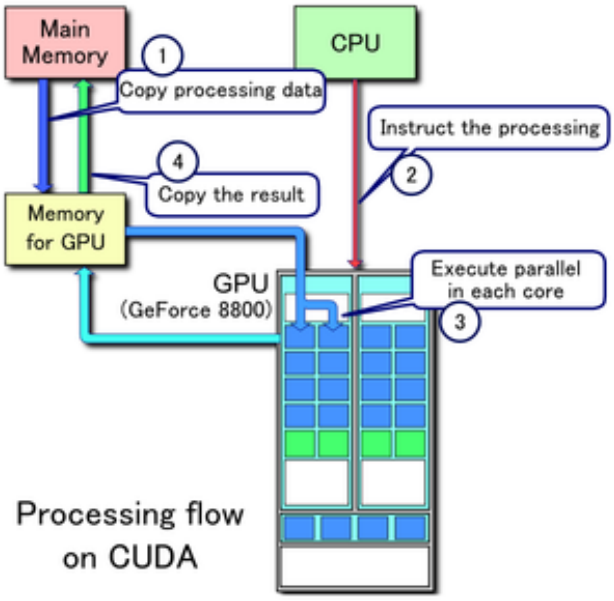


Σχήμα 2.1: GPU Αρχιτεκτονική [4]

Αυτή η ξεχωριστή αρχιτεκτονική των GPUs σύντομα οδήγησε στο να ξεφύγουν από τις αποκλειστικά γραφικές εφαρμογές και την αξιοποίηση τους σε γενικότερης χρήσης εφαρμογές / υπολογισμούς – General Purpose computation on GPU (GPGPU)[4,7]. Το GPGPU αντιπροσωπεύει το γενικής χρήσης υπολογισμό στις Graphics Processing Units, γνωστές ως υπολογισμός GPU. Τα GPUs με τη χρήση επεξεργαστών πολύ-πυρήνων (many-core), είναι ικανά να πετύχουν υψηλή απόδοση στους υπολογισμούς και έξοδο των δεδομένων. Ένα ειδικά σχεδιασμένο για τις γραφιστικές παραστάσεις και δύσκολο να προγραμματιστεί. Σήμερα τα GPUs είναι γενικής χρήσης παράλληλοι επεξεργαστές για την υποστήριξη διεπαφών σε γλώσσες προγραμματισμού όπως η C. Υπεύθυνοι για την ανάπτυξη των εφαρμογών σε GPUs επιτυγχάνουν συχνά speedups εναντίον των βελτιστοποιημένων εφαρμογών CPU. Ο στόχος είναι να καταχωρηθεί η τωρινή και ιστορική χρήση GPUs για το γενικής χρήσης υπολογισμό, και να προωθήσουν ένα κεντρικό πόρο για τους προγραμματιστές λογισμικού GPGPU.

Αυτό με τη σειρά του οδήγησε τους κατασκευαστές GPUs όπως η AMD και η NVIDIA να μετεξελίξου τα νεότερα μοντέλα τους σε ισχυρούς συν-επεξεργαστές (co-processors) για τα CPUs. Συγκεκριμένα προσφέρουν δυνατότητες παράλληλης επεξεργασίας γενικών εφαρμογών συμπεριλαμβανόμενης υποστήριξης για διασκορπισμένες εφαρμογές και επικοινωνία μεταξύ των επεξεργαστών. Ταυτόχρονα δόθηκαν γλώσσες προγραμματισμού για GPGPUs όπως η NVIDIA CUDA [4,6] και η ATI CTM [18]. Αυτές απάλλαξαν τους προγραμματιστές GPGPUs από τη χρήση γραφικών APIs καθώς το τελευταίο είχε το μειονέκτημα της απεικόνισης μη γραφικών υπολογισμών σε γραφικά APIs.

Το πακέτο CUDA από την NVIDIA είναι το πιο διαδεδομένο για εφαρμογές GPGPU. Βασίζεται σε μια επεκταμένη έκδοση της γλώσσας C και υποστηρίζεται σε προϊόντα NVIDIA GeForce, Quadro και Tesla. Στο Σχήμα 2.2 παρουσιάζεται ένα διάγραμμα για τη ροή επεξεργασίας στη CUDA.



Σχήμα 2.2: Διάγραμμα ροής επεξεργασίας σε CUDA [13]

Στο Σχήμα 2.2 βλέπουμε καταρχάς ότι γίνεται η αντιγραφή δεδομένων από την κύρια μνήμη στη μνήμη GPU. Έπειτα το CPU δίνει εντολή για εφαρμογή της διαδικασίας στο GPU. Στη συνέχεια το GPU εκτελεί παράλληλα σε κάθε επεξεργαστή κάποιους υπολογισμούς και τέλος γίνεται η αντιγραφή των αποτελεσμάτων από τη μνήμη του GPU στην κύρια μνήμη.

Μια βάση δεδομένων αποθηκεύει, οργανώνει και εκτελεί κάποιες επερωτήσεις (queries), για κάποια δεδομένα. Η βάση δεδομένων επικοινωνεί με το σύστημα λογισμικού. Οι διάφορες επερωτήσεις (queries) περιλαμβάνουν selections, aggregations και joins operations. Οι υψηλές απαιτήσεις μνήμης και οι υπολογισμοί βάσεων δεδομένων απαιτούν υψηλή υπολογιστική δύναμη λόγο του μεγάλου πλήθους δεδομένων που χρησιμοποιούν. Αυτή την υπολογιστική δύναμη μπορούμε να την αντλήσουμε μέσω των πολλαπλών επεξεργαστών και της ξεχωριστής αρχιτεκτονικής των GPUs, καθιστώντας έτσι τα queries ιδανικές εφαρμογές GPGPU.

Το Transaction Processing Council (TPC) [1], είναι ένας μη κερδοσκοπικός οργανισμός που ιδρύθηκε για να ορίσει σημεία αναφοράς μεταποίησης και της συναλλαγής βάσης δεδομένων. Αυτό έχει ως σκοπό τη διάδοση των δεδομένων επιδόσεων TPC στον κλάδο παραγωγής. Τα σημεία αναφοράς TPC χρησιμοποιούνται ευρέως σήμερα καθώς παρέχει μετρήσεις αντικειμενική απόδοση των συστημάτων πληροφορικής. Ένα από τα σημεία αναφοράς TPC είναι TPC Benchmark H (TPC-H). TPC-H είναι ένα Decision Support System (DSS). Οι επερωτήσεις(queries) που απασχολούνται από αυτό το σημείο αναφοράς είναι σχετικά με τις επιχειρήσεις υποστήριξης της απόφασης βιομηχανίας. Η συγκριτική μέτρηση επιδόσεων (benchmarks) χρησιμοποιείται ενάντια στους μεγάλους όγκους δεδομένων, και αποτελείται από ερωτήματα με υψηλό βαθμό πολυπλοκότητας.

2.2 Παράλληλος υπολογισμός και εφαρμογές του (GPGPU), σε CUDA

Ο παράλληλος υπολογισμός αποτελεί την συνεταιριστική και ταυτόχρονη επεξεργασία δεδομένων από περισσότερους από ένα επεξεργαστές που αποσκοπεί στη γρήγορη επίλυση σύνθετων υπολογιστικών προβλημάτων [2,3].

Επιπρόσθετα η βασική διαφορά μεταξύ του παράλληλου και σειριακού υπολογισμού έγκειται στο γεγονός ότι σε ένα συνηθισμένο ακολουθιακό επεξεργαστή η επεξεργασία μπορεί να γίνεται κάθε φορά σε μια φυσική τοποθεσία. Σε μια παράλληλη όμως μηχανή η επεξεργασία μπορεί να γίνει ταυτόχρονα σε πολλές τοποθεσίες. Πολλές υπολογιστικές λειτουργίες μπορούν να γίνουν ανά δευτερόλεπτο.

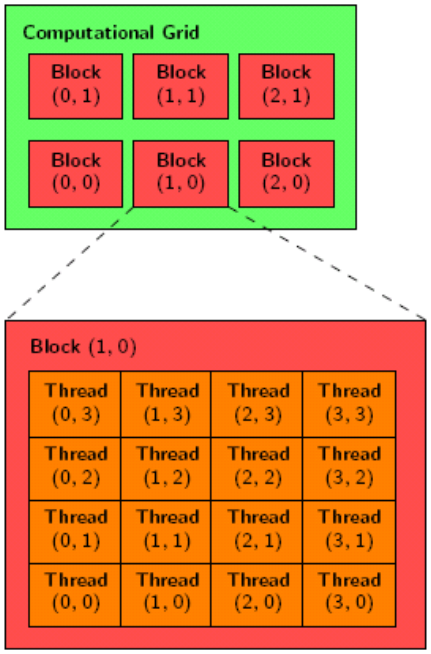
Ο παραλληλισμός μπορεί να υλοποιηθεί στο GPU, το οποίο έχει εξελιχθεί σε έναν παράλληλο, πολύπλοκο, επεξεργαστή πολύ-πυρήνων (manycore) με την τεράστια υπολογιστική δύναμη και το πολύ υψηλό εύρος ζώνης μνήμης (bandwidth).

Σχήμα 2.3: Το GPU αφιερώνει περισσότερα τρανζίστορ για την επεξεργασία δεδομένων [6]

Πιο συγκεκριμένα, όπως φαίνεται και στο Σχήμα 2.3 το GPU είναι ιδιαίτερα καλοσχεδιασμένο για να εξετάσει τα προβλήματα που μπορούν να εκφραστούν όπως στοιχείο-παράλληλοι υπολογισμοί, το ίδιο πρόγραμμα εκτελείται για πολλά δεδομένα στοιχείων παράλληλα με την υψηλή αριθμητική ένταση. Επειδή το ίδιο πρόγραμμα εκτελείται για κάθε ένα δεδομένο στοιχείων, υπάρχει μια χαμηλότερη απαίτηση για τον περίπλοκο έλεγχο ροής και επειδή εκτελείται σε πολλά δεδομένα στοιχείων και έχει την υψηλή αριθμητική ένταση, η λανθάνουσα πρόσβασης μνήμης μπορεί να κρυφτεί (hidden) με τους υπολογισμούς έναντι των μεγάλων data caches.

Παράλληλα δεδομένα στοιχείων μέσω των threads εφαρμόζουν παράλληλη επεξεργασία. Πολλές εφαρμογές που επεξεργάζονται τα μεγάλα σύνολα στοιχείων μπορούν να χρησιμοποιήσουν ένα στοιχείο-παράλληλο πρότυπο προγραμματισμού για να επιταχύνουν τους υπολογισμούς.

Ο παραλληλισμός στη CUDA, μπορεί να υλοποιηθεί με τη χρήση των Kernel Functions στο GPU.



Σχήμα 2.4: Grid of Thread Blocks [17]

Όπως φαίνεται από το Σχήμα 2.4 το computational grid αποτελείται από grid των thread blocks. Κάθε thread εκτελεί τον πυρήνα. Τα grid layouts μπορούν να είναι 1, 2, ή 3 διαστάσεων. Τα μέγιστα μεγέθη καθορίζονται από τη μνήμη GPU και την πολυπλοκότητα πυρήνων κάθε μπλοκ έχει μια μοναδική ταυτότητα μπλοκ, block ID, που κάθε thread έχει μια μοναδική ταυτότητα threads, thread ID, (μέσα στο μπλοκ).

Όλα τα threads εκτελούν τον ίδιο κώδικα, παράλληλα. Κάθε thread έχει τη δική του θέση μνήμης (threadId) όπου αποθηκεύει τα αποτελέσματα κάθε υπολογισμού.

Σε κάποια προηγούμενη μελέτη που έγινε [4] μελέτησαν την απόδοση του GPU και του CPU σε CUDA μέσω κάποιων αλγορίθμων που χρησιμοποιούν τον υψηλό παραλληλισμό καθώς και το υψηλό εύρος ζώνης μνήμης του GPU, και χρησιμοποιούν τις παράλληλες βελτιστοποιήσεις υπολογισμού και μνήμης για να μειώσουν αποτελεσματικά τις κυψελίδες μνήμης.

Κάποια αποτελέσματα από τη μελέτη αυτή [4] για τους τέσσερις join αλγορίθμους, σε CUDA, φαίνονται στον πιο κάτω πίνακα.

Πίνακας 2.1 Χρόνος που χρειάζεται κάθε ένα από τα τέσσερα relational joins σχετικά με την GPU και της CPU [4]

Joins	CPU (sec)	GPU (sec)	Speedup
NINLJ	528.0	75.0	7.0
INLJ	4.2	0.7	6.1
SMJ	5.0	2.0	2.4
HJ	2.5	1.3	1.9

Ο πίνακας παρουσιάζει το χρόνο που χρειάζεται κάθε ένα από τα τέσσερα Relational Joins στο GPU και στο CPU. Ο χρόνος για το GPU περιλαμβάνει το χρόνο μεταφοράς δεδομένων μεταξύ της μνήμης συσκευών (device) και της κύριας μνήμης. Συνολικά, το GPU είναι 2-7X πιο αποδοτικός από το CPU.

Στον επόμενο πίνακα παρουσιάζονται κάποια αποτελέσματα σύγκρισης του χρόνου που χρειάζονται οι τέσσερις Relational Joins αλγόριθμοι εφαρμοσμένα σε CUDA και DX (DirectX)

Πίνακας 2.2: αποτελέσματα σύγκρισης του χρόνου που χρειάζονται οι τέσσερις Relational Joins αλγόριθμοι εφαρμοσμένα σε CUDA και DX (DirectX) [4].

	DX (join)	DX (total)	CUDA (join)	CUDA (total)
NINLJ	72.3	74.1	75.0	75.0
INLJ	0.7	0.9	0.6	0.7
SMJ	3.8	4.7	1.9	2.0
HJ	2.3	2.7	1.2	1.3

- Το directX για NINLJ και INLJ επιτυγχάνει μια παρόμοια απόδοση με το CUDA.
- Αντίθετα, το directX για SMJ και HJ είναι περίπου δύο φορές πιο αργό από το CUDA. Αυτές οι εφαρμογές DirectX χρειάζονται περισσότερο χρόνο για GPU όπως η κωδικοποίηση/η αποκωδικοποίηση σύστασης.

Οι μονάδες επεξεργασίας γραφικών (GPUs) έχουν πρόσφατα προσέλκυσε πολύ ενδιαφέρον για διάφορους τομείς που αγωνίζονται με τους μαζικά μεγάλους στόχους υπολογισμού.

Σε μια έρευνα που έγινε [19] αναφέρουν ότι η GPU μπορεί να χρησιμοποιηθεί για την αναπαραγωγή τρισδιάστατων οπτικών εικόνων. Μέσα από την έρευνα τους παρουσιάζουν ότι η χρήση GPU είναι η πλέον κατάλληλη για αναπαραγωγή τέτοιων εικονικών αναπαραστάσεων και ολογραμμάτων λόγω της δομής τους που τους επιτρέπει υψηλές επιδόσεις στους υπολογισμούς, ακόμα και σε περιπτώσεις ολογραμμάτων ανάλυσης υψηλής ευκρίνειας (1400X1050 pixels).

Σε μια άλλη έρευνα που έγινε [20] χρησιμοποιούν GPU και CPU για να τρέξουν ένα πρόγραμμα τρισδιάστατων γραφικών «3D Richardson-Lucy» και κάνουν σύγκριση των αποτελεσμάτων απόδοσης. Τα αποτελέσματα έδειξαν ότι η χρήση GPU υπερτερούσε σημαντικά της CPU σε απόδοση.

Στην έρευνα του ο M. Andrecut [21] χρησιμοποιεί GPU και CPU για να τρέξει αλγόριθμους ανάλυσης δεδομένων πολλαπλών μεταβλητών (NIPALS-PCA και GS-PCA). Μέσα από την εργασία αυτή παρουσιάζεται ότι η χρήση GPU για την εφαρμογή τέτοιων αλγορίθμων είναι πολύ πιο αποδοτική (έως και 12 φορές γρηγορότερη) από μια CPU.

Μελέτες έδειξαν ότι η CUDA και γενικά η χρήση του GPU βοήθησε και στη ιατρική (Ακτινοθεραπεία) [22]. Δημιουργήθηκαν αλγόριθμοι για αυτό και απέδειξαν από κάποιες έρευνες [22], ότι η ανίχνευση ακτινών μπορεί να εκτελεστεί ακριβώς στο GPU. Πέτυχε επιτάχυνση περίπου 2.1 - 10.1sec (6 κατά μέσον όρο).

Από μια άλλη μελέτη που έγινε [18] αναπτύχθηκαν αρκετοί αλγόριθμοι (πχ MERL's state-of-the art Bayesian background generation) για πολύ γρήγορη επεξεργασία εικόνας και βίντεο εκμεταλλευμένοι το GPU (Graphics Processing Units). Τελικά αποδείχθηκε ότι σε σύγκριση με την έκδοση CPU του ίδιου αλγορίθμου, η εφαρμογή GPU είναι περισσότερες από 20 φορές γρηγορότερη.

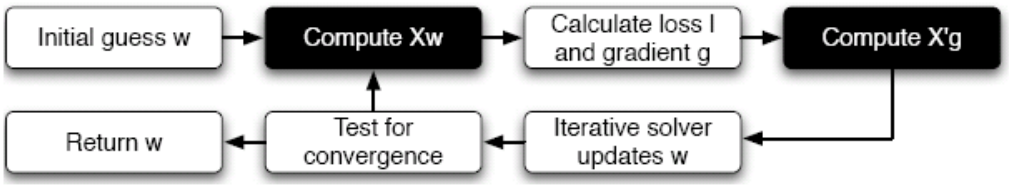
Τα GPUs είδη χρησιμοποιούνται εδώ και καιρό, στους προσωπικούς υπολογιστές μέσω των τηλεοπτικών παιχνιδιών και των πολυμέσων, στους υπολογισμούς και πλέον και στη ιατρική. Με τις όλο και αυξανόμενες ικανότητες προγραμματισμού, τα GPUs είναι σε θέση να εκτελούν περισσότερο από τους συγκεκριμένους υπολογισμούς γραφικών για τους οποίους και σχεδιάστηκαν. Ο υπολογισμός με το GPU με την CUDA είναι μια νέα προσέγγιση στον υπολογισμό όπου εκατοντάδες on-chip επεξεργαστών πυρήνα, επικοινωνούν ταυτόχρονα και συνεργάζονται για να λύσουν σύνθετα προβλήματα υπολογισμού, ειδικά για τις χρονοβόρες διεργασίες ανάλυσης βίντεο και 3D δεδομένων.

Μια από τις σημαντικότερες καινοτομίες που προσφέρονται από την CUDA είναι η δυνατότητα για τα threads σε GPUs. Επιτρέποντας σε threads να επικοινωνούν, η τεχνολογία της CUDA επιτρέπει στις εφαρμογές να λειτουργούν αποτελεσματικότερα. Το GPUs που χαρακτηρίζει την τεχνολογία CUDA έχει μια on-chip Parallel Data Cache που οι υπεύθυνοι για την ανάπτυξη μπορούν να χρησιμοποιήσουν για να αποθηκεύσουν

τις συχνά χρησιμοποιημένες πληροφορίες άμεσα για το GPU. Η αποθήκευση των πληροφοριών για το GPU επιτρέπει στα υπολογιστηκά threads να μοιράζονται αμέσως τις πληροφορίες παρά να περιμένει τα στοιχεία από τα πολύ πιό αργά off-chip DRAM. Αυτή η πρόοδος στην τεχνολογία επιτρέπει στους χρήστες να βρουν τις απαντήσεις στα σύνθετα υπολογιστικά προβλήματα γρηγορότερα από ότι χρησιμοποιώντας τις παραδοσιακές αρχιτεκτονικές.

2.3 Χρήση της πλατφόρμας CUDA για εφαρμογές στατιστικής μηχανικής με GPU GeForce 8800 GTX

Σε μια άλλη μελέτη [12] οι συγγραφείς αποτιμούν τη χρήση της πλατφόρμας CUDA για εφαρμογές στατιστικής μηχανικής μάθησης (machine learning - ML). Καταμέτρησαν τον ρυθμό μεταφοράς από και προς την GPU καθώς και την απόδοση διανυσματικού γινομένου πινάκων στην GPU. Η GPU που χρησιμοποιήθηκε είναι η GeForce 8800 GTX. Η εφαρμογή στατιστικής μηχανικής μάθησης που χρησιμοποίησαν αποτελείται από ένα επαναληπτικό αλγόριθμο επίλυσης (iterative solver) που εκτελεί επαναλαμβανόμενους πολλαπλασιασμούς πινάκων. Το πιο κάτω Σχήμα 2.6 αναπαριστά γραφικά τον επαναληπτικό αλγόριθμο επίλυσης.



Σχήμα 2.6: επαναληπτικό αλγόριθμο επίλυσης (iterative solver) [12]

Τα μαύρα κουτιά στο Σχήμα 2.6 αντιστοιχούν σε διανυσματικές πράξεις πινάκων οι οποίες μπορούν να επιταχυνθούν κάνοντας χρήση GPU.

Το host του GPU που χρησιμοποιήθηκε είναι ένας 2GHz dual core AMD Athlon64 3800+ με 2GB PC3200 μνήμη DDR. Ο επεξεργαστής έχει 128KB L1 cache και 1 MB L2 cache. Για όλα τα benchmarks και όλα τα πειράματα, ο κώδικας έτρεξε για 100 επαναλήψεις και ο μέσος χρόνος χρησιμοποιήθηκε για τον υπολογισμό του εύρους ζώνης

και των FLOPS. Οι σποραδικοί πίνακες χρησιμοποιήθηκαν για την αποτίμηση του ρυθμού μεταφοράς δεδομένων από την κύρια μνήμη στη μνήμη του GPU και αντίστροφα. Τα αποτελέσματα φαίνονται στον πιο κάτω Πίνακας 2.3 και η μονάδα μέτρησης είναι GB ανά δευτερόλεπτο. Η πρώτη στήλη "latency" αφορά το χρόνο αντίδρασης και μετριέται σε μικρό-δευτερόλεπτα.

Πίνακας 2.3: Host initiated memory transfer rates (GB/s) [12]

	Latency μs	1KB	1MB	100MB
Main Memory to GPU	22	0.03	0.80	1.10
Main Memory (pinned) to GPU	18	0.04	2.70	3.10
GPU to Main Memory	18	0.04	0.40	0.50
GPU to Main Memory (pinned)	15	0.05	2.80	3.00
GPU Memory to GPU Memory	12	0.14	50.59	71.17

Οι πυκνοί πίνακες (ελάχιστα έως καθόλου μηδενικά) χρησιμοποιήθηκαν για τη σύγκριση της απόδοσης μεταξύ του host και του GPU. Το πακέτο (library) BLAS του CUDA χρησιμοποιήθηκε στο GPU ενώ το λογισμικό ATLAS στο host. Χρησιμοποιήθηκαν δύο ρουτίνες για πράξεις πινάκων, SGEMV και SGEMM. Η ρουτίνα SGEMV στο ATLAS είχε πάντοτε καλύτερη απόδοση από τη SGEMM (συνεπώς εξαιρείται από τα αποτελέσματα). Οι πίνακες που λήφθηκαν υπόψη έχουν διαστάσεις πολλαπλάσια του 16 καθώς διαπιστώθηκε ότι σε διαφορετική περίπτωση η απόδοση του CUBLAS πέφτει δραματικά. Οι πίνακες εισήχθηκαν τόσο σε κανονική (N) όσο και σε ανεστραμμένη (transposed (T)) διάταξη.

Για τετραγωνικούς πίνακες η απόδοση του host είναι περίπου σταθερή για όλες τις διαστάσεις πινάκων με καλύτερη απόδοση για κανονική διάταξη πινάκων παρά ανεστραμμένη. Αντίθετα στη GPU η απόδοση μεταβάλλεται καθώς οι διαστάσεις μεταβάλλονται. Η απόδοση της ρουτίνας SGEMV για κανονική διάταξη πινάκων αυξάνεται καθώς οι διαστάσεις αυξάνονται ενώ για ανεστραμμένη διάταξη η απόδοση παραμένει περίπου σταθερή. Σε όλες σχεδόν τις περιπτώσεις η ρουτίνα SGEMV αποδίδει καλύτερα από την SGEMM. Στην καλύτερη περίπτωση η απόδοση του GPU είναι 4.5 φορές γρηγορότερη από την απόδοση του host.

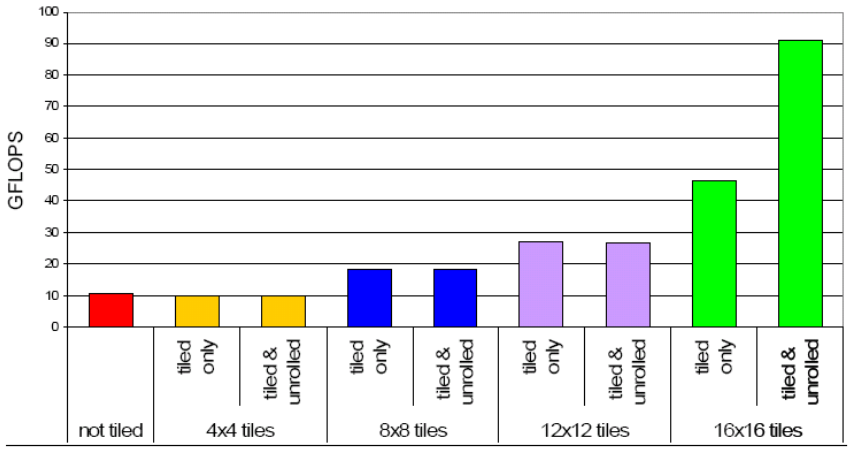
Σε γενικότερους πίνακες παρουσιάζεται μια πτώση στην απόδοση τόσο στο host όσο και στον GPU όταν ο αριθμός των στηλών του πίνακα είναι μεγαλύτερος από τον αριθμό των γραμμών, ιδιαίτερα όταν χρησιμοποιείται ανεστραμμένη διάταξη πινάκων. Αυτό συμβαίνει σε μεγαλύτερο βαθμό στη ρουτίνα SGEMV αλλά εξακολουθεί να αποδίδει

καλύτερα από τη SGEMM. Οι συγγραφείς προτείνουν να γίνεται χρήση της ρουτίνας SGEMV στο CUBLAS.

Από τη μελέτη κάποιου άλλου άρθρου [11], εξετάζεται ο GeForce 8800 GTX GPU, η οργάνωση, τα χαρακτηριστικά καθώς και μέθοδοι βελτιστοποίησης της απόδοσης του. Χρησιμοποιείται η γλώσσα προγραμματισμού NVIDIA CUDA. Το ενδιαφέρον των συγγραφέων επικεντρώνεται σε χρήση πολύ- multithreaded διαδικασιών αξιοποιώντας κατ' αυτό τον τρόπο τον μεγάλο αριθμό επεξεργαστών του GPU.
Κατά τη διαδικασία βελτιστοποίηση της απόδοσης του GPU στηρίζονται στα τρία ακόλουθα σημαντικά σημεία:

1. Το κινητό σημείο (floating point) εξόδου μιας εφαρμογής εξαρτάται από το ποσοστό των εντολών της εφαρμογής οι οποίες είναι πράξεις κινητού σημείου.
2. Όταν επιχειρείται να επιτευχθεί η μέγιστη απόδοση μιας εφαρμογής, το ενδιαφέρον στρέφεται κυρίως στη διαχείριση του διαστήματος αντίδρασης (latency) της κύριας μνήμης (global memory).
3. Το εύρος ζώνης (bandwidth) της global memory μπορεί να περιορίσει την παραγωγή του συστήματος.

Αρχικά διερευνούν την απόδοση σε μια πειραματική εφαρμογή πολλαπλασιασμού πινάκων. Τα αποτελέσματα παρουσιάζονται στο πιο κάτω Σχήμα 2.7.



Σχήμα 2.7: Απόδοση των πινάκων πολλαπλασιασμού πυρήνων [11]

Όπως φαίνεται στο Σχήμα 2.7, οι πίνακες αποτελούντο από blocks υπό-πινάκων (tiles). Κατέληξαν ότι η χρήση πινάκων με 16x16 tiles ελαττώνει τα αρχικά ολικά φορτία κατά

ένα παράγοντα της τάξης του 16. Όταν ο πίνακας αποτελείται από 16x16 tiles είναι η μοναδική διαμόρφωση στο πείραμα η οποία αξιοποιεί όλα τα warps του GPU. Η απόδοση της διαμόρφωσης 16x16 tiles είναι 4.5 φορές μεγαλύτερη όταν δεν χωρίσουμε τον πίνακα σε tiles.

Ακολούθως βάσει των συμπερασμάτων που εξήγαγαν από την πειραματική εφαρμογή προχώρησαν σε πραγματικές εφαρμογές. Αυτές έδειξαν ότι η χρήση του GPU οδηγεί σε αύξηση της απόδοσης από 10.5x – 457x.

Κεφάλαιο 3

Περιγραφή γλώσσας προγραμματισμού CUDA - Compute Unified Device Architecture

---

3.1 Διαδικασίες βάσεων δεδομένων που χρησιμοποιούν το GPU

19

3.2 Γλώσσα προγραμματισμού CUDA

21

3.3 Επεξήγηση προγραμματιστικών εντολών σε CUDA

23

3.4 Περιγραφή και υλοποίηση συνάρτησης που παρουσιάζει το παραλληλισμό στο GPU σε CUDA

30

Στόχος του Κεφαλαίου 3 στο Υποκεφάλαιο 3.1 παρουσιάζεται το πώς μπορεί να χρησιμοποιηθεί το GPU στις διαδικασίες βάσεων δεδομένων. Στο Υποκεφάλαιο 3.2 είναι η παρουσίαση της προγραμματιστικής γλώσσας CUDA και στη συνέχεια στο Υποκεφάλαιο 3.3 γίνεται επεξήγηση διάφορων προγραμματιστικών εντολών που υπάρχουν σε αυτή. Ακολούθως στο Υποκεφάλαιο 3.4 παρουσιάζεται μιας συνάρτηση του reduction αλγορίθμου που θα παρουσιάζει το παραλληλισμό στη CUDA και εκτελείτε στο GPU.

3.1 Διαδικασίες βάσεων δεδομένων που χρησιμοποιούν το GPU

Οι διαδικασίες βάσεων δεδομένων αν και μη σύνθετες, εξαιτίας του γεγονότος ότι το μεγαλύτερο μέρος τους εκτελείται ενάντια στους μεγάλους ή ακόμα και τεράστιους όγκους των στοιχείων, μπορούν να χαρακτηριστούν ως χρονοβόρες διαδικασίες. Σε τέτοιες περιπτώσεις, μια απλή λειτουργία βάσεων δεδομένων μπορεί να πάρει αρκετό χρόνου για να εκτελεστεί. Σχετική βιβλιογραφία παρουσιάζει τις σημαντικές προσπάθειες της χρησιμοποίησης του GPU για τη βελτιστοποίηση των παραδοσιακών αλγορίθμων βάσεων δεδομένων.

Οι χωρικές διαδικασίες βάσεων δεδομένων χρησιμοποιούνται στις εφαρμογές όπως τα γεωγραφικά συστήματα πληροφοριών (geographical information systems - GIS) και με τη βοήθεια υπολογιστή (computer-aided - CAD). Αυτές οι διαδικασίες, λόγω η φύση τους, περιλαμβάνει τη χρήση των υπολογιστικών αλγορίθμων γεωμετρίας. Αυτό το είδος αλγορίθμων είναι σημαντικής πολυπλοκότητας, δεδομένου ότι η CPU είναι αρμόδια για την εκτέλεση των δύσκολων υπολογισμών που περιλαμβάνουν την επεξεργασία της σύνθετης γεωμετρίας όπως τα πολύγωνα. Στις περισσότερες περιπτώσεις δεν είναι μόνο

η πολυπλοκότητα τέτοιων μορφών, αλλά και το πλήθος των στοιχείων που απαιτείται να υποβληθούν σε επεξεργασία. Σε έρευνες που έγιναν [5] προσπάθησαν να εξετάσουν αυτό το πρόβλημα, με τη χρησιμοποίηση της αρχιτεκτονικής υλικού GPU για την επιτάχυνση των χωρικών (spatial) selections και joins. Τα πειραματικά αποτελέσματά τους για ένα 8x8 ανάλυση παραθύρων (window resolution) παρείχαν ενδιαφέροντα speedups. Για τα σύνθετα στοιχεία πολυγώνων οι τεχνικές τους παρείχαν ότι 4.8x speedup για intersection joins, και μέχρι 5.9x speedup για το within distance join operations. Αυτά τα αποτελέσματα ενθαρρύνουν αρκετά, διαδικασίες βάσεων δεδομένων μεγάλης πολυπλοκότητας.

Η ταξινόμηση είναι ένα αναπόσπαστο τμήμα των περισσότερων DBMSs. Σε πολλές περιπτώσεις, ο χρόνος εκτέλεσης των διαδικασιών βάσεων δεδομένων εξαρτάται πλήρως από το χρονικό διάστημα που απαιτείται για την εκτέλεση των αλγορίθμων ταξινόμησης. Οι μελέτες έδειξαν ότι η ταξινόμηση μπορεί να είναι και υπολογισμός καθώς επίσης και memory-intensive. Για αυτόν τον λόγο, διάφοροι αλγόριθμοι προτάθηκαν. Αυτές οι τεχνικές βασίστηκαν στην εκτέλεση CPU και έτσι, αντιμετώπισε τα γενικά έξοδα της διαδοχικής εκτέλεσης σε CPU. Με την εμφάνιση του προγραμματισμού GPU, ερευνητές βάσεων δεδομένων εστίασαν στην εκμετάλλευση των ισχυρών υπολογιστικών χαρακτηριστικών γνωρισμάτων τους, για την επιτάχυνση των ερωτήσεων βάσεων δεδομένων [5]. Από έρευνες που έγιναν, παρουσίασαν έναν αλγόριθμο για τους υπολογισμούς βάσεων δεδομένων και εξόρυξη δεδομένων χρησιμοποιώντας το GPU. Αυτός ο αλγόριθμος χρησιμοποιήθηκε για να επιταχύνει τον υπολογισμό equi-join και nonequi-join επερωτήσεις στις βάσεις δεδομένων και τις αριθμητικές επερωτήσεις στατιστικής στα data streams. Η εφαρμογή CPU εκτελέσθηκε σε έναν ηλεκτρονικό υπολογιστή με το Intel Pentium 4 3.4GHz και Εφαρμογή GPU στον ίδιον ηλεκτρονικό υπολογιστή με ένα NVIDIA GeForce 6800 υπερβολικό GPU. Τα πειραματικά αποτελέσματά τους έδειξαν μια βελτίωση 30% στη γενική υπολογιστική απόδοση.

Στον ίδιο τομέα της έρευνας βάσεων δεδομένων, ερεύνησαν την παράλληλη ταξινόμηση στις stream processing αρχιτεκτονικές. Η ιδέα της εργασίας τους βασίστηκε στην προσαρμοστική bitonic ταξινόμηση. Η εφαρμογή τους που παρουσίασε speedups γύρω από 2.5x που αποδεικνύει ότι το GPU μπορεί να χρησιμοποιηθεί αποτελεσματικά ως συνεπεξεργαστής για τέτοιες διαδικασίες.

Επίσης από άλλες έρευνες, παρουσίασαν το GPUTeraSort [14], έναν αλγόριθμο που χρησιμοποιείται για την ταξινόμηση δισεκατομμυρίων αρχείων που χρησιμοποιούν το GPU. Ο αλγόριθμός τους εκτελέσθηκε ενάντια στο nsort, ένας εμπορικός αλγόριθμος ταξινόμησης, βασισμένος στη CPU με την υψηλή I/O απόδοση. Η πειραματική τους οργάνωση σχετικά με την εκτέλεση GPU αποτελέστηκε από τρία διαφορετικά GPUs (NVIDIA GeForce 6800, NVIDIA GeForce 6800 εξαιρετικά και NVIDIA GeForce 7800GT). Οι αλγόριθμοι που ήταν βασισμένοι στη CPU έτρεχαν σε έναν high-end Dual Xeon server. Τα πειραματικά αποτελέσματά τους έδειξαν ότι η γενική απόδοση GPUTeraSort με ένα μεσαίο GPU είναι συγκρίσιμη με αυτήν ενός high-end Dual Xeon server. Εάν λαμβάνουμε υπόψη την τεράστια διαφορά σχετικά με το κόστος μεταξύ των δύο οργανώσεων, μπορούμε να δούμε ότι το GPU περιλαμβάνει μια φτηνή και καλή επιλογή για τέτοιες διαδικασίες.

Στην εργασία που προαναφέρθηκε, έχω εκθέσει την έρευνα που πραγματοποιείται για τις χωρικές διαδικασίες και διαδικασίες ταξινόμησης για μεγάλες βάσεις δεδομένων. Αν και, δεν ανέφερα την περίπτωση των διαδικασιών βάσεων δεδομένων όπως: conjunctive selections, aggregations και semi-linear επερωτήσεις. Ήδη έχουν πραγματοποιήσει την έρευνα σε αυτήν την περιοχή των βάσεων δεδομένων και έχουν πειραματιστεί με τους επεξεργαστές γραφικής παράστασης (GPU), για την επιτάχυνση τέτοιων διαδικασιών. Έχουν χρησιμοποιήσει το GPU και παρουσίασαν τους νέους αλγορίθμους που μπορούν να χρησιμοποιηθούν αντί των αντίστοιχων παραδοσιακών αλγορίθμων βάσεων δεδομένων, οι οποίοι παρέχουν πιο σύντομους χρόνους εκτέλεσης για τις επερωτήσεις στόχου (targeted query). Αυτές οι διαδικασίες αποτελούντα από: predicate evaluation (σύγκριση μεταξύ ενός χαρακτηριστικού και μιας σταθεράς, σύγκριση μεταξύ δύο χαρακτηριστικών), boolean combination και aggregations (συναθροίσεις) (COUNT, MIN-MAX, SUMAVG)

Αν και υπάρχει ένα ουσιαστικό ποσό σχετικής ερευνητικής εργασίας όπως παρουσιάζεται σε αυτήν την υποενότητα, υπάρχουν ακόμα πολλά ζητήματα που αντιμετωπίζουν στην εκτέλεση των διαδικασιών βάσεων δεδομένων σε ένα GPU. Ένα τέτοιο ζήτημα είναι αλγόριθμοι βάσεων δεδομένων γνωστοί ως database benchmark’s TCP-H στο GPU. Για αυτόν τον λόγο, έχω εκτελέσει κάποιο αλγόριθμο για την benchmark’s TCP-H τον Query6. Έχει χρησιμοποιηθεί CUDA, ένα σχετικά νέο, ισχυρό

υψηλού επιπέδου περιβάλλον προγραμματισμού GPU από όπου και πήρα τις σχετικές μετρήσεις και αποτελέσματα.

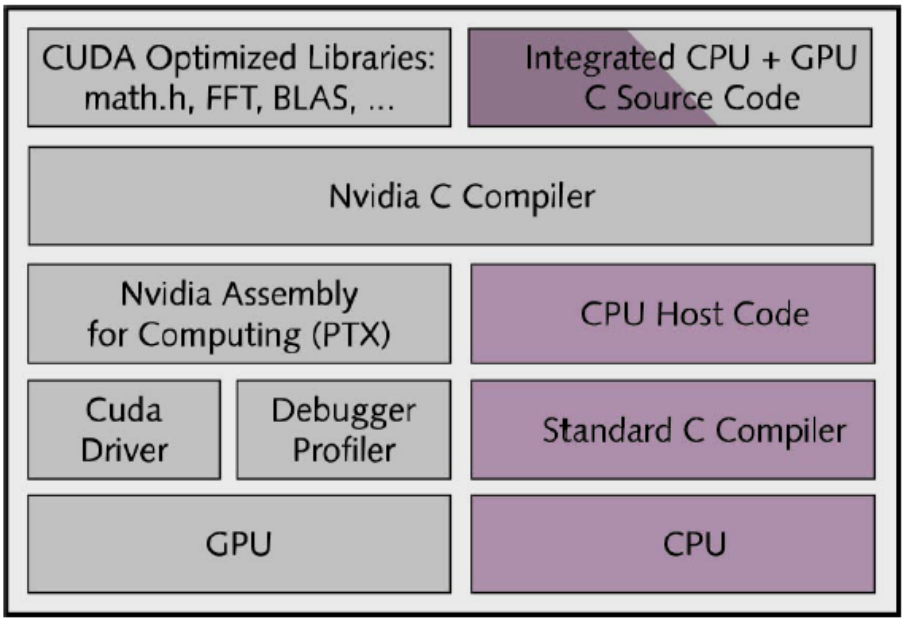
Η σχετική εργασία που παρουσιάζεται πιο πάνω, δικαιολογεί τις σύγχρονες ερευνητικές τάσεις όπου το GPU μπορεί να χρησιμοποιηθεί για την επιτάχυνση των διαδικασιών βάσεων δεδομένων. Έτσι προκύπτει η ανάγκη για τους αποδοτικούς αλγορίθμους βάσεων δεδομένων GPU που μπορούν στο μέλλον να γίνουν απόγονοι των τρεχόντων παραδοσιακών αλγορίθμων βάσεων δεδομένων CPU.

3.2 Γλώσσα προγραμματισμού CUDA και επεξήγηση προγραμματιστικών εντολών

Στόχος στο σημείο αυτό είναι να γίνει κάποια παρουσίαση της γλώσσας προγραμματισμού που χρησιμοποιείται. Συγκεκριμένα χρησιμοποιείται η γλώσσα προγραμματισμού CUDA. Προσφέρει υψηλό παραλληλισμό με τη χρήση των threads και έχει την δυνατότητα να εκτελεί κάποιες εργασίες στο GPU. Είναι επέκταση της γλώσσας προγραμματισμού C και υποστηρίζεται από προϊόντα της NVIDIA. Στην εργασία αυτή χρησιμοποιήθηκε κάρτα γραφικών GeForce 8800 GTS.

Με τον όρο CUDA, η NVIDIA αναφέρεται σε μία αρχιτεκτονική παράλληλου υπολογισμού computing (Compute Unified Device Architecture) η οποία έχει ως στόχο να εκμεταλλευθεί την εξαιρετική δύναμη των επεξεργαστών γραφικών της εταιρείας, για εργασίες που δεν σχετίζονται με την απεικόνιση γραφικών.

Σε πιο τεχνικό επίπεδο, θα λέγαμε ότι η αρχιτεκτονική CUDA αποτελεί μία μικρή επέκταση της γλώσσας C, βάσει της οποίας οι προγραμματιστές αποκτούν πρόσβαση στις δυνατότητες της κάρτας γραφικών και έτσι μπορούν να εκμεταλλευθούν την ισχύ της για την επίλυση διαφόρων προβλημάτων, συνήθως επιστημονικής φύσης. Αξίζει να σημειωθεί ότι η NVIDIA δίνει τη δυνατότητα στους προγραμματιστές να εκμεταλλευθούν παράλληλα και τον κεντρικό επεξεργαστή του συστήματος (όπως γίνεται άλλωστε με κάθε γλώσσα προγραμματισμού). Αξίζει να σημειωθεί ότι η NVIDIA δίνει τη δυνατότητα στους προγραμματιστές να εκμεταλλευθούν παράλληλα και τον κεντρικό επεξεργαστή του συστήματος και έτσι να μπορούν να μοιράσουν τον φόρτο εργασίας του προγράμματός τους, τόσο στη CPU (Κεντρική Μονάδα Επεξεργασίας) , όσο και στη GPU (Graphics Processing Unit).



Σχήμα 3.1: CUDA Software Development Kit [17]

Όπως βλέπουμε από το Σχήμα 3.1 τα εργαλεία ανάπτυξης CUDA λειτουργούν παράλληλα με έναν συμβατικό μεταγλωττιστή C/C++, έτσι οι προγραμματιστές μπορούν να αναμίξουν τον κώδικα GPU με το γενικής χρήσης κώδικα για το host CPU.

Η αρχιτεκτονική CUDA και το σχετικό λογισμικό της αναπτύχθηκαν με διάφορους στόχους:

- Παρέχετε ένα μικρό σύνολο επεκτάσεων στις τυποποιημένες γλώσσες προγραμματισμού, όπως η C, που επιτρέπει μια απλή εφαρμογή των παράλληλων αλγορίθμων. Με τη CUDA και τη C για CUDA, οι προγραμματιστές μπορούν να εστιάσουν σαν στόχο το παραλληλισμό αλγορίθμων παρά το χρόνο που θα χρειαστούν για την εφαρμογή τους.
- Η CUDA προσφέρει το συνδυασμό του CPU και του GPU. Η CPU και το GPU αντιμετωπίζονται ως χωριστές συσκευές με διαφορετική μνήμη. Αυτή η διαμόρφωση επιτρέπει επίσης ταυτόχρονο υπολογισμός και στην CPU και σε GPU.

Η CUDA GPUs έχει εκατοντάδες πυρήνες που μπορούν συλλογικά να τρέξουν χιλιάδες threads.

Η γλώσσα αυτή, CUDA, διαθέτει πολλές συναρτήσεις που βοηθούν στο παραλληλισμό. Το πρότυπο για τον υπολογισμό GPU που διαθέτει η CUDA είναι να χρησιμοποιηθεί μια

CPU και ένα GPU μαζί σε ένα ετερογενές πρότυπο υπολογισμού. Το διαδοχικό μέρος των εκτελέσεων εφαρμογής εκτελούνται στην CPU και το υπολογιστικό μέρος τρέχουν στο GPU. Από την προοπτική του χρήστη, η εφαρμογή τρέχει ακριβώς γρηγορότερα επειδή χρησιμοποιεί τον υψηλής απόδοσης του GPU για τους υπολογισμούς.
Για να τρέξουν τα προγράμματα σε CUDA χρησιμοποιήθηκαν οι πιο κάτω εντολές για την μεταγλώττιση και εκτέλεση των προγραμμάτων [6]:

Μεταγλώττιση του προγράμματος:

- nvcc file.cu

Το μπροστινό μέρος του μεταγλωττιστή επεξεργάζεται τα αρχεία πηγής CUDA σύμφωνα με τους C++ κανόνες σύνταξης. Εντούτοις, μόνο το υποσύνολο C/C++ υποστηρίζεται. Αυτό σημαίνει ότι C++ τα συγκεκριμένα χαρακτηριστικά γνωρίσματα όπως οι κατηγορίες (classes), η κληρονομιά (inheritance), ή η δήλωση των μεταβλητών μέσα στους βασικά μπλοκ δεν υποστηρίζονται. Συνεπεία της χρήσης C++ των κανόνων σύνταξης, void δείκτες (π.χ. επιστρεφόμενος από το malloc ()) δεν μπορεί να οριστείτε στους non-void δείκτες χωρίς ένα typecast.

Εκτέλεση του προγράμματος:

> ./a.out

3.3 Επεξήγηση προγραμματιστικών εντολών σε CUDA

Πιο κάτω παρουσιάζονται κάποιες από τις βασικές εντολές που χρησιμοποίησα σε αυτή τη ατομική διπλωματική εργασία:

- Δήλωση μεταβλητών και δέσμευση μνήμης στο Device (GPU). Η συνάρτηση CUDAMalloc δεσμεύει χώρο, σε bytes, ικανό για την αποθήκευση όλων των δεδομένων. Το δεδομένο data_d είναι ένας πίνακας που θα χρησιμοποιηθεί σαν δεδομένο για τους υπολογισμούς στο GPU.

```
float *data_d;
CUDAMalloc((void **) &data_d, size);
```

- Δήλωση μεταβλητών και δέσμευση μνήμης στο Host (CPU). Με τη malloc δεσμεύεται χώρος, σε bytes, ικανό για την αποθήκευση όλων των δεδομένων. Το δεδομένο data_h είναι ένας πίνακας που θα χρησιμοποιηθεί σαν δεδομένο για τους υπολογισμούς στο CPU.

```
float *data_h;
data_h = (float *)malloc(size);
```

- Αποδέσμευση μεταβλητών του Host (CPU) και του Device (GPU) αντίστοιχα.

```
free(data_h);
CUDAFree(data_d);
```

- Εντολή που αντιγράφει μια μεταβλητή από το Host (CPU) στο Device (GPU). Στη συγκεκριμένη εντολή αντιγράφει το πίνακα data_h που είναι μια μεταβλητή που εκτελείτε στο CPU στο πίνακα data_d που είναι μια μεταβλητή που εκτελείτε στο GPU. Αυτές οι μεταβλητές (πίνακες) είναι μεγέθους sizeof(float)*N όπου N είναι το μέγεθος του αρχείου από όπου θα γεμίσουν οι μεταβλητές. Και η συνάρτηση CUDAMemcpyHostToDevice υποδηλώνει ότι θα γίνει αντιγραφή μεταβλητής από το CPU σε μεταβλητή στο GPU.

```
CUDAMemcpy(data_d,data_h,sizeof(float)*N,CUDAMemcpyHostToDevice);
```

- Εντολή που αντιγράφει μια μεταβλητή από το Device (GPU) στο Host (CPU). Στη συγκεκριμένη εντολή αντιγράφει το πίνακα data_d που είναι μια μεταβλητή που εκτελείτε στο GPU στο πίνακα data_h που είναι μια μεταβλητή που εκτελείτε στο CPU. Αυτές οι μεταβλητές (πίνακες) είναι μεγέθους sizeof(float)*N όπου N είναι το μέγεθος του αρχείου από όπου θα γεμίσουν οι μεταβλητές. Και η συνάρτηση CUDAMemcpyDeviceToHost υποδηλώνει ότι θα γίνει αντιγραφή μεταβλητής από το GPU σε μεταβλητή στο CPU.

```
CUDAMemcpy(data_h,data_d,sizeof(float)*N,CUDAMemcpyDeviceToHost);
```

- Εντολή που αντιγράφει μια μεταβλητή από το Device (GPU) στο Device (GPU). Στη συγκεκριμένη εντολή αντιγράφει το πίνακα data1_d που είναι μια μεταβλητή που

εκτελείτε στο GPU στο πίνακα data2_d που είναι μια μεταβλητή που εκτελείτε στο GPU. Είναι για περιπτώσεις που από μια συνάρτηση που εκτελείτε στο GPU επιστραφεί ένα αποτέλεσμα που θα το χρειαστεί σαν είσοδο (παράμετρο) μια άλλη συνάρτηση που θα εκτελεστεί στο GPU. Αυτές οι μεταβλητές (πίνακες) είναι μεγέθους sizeof(float)*N όπου N είναι το μέγεθος του αρχείου από όπου θα γεμίσουν οι μεταβλητές. Και η συνάρτηση CUDAMemcpyDeviceToDevice υποδηλώνει ότι θα γίνει αντιγραφή μεταβλητής από το GPU σε μεταβλητή στο GPU.

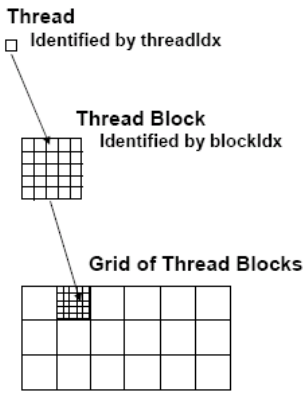
```
CUDAMemcpy(data1_d,data2_d,sizeof(float)*N,CUDAMemcpyDeviceToDevic
e);
```

- Δήλωση του μεγέθους του block και του αριθμού των Blocks, όπου το N είναι ίσο με τον αριθμό του μεγέθους του πίνακα.

```
int blockSize = 4;
int nBlocks = N/blockSize + (N%blockSize == 0?0:1);
```

- Δήλωση threads (η χρήση threads βοηθά στην καλύτερη απόδοση του αλγορίθμου) [7].

```
int idx = blockIdx.x*blockDim.x + threadIdx.x;
```



Σχήμα 3.3: Πολλαπλά επίπεδα παραλληλισμού [7]

Όπως φαίνεται και από το Σχήμα 3.3:

- Ένα threadIdx περιέχει το thread index από το block.
- Ένα blockIdx περιέχει το block index του grid.
- Και το blockDim περιέχει τον αριθμό των threads σε ένα block.

- Αυτή η συνάρτηση καλείται για να συντονίσει την επικοινωνία μεταξύ των threads στο ίδιο block. Όταν μερικά threads μέσα σε έναν block έχουν πρόσβαση στις ίδιες διευθύνσεις στην shared ή global memory, έτσι υπάρχει κίνδυνος να γίνουν κάποιες παραβιάσεις μνήμης όπως read-after-write, write-after-read, or write-after-write . Αυτοί οι κίνδυνοι στοιχείων μπορούν να αποφευχθούν με το συγχρονισμό των threads μεταξύ αυτών των προσβάσεων.

  __syncthreads()

- Κλήση συνάρτησης στο GPU, δηλαδή η συνάρτηση functionKernel εκτελείτε στο GPU με παραμέτρους που θα στείλουν και θα επιστρέψουν τη τιμή τους που είναι η A και B και με [αριθμό Blocks] = nBlocks και [μέγεθος block] = blockSize. Συνήθως οι παράμετροι που επιστρέφονται οι τιμές τους και γίνονται σε αυτές οι διάφοροι υπολογισμοί είναι πίνακες. Αλλά μπορούν να σταλούν και μεταβλητές οι οποίες χρησιμοποιούνται χωρίς να επιστρέψουν την τιμή τους.

  functionKernel <<< nBlocks, blockSize >>> (A, B);

- Συνάρτηση στο GPU, δηλαδή αυτή η συνάρτηση (functionKernel) εκτελείτε στο GPU. Οι τιμές των παραμέτρων επιστρέφονται. Όμως συνήθως οι υπολογισμοί γίνονται σε πίνακες.

  __global__ void functionKernel(float *A, int B){…}

  Η __global__ δήλωση της συνάρτησης εκτελείται στο device αλλά καλείται μόνο από host. Σε αυτή τη συνάρτηση δεν μπορούμε να εκτυπώσουμε, αλλά ούτε να κρατήσουμε θέση μνήμης σε πίνακες γιατί εκτελείτε σε GPU όπου όλα γίνονται παράλληλα και δεν έχουν διευθύνσεις μνήμης τις ίδιες με του CPU.

  Η __device__ δήλωση της συνάρτησης εκτελείται και καλείται μόνο από το device. Σε αυτή τη συνάρτηση δεν μπορούμε να εκτυπώσουμε, αλλά ούτε να κρατήσουμε θέση μνήμης σε πίνακες γιατί εκτελείτε σε GPU όπου όλα γίνονται παράλληλα και δεν έχουν διευθύνσεις μνήμης τις ίδιες με του CPU.

Η __host__ δήλωση της συνάρτησης εκτελείται και καλείται μόνο από το host. Σε αυτή τη συνάρτηση μπορούμε να την υλοποιήσουμε όπως υλοποιούμε σε CPU. Είναι συνάρτηση που εκτελείτε σε CPU και έχει τις ίδιες δυνατότητες με μια συνάρτηση σε C.

- Στη συνέχεια παρουσιάζονται οι τοπικοί τύποι μνήμης πολυεπεξεργαστών (multi-processor) με την ικανότητα ανάγνωσης/γραφής και τη διάρκεια ζωής τους [8]:

Registers:

- Η γρηγορότερη μορφή μνήμης στον πολυεπεξεργαστή (multi-processor).
- Είναι μόνο προσιτή από το thread.
- Έχει τη διάρκεια ζωής του thread.

Shared memory (Κοινή μνήμη):

- Μπορεί να είναι τόσο γρήγορη όσο ένας Register όταν δεν υπάρχει καμία σύγκρουση ή όταν διαβάζουν από την ίδια διεύθυνση.
- Προσιτή από οποιοδήποτε thread του block από τον οποίο δημιουργήθηκε.
- Έχει τη διάρκεια ζωής του block.

Global memory (Σφαιρική μνήμη):

- Ενδεχομένως 150x πιο αργός από τον Register ή την Global.
- Προσιτή είτε από το host είτε το device.
- Έχει τη διάρκεια ζωής της εφαρμογής.

Local memory (Τοπική μνήμη):

- Ανήκει στη Global memory και μπορεί να είναι 150x πιο αργή από το register ή τη shared memory.
- Είναι μόνο προσιτή από το thread.

- Έχει τη διάρκεια ζωής του thread.

- Η CUDA παρέχει τη δυνατότητα εντοπισμού σφαλμάτων και την επιστροφή ενός error code τύπου CUDAError_t. Με τον τρόπο αυτό γίνεται μια καλή διαχείριση των σφαλμάτων για την διόρθωσή τους.

  char *CUDAGetErrorString(CUDAError_t code);

- Μέσω της CUDA έχουμε τη δυνατότητα να καθορίσουμε πόσες φορές θα εκτελεστεί ένα συγκεκριμένο loop. Φυσικά αυτή η δυνατότητα δίνεται μόνο για μικρό αριθμό φορών που θα εκτελεστεί το συγκεκριμένο loop. Η εντολή αυτή είναι:

  ```
  #pragma unroll c
  ```

  Όπου c είναι ένας μικρός ακέραιος αριθμός. Για παράδειγμα:

  ```
  #pragma unroll 5
  for (int i = 0; i < n; ++i){…}
  ```

  το συγκεκριμένο loop θα εκτελεστεί πέντε φορές.

- Όταν η εντολή clock_t clock(); εκτελείται στο device κώδικα, επιστρέφει τη τιμή ενός μετρητή για κάθε πολυεπεξεργαστή, που αυξάνεται κάθε κύκλο ρολογιού (clock cycle). Η δειγματοληψία αυτού του μετρητή λαμβάνεται στην αρχή και στο τέλος ενός πυρήνα, που παίρνει τη διαφορά των δύο δειγμάτων, και που καταγράφει το αποτέλεσμα ανά thread παρέχει ένα μέτρο για κάθε thread του αριθμού κύκλων ρολογιών που λαμβάνονται από το device για να εκτελέσουν εντελώς τα threads, αλλά όχι του αριθμού κύκλου ρολογιού που ξοδεύεται στο device πραγματικά κατά την εκτέλεση των εντολών των threads.

- Γεμίζει τη μνήμη που υποδεικνύεται από το devPtr (Pointer to device memory - Δείκτης στη μνήμη της συσκευής) με τη σταθερή τιμή byte value. Γεμίζει όσες θέσεις υποδηλώνει το count το οποίο είναι μέγεθος σε byte.

  ```
  CUDAMemset(void *devptr, int value, size_t count)
  ```

Χρήση των threads για παράλληλους υπολογισμούς

Σειριακός υπολογισμός στο CPU

C program

```
void increment_cpu(float *a, float b, int N)
{
        int idx;
        for (idx=0; idx<N; idx++)
                a[idx]=a[idx]+b;
}
```

Παράλληλος υπολογισμός στο GPU

CUDA program

```
__global__ void increment_cpu(float *a, float b, int N)
{
    int idx=blockIdx.x*blockDim.x+thread.x;
    if (idx<N)
        a[idx]=a[idx]+b;
}
```

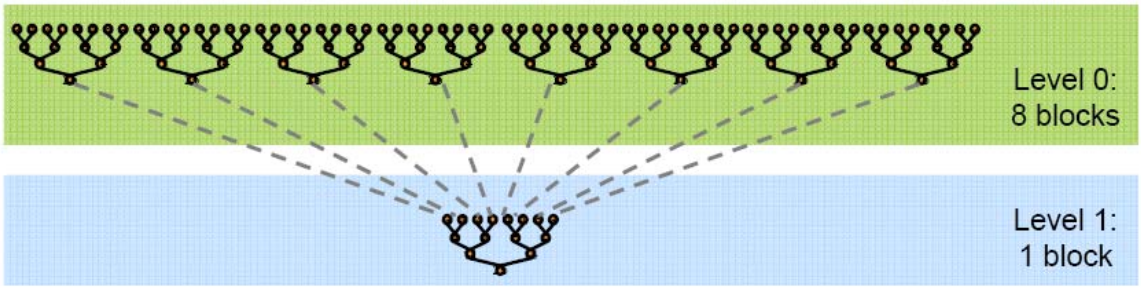
3.4 Περιγραφή και υλοποίηση συνάρτησης που παρουσιάζει το παραλληλισμό στο GPU σε CUDA

Η πιο κάτω συνάρτηση εκτελεί τους υπολογισμούς του προγράμματος και εκτελείτε στη κάρτα γραφικών - GPU. Κάνει κάποιο έλεγχο με βάση κάποια δεδομένα που αποθηκευτήκαν στις μεταβλητές που δεσμεύονται από το device (GPU). Η εκτέλεση αυτών των ελέγχων και οι υπολογισμοί γίνεται παράλληλα μέσω των threads ο αριθμός των οποίων είναι ίσος με το μέγεθος του πίνακα. Κάθε επεξεργαστής εκτελεί ταυτόχρονα με τους υπόλοιπους στο kernel κάποιες λειτουργίες και τα αποτελέσματα αποθηκεύονται στην μνήμη του κάθε επεξεργαστή.

Parallel Reduction [9]

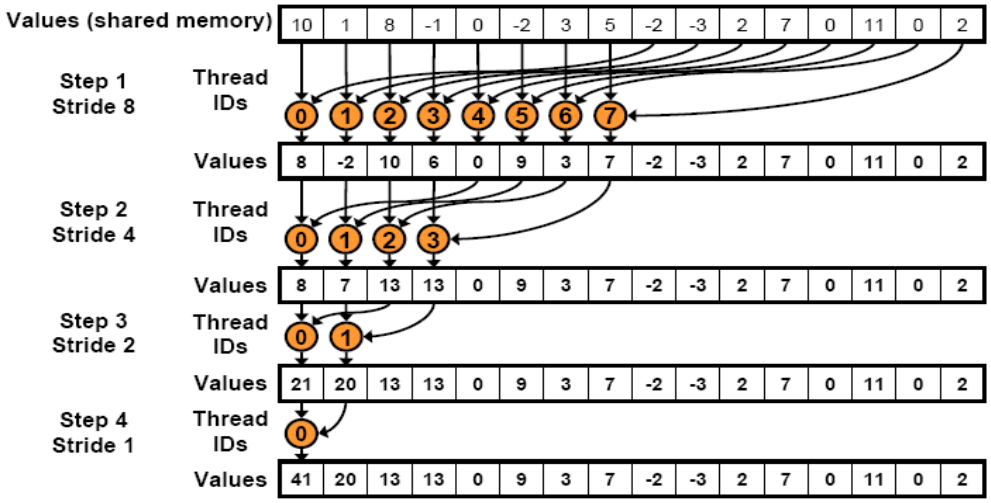
Υποθέστε ότι μας δίνεται μια ακολουθία ακέραιων αριθμών N και πρέπει να συνδυαστείτε σε κάποια μόδα (π.χ., ένα άθροισμα). Αυτό εμφανίζεται σε ποικίλους αλγορίθμους, γραμμική άλγεβρα που είναι ένα κοινό παράδειγμα. Σε έναν serial processor, γράψτε έναν απλό βρόχο με μια ενιαία μεταβλητή συσσωρευτών (accumulator) για να κατασκευάσουν το ποσό όλων των στοιχείων στη σειρά. Σε μια παράλληλη μηχανή, που χρησιμοποιεί μια ενιαία μεταβλητή accumulator θα δημιουργούσε ένα global serialization σημείο και θα οδηγούσε πολύ φτωχή επίδοση.

Μια γνωστή λύση σε αυτό το πρόβλημα είναι ο αποκαλούμενος παράλληλος reduction αλγόριθμος. Κάθε ένα παράλληλο thread αθροίζει ένα καθορισμένου μήκους subsequence της εισόδου. Συγκεντρώνουμε έπειτα αυτά τα μερικά αθροίσματα, με να αθροίσουμε τα ζευγάρια των μερικών αθροισμάτων παράλληλα. Κάθε βήμα αυτού του pair-wise αθροίσματος κόβει τον αριθμό μερικών αθροισμάτων στο μισό και παράγει τελικά το τελικό ποσό μετά από $\log_2 N$ βήματα. Σημειώστε ότι αυτό χτίζει μια δομή δέντρων πέρα από τα αρχικά μερικά αθροίσματα (Σχήμα 3.3).



Σχήμα 3.3: Δομή δέντρου παράλληλης άθροιση [15]

Πιο κάτω φαίνεται σχηματικά (Σχήμα 3.4) η υλοποίηση του παραδείγματος του αθροίσματος αυτού του αλγορίθμου [10].



Σχήμα 3.4: Parallel Reduction: Sequential Addressing [15]

Όπως φαίνεται στο Σχήμα 3.4 κάθε thread προσθέτει δυο αριθμούς παράλληλα με τα υπόλοιπα αθροίσματα των άλλων threads και αυτό γίνεται μέχρι να προστεθούν οι τελευταίοι δυο αριθμοί μέσω ενός thread.

Στη συνέχεια παρουσιάζεται η υλοποίηση του reduction αλγορίθμου για το άθροισμα κάποιων αριθμών [9].

```
__global__ void plus_reduce(int *input, unsigned int N, int *total)
{
        unsigned int tid = threadIdx.x;
        unsigned int i = blockIdx.x*blockDim.x + threadIdx.x;
        // Each block loads its elements into shared memory, padding
        // with 0 if N is not a multiple of blocksize
        __shared__ int x[blocksize];
        x[tid] = (i<N) ? input[i] : 0;
        __syncthreads();
        // Every thread now holds 1 input value in x[]
        // Build summation tree over elements. See attached figure.
        for(int s=blockDim.x/2; s>0; s=s/2)
        {
                if(tid < s)
                        x[tid] += x[tid + s];
                __syncthreads();
        }
        // Thread 0 now holds the sum of all input values
        // to this block. Have it add that sum to the running total
        if( tid == 0 )
                atomicAdd(total, x[tid]);
}
```

Κάποιες έτοιμες συναρτήσεις στη CUDA είναι οι ακόλουθες:

Arithmetic Functions (αριθμητικές συναρτήσεις)

- Διαβάζει μια 32-bit λέξη από κάποια διεύθυνση στη global μνήμης (address), προσθέτει έναν ακέραιο αριθμό σε αυτή (έστω val), και γράφει το αποτέλεσμα πίσω στην ίδια διεύθυνση. Η λειτουργία είναι ατομική υπό την έννοια ότι είναι εγγυημένο να εκτελεστεί χωρίς παρέμβαση από άλλα threads. Η ατομική διαδικασία δουλεύει μόνο με 32-bit ακέραιους αριθμούς και μέχρι η λειτουργία να είναι πλήρης.

  ```
  int atomicAdd(int* address, int val)
  ```

- Διαβάζει μια 32-bit λέξη από κάποια διεύθυνση στη global μνήμης (address), αντικαθιστά έναν ακέραιο αριθμό σε αυτή (val), και γράφει το αποτέλεσμα πίσω στην ίδια διεύθυνση. Η λειτουργία είναι ατομική υπό την έννοια ότι είναι εγγυημένο να εκτελεστεί χωρίς παρέμβαση από άλλα threads. Η ατομική

διαδικασία δουλεύει μόνο με 32-bit ακέραιους αριθμούς και μέχρι η λειτουργία να είναι πλήρης.

int atomicSub(int* address, int val)

- Διαβάζει μια 32-bit λέξη από κάποια διεύθυνση στη global μνήμης (address), και συγκρίνει κάθε φορά έναν ακέραιο αριθμό σε αυτή (val) βρίσκει το μικρότερο, και γράφει το αποτέλεσμα πίσω στην ίδια διεύθυνση. Η λειτουργία είναι ατομική υπό την έννοια ότι είναι εγγυημένο να εκτελεστεί χωρίς παρέμβαση από άλλα threads. Η ατομική διαδικασία δουλεύει μόνο με 32-bit ακέραιους αριθμούς και μέχρι η λειτουργία να είναι πλήρης.

  int atomicMin(int* address, int val)

- Διαβάζει μια 32-bit λέξη από κάποια διεύθυνση στη global μνήμης (address), και συγκρίνει κάθε φορά έναν ακέραιο αριθμό σε αυτή (val) βρίσκει το μεγαλύτερο, και γράφει το αποτέλεσμα πίσω στην ίδια διεύθυνση. Η λειτουργία είναι ατομική υπό την έννοια ότι είναι εγγυημένο να εκτελεστεί χωρίς παρέμβαση από άλλα threads. Η ατομική διαδικασία δουλεύει μόνο με 32-bit ακέραιους αριθμούς και μέχρι η λειτουργία να είναι πλήρης.

  int atomicMax(int* address, int val)

Bitwise Functions

- Διαβάζει τη τιμή της διεύθυνσης του address και το συγκρίνει με το val και συνεχίζει και συγκρίνει κάθε address με το κάθε νέο val για να κάνει το λογικό τους And.

  int atomicAnd(int* address, int val)

- Διαβάζει τη τιμή της διεύθυνσης του address και το συγκρίνει με το val και συνεχίζει και συγκρίνει κάθε address με το κάθε νέο val για να κάνει το λογικό τους Or.

  int atomicOr(int* address, int val)

- Διαβάζει τη τιμή της διεύθυνσης του address και το συγκρίνει με το val και συνεχίζει και συγκρίνει κάθε address με το κάθε νέο val για να κάνει το λογικό τους Xor.

  int atomicXor(int* address, int val)

Κεφάλαιο 4

Περιγραφή του αλγορίθμου Query 6 (Q6)

4.1 Περιγραφή αλγορίθμου

36

4.2 Ψευδοκώδικας αλγορίθμου

36

4.3 Περιγραφή κώδικα αλγορίθμου Q6
37
4.4 Αποτελέσματα αλγορίθμου Q6 σε C++ και CUDA
41

Σκοπός του Κεφαλαίου 4 είναι να παρουσιάσει τον αλγόριθμο Q6 (Q6: Only a single scan operation). Στο πρώτο υποκεφάλαιο 4.1, παρουσιάζεται μια σύντομη περιγραφή αυτού του αλγορίθμου. Δηλαδή τι κάνει αυτός ο αλγόριθμος και τι υπολογίζει. Στη συνέχεια στο δεύτερο υποκεφάλαιο 4.2 Ψευδοκώδικας αλγορίθμου, παρουσιάζεται ο ψευδοκώδικας του αλγορίθμου με λίγα λόγια για αυτόν. Στο Υποκεφάλαιο 4.3 παρουσιάζονται κάποια κομμάτια από το υλοποιημένο κώδικα του αλγορίθμου Q6 που υλοποίησα. Στο υποκεφάλαιο 4.4 Αποτελέσματα αλγορίθμου Q6 σε C++ και CUDA παρουσιάζονται τα αποτελέσματα που πήρα από την εκτέλεση του αλγορίθμου Q6 σε δύο διαφορετικές γλώσσες προγραμματισμού την C++ και τη CUDA για διαφορετικό πλήθος δεδομένων. Παρουσιάζονται και με κάποιες γραφικές παραστάσεις.
Σε αυτό το πείραμα έχει εφαρμοστεί το TPC-H Query 6 (Q6) για εκτέλεση GPU και μελέτησα τις επιδόσεις ενάντια της εφαρμογής CPU.
Ο λόγος που χρησιμοποίησα το Query6 (Q6) είναι γιατί αυτή η διαδικασία βάσεων δεδομένων αποτελεί μια από τις πιο σημαντικές διαδικασίες που συναντιούνται σε σημερινό σύγχρονο DBMSs.

4.1 Περιγραφή αλγορίθμου

Αυτή η επερώτηση υπολογίζει το ποσό αύξησης εισοδήματος που θα είχε προκύψει από την εξάλειψη ορισμένων επιχειρησιακών εκπτώσεων σε μια δεδομένη σειρά ποσοστού σε ένα δεδομένο έτος. Ρωτώντας αυτόν τον τύπο «τι εάν» η επερώτηση μπορεί να χρησιμοποιηθεί για να ψάξει τους τρόπους να αυξηθούν τα εισοδήματα. Οι έλεγχοι γίνονται πάνω σε τεράστιους όγκους δεδομένων.

Η Forecasting Revenue Change (Q6) επερώτηση εξετάζει όλα τα lineitems, το μέγεθος του οποίου είναι 240000, που στέλνονται σε ένα δεδομένο έτος με τις εκπτώσεις μεταξύ έκπτωση-0.01 και DISCOUNT+0.01. Η επερώτηση απαριθμεί το ποσό από το οποίο το συνολικό εισόδημα θα είχε αυξηθεί εάν αυτές οι εκπτώσεις ήταν για τα lineitems με την ποσότητα λιγότερο από l_quantity. Η πιθανή αύξηση εισοδήματος είναι ίση με το ποσό [l_extendedprice * l_discount] για όλα τα lineitems με τις εκπτώσεις και τις ποσότητες στην κατάλληλη σειρά. [1]

4.2 Ψευδοκώδικας αλγορίθμου

Πιο κάτω περιγράφεται ο κώδικας SQL για το Q6.

```
SELECT
        sum(l_extendedprice*l_discount) as revenue
FROM
        lineitem
WHERE
        l_shipdate >= date '[DATE]'
        and l_shipdate < date '[DATE]' + interval '1' year
        and l_discount between [DISCOUNT] - 0.01 and [DISCOUNT] + 0.01
        and l_quantity < [QUANTITY];
```

Τιμές παραμέτρων για τις πιο πάνω παραμέτρους πρέπει να δημιουργούνται και να χρησιμοποιούνται για το εκτελέσιμο αρχείο Query6 [1]:

1. DATE είναι η 1η Ιανουαρίου του ενός τυχαία επιλεγμένου έτους εντός [1993.. 1997]
2. DISCOUNT επιλέγεται τυχαία μέσα [0.02.. 0.00]
3. QUANTITY επιλέγεται τυχαία εντός [24.. 25].

Όπου τα δεδομένα των μεταβλητών αυτών τα πήρα από το αρχείο Lineitem.

Ο πιο πάνω αλγόριθμος (Q6) υλοποιήθηκε σε C++ και CUDA και στη συνέχεια υπολογίστηκε ο χρόνος εκτέλεσης για κάθε ένα από τα δυο προγράμματα. Στην πρώτη

περίπτωση δηλαδή η υλοποίηση σε C++ όλοι οι υπολογισμοί έγιναν στο CPU. Στη δεύτερη περίπτωση οι έλεγχοι του where statement και οι υπολογισμοί του select statement έγιναν στο GPU όπου οι έλεγχοι και οι υπολογισμοί έγιναν παράλληλα με τη χρήση threads. Για να γίνει αυτό χρειάστηκε να καλέσουμε μια συνάρτηση η οποία όπως είναι υλοποιημένη στη γλώσσα προγραμματισμού CUDA εκτελείται στο GPU με κατάλληλο αριθμό από threads. Επίσης οι μεταβλητές που στέλνονται και επιστρέφονται πρέπει να αντιγράφονται σε κατάλληλες μεταβλητές που είναι δηλωμένες για να εκτελούνται σε CPU και GPU αντίστοιχα. Αυτό γίνεται με τη χρήση ιδικών συναρτήσεων της CUDA.

4.3 Περιγραφή κώδικα αλγορίθμου Q6

- Συνάρτηση που κάνει κάποιους υπολογισμούς παράλληλα. Ουσιαστικά εκτελέι κάποιους ελέγχους και εάν ισχύουν κάνει κάποιους υπολογισμούς. Λόγω του ότι στο GPU κάποια πράγματα δεν μπορούν να υλοποιηθούν όπως για παράδειγμα:
    - Να προσθέσουμε το result_count = result_count + 1. Αυτό δεν μπορούμε να το κάνουμε γιατί όλοι οι υπολογισμοί εκτελούνται παράλληλα μέσω threads και η επιστρεφόμενη μεταβλητή πρέπει να είναι πίνακας. Έτσι δεν μπορούν να αναγνωρίσουν μια συγκεκριμένη θέση μνήμης επειδή εκτελούνται στο GPU. Ο τρόπος να υλοποιήσουμε τη πράξη αυτή είναι να υποθηκεύσει το κάθε thread στη θέση μνήμης του threadId 1 εάν ισχύει ο έλεγχος και 0 εάν δεν ισχύει και στο CPU όταν επιστραφεί ο πίνακας με τα αποτελέσματα να προσθέσουμε τη κάθε διεύθυνση μνήμης. Έτσι θα μας δώσει το ζητούμενο αποτέλεσμα.
    - Επίσης το ίδιο συμβαίνει και με το sum[i] = sum[i] + line_item_ex_price[i] * line_item_disc[i] όπου πάλι είναι μια πράξη που περιέχει πρόσθεση του αποτελέσματος με μια άλλη πράξη πολλαπλασιασμού κάθε φορά. Όμως λόγω του ότι εκτελούνται όλες οι πράξεις παράλληλα μέσω threads και τα threads έχουν διαφορετική θέση μνήμης δεν μπορούν να αθροίσουν κάθε φορά το αποτέλεσμα με την υπόλοιπη πράξη. Έτσι απλά αποθηκεύεται το αποτέλεσμα κάθε πράξης στη ξεχωριστή θέση μνήμης του thread και όταν επιστραφεί ο πίνακας με

τα αποτελέσματα στο CPU να προσθέσουμε τη κάθε διεύθυνση μνήμης και θα μας δώσει το ζητούμενο αποτέλεσμα.

```
__global__ void computation(float *line_item_ex_price, float
*line_item_quant, float *line_item_year, float *line_item_disc, float*
result_count, float* sum)
{

  int i=blockIdx.x * blockDim.x + threadIdx.x;


 if (i<LINEITEM){

      // DATE1 and DATE2 means years in float format, eg. 2007.0
      // DISCOUNT AND QUANTITY are other constants given by the user
      if((line_item_year[i]>=(float)DATE1)
            && (line_item_year[i]<(float)DATE2)
            && ((line_item_disc[i]>(float)(DISCOUNT-0.01))
            &&(line_item_disc[i]<(float)(DISCOUNT+0.01)))
            &&(line_item_quant[i]<(float)QUANTITY)){
                  sum[i]=line_item_ex_price[i]*line_item_disc[i];
                  result_count[i]=1;
      }else{
                  sum[i]=0;
                  result_count[i]=0;
      }
 }
}
```

- Δήλωση του mem_size που είναι τύπου size_t και μεγέθους όσο το μέγεθος του πίνακα επί το αριθμό των bytes που έχει κάθε θέση του πίνακα (float). Αυτό θα χρησιμοποιηθεί στη συνέχεια του κώδικα για δέσμευση μεταβλητών που θα χρησιμοποιηθούν είτε στο CPU είτε στο GPU, καθώς επίσης και στη αντιγραφή των μεταβλητών από το Device (GPU) στο Host (CPU) και το αντίθετο.

  ```
  size_t mem_size = N*sizeof(float);
  ```

- Δήλωση μεταβλητών και δέσμευση μνήμης των μεταβλητών αυτών που θα χρησιμοποιηθούν είτε στο CPU είτε στο GPU. Οι μεταβλητές που έχουν το γράμμα «d» μπροστά υποδηλώνουν μεταβλητές που θα χρησιμοποιηθούν στο GPU ενώ αυτές με το «h» στο CPU.

  ```
  float *d_line_item_ex_price;
  ```

```
float *d_line_item_quant;
float *d_line_item_year;
float *d_line_item_disc;
float *h_line_item_ex_price;
float *h_line_item_quant;
float *h_line_item_year;
float *h_line_item_disc;
float *d_sum;
float *d_result_count;
float *h_sum;
float *h_result_count;
```

- Η συνάρτηση cudaMalloc() δεσμεύει μνήμη για το GPU ενώ για το CPU είναι η εντολή malloc() που χρησιμοποιείται σε πολλές γλώσσες προγραμματισμού όπως η C/C++.

```
cudaMalloc((void**) &d_line_item_ex_price, mem_size);
cudaMalloc((void**) &d_line_item_quant, mem_size);
cudaMalloc((void**) &d_line_item_year, mem_size);
cudaMalloc((void**) &d_line_item_disc, mem_size);
cudaMalloc((void**) &d_sum, mem_size);
cudaMalloc((void**) &d_result_count, mem_size);

h_line_item_ex_price = (float*) malloc( mem_size);
h_line_item_quant = (float*) malloc( mem_size);
h_line_item_year = (float*) malloc( mem_size);
h_line_item_disc = (float*) malloc( mem_size);
h_sum = (float*) malloc( mem_size);
h_result_count = (float*) malloc( mem_size);
```

- Με τη συνάρτηση cudaMemcpy() αντιγράφει τις μεταβλητών από το Device (GPU) στο Host (CPU) και το αντίθετο. Συγκεκριμένα με την cudaMemcpyHostToDevice αντιγράφονται τα δεδομένα από το Host στο Device, ενώ με την cudaMemcpyDeviceToHost από το Device στο Host. Ο λόγος που στο πρόγραμμα μου μεταφέρω περισσότερα δεδομένα από το Host στο Device είναι γιατί χρειάζονται στους ελέγχους ενώ επιστρέφονται λιγότερα από το Device στο Host γιατί μόνο αυτά χρειάζομαι στα αποτελέσματα.

```
cudaMemcpy(d_line_item_ex_price, h_line_item_ex_price, mem_size,
                  cudaMemcpyHostToDevice);
cudaMemcpy(d_line_item_quant, h_line_item_quant , mem_size,
                  cudaMemcpyHostToDevice);
cudaMemcpy(d_line_item_year, h_line_item_year, mem_size,
                  cudaMemcpyHostToDevice);
```

```
cudaMemcpy(d_line_item_disc, h_line_item_disc, mem_size,
                  cudaMemcpyHostToDevice);
cudaMemcpy(d_sum , h_sum , mem_size,cudaMemcpyHostToDevice);
cudaMemcpy(d_result_count, h_result_count, mem_size,
                  cudaMemcpyHostToDevice);

cudaMemcpy( h_sum, d_sum, mem_size, cudaMemcpyDeviceToHost);
cudaMemcpy(h_result_count,d_result_count,mem_size,
                  cudaMemcpyDeviceToHost);
```

- Κλήση της συνάρτηση που θα εκτελεστεί στο GPU. Δηλώνουμε το αριθμό και το μέγεθος των Blocks και στέλνουμε σαν παραμέτρους μεταβλητές που μπορούν να εκτελεστούν στο GPU (device) και οι οποίες επιστρέφουν τα αποτελέσματα που θα έχουν και τα οποία στη συνέχεια θα μπορούν να αντιγραφούν από το Device στο Host.

```
computation<<<nBlocks, blockSize >>>(d_line_item_ex_price,
                d_line_item_quant, d_line_item_year,
                      d_line_item_disc,d_result_count,d_sum);
```

- Αποδέσμευση μεταβλητών τόσο του GPU όσο και του CPU. Με την συνάρτηση cudaFree() αποδεσμεύεται μια μεταβλητή του GPU ενώ με τη free() μια μεταβλητή του CPU η οποία χρησιμοποιείται και αυτή σε διάφορες γλώσσες προγραμματισμού όπως η C/C++.

```
// cleanup memory
cudaFree(d_line_item_ex_price);
cudaFree(d_line_item_quant);
cudaFree(d_line_item_year);
cudaFree(d_line_item_disc);
cudaFree(d_result_count);
cudaFree(d_sum);

free(h_line_item_ex_price);
free(h_line_item_quant);
free(h_line_item_year);
free(h_line_item_disc);
free(h_result_count);
free(h_sum);
```

4.4 Αποτελέσματα αλγορίθμου Q6 σε C++ και CUDA

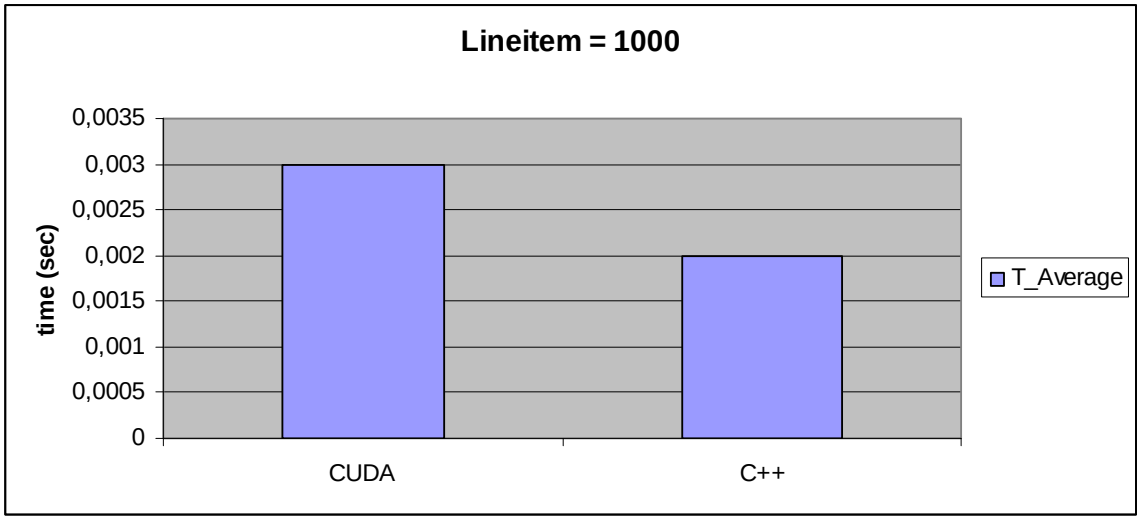
Πιο κάτω στο πίνακα 4.1 παρουσιάζονται οι μετρήσεις για την εκτέλεση του αλγορίθμου Q6 σε CUDA και C++ για πλήθος δεδομένων ίσο με 1000 πλειάδες. Ο αλγόριθμος αυτός εκτελέστηκε δέκα φορές και στο τέλος υπολογίστηκε ο μέσος χρόνος. Όπως φαίνεται ο χρόνος είναι σχεδόν σταθερός σε όλες τις εκτελέσεις είναι περίπου 0.003 sec για CUDA και 0.002 sec σε C++. Ο λόγος που το εκτέλεσα δέκα φορές είναι για να παραχθούν πιο ρεαλιστικά αποτελέσματα.

Πίνακας 4.1

Από το πιο πάνω πίνακα παράχθηκε η πιο κάτω γραφική παράσταση.

Γραφική 4.1

Time	T0	T1	T2	T3	T4	T5	T6	T7	T8	T9	T_Average
CUDA	0	0	0.01	0.01	0	0	0	0	0.01	0	0.003
C++	0	0	0	0	0.01	0	0	0	0	0.01	0.002



Στο πιο πάνω Σχήμα 4.1 βλέπουμε τα συγκριτικά αποτελέσματα του αλγορίθμου Q6 σε CUDA και C++ για μέγεθος του αρχείου Lineitem = 1000 δηλαδή 1000 πλειάδες. Όπως βλέπουμε για μικρό αριθμό δεδομένων το C++ επιτυγχάνει καλύτερο χρόνο από το CUDA αλλά η διαφορά είναι πού μικρή. Ο λόγος είναι γιατί και σε C++ και σε CUDA εκτελούνται σειριακοί υπολογισμοί. Παρόλο που σε CUDA παρέχεται η δυνατότητα της εκτέλεσης παράλληλων υπολογισμών παρόλα αυτά λόγο του γεγονότος ότι δεν μπορούν να υπολογιστούν τα αθροίσματα μέσα στο GPU με παράλληλη εκτέλεση μέσω threads αναγκαζόμαστε να τα υπολογίσουμε σειριακά στο CPU.

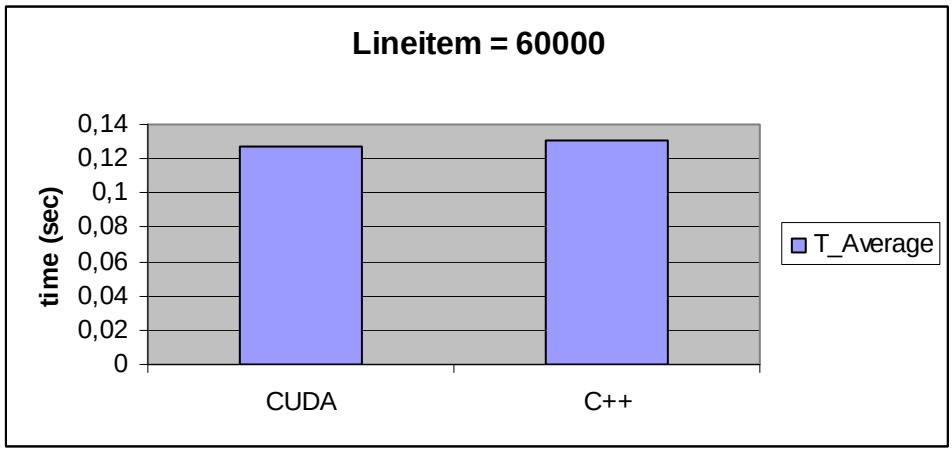
Στη συνέχεια στο πίνακα 4.2 παρουσιάζονται οι μετρήσεις για την εκτέλεση του αλγορίθμου Q6 σε CUDA και C++ για πλήθος δεδομένων ίσο με 60000 πλειάδες. Ο αλγόριθμος αυτός εκτελέστηκε δέκα φορές και στο τέλος υπολογίστηκε ο μέσος χρόνος. Όπως φαίνεται ο χρόνος είναι σχεδόν σταθερός σε όλες τις εκτελέσεις είναι περίπου 0.127 sec για CUDA και 0.131 sec σε C++. Ο λόγος που το εκτέλεσα δέκα φορές είναι για να παραχθούν πιο ρεαλιστικά αποτελέσματα.

Πίνακας 4.2

Time	T0	T1	T2	T3	T4	T5	T6	T7	T8	T9	T_Average
CUDA	0.12	0.14	0.12	0.12	0.14	0.12	0.14	0.12	0.13	0.12	0.127
C++	0.12	0.13	0.12	0.13	0.13	0.14	0.13	0.14	0.14	0.13	0.131

Από το πιο πάνω πίνακα παράχθηκε η πιο κάτω γραφική παράσταση.

Γραφική 4.2



Στο πιο πάνω Σχήμα 4.2 βλέπουμε τα συγκριτικά αποτελέσματα του αλγορίθμου Q6 σε CUDA και C++ για μέγεθος του αρχείου Lineitem = 60000 δηλαδή 60000 πλειάδες. Όπως βλέπουμε για μικρό αριθμό δεδομένων το CUDA επιτυγχάνει καλύτερο χρόνο από το C++ αλλά η διαφορά είναι πού μικρή. Ο λόγος είναι γιατί και σε C++ και σε CUDA εκτελούνται σειριακοί υπολογισμοί. Παρόλο που σε CUDA παρέχεται η δυνατότητα της εκτέλεσης παράλληλων υπολογισμών, παρόλα αυτά λόγο του γεγονότος ότι δεν μπορούν να υπολογιστούν τα αθροίσματα μέσα στο GPU με παράλληλη εκτέλεση μέσω threads αναγκαζόμαστε να τα υπολογίσουμε σειριακά στο CPU.

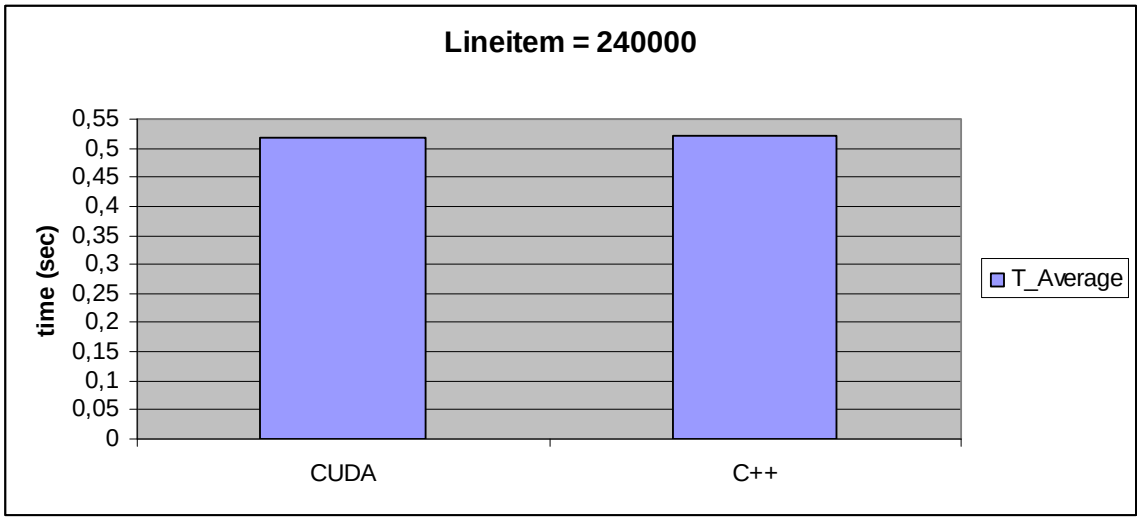
Πιο κάτω στο πίνακα 4.3 παρουσιάζονται οι μετρήσεις για την εκτέλεση του αλγορίθμου Q6 σε CUDA και C++ για πλήθος δεδομένων ίσο με 24000 πλειάδες. Ο αλγόριθμος αυτός εκτελέστηκε δέκα φορές και στο τέλος υπολογίστηκε ο μέσος χρόνος. Όπως φαίνεται ο χρόνος είναι σχεδόν σταθερός σε όλες τις εκτελέσεις είναι περίπου 0.517 sec για CUDA και 0.522 sec σε C++. Ο λόγος που το εκτέλεσα δέκα φορές είναι για να παραχθούν πιο ρεαλιστικά αποτελέσματα.

Πίνακας 4.3

Time	T0	T1	T2	T3	T4	T5	T6	T7	T8	T9	T_Average
CUDA	0.53	0.51	0.52	0.51	0.52	0.52	0.51	0.51	0.51	0.52	0.517
C++	0.51	0.51	0.53	0.53	0.53	0.52	0.52	0.53	0.51	0.53	0.522

Από το πιο πάνω πίνακα παράχθηκε η πιο κάτω γραφική παράσταση.

Γραφική 4.3



Στο πιο πάνω Σχήμα 4.3 βλέπουμε τα συγκριτικά αποτελέσματα του αλγορίθμου Q6 σε CUDA και C++ για μέγεθος του αρχείου Lineitem = 240000 δηλαδή 240000 πλειάδες. Όπως βλέπουμε για μικρό αριθμό δεδομένων το CUDA επιτυγχάνει καλύτερο χρόνο από το C++ αλλά η διαφορά είναι πού μικρή. Ο λόγος είναι γιατί και σε C++ και σε CUDA εκτελούνται σειριακοί υπολογισμοί. Παρόλο που σε CUDA παρέχεται η δυνατότητα της εκτέλεσης παράλληλων υπολογισμών, παρόλα αυτά λόγο του γεγονότος ότι δεν μπορούν να υπολογιστούν τα αθροίσματα μέσα στο GPU με παράλληλη εκτέλεση μέσω threads αναγκαζόμαστε να τα υπολογίσουμε σειριακά στο CPU.

Πιο κάτω στο πίνακα 4.4 παρουσιάζονται οι μετρήσεις για την εκτέλεση του αλγορίθμου Q6 σε CUDA και C++ για πλήθος δεδομένων ίσο με 1000 πλειάδες. Ο αλγόριθμος αυτός εκτελέστηκε δέκα φορές και στο τέλος υπολογίστηκε ο μέσος χρόνος. Όπως φαίνεται ο χρόνος είναι σχεδόν σταθερός σε όλες τις εκτελέσεις είναι περίπου 1.065 sec για CUDA και 1.062 sec σε C++. Ο λόγος που το εκτέλεσα δέκα φορές είναι για να παραχθούν πιο ρεαλιστικά αποτελέσματα.

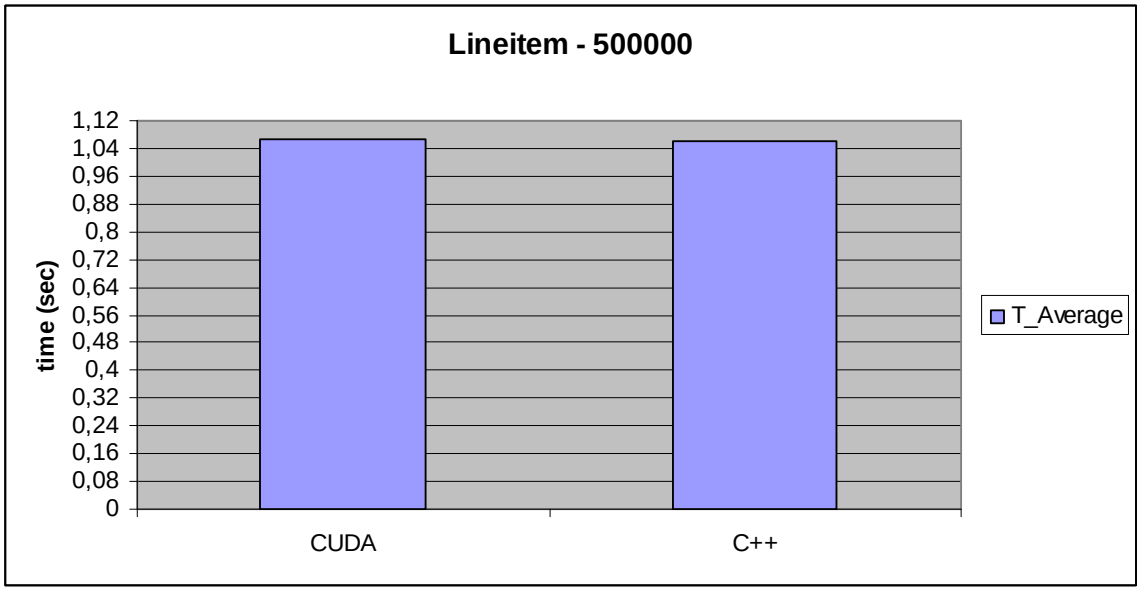
Πίνακας 4.4

Time	T0	T1	T2	T3	T4	T5	T6	T7	T8	T9	T_Average
CUDA	1.07	1.07	1.06	1.07	1.06	1.06	1.07	1.06	1.07	1.06	1.065

C++	1.06	1.07	1.06	0.07	1.07	1.07	1.06	1.07	1.07	0.07	1.067

Από το πιο πάνω πίνακα παράχθηκε η πιο κάτω γραφική παράσταση.

Γραφική 4.4

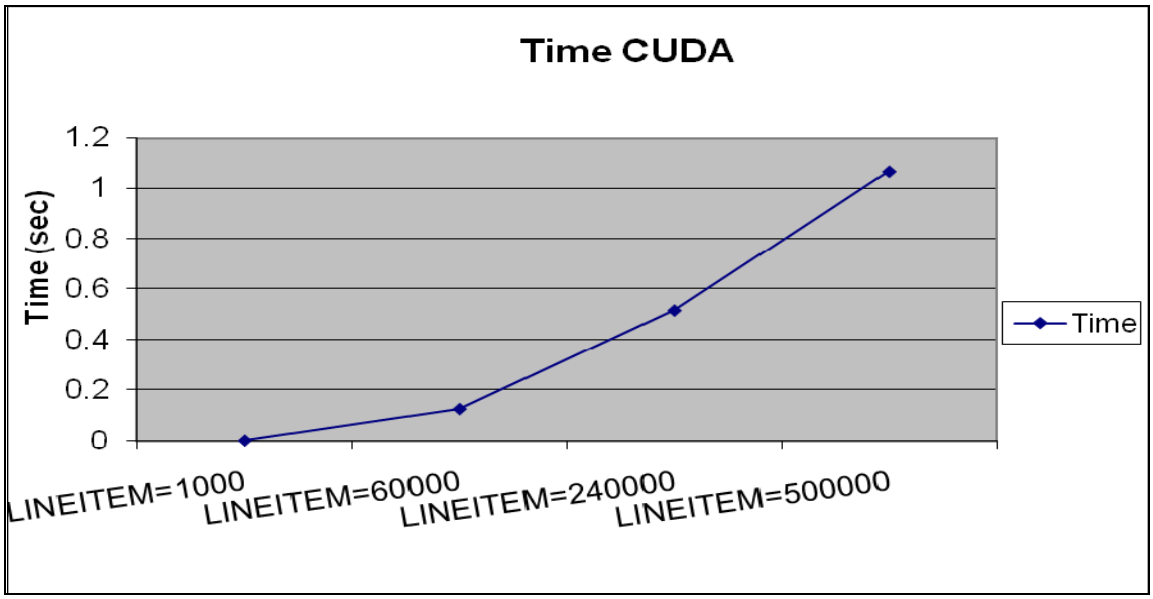


Στο πιο πάνω Σχήμα 4.4 βλέπουμε τα συγκριτικά αποτελέσματα του αλγορίθμου Q6 σε CUDA και C++ για μέγεθος του αρχείου Lineitem = 500000 δηλαδή 500000 πλειάδες. Όπως βλέπουμε για μικρό αριθμό δεδομένων το CUDA επιτυγχάνει καλύτερο χρόνο από το C++ αλλά η διαφορά είναι πού μικρή. Ο λόγος είναι γιατί και σε C++ και σε CUDA εκτελούνται σειριακοί υπολογισμοί. Παρόλο που σε CUDA παρέχεται η δυνατότητα της εκτέλεσης παράλληλων υπολογισμών, παρόλα αυτά λόγο του γεγονότος ότι δεν μπορούν να υπολογιστούν τα αθροίσματα μέσα στο GPU με παράλληλη εκτέλεση μέσω threads αναγκαζόμαστε να τα υπολογίσουμε σειριακά στο CPU.

Για μεγαλύτερο μέγεθος αρχείου (Lineitem > 500000) δεν μπορούν να γίνουν υπολογισμοί στο GPU.

Επίσης με την αλλαγή του block size καθώς και του αριθμού των blocks δεν παρατηρείτε καμία αλλαγή και ο λόγος είναι ότι σε κάθε block υπάρχει ένας αριθμός από threads που καθορίζεται από το GPU. Έτσι εάν είναι πιο λίγα blocks θα περιέχονται περισσότερα threads σε κάθε block ή το αντίθετο.

Σχήμα 4.1



Στο πιο πάνω Σχήμα 4.1 παρατηρούμε ότι κατά την αύξηση των δεδομένων (πλειάδων) του αρχείου αυξάνεται γραμμικά και ο χρόνος. Αυτό ήταν αναμενόμενο αφού θα αυξηθούν οι υπολογισμοί τόσο στο GPU όπου θα εκτελεστούν παράλληλα οι υπολογισμοί όσο και στο CPU που εκτλούντε οι σειριακοί υπολογισμοί.

Κεφάλαιο 5

Συμπεράσματα

5.1 Συμπεράσματα
48
5.2 Μελλοντική εργασία
49

5.1 Συμπεράσματα

Μέσα από την ανασκόπηση της βιβλιογραφίας (βλ. Κεφάλαιο 2) προέκυψε ότι το GPU μπορεί να χρησιμοποιηθεί ως αποδοτικός συνεπεξεργαστής (co-processor) για διαδικασίες βάσεων δεδομένων, για την επεξεργασία εικόνων και βίντεο, για τις ακτινοθεραπείες στην ιατρική, στα τηλεοπτικά παιχνίδια και πολυμέσα κοκ. Πέρα από τα ανωτέρω, διαφάνηκε ότι τα μοντέλα με ισχυρούς συν-επεξεργαστές (co-processors) για τα CPUs προσφέρουν δυνατότητες παράλληλης επεξεργασίας γενικών εφαρμογών συμπεριλαμβανόμενης υποστήριξης για διασκορπισμένες εφαρμογές και επικοινωνία μεταξύ των επεξεργαστών. Ένα τέτοιο μοντέλο είναι η γλώσσα προγραμματισμού για GPGPUs όπως η NVIDIA CUDA. Αξιοσημείωτο είναι το γεγονός ότι διάφορες έρευνες, για τις οποίες έγινε εκτενής αναφορά στο Κεφάλαιο 2, κατέληξαν στο συμπέρασμα ότι η γλώσσα προγραμματισμού CUDA έχει καλύτερη απόδοση σε σχέση με άλλες γλώσσες προγραμματισμού που προσφέρουν παραλληλισμό.

Στόχος της παρούσας εργασίας ήταν να αποδείξει ότι το πακέτο CUDA, μέσω της χρήση του GPGPU, μπορεί να επιτύχει υψηλότερη απόδοση σε σχέση με άλλες γλώσσες προγραμματισμού. Ωστόσο λόγω προγραμματιστικών σφαλμάτων αυτό δεν επιτεύχθηκε.

5.2 Μελλοντική εργασία

Θα ήταν πολύ χρήσιμο στο μέλλον να υλοποιηθούν και άλλοι αλγόριθμοι βάσεων δεδομένων, benchmarks TPC-H, σε CUDA έτσι ώστε να υπάρχουν περισσότερα αποτελέσματα και να γίνει μια καλύτερη σύγκριση. Να εξαχθούν περισσότερα αποτελέσματα και συμπεράσματα. Επίσης θα μπορούσαν οι ίδιοι αλγόριθμοι να υλοποιηθούν και σε άλλη γλώσσα προγραμματισμού που προσφέρει τον παραλληλισμό έτσι ώστε να διαπιστωθεί εάν η CUDA προσφέρει καλά αποτελέσματα στο παραλληλισμό. Να γίνει σύγκριση παράλληλων προγραμμάτων, τα οποία θα εκτελούν ακριβώς τους ίδιους υπολογισμούς στο GPU.

Βιβλιογραφία

[1] *Jack Stephens, member companies in developing* , "Transaction Processing Council – TPC-H, 2007", http://www.tpc.org/tpch/

[2] *Χρύσης Γεωργίου,* "Σημειώσεις Μαθήματος ΕΠΛ 431 – Σύνθεση Παράλληλων Αλγόριθμων, Τμήμα Πληροφορικής, Πανεπιστήμιο Κύπρου, 2008", http://www2.cs.ucy.ac.cy/~chryssis/EPL431/

[3] *JaJa Joseph*, An Introduction to Parallel Algorithms, Addison-Wesley Publishing Company, Reprinted November 1992

[4] *Bingsheng He, Ke Yang, Rui Fang, Mian Lu, Naga K. Govindaraju, Qiong Luo, and Pedro V. Sander*, "Relational Joins on Graphics Processors.", http://portal.acm.org/citation.cfm?id=1376670/

[5] *N. Bandi, C. Sun, D. Agrawal, and A. El Abbadi,* "Hardware acceleration in commercial databases: A case study of spatial operations. Technical report,Computer Science Department, University of California, Santa Barbara, 2004. ", http://www.cs.toronto.edu/vldb04/protected/eProceedings/contents/pdf/IND2P1.PDF

[6] *JOHN NICKOLLS, IAN BUCK, AND MICHAEL GARLAND, NVIDIA, KEVIN SKADRON, UNIVERSITY OF VIRGINIA,*"Programming Guide, NVIDIA CUDA", http://www.nvidia.com/object/cuda_develop.html

[7] *Ian Buck, Bernard Deschizeaux, Mark Harris, John Owens, Jim Phillips, John Stone,*

"High Performance Computing with CUDA", http://gpgpu.org/sc2007/

[8] *Rob Farber,* "CUDA, Supercomputing for the Masses, Dr. Dobb's, CUDA", http://www.ddj.com/cpp/207200659/

[9] *JOHN NICKOLLS, IAN BUCK, AND MICHAEL GARLAND, NVIDIA, KEVIN SKADRON, UNIVERSITY OF VIRGINIA,* "Scalable Parallel PROGRAMMING with CUDA.", http://portal.acm.org/citation.cfm?id=1365500

[10] *Mark Harris NVIDIA Developer Technology,* "High Performance Computing with CUDA, Optimizing CUDA", http://gpgpu.org/static/sc2007/

[11] *Shane Ryoo, Christopher I. Rodrigues, Sara S. Baghsorkhi, Sam S. Stone, David B. Kirk, Wen-mei W. Hwu,* "Optimization Principles and Application Performance Evaluation of a Multithreaded GPU Using CUDA.",

http://portal.acm.org/citation.cfm?id=1345220

[12] *Ahmed El Zein, Eric McCreath, Alistair Rendell, and Alex* ***Smola2,*** "Performance Evaluation of the NVIDIA GeForce 8800 GTX GPU for Machine Learning.", http://www.springerlink.com/content/d4583830557k837u/

[13] Jimmy Wales, "CUDA", http://en.wikipedia.org/wiki/CUDA

[14] *Naga Govindaraju, Jim Gray, Ritesh Kumar, Dinesh Manocha*, "GPUTeraSort: high performance graphics co-processor sorting for large database management", http://portal.acm.org/citation.cfm?id=1142511

[15] *Tom R. Halfhill,* "PARALLEL PROCESSINGWITH CUDA", http://www.nvidia.com/docs/IO/55972/220401_Reprint.pdf

[16] *Mark Harris,* "Optimizing Parallel Reduction in CUDA", http://developer.download.nvidia.com/compute/cuda/1_1/Website/projects/reduction/doc/reduction.pdf

[17] *Johan Seland*, "CUDA Programming", http://heim.ifi.uio.no/~knutm/geilo2008/seland.pdf

[18] "ATI CTM Guide, Technical Reference Manual" http://ati.amd.com/companyinfo/researcher/documents/ATI_CTM_Guide.pdf

[18] Fatih Porikli, Jay Thornton , "GPU for Surveillance", http://www.merl.com/projects/gpusurveillance/

[19] Atsushi Shiraki, Naoki Takada, Masashi Niwa, Yasuyuki Ichihashi, Tomoyoshi Shimobaba, Nobuyuki Masuda,Tomoyoshi Ito, "Simplified electroholographic color reconstruction system using graphics processing unit and liquid crystal display projector", http://www.opticsinfobase.org/DirectPDFAccess/CA20F62F-BDB9-137E-CAD99DC7311F1AE4_185577.pdf?da=1&id=185577&seq=0

[20] Luke Domanski, Pascal Vallotton and Dadong Wang ,"Two and Three-Dimensional Image Deconvolution on Graphics Hardware", http://www.mssanz.org.au/modsim09/C5/domanski.pdf

[21] M. Andrecut, "Parallel GPU Implementation of Iterative PCA Algorithms", http://arxiv.org/PS_cache/arxiv/pdf/0811/0811.1081v1.pdf

[22] Martijn de Greef, Jordy van Eijk, Ren´e Pool, Hans Crezee, Davi Correia and Arjan Bel, “GPU accelerated dose calculations for radiotherapy”, http://www.amc.nl/upload/teksten/radiotherapie/hyperthermie/RayForDose-NVIDIA.pdf

Παράρτημα Α

Παρουσίαση υλοποιημένων αλγορίθμων

Α.1 Αλγόριθμος Q6 σε CUDA

```
#include <stdio.h>
#include <cuda.h>

#define LINEITEM 240000


const int DATE1 = 1994;
const int DATE2 = 1995;
const float DISCOUNT = 0.01;
const int QUANTITY = 24;

__global__ double duration;
int result_count=0;
clock_t start, finish;

using namespace std;

// DATE1 and DATE2 means years in float format, eg. 2007.0
// DISCOUNT AND QUANTITY are other constants given by the user

void readLineitem(float *line_item_ex_price, float *line_item_quant,
float *line_item_year, float *line_item_disc){

      int i, j;
      int c = LINEITEM;
      int count=0;

      FILE *fp;
      char line[200];
      char *tock;
      char *year, *month, *day;
      char *ext_price, *quantity, *discount;
      fp=fopen("lineitem.tbl","r"); /*Open the file Input*/

      if(fp== NULL)
      {
            fprintf(stderr,"The file lineitem can not open!!\n");
            exit(-1);
      }

      while ((fp != NULL ) && (count<c)){

                  fgets ( line, sizeof line, fp);


            //the 7th tocken is the discount
            tock = strtok(line, "|");
```

```
            for(i=0; i<5; i++){
                  tock = strtok(NULL, "|");
            }

            ext_price = tock;

            quantity = strtok(NULL, "|");

            discount = strtok(NULL, "|");



            //the 11th is the date int the form yyyy-mm-dd
            for(i=7; i<11; i++){
                  tock = strtok(NULL, "|");
            }

            year = strtok(tock, "-");
            month = strtok(NULL, "-");
            day = strtok(NULL, "-");


            line_item_ex_price[count]=atof(ext_price); //extended price

            line_item_quant[count]=atof(quantity);  //quantity

            line_item_year[count]=atof(year);        //year

            line_item_disc[count]=atof(discount); //discount
            count=count+1;


            }

      fclose(fp);
}



__global__ void computation(float *line_item_ex_price, float
*line_item_quant, float *line_item_year, float *line_item_disc, float*
result_count, float* sum)
{

  int i=blockIdx.x * blockDim.x + threadIdx.x;


 if (i<LINEITEM){

      // DATE1 and DATE2 means years in float format, eg. 2007.0
      // DISCOUNT AND QUANTITY are other constants given by the user
      if((line_item_year[i]>=(float)DATE1) &&
(line_item_year[i]<(float)DATE2)          &&
((line_item_disc[i]>(float)(DISCOUNT-0.01))
            &&(line_item_disc[i]<(float)(DISCOUNT+0.01)))
            &&(line_item_quant[i]<(float)QUANTITY)){
                  sum[i]=line_item_ex_price[i]*line_item_disc[i];
```

```
                result_count[i]=1;
        } else {
                sum[i]=0;
                result_count[i]=0;
        }
 }
__syncthreads()
}



int main(){

        // Timing Variables
        float duration=0;
        clock_t start[10];
        clock_t finish[10];


        int N=LINEITEM;
        size_t mem_size = N*sizeof(float);

        float *d_line_item_ex_price;
        float *d_line_item_quant;
        float *d_line_item_year;
        float *d_line_item_disc;
        float *h_line_item_ex_price;
        float *h_line_item_quant;
        float *h_line_item_year;
        float *h_line_item_disc;
        float *d_sum;
        float *d_result_count;
        float *h_sum;
        float *h_result_count;

        cudaMalloc((void**) &d_line_item_ex_price, mem_size);
        cudaMalloc((void**) &d_line_item_quant, mem_size);
        cudaMalloc((void**) &d_line_item_year, mem_size);
        cudaMalloc((void**) &d_line_item_disc, mem_size);
        cudaMalloc((void**) &d_sum, mem_size);
        cudaMalloc((void**) &d_result_count, mem_size);

        h_line_item_ex_price = (float*) malloc( mem_size);
        h_line_item_quant = (float*) malloc( mem_size);
        h_line_item_year = (float*) malloc( mem_size);
        h_line_item_disc = (float*) malloc( mem_size);
        h_sum = (float*) malloc( mem_size);
        h_result_count = (float*) malloc( mem_size);

        for(int k=0; k<10; k++){

        start[k]=clock();

        readLineitem(h_line_item_ex_price, h_line_item_quant,
h_line_item_year,             h_line_item_disc);

        h_sum[1]=0;
```

```
      h_result_count[1]=0;

      cudaMemcpy(d_line_item_ex_price, h_line_item_ex_price, mem_size,
                        cudaMemcpyHostToDevice);
      cudaMemcpy(d_line_item_quant, h_line_item_quant , mem_size,
                        cudaMemcpyHostToDevice);
      cudaMemcpy(d_line_item_year, h_line_item_year, mem_size,
                        cudaMemcpyHostToDevice);
      cudaMemcpy(d_line_item_disc, h_line_item_disc, mem_size,
                        cudaMemcpyHostToDevice);
      cudaMemcpy(d_sum , h_sum , mem_size,cudaMemcpyHostToDevice);
      cudaMemcpy(d_result_count, h_result_count, mem_size,
                        cudaMemcpyHostToDevice);

      int blockSize = 4;
      int nBlocks = N/blockSize + (N%blockSize == 0?0:1);

      computation<<<nBlocks, blockSize >>>(d_line_item_ex_price,
d_line_item_quant, d_line_item_year, d_line_item_disc,
d_result_count,d_sum);

      cudaMemcpy( h_sum, d_sum, mem_size, cudaMemcpyDeviceToHost);
      cudaMemcpy(h_result_count,d_result_count,mem_size,cudaMemcpyDevic
eToHost);

      // Time - End
      finish[1]=clock();
      duration= (double)(finish[1]-start[1])/CLOCKS_PER_SEC;

      }

      float sum = 0.0;
      float r_count = 0;
      printf("GPU results \n----------\n");
      for(int i=0;i<N;i++){
            sum+=h_sum[i];
            r_count+=h_result_count[i];
      }
      printf("Sum_d=%.1f\n", sum);
      printf("Result count_d=%.1f\n", r_count);


      sum = 0.0;
      float result_count = 0.0;

      for (int i=0;i<N;i++){

      if((h_line_item_year[i]>=(float)DATE1) &&
(h_line_item_year[i]<(float)DATE2) &&
((h_line_item_disc[i]>(float)(DISCOUNT-0.01)) &&
(h_line_item_disc[i]<(float)(DISCOUNT+0.01))) &&
(h_line_item_quant[i]<(float)QUANTITY)){
                  sum=sum +
h_line_item_ex_price[i]*h_line_item_disc[i];
                  result_count=result_count+1.0;
            }
      }
```

```
      printf("CPU results \n----------\n");
      printf("Sum = %.1f \n Result count=%.1f\n", sum, result_count);



// Time - End

 finish[k]=clock();
 duration= (double)(finish[k]-start[k])/CLOCKS_PER_SEC;
 clk[k]=duration;
 }

 float average=0;
 for (int k=0; k<10;k++){
 printf("****** Time[%d] =%.5f ******\n",k,clk[k]);
 average=average+clk[k];
}
 average = average/10;
 printf("\n****** Average Time = %.3f ******\n",average);

      // cleanup memory
      cudaFree(d_line_item_ex_price);
      cudaFree(d_line_item_quant);
      cudaFree(d_line_item_year);
      cudaFree(d_line_item_disc);
      cudaFree(d_result_count);
      cudaFree(d_sum);

      free(h_line_item_ex_price);
      free(h_line_item_quant);
      free(h_line_item_year);
      free(h_line_item_disc);
      free(h_result_count);
      free(h_sum);


      return 0;
}
```

A.2 Αλγόριθμος Q6 σε C++

```
#include <stdio.h>
#include <string.h>
#include <ctime>
#include <iostream>
#include <iomanip>

void compute_average(float times[]);
float times[10];

const int  LINEITEM = 240000;
const int  DATE1 = 1994;
const int  DATE2 = 1995;
const float  DISCOUNT = 0.01;
const int  QUANTITY = 24;
const int  ITERATIONS = 1;
const int  WARMUP_ITERATIONS = 3;
//const int  LINE_MAX = 200;


float line_item[LINEITEM][4];


using namespace std;

// DATE1 and DATE2 means years in float format, eg. 2007.0
// DISCOUNT AND QUANTITY are other constants given by the user


int main(){

      // Timing Variables
```

```
double process_data_time;
float sum=0;
int k=0;

// Other Variables
int i=0,j=0;
int count = 1;
int result_count=0;
FILE *fp;
char line[200];
char *tock;
char *year, *month, *day;
char *ext_price, *quantity, *discount;
double duration;
clock_t start, finish;

//Time - Start
start=clock();

fp=fopen("lineitem.tbl","r"); /*Open the file Input*/

if(fp== NULL)
{
      cout<<"The file lineitem can not open!!\n";
      exit(-1);
}


/* Antigrafei olon ton dedomenon apo to arxeio LINEITEM
   se kapoies metavlites */

while ((fp != NULL ) && (count<LINEITEM)){

      fgets ( line, sizeof line, fp); / *antigrafei stin
      metavliti line  ka8e fora mia grami apo to arxeio pou
      dixnei o diktis fp */

      tock = strtok(line, "|"); /* tha parei to string prin apo
      to prwto "|" kai tha to apothikeusei sto tock. */

      for(i=0; i<5; i++){
            tock = strtok(NULL, "|");

       }
      ext_price = tock;

      quantity = strtok(NULL, "|");

      discount = strtok(NULL, "|");

      line_item[count][3]=atof(discount); //discount

      //the 11th is the date int the form yyyy-mm-dd
      for(i=7; i<11; i++){
            tock = strtok(NULL, "|");
      }
```

```
            year = strtok(tock, "-");
            month = strtok(NULL, "-");
            day = strtok(NULL, "-");

            line_item[count][0]=atof(ext_price); //extended price
            line_item[count][1]=atof(quantity);  //quantity
            line_item[count][2]=atof(year);       //year
            count++;
      }

      fclose(fp);

      sum=0;

            for(i=0;i<LINEITEM;++i){

                  // line_item[i][0]: l_extendedprice
                  // line_item[i][1]: l_discount
                  // line_item[i][2]: l_shipdate
                  // line_item[i][3]: l_quantity

                  // DATE1 and DATE2 means years in float format, eg.
2007.0
                  // DISCOUNT AND QUANTITY are other constants given by
the user

                  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(k<ITERATIONS-1)
                  sum=0;

            // Time - End
            finish=clock();
            duration= (double)(finish-start)/CLOCKS_PER_SEC;
            //cout << "DURATION: "<<duration<<"\n\n";
            times[k]=duration;

      compute_average(times);

      cout<<"\nResult:  Sum is "<< setprecision(4) << setw(6)<< sum<<"
      (count = "<<result_count<<").\n\n";

      cout<<"Time: "<<duration<<" seconds.\n";

      return 0;
}

void compute_average(float times[]){
```

```
	float min=times[0];
	float max=times[9];
	int min_index=0,max_index=9;
	float average_time=0;
	int i=0;

	cout<<"\nVarious Execution Times\n";
	cout<<"-----------------------";


	//*****************
	// Find min and max *
	//*****************
	for(i=0;i<10;++i){

		if (times[i]<min){
			min=times[i];
			min_index=i;
		}

		else if (times[i]>max){
			max=times[i];
			max_index=i;
		}
	}



	cout<<"\nMin is: "<<min<<" - Min index: "<<min_index<<"\n";
	cout<<"Max is: "<<max <<" - Max index: "<< max_index <<"\n";

	//**************
	// Compute Sum *
	//**************
	for(i=0;i<10;++i){
		if((i!=min_index)&&(i!=max_index))
			average_time+=times[i];
	}

	//*******************************
	// Compute Average Execution Time *
	//*******************************
	average_time=average_time/8.0;
	cout<<"Average Execution Time: "<<average_time;
	cout<<"\n";

}
```