Πανεπιστήμιο Θεσσαλίας
Πολυτεχνική Σχολή
Τμήμα Ηλεκτρολόγων Μηχανικών & Μηχανικών Υπολογιστών

Επεξεργασία Χωρικών Ερωτημάτων σε Κάρτα Γραφικών

Διπλωματική Εργασία

ΑΓΓΕΛΟΥ ΣΠΑΘΑΡΑ

Επιβλέπων
Μιχαήλ Βασιλακόπουλος
Αναπληρωτής Καθηγητής

Βόλος, Ιούνιος 2020

Πανεπιστήμιο Θεσσαλίας
Πολυτεχνική Σχολή
Τμήμα Ηλεκτρολόγων Μηχανικών & Μηχανικών Υπολογιστών

Επεξεργασία Χωρικών Ερωτημάτων σε Κάρτα Γραφικών

Διπλωματική Εργασία

ΑΓΓΕΛΟΥ ΣΠΑΘΑΡΑ

Επιτροπή επίβλεψης

Επιβλέπων	Συνεπιβλέπων	Συνεπιβλέπων
Μιχαήλ Βασιλακόπουλος	Γεώργιος Σταμούλης	Γεώργιος Θάνος
Αναπληρωτής Καθηγητής	Καθηγητής	Μέλος Ε.ΔΙ.Π.

Βόλος, Ιούνιος 2020

Πανεπιστήμιο Θεσσαλίας
Πολυτεχνική Σχολή
Τμήμα Ηλεκτρολόγων Μηχανικών & Μηχανικών Υπολογιστών

Ο συγγραφέας αυτής της εργασίας βεβαιώνει ότι κάθε βοήθεια την οποία είχε για την προετοιμασία της είναι πλήρως αναγνωρισμένη και αναφέρεται στην εργασία. Επίσης βεβαιώνει ότι έχει αναφέρει τις όποιες πηγές από τις οποίες έκανε χρήση δεδομένων, ιδεών ή λέξεων, είτε αυτές αναφέρονται επακριβώς, είτε παραφρασμένες.

University of Thessaly
Faculty of Engineering
Department of Electrical & Computer Engineering

GPU-based Processing of Spatial Queries

Diploma Thesis

ANGELOS SPATHARAS

Supervisor
Michael Vassilakopoulos
Associate Professor

Volos, June 2020

Περίληψη

Οι πρόσφατες εξελίξεις στις Μονάδες Επεξεργασίας Γραφικών (ΜΕΓ) επέτρεψαν τον φθηνό υπολογισμό υψηλής απόδοσης για εφαρμογές γενικού σκοπού. Λόγω της τεράστιας υπολογιστικής ικανότητας της ΜΕΓ, έχει αναδειχθεί ως συν-επεξεργαστής της Κεντρικής Μονάδας Επεξεργασίας (ΚΜΕ) για να επιτύχει υψηλή συνολική απόδοση. Το μοντέλο προγραμματισμού CUDA παρέχει στους προγραμματιστές επαρκή μέσα για την καλύτερη εκμετάλλευση της παράλληλης ισχύος της ΜΕΓ. Το ερώτημα εύρεσης "Όλων των k-Πλησιέστερων Γειτόνων" (ΟkΠΓ) ανήκει στην οικογένεια των χωρικών ερωτημάτων και η υπολογιστική-εντατική φύση του απαιτεί υλοποίηση υψηλής απόδοσης. Αποτελεί επέκταση του ερωτήματος "Όλων των k-Πλησιέστερων Γειτόνων", μιας ευρέως διαδεδομένης τεχνικής κατηγοριοποίησης. Υλοποιήθηκαν αλγόριθμοι παράλληλης επεξεργασίας του, για δυσδιάστατα σύνολα δεδομέων σε κάρτες γραφικών Nvidia, κάνοντας χρήση αποκλειστικά της βιβλιοθήκης προτύπων C++ του CUDA, τη Thrust. Βασίζεται στη Τυπική Βιβλιοθήκη Προτύπων (STL) και παρέχει παράλληλους αλγορίθμους και δομές δεδομένων, απαλλάσσοντας τον προγραμματιστή από τη διαχείριση των πόρων της ΜΕΓ και το διαμοιρασμό εργασιών και δεδομένων μεταξύ των νημάτων CUDA. Αξιολογήθηκαν οι επιδόσεις των παράλληλων αλγορίθμων μέσα από μία σειρά πειραμάτων χρησιμοποιώντας φυσικά σύνολα δεδομένων και τυχαία σύνολα ομοιόμορφης κατανομής που χωράν εξ' ολοκλήρου στη μνήμη συσκευής. Έγινε σύγκριση αυτών με αντίστοιχες παράλληλες υλοποιήσεις ΟkΠΓ σε ΚΜΕ και ΜΕΓ. Τα αποτελέσματα έδειξαν ότι το μεγαλύτερο ποσοστό του συνολικού χρόνου εκτέλεσης καταλαμβάνει η μεταφορά δεδομένων από τη μνήμη κεντρικού υπολογιστή στη μνήμη της κάρτας γραφικών και το αντίστροφο.

Λέξεις Κλειδιά

Όλοι k-Πλησιέστεροι Γείτονες, Σάρωση επιπέδου, Κάρτα γραφικών, Βιβλιοθήκη Thrust

Abstract

Recent developments in GPUs have allowed the inexpensive computing of high performance for general purpose applications. GPU's enormous computing power, led to its emergence as a co-processor of the CPU to achieve high overall performance. The CUDA programming model provides developers with sufficient means to make the best use of the parallel power of the GPU. The query of finding "All of the k-Nearest Neighbors" (AkNN) belongs to the family of spatial queries and its computational-intensive nature requires high performance implementation. It is an extension of the query "All of the k-Nearest Neighbors", a widely used classification technique. Its parallel processing algorithms have been implemented for two-dimensional datasets on Nvidia graphics cards, using exclusively the CUDA C++ standard library, Thrust. It is based on the Standard Template Library (STL) and provides parallel algorithms and data structures, freeing the developer from managing GPU resources and from sharing work and data among the CUDA threads. The performance of the parallel algorithms was evaluated through a series of experiments using natural datasets and random sets of uniform distribution that fit into the device's memory and were compared with similar parallel implementations in CPU and GPU. In conclusion, the largest percentage of total execution time is occupied by the transfer of data from the computer's memory to the graphics card's memory and vice versa.

Keywords

All k-Nearest Neighbors, Plane Sweep, Graphics card, Thrust Library

Ευχαριστίες

Θα ήθελα καταρχήν να ευχαριστήσω τον αναπληρωτή καθηγητή κ. Μιχαήλ Βασιλακόπουλο για την επίβλεψη αυτής της διπλωματικής εργασίας. Επίσης ευχαριστώ ιδιαίτερα το μέλος επιτροπής ερευνών Πολυχρόνη Βελέτζα για τη συνεργασία και την πολύτιμη συμβολή του στην ολοκλήρωση της εργασίας. Τέλος θα ήθελα να ευχαριστήσω τους γονείς μου για την καθοδήγηση και την ηθική συμπαράσταση που μου προσέφεραν όλα αυτά τα χρόνια.

Περιεχόμενα

Κατάλογος σχημάτων

Κατάλογος πινάκων

Κεφάλαιο 1

Εισαγωγή

Μετά από δεκαετίες επίτευξης σταθερών κερδών στην τιμή και την απόδοση, ο νόμος του Moore έφτασε τελικά στο απόγειο του για τις μονάδες κεντρικής επεξεργασίας, ή CPUs. Ευτυχώς, για τις βάσεις δεδομένων, την ανάλυση μεγάλων δεδομένων και τη μηχανική μάθηση, υφίσταται πλέον μια πιο ικανή και οικονομική εναλλακτική λύση για την κλιμάκωση της απόδοσης υπολογισμού, η μονάδα επεξεργασίας γραφικών, ή GPU. Ενώ οι μεμονωμένοι πυρήνες CPU είναι γρηγορότεροι και πιο έξυπνοι από τους μεμονωμένους πυρήνες GPU, ο τεράστιος αριθμός πυρήνων GPU καθιστά τις κάρτες γραφικών ιδανικές για επαναλαμβανόμενες και εξαιρετικά παράλληλες υπολογιστικές εργασίες. Αποδεικνύονται έμπρακτα πως οι κάρτες γραφικών διαθέτουν μεγάλη ποικιλία εφαρμογών και με τις εξελίξεις στο σχεδιασμό τους έχουν πλέον συμβαδίσει με την αδυσώπητη αύξηση του όγκου, της ποικιλίας και ταχύτητας των δεδομένων που αντιμετωπίζουν οι οργανισμοί σήμερα.

Μια κατηγορία που απαιτεί ιδιαίτερα μεγάλη υπολογιστική ισχύ είναι τα χωρικά ερωτήματα. Ο όρος "απαιτητικά χωρικά ερωτήματα" αναφέρεται σε μια κατηγορία ερωτημάτων προς χωρικές βάσεις δεδομένων. Η παράλληλη επεξεργασία περιγράφεται ως η διαδικασία διάσπασης ενός προβλήματος σε επιμέρους υποπροβλήματα με στόχο την ταυτόχρονη επίλυσή τους από διαφορετικές υπολογιστικές μονάδες και τη σύνθεση των επιμέρους λύσεων ώστε να σχηματίσουν την τελική λύση στο αρχικό πρόβλημα. Τα χωρικά ερωτήματα είναι μια κατηγορία ερωτημάτων βάσεων δεδομένων κάνουν αναζήτηση αντικειμένων του δυσδιάστατου, τρισδιάστατου ή ν-διάστατου χώρου: σημεία, ευθείες, ή σχήματα. Ερωτήματα αυτού του είδους βρίσκουν εφαρμογή κυρίως σε γεωγραφικά συστήματα πληροφοριών (GIS) αλλά και σε ποικίλους τομείς της πληροφορικής όπως η εξόρυξη γνώσης.

1.1 Αντικείμενο της διπλωματικής

Το ερώτημα εύρεσης "Όλων των k πλησιέστερων γειτόνων" ("All k Nearest Neighbors") αποτελεί ένα χωρικό ερώτημα που απαιτεί ιδιαίτερα μεγάλο όγκο υπολογισμών. Τόσο το πλήθος, όσο και η ανεξαρτησία των επαναλαμβανόμενων εργασιών ελκύουν την παραλληλοποίησή του. Οι μονάδες επεξεργασίας γραφικών λόγω του μεγάλου πλήθους αριθμητικών και λογικών μονάδων φαίνεται να εφάπτονται στις υπολογιστικές ανάγκες αυτού του ερωτήματος. Ωστόσο, η χρήση μιας

κάρτας γραφικών δημιουργεί επιπλέον ανάγκες. Το μεγάλο πλήθος πυρήνων και πόρων απαιτεί προσεκτική διαχείριση και συγχρονισμό, ώστε να μην υπάρχουν αλληλοσυγκρουέμενες εργασίες μεταξύ των πυρήνων. Επίσης, εξαιτίας της ανεξαρτησίας της μνήμης κάρτας γραφικών, από εκείνη του κεντρικού υπολογιστή, κρίνεται αναγκαία η μεταφορά όλων των απαραίτητων δεδομένων από και προς αυτή. Συνεπώς, για να είναι ωφέλιμη μία τέτοια προσέγγιση θα πρέπει το χρονικό κόστος συχρονισμού και αντιγραφής δεδομένων να είναι μικρότερο από την χρονική βελτίωση που πιθανόν να δίνει η παραλληλοποίηση. Το στοίχημα αυτό εξετάζει η εν λόγω εργασία, υλοποιώντας το ΟκΠΓ ερώτημα με μια σειρά παράλληλων αλγορίθμων που εκτελούνται σε κάρτες γραφικών κατασκευαστή Nvidia με τη χρήση της βιβλιοθήκης Cuda, Thrust.

1.1.1 Συνεισφορά

Η συνεισφορά της διπλωματικής αυτής εργασίας συνοψίζεται ως εξής:

1. Υλοποιήθηκαν έντεκα παράλληλοι αλγόριθμοι εύρεσης "Όλων των k Πλησιέστερων Γειτόνων" σε κάρτα γραφικών.

2. Χρησιμοποιήθηκε η βιβλιοθήκη Thrust του CUDA API για τον προγραμματισμό της κάρτας γραφικών Nvidia.

3. Αξιολογήθηκε η επίδοση των αλγορίθμων και έγινε σύγκριση της χρονικής απόκρισης με αντίστοιχες παράλληλες υλοποιήσεις σε επεξεργαστές και κάρτες γραφικών.

1.2 Οργάνωση του τόμου

Η διπλωματική είναι δομημένη σε οχτώ κεφάλαια. Το κεφάλαιο 1 πραγματεύεται το θέμα της έρευνας και την συνεισφορά αυτής στην παραλληλοποίηση χωρικών ερωτημάτων σε κάρτες γραφικών. Στο κεφάλαιο 2 παρατίθενται δεδομένα από εργασίες σχετικές με το αντικείμενο της εν λόγω έρευνας, που συλλέχθηκαν μέσα από την βιβλιογραφική ανασκόπηση. Το θεωρητικό υπόβαθρο εντοπίζεται στο κεφάλαιο 3, στο οποίο αποσαφηνίζονται βασικοί ορισμοί που αξιοποιήθηκαν στην εργασία και γίνεται μια λεπτομέρη ανάλυση τόσο στην αρχιτεκτονική όσο και στον προγραμματισμό των καρτών γραφικών Nvidia. Οι ακολουθιακοί αλγόριθμοι και οι τεχνικές που αποτέλεσαν τον πυλώνα της εργασίας περιγράφονται στο Κεφάλαιο 4. Το Κεφάλαιο 5 παρουσιάζει τη μεθοδολογία παραλληλοποίησης του χωρικού ερωτήματος ΟκΠΓ σε κάρτα γραφικών με τη χρήση της βιβλιοθήκης Thrust. Ακολουθεί το Κεφάλαιο 6 στο οποίο παρατίθενται η πειραματική αξιολόγιση των παράλληλων αλγορίθμων και η σύγκριση αυτών με άλλες παράλληλες υλοποιήσεις σε επεξεργαστή και κάρτα γραφικών. Το Κεφάλαιο 7 αναλύει τα προγραμματιστικά εργαλεία και πλατφόρμες που χρησιμοποιήθηκαν και συμπεριλαμβάνει οδηγίες εγκατάστασης και εκτέλεσης των αλγορίθμων. Τέλος, στο Κεφάλαιο 8, γίνεται αναφορά τα αποτελέσματα και συμπεράσματα της μελέτης και προτείνονται μελλοντικές επεκτάσεις για μείωση χρονικών υπολογισμών και επιτάχυνση της χρονικής απόκρισης. Στο παράρτημα Ι απαρτίζεται απο τους ψευδοκώδικες των βασικών σειριακών αλγορίθμων.

Κεφάλαιο 2

Συγγενικές εργασίες

2.1 Εισαγωγή

Την τελευταια δεκαετία παρατηρείται ραγδαία αύξηση του όγκου των δεδομένων που διατίθενται προς επεξεργασία. Τα δεδομένα αυτά έχουν συλλεχθεί από δορυφόρους, κινητικές συσκευές, συστήματα IoT κλπ. Για την επεξεργασία αυτών των δεδομένων απαιτείται συνεχώς αυξανόμενη υπολογιστική ισχύ που κατά γενική ομολογία δεν είναι διαθέσιμη σε ένα απλό υπολογιστικό σύστημα, φέροντας ως συνέπεια την καθυστέρηση της εκτέλεσης ενός χωρικού ερωτήματος. Η λύση του προβλήματος δεν είναι άλλη από την παράλληλη επεξεργασία, με λίγα λόγια την ταυτόχρονη αξιοποίηση πολλών υπολογιστικών μονάδων για την εκτέλεση του ίδιου χωρικού ερωτήματος.

Αρκετές μελέτες έχουν πραγματοποιηθεί γύρω από το πρόβλημα της παράλληλης επεξεργασίας απαιτητικών χωρικών ερωτημάτων τόσο στην ελληνική όσο και διεθνή βιβλιογραφία. Παράλληλες υλοποιήσεις χωρικών ερωτημάτων αναπτύχθηκαν σε κατανεμημένα συστήματα, πολυπύρηνους επεξεργαστές και κάρτες γραφικών καθώς και στο συνδυασμό των δύο τελευταίων. Άλλες συγγενικές εργασίες διερευνούν μεθόδους προσέγγισης απαιτητικών χωρικών ερωτημάτων αυξάνοντας την απόδοση ή και την παραλληλησιμότητά τους. Πιο συγκεκριμένα και συγγενικά με την εν λόγω εργασία έχουν γίνει έρευνες παραλληλοποίησης των ερωτημάτων kΠΓ και OkΠΓ. Ο γενικότερος κλάδος των δεδομένων και της πληροφορίας έχει κατακλύσει την επικαιρότητα και αποτελεί αντικείμενο πλήθους ερευνών που δε μπορεί παρά να υποσχεθεί ριζικές αλλαγές στο μέλλον.

2.2 Παράλληλη επεξεργασία χωρικών ερωτημάτων

Στο [14], οι Zhang et al. (2014) κάνουν χρήση πολυπύρηνων επεξεργαστών και καρτών γραφικών για την ανάλυση μεγάλου όγκου χωροχρονικών δεδομένων που περιέχουν καταγραφές διαδρομών ταξί της πόλης της Νέας Υόρκης (επιβιβάσεις/αποβιβάσεις). Για την ανάλυση των δεδομένων εκτελούνται ερωτήματα τύπου OLAP με αθροίσεις, ομαδοποιήσεις και συνενώσεις. Υλοποιούν σε C++ περιβάλλον εξειδικευμένους αλγορίθμους που χρησιμοποιούν τη βιβλιοθήκη Intel TBB και Thrust, για την παράλληλη επεξεργασία με πολυπύρηνους επεξεργαστές και την αξιοποίηση των πολλών πυρήνων των καρτών γραφικών Nvidia. Εκμεταλεύονται, επίσης, τις δυνατότη-

τες SIMD των σύγχρονων επεξεργαστών να χειρίζονται πολλές λέξεις συγχρόνως με μία εντολή. Συμπεραίνουν ότι η κατασκευή ενός συστήματος επεξεργασίας χωροχρονικών ερωτημάτων τύπου OLAP, αξιοποιώντας τους πολυπύρηνους επεξεργαστές και κάρτες γραφικών, είναι εφικτή, με χρόνο απόκρισης μερικών δευτερολέπτων για εκατοντάδες εκατομμύρια εγγραφές.

Δύο εκ των τριών ερευνητών σε επόμενη εργασία τους [15], οι Zhang και You (2014) κάνουν πειραματική σύγκριση των αλγορίθμων που ανέπτυξαν σε πολυπύρηνους επεξεργαστές, κάρτες γραφικών και συνεπεξεργαστές αρχιτεκτονικής Intel MIC, όπως ο Xeon Phi. Καταλήγουν στο ότι οι πολυπύρηνοι επεξεργαστές νέας γενιάς οι οποίοι διαθέτουν μονάδα διανυσματικής επεξεργασίας (VPU) με σετ εντολών SIMD, αποδίδουν αρκετά καλύτερα από τις πολυπύρηνες κάρτες γραφικών εφόσον χρησιμοποιηθεί αυτή τους η δυνατότητα.

Σε μεταγενέστερη εργασία (2017) [16], οι ίδιοι ερευνητές συνδυάζοντας τις παραπάνω τεχνικές αναπτύσουν ένα σύστημα Web GIS με αρχιτεκτονική client-server. Το γραφικό περιβάλλον φυλλομετρητή ιστού (web browser), χρησιμοποιώντας την υπηρεσία Google Maps, υποστηρίζει την υποβολή ad-hoc χωρικών ερωτημάτων.

Στο [9], οι Gang Mei et al. (2015) επικεντρώνονται στο σχεδιασμό και την υλοποίηση του αλγόριθμου παρεμβολής επιτάχυνσης αντίστροφης στάθμισης (AIDW) σε GPU. Ο AIDW είναι μια βελτιωμένη έκδοση του τυπικού IDW, η οποία μπορεί προσαρμοστικά να καθορίσει την παράμετρο ισχύος σύμφωνα με το μοτίβο κατανομής των χωρικών σημείων και να επιτύχει πιο ακριβείς προβλέψεις από αυτές του IDW. Υλοποιούνται δύο εκδόσεις του AIDW με επιτάχυνση GPU, η απλοϊκή (naive) έκδοση χωρίς να επωφελείτε από την κοινόχρηστη μνήμη και η έκδοση με πλακάκια (tiled) που εκμεταλλεύεται την κοινόχρηστη μνήμη. Εφαρμόζονται επίσης οι δύο εκδόσεις χρησιμοποιώντας τις διατάξεις δεδομένων, τη δομή πινάκων (SoA) και τον πίνακα ευθυγραμμισμένων δομών (AoaS), σε μονή και διπλή ακρίβεια. Αξιολογείται η απόδοση του AIDW με επιτάχυνση GPU σε σύγκριση με την αρχική του έκδοση σε CPU. Τα πειραματικά αποτελέσματα δείχνουν ότι σε μονή ακρίβεια, η απλοϊκή έκδοση και η έκδοση με πλακάκια μπορούν να επιτύχουν πολύ μεγάλες ταχύτητες και ότι οι υλοποιήσεις που χρησιμοποιούν τη διάταξη SoA είναι πάντα ελαφρώς ταχύτερες από αυτές που χρησιμοποιούν τη διάταξη AoaS. Ωστόσο, με διπλή ακρίβεια, η επιτάχυνση είναι πολύ μικρότερη. Καταλήγουν επίσης στο ότι δεν έχουν ληφθεί κέρδη απόδοσης από την έκδοση με πλακάκια έναντι της αφελής έκδοσης. και ότι η χρήση των SoA και AoaS δεν οδηγεί σε σημαντικές διαφορές στην υπολογιστική απόδοση.

Στο [11], οι Ogden et al. (2016) δημιουργούν μία σύνθετη υπολογιστική τεχνική την οποία αποκαλούν "προσεταιριστικό μορφοτροπέα" (asociative transducer) , η ανάπτυξη της οποίας στηρίχθηκε στη θεωρία μηχανών πεπερασμένων καταστάσεων. Η συγκεκριμένη τεχνική αξιοποιήθηκε απο τους ερευνητές προκειμένου να μετασχηματίσουν ένα χωρικό ερώτημα από την αρχική SQL-GIS μορφή του σε επιμέρους στάδια επεξεργασίας, που δύνανται να εκτελούνται παράλληλα για διαφορετικά μέρη των δεδομένων εισόδου. Τα διάφορα στάδια επεξεργασίας είναι κώδικες σε γλώσσα προγραμματισμού C++, που προκύπτουν δυναμικά από πρότυπα κλάσεων (C++ templates) και μεταγλωττίζονται σε εκτελέσιμα προγράμματα κατά την διάρκεια εκτέλεσης του χωρικού ερωτήματος. Το συγκεκριμένο σύστημα εκτελείται σε απλό υπολογιστικό σύστημα με πολυπύρηνο επεξεργαστή επιτυγχάνοντας υψηλότερες αποδόσεις σε σύγκριση με άλλα κατανεμημένα συστήματα που αφορούν γεωγραφικά σύνολα δεδομένων μεγέθους λιγότερων των 200

εκατομμυρίων στοιχείων.

Οι Roumelis et al. (2016), στο [12], ερευνούν την εφαρμογή διαφόρων παραλλαγών του αλγορίθμου Σάρωση Επιπέδου (Plane Sweep) για την αναθεώρηση μιας συγκεκριμένης κατηγορίας χωρικών ερωτημάτων χωρίς την χρήση ευρετηρίου (index) για τα δεδομένα εισόδου. Στην έρευνά τους δεν κάνουν χρήση παράλληλης επεξεργασίας, αντ' αυτού κατανέμουν τον χώρο σε λωρίδες, το οποιο πυροδοτεί την ανάπτυξη παράλληλων αλγορίθμων. Τα χωρικά ερωτήματα που διερευνώνται ποικίλουν από το ΟκΠΓ, ωστόσο το κοινό τους γνώρισμα είναι η δυνατότητα υπολογισμού των ευκλείδιων αποστάσεων μεταξύ των σημείων. Συμπερασματικά, τα ευρήματα της έρευνας αποδεικνύουν ότι με την βέλτιστη διαίρεση του χώρου σε λωρίδες οι διάφορες εκδοχές του αλγορίθμου Plane Sweep μπορούν να επιτύχουν καλύτερες αποδόσεις σε σχέση με τους αλγορίθμους που κάνουν χρήση διαφόρων ευρετηρίων (index).

Στο [7], οι Du et al. (2017) δημιουργούν έναν αλγόριθμο για συνένωση πολλαπλών εισόδων (multiway join) μεταξύ συνόλων γεωγραφικών δεδομένων, ο οποίος εφαρμόζεται σε κατανεμημένο σύστημα που βασίζεται στο Apache Spark. Αξιοποιούν ένα ομοιόμορφο πλέγμα για να κατακερματίσουν το χώρο, πράττοντας διαδοχικές συνενώσεις συνόλων με τη χρήση ευρετηρίων R*-tree. Αξιοποιώντας τις δυνατότητες του Spark, καταφέρνουν να μειώσουν σε σημαντικό βαθμό την ανάγκη πρόσβασης σε δίκτυο ή δίσκο. Με την έρευνα τους, αποδεικνύουν ότι το σύστημα που ανέπτυξαν διαθέτει υψηλότερες αποδόσεις από αντίστοιχα συστήματα που βασίζονται στο MapReduce αλλά και γραμμική εξάρτηση από το μέγεθος των δεδομένων εισόδου.

Στο [10], οι Moutafis et al. (2017) κάνουν μελέτη του ερωτήματος AkNN σε κατανεμημένο σύστημα σε περιβάλλον Apache Hadoop. Με τη χρήση δομών τύπου πλέγματος και τετραδικού δένδρου (quad-tree) για να ταξινομούν τα σημειοσύνολα και εφαρμόζουν τον αλγόριθμο Plane Sweep για τον περιορισμό του αριθμού των υποψήφιων προς έλεγχο γειτόνων. Αναπτύσσονται τεχνικές περιορισμού του όγκου δεδομένων που ανταλλάσσεται μεταξύ των κόμβων του κατανεμημένου συστήματος. Καταλήγουν στο συμπέρασμα ότι η επίδραση του αλγορίθμου Plane Sweep σε συνδυασμό με τα quad-trees επιταχύνουν την εκτέλεση του ερωτήματος. Επιπλέον παρατηρούν αυξάνοντας τον αριθμό των κόμβων του κατανεμημένου συστήματος η επίδραση στο χρόνο εκτέλεσης του αλγορίθμου δεν είναι γραμμική, δηλ. μετά από κάποιο αριθμό κόμβων η βελτίωση της ταχύτητας εκτέλεσης του ερωτήματος που επυτιγχάνεται δεν είναι σημαντική.

Στο [17] οι Χριστοφής και Βασιλακόπουλος (2018) κάνουν μελέτη του προβλήματος εύρεσης "Όλων των k πλησιέστερων γειτόνων" για δυσδιάστατα σύνολα δεδομένων. Αναπτύσσουν σε προγραμματιστικό περιβάλλον C++ με τη χρήση των μεθόδων παραλληλισμού OpenMP και Intel TBB, παράλληλες υλοποιήσεις των αλγορίθμων Brute Force, Plane Sweep και Plane Sweep με λωρίδες. Την καλύτερη απόκριση δίνει ο Plane Sweep με λωρίδες, ο οποίος με την κατάλληλη επιλογή του πλήθους λωρίδων παρουσιάζει σχεδόν γραμμική πολυπλοκότητα. Ο τελευταίος αλγόριθμος προσαρμόστηκε ώστε σύνολα δεδομένων που δεν χωράνε στην κύρια μνήμη εξ'ολοκλήρου, να επεξεργάζονται με τμηματική ανάγνωση και εγγραφή στο δίσκο μέσω της βιβλιοθήκης STXXL. Συμπερεσματικά, ο αλγόριθμος εξωτερικής μνήμης διατηρεί τη γραμμική πολυπλοκότητα παραμένοντας, δηλαδή, ανεπηρέαστος απ'τη διαθέσιμη ενδιάμεση μνήμη.

Στο [13], οι Velentzas et al. προτείνουν κι εφαρμόζουν δύο αλγόριθμους σε κάρτες γραφικών για το ερώτημα "k Πλησιέστεροι Γείτονες" χρησιμοποιώντας το CUDA API και τη βιβλιοθήκη

Thrust. Ο πρώτος βασίζεται σε μια προσέγγιση ωμής βίας και ο δεύτερος χρησιμοποιεί ευρετήρια για να ελαχιστοποιήσει τα σημεία αναφοράς κοντά σε ένα σημείο ερωτήματος. Κάνουν σύγκριση του αλγορίθμου τους με έξι υπάρχοντες αλγορίθμους ωμής βίας πάνω σε πειράματα τύπου ΟkΠΓ. Καταλήγουν στο ότι οι δύο αλγόριθμοι υπερτερούν όσον αφορά το χρόνο εκτέλεσης καθώς και τον συνολικό όγκο των σημείων αναφοράς στη μνήμη που μπορούν να επεξεργαστούν.

Στο [8], οι Shenshen Liang et al. παρουσιάζουν μια παράλληλη υλοποίηση, με βάση το CUDA API, του "k Πλησιέστεροι Γείτονες" ερωτήματος, με την ονομασία CUKNN, χρησιμοποιώντας το μοντέλο πολλαπλών νημάτων CUDA, όπου τα στοιχεία δεδομένων υποβάλλονται σε επεξεργασία με παράλληλο τρόπο δεδομένων. Χρησιμοποιούνται διάφορες τεχνικές βελτιστοποίησης CUDA για τη μεγιστοποίηση της χρήσης της GPU. Το CUKNN ξεπερνά σημαντικά το σειριακό kNN συμπεριλαμβανομένου του χρόνου εισόδου/εξόδου. Παρουσιάζει επίσης καλή επεκτασιμότητα κατά τη μεταβολή της διάστασης του συνόλου δεδομένων αναφοράς, τον αριθμό των εγγραφών στο σύνολο δεδομένων αναφοράς και τον αριθμό των εγγραφών στο σύνολο δεδομένων ερωτήματος.

Κεφάλαιο 3

Θεωρητικό υπόβαθρο

3.1 Εισαγωγή

Το ερώτημα εύρεσης "Όλων των k-πλησιέστερων γειτόνων" χρησιμοποιεί τους ακόλουθους ορισμούς που αναφέρονται στη βιβλιογραφία [10], από τους Moutafis, Mavrommatis, Vassilakopoulos και Sioutas (2017). Στη συνέχεια ακολουθεί ανάλυση των μονάδων επεξεργασίας γραφικών του κατασκευαστή Nvidia.

3.2 Ορισμοί

Ευκλείδεια Απόσταση

Δεδομένων δύο σημείων p(x1, y1) και q(x2, y2), η Ευκλείδεια απόστασή τους ορίζεται ως

$$dist(p,q) = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$$

k Πλησιέστεροι Γείτονες

Δεδομένων ενός σημείου p, ενός συνόλου σημείων S και ενός ακεραίου k, οι k πλησιέστεροι γείτονες του p από το S είναι ένα σύνολο σημείων kΠΓ(p, S) τέτοιο ώστε

$$\forall r \in k(p,S), \forall q \in S - k(p,S), dist(p,r) \leq dist(p,q)$$

Όλοι οι k Πλησιέστεροι Γείτονες

Δεδομένων δύο συνόλων σημείων R, S και ενός ακεραίου k, όλοι οι k πλησιέστεροι γείτονες του R από το S είναι ένα σύνολο ζευγών OkΠΓQ(R, S) που ορίζεται ως

$$(r,s) : r \in R, s \in k(r,S)$$

Συνήθως γίνεται αναφορά στο R ως σύνολο input και στο S ως σύνολο training.

3.3 Μονάδες Επεξεργασίας γραφικών Nvidia

Μια μονάδα επεξεργασίας γραφικών (GPU) είναι κυρίως γνωστή ως η συσκευή υλικού που χρησιμοποιείται κατά την εκτέλεση εφαρμογών που δίνουν βαρύτητα στα γραφικά, όπως λογισμικά μοντελοποίησης 3D. Στην καταναλωτική αγορά, μια GPU χρησιμοποιείται κυρίως για την επιτάχυνση των γραφικών παιχνιδιών. Σήμερα, η γενικής χρήσης GPU (General Purpose GPU) είναι η επιλογή υλικού για την επιτάχυνση του υπολογιστικού φόρτου εργασίας σε σύγχρονα τοπία υψηλής απόδοσης υπολογιστών (High-Performance Computing).

Μία κάρτα γραφικών παρέχει πολύ υψηλότερη ροή εντολών (instruction throughput) και ταχύτητα μετάδοσης δεδομένων από τη μνήμη (memory bandwidth) από την CPU σε παρόμοια τιμή και κατανάλωση ισχύος. Πολλές εφαρμογές αξιοποιούν αυτές τις υψηλότερες δυνατότητες για να τρέχουν γρηγορότερα στην GPU από ότι στην CPU. Αυτή η διαφορά στις δυνατότητες μεταξύ της GPU και της CPU υπάρχει επειδή έχουν σχεδιαστεί με διαφορετικούς στόχους κατά νου. Ενώ η CPU έχει σχεδιαστεί για να υπερέχει στην εκτέλεση μιας ακολουθίας λειτουργιών, που ονομάζεται νήμα (thread), το συντομότερο δυνατό και μπορεί να εκτελέσει παράλληλα μερικές δεκάδες από αυτά τα νήματα, η GPU έχει σχεδιαστεί για να υπερέχει στην εκτέλεση χιλιάδων ακολουθιών παράλληλα.

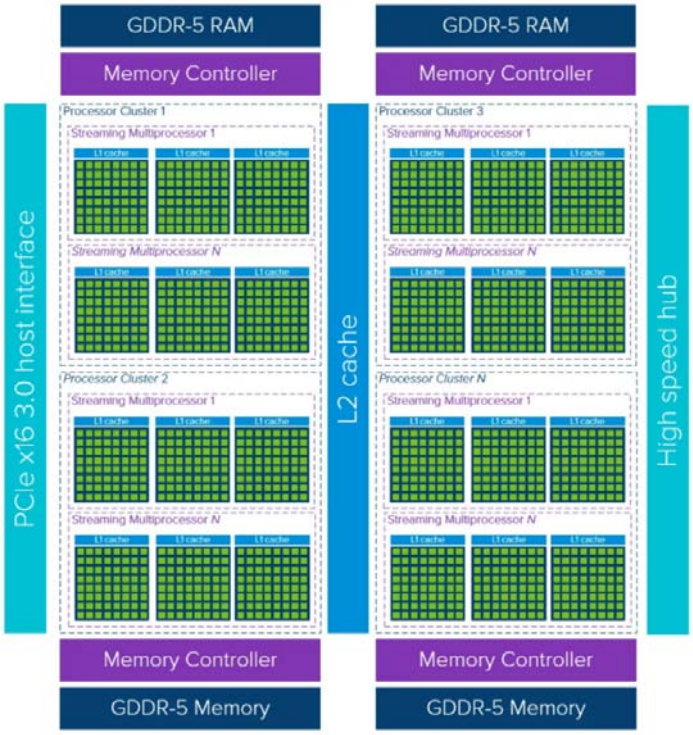
Ως εκ τούτου έχει κατασκευαστεί έτσι ώστε περισσότερα τρανζίστορ να αφιερώνονται στην επεξεργασία δεδομένων και όχι στην αποθήκευση δεδομένων και στον έλεγχο ροής. Η αφιέρωση περισσότερων τρανζίστορ στην επεξεργασία δεδομένων, όπως για παράδειγμα υπολογισμοί κινητής υποδιαστολής, είναι επωφελής για εξαιρετικά παράλληλους υπολογισμούς. Η GPU μπορεί να καλύψει τους χρόνους καθυστέρησης πρόσβασης στη μνήμη με υπολογισμούς, αντί να βασίζεται σε μεγάλες κρυφές μνήμες δεδομένων (cache memories) και πολύπλοκο έλεγχο ροής για να αποφευχθούν οι καθυστερήσεις αυτες, και τα δύο εκ των οποίων είναι ακριβά όσον αφορά τα τρανζίστορ.

Σε γενικές γραμμές, μια εφαρμογή έχει ένα συνδυασμό παράλληλων μερών και διαδοχικών τμημάτων, έτσι τα συστήματα έχουν σχεδιαστεί με ένα συνδυασμό GPU και CPU με σκοπό τη μεγιστοποίηση της συνολικής απόδοσης. Εφαρμογές με υψηλό βαθμό παραλληλισμού μπορούν να εκμεταλλευτούν αυτόν τον μαζικά παράλληλο χαρακτήρα της GPU για να επιτύχουν υψηλότερη απόδοση από ότι στην CPU.

3.3.1 Αρχιτεκτονική GPU

Η επισκόπηση υψηλού επιπέδου της αρχιτεκτονικής μιας GPU (η οποία εξαρτάται σε μεγάλο βαθμό από το μοντέλο), καθιστά εμφανές ότι η φύση μιας GPU έχει να κάνει με τη λειτουργία των διαθέσιμων πυρήνων και λιγότερο επικεντρώνεται στην πρόσβαση στη κρυφή μνήμη (cache) χαμηλής απόκρισης.

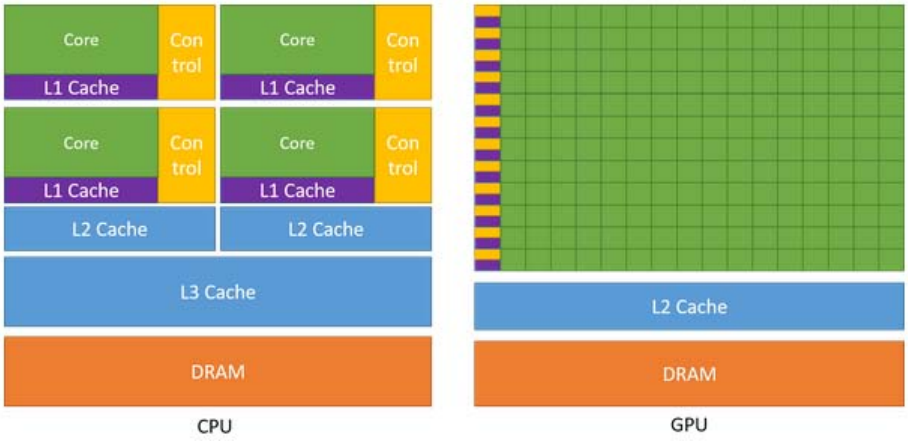
Μια μεμονωμένη συσκευή GPU αποτελείται από πολλαπλές συστάδες επεξεργαστών (Processor Clusters) που περιέχουν πολλούς πολυεπεξεργαστές ροής (Streaming Multiprocessors). Κάθε SM φιλοξενεί επίπεδο cache layer1 με τους αντίστοιχους πυρήνες του. Συνήθως, ένας SM χρησιμοποιεί μια αποκλειστική cache layer1 και μια κοινόχρηστη cache layer2 πριν αντλήσει δεδομένα από την παγκόσμια μνήμη GDDR-5. Η αρχιτεκτονική του είναι ανεκτική στην απόκριση μνήμης.



Σχήμα 3.1: Αρχιτεκτονική GPU (πηγή [3])

3.3.2 Σύγκριση GPU και CPU

Ο αριθμός των πυρήνων αποκαλύπτει τις δυνατότητες παραλληλισμού. Οι εργασίες δεν είναι προγραμματισμένες σε μεμονωμένους πυρήνες, αλλά σε ομάδες επεξεργαστών και SM.

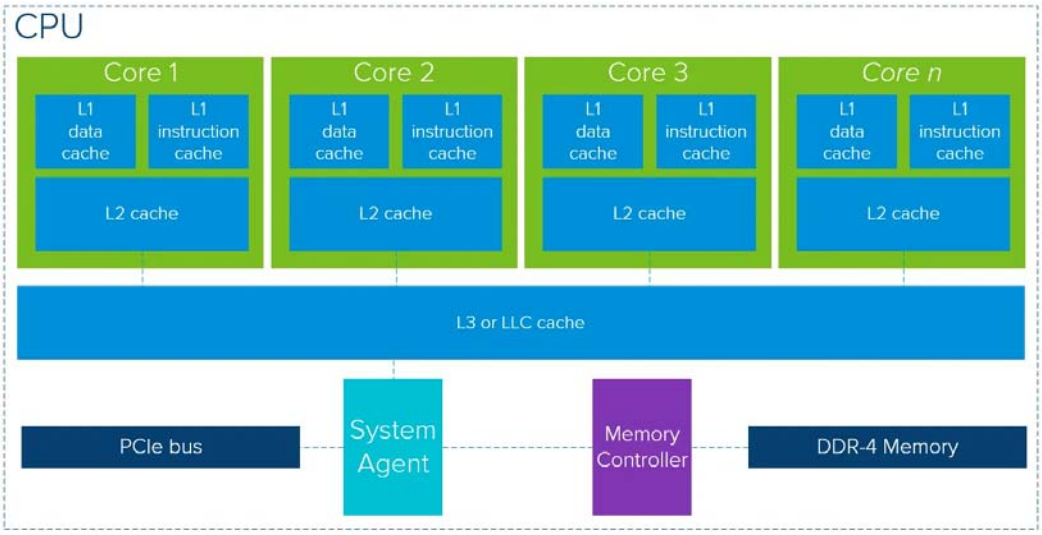


Σχήμα 3.2: Πλήθος πυρήνων CPU και GPU (πηγή [2])

Ωστόσο, δεν αφορά μόνο τον αριθμό των πυρήνων. Οι πυρήνες GPU NVIDIA είναι πυρήνες CUDA που αποτελούνται από ALU (Arithmetic Logic Unit).

Υπάρχουν πολλές ομοιότητες της συνολικής αρχιτεκτονικής μιας CPU και GPU. Και οι δύο χρησιμοποιούν τις δομές μνήμης των επιπέδων μνήμης cache, του ελεγκτή μνήμης και της κα-

θολικής μνήμης. Μια επισκόπηση υψηλού επιπέδου των σύγχρονων αρχιτεκτονικών CPU δείχνει ότι έχει να κάνει με τη χαμηλή απόκριση πρόσβασης στη μνήμη χρησιμοποιώντας σημαντικά τα επίπεδα μνήμης cache.



Σχήμα 3.3: Αρχιτεκτονική CPU (πηγή [3])

Ένα ενιαίο πακέτο CPU αποτελείται από πυρήνες που περιέχουν ξεχωριστά δεδομένα και οδηγίες cache layer-1, που υποστηρίζονται από την cache layer-2. Η προσωρινή μνήμη layer-3 ή cache τελευταίου επιπέδου μοιράζεται σε πολλούς πυρήνες. Εάν τα δεδομένα δεν βρίσκονται στα επίπεδα της μνήμης cache, θα ανακτήσει τα δεδομένα από την καθολική μνήμη DDR-4.

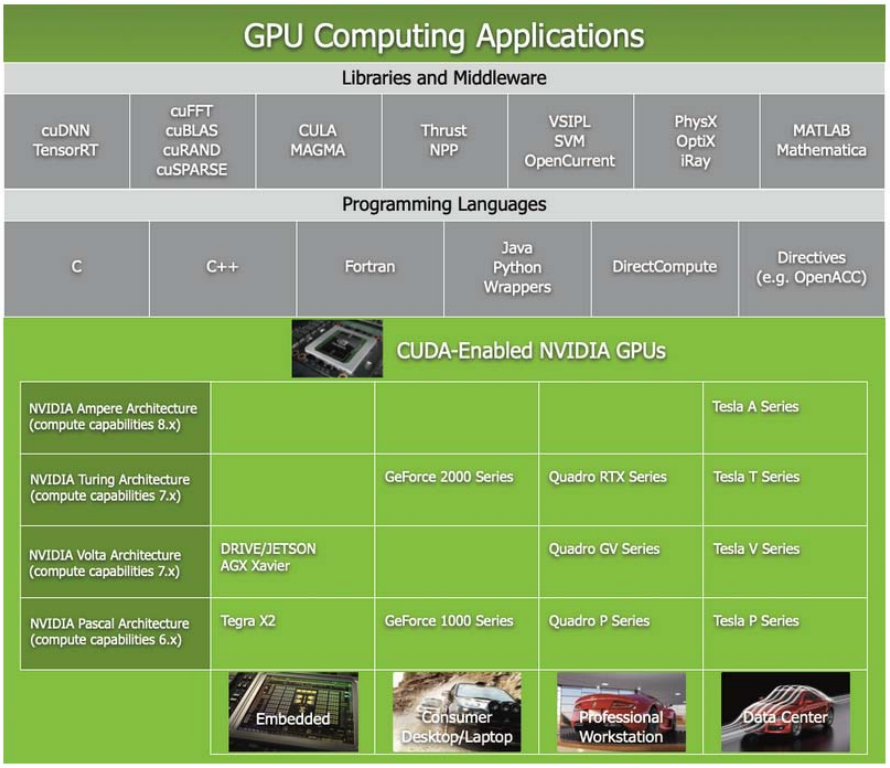
3.3.3 Προγραμματισμός σε GPU

Τον Νοέμβριο του 2006, η NVIDIA παρουσίασε το CUDA®, μια πλατφόρμα παράλληλου υπολογισού γενικής χρήσης και μοντέλο προγραμματισμού που αξιοποιεί τον παράλληλο υπολογιστικό κινητήρα σε GPU NVIDIA για την επίλυση πλήθους πολύπλοκων υπολογιστικών προβλημάτων με πιο αποτελεσματικό τρόπο από ότι σε έναν CPU. Στο Σχήμα 3.4 παρουσίαζονται εφαρμογές της GPU καθώς και διάφορες γλώσσες και διεπαφές προγραμματισμού εφαρμογών που το CUDA έχει σχεδιαστεί να υποστηρίζει.

Πυρήνες

Το CUDA C++ επεκτείνει το C++ επιτρέποντας στον προγραμματιστή να ορίσει συναρτήσεις C++, που ονομάζονται πυρήνες (kernels), οι οποίες, όταν καλούνται, εκτελούνται N φορές παράλληλα από N διαφορετικά νήματα CUDA, σε αντίθεση με μία μόνο φορά όπως οι κανονικές λειτουργίες C++.

Ο πυρήνας ορίζεται χρησιμοποιώντας τον προσδιοριστή δήλωσης __global__ και ο αριθμός των νημάτων CUDA που εκτελούν αυτόν τον πυρήνα για μια δεδομένη κλήση πυρήνα καθορίζεται χρησιμοποιώντας μια νέα σύνταξη διαμόρφωσης $\lll \cdots \ggg$ Σε κάθε νήμα που εκτελεί τον πυρήνα δίνεται ένα μοναδικό αναγνωριστικό νήματος που είναι προσβάσιμο εντός του πυρήνα μέσω



Σχήμα 3.4: Εφαρμογές της Nvidia GPU (πηγή [2])

ενσωματωμένων μεταβλητών.

Ως παράδειγμα, ο ακόλουθος ενδεικτικός κώδικας, χρησιμοποιεί την ενσωματωμένη μεταβλητή threadIdx, προσθέτει δύο διανύσματα A και B μεγέθους N και αποθηκεύει το αποτέλεσμα στο διάνυσμα C:

```
// Kernel definition
__global__ void VecAdd(float* A, float* B, float* C)
{
    int i = threadIdx.x;
    C[i] = A[i] +B[i];
}

int main()
{
    ...
    // Kernel invocation with N threads
    VecAdd<<<1, N>>>(A, B, C);
    ...
}
```

Εδώ, κάθε ένα από τα N νήματα που εκτελεί το VecAdd() εκτελεί μια προσθήκη ανά ζεύγος.

Ιεραρχία των Νημάτων

Για ευκολία, το threadIdx είναι ένας φορέας τριών συστατικών, έτσι ώστε τα νήματα να μπορούν να ταυτοποιηθούν χρησιμοποιώντας έναν μονοδιάστατο, δισδιάστατο ή τρισδιάστατο δείκτη νημάτων, σχηματίζοντας ένα μονοδιάστατο, δισδιάστατο ή τρισδιάστατο μπλοκ από νήματα(block of threads).

Το ευρετήριο ενός νήματος και το αναγνωριστικό του νήματος σχετίζονται μεταξύ τους με έναν απλό τρόπο: Για ένα μονοδιάστατο μπλοκ, είναι το ίδιο. για ένα δισδιάστατο μπλοκ μεγέθους (Dx, Dy), το ID νημάτων ενός νήματος του ευρετηρίου (x, y) είναι (x + y Dx). για ένα τρισδιάστατο μπλοκ μεγέθους (Dx, Dy, Dz), το ID νημάτων ενός νήματος του ευρετηρίου (x, y, z) είναι (x + y Dx + z Dx Dy).

Για παράδειγμα, ο ακόλουθος κώδικας προσθέτει δύο πίνακες A και B μεγέθους NxN και αποθηκεύει το αποτέλεσμα στον πίνακα C:

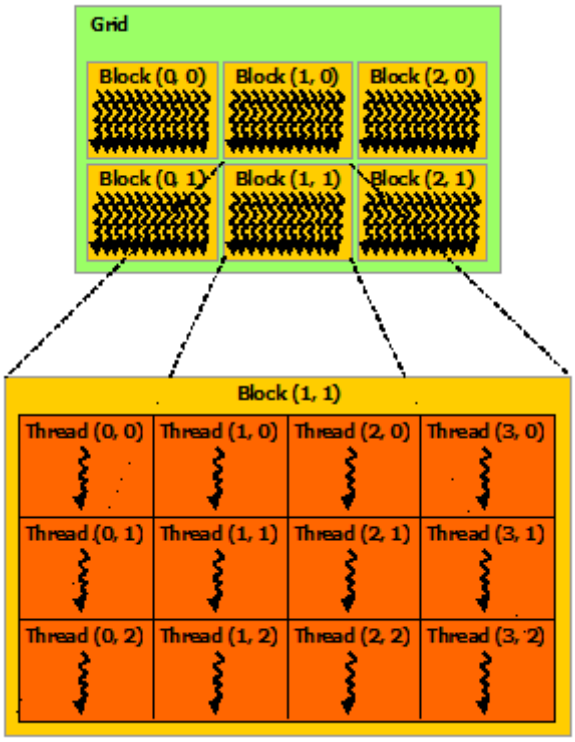
```
// Kernel definition
__global__ void MatAdd(float* A[N][N], float* B[N][N],
                       float* C[N][N])
{
    int i = threadIdx.x;
    int j = threadIdx.y;
    C[i][j] = A[i][j] +B[i][j];
}

int main()
{
    ...
    // Kernel invocation with one block of N * N * 1 threads
    int numBlocks = 1;
    dim3 threadPerBlock(N, N);
    MatAdd<<<numBlocks, threadPerBlock>>>(A, B, C);
    ...
}
```

Υπάρχει ένα όριο στον αριθμό νημάτων ανά μπλοκ, καθώς όλα τα νήματα ενός μπλοκ αναμένεται να βρίσκονται στον ίδιο πυρήνα επεξεργαστή και πρέπει να μοιράζονται τους περιορισμένους πόρους μνήμης αυτού του πυρήνα.

Ωστόσο, ένας πυρήνας μπορεί να εκτελεστεί από πολλαπλά μπλοκ νημάτων ίσου σχήματος, έτσι ώστε ο συνολικός αριθμός νημάτων να είναι ίσος με τον αριθμό νημάτων ανά μπλοκ επί τον αριθμό μπλοκ.

Τα μπλοκ είναι οργανωμένα σε μονοδιάστατο, δισδιάστατο ή τρισδιάστατο πλέγμα σπειρωμάτων όπως φαίνεται στο Σχήμα 3.5. Ο αριθμός μπλοκ νήματος σε ένα πλέγμα υπαγορεύεται συνήθως από το μέγεθος των δεδομένων που υποβάλλονται σε επεξεργασία, το οποίο συνήθως υπερβαίνει τον αριθμό των επεξεργαστών στο σύστημα.



Σχήμα 3.5: Πλέγμα μπλοκ νημάτων (πηγή [2])

Ο αριθμός νημάτων ανά μπλοκ και ο αριθμός μπλοκ ανά πλέγμα μπορεί να είναι τύπου int ή dim3 και καθορίζεται στη σύνταξη $\lll \cdots \ggg$. Τα δισδιάστατα μπλοκ ή πλέγματα μπορούν να καθοριστούν όπως στο παραπάνω παράδειγμα.

Κάθε μπλοκ εντός του πλέγματος μπορεί να αναγνωριστεί από ένα μονοδιάστατο, δισδιάστατο ή τρισδιάστατο μοναδικό ευρετήριο προσβάσιμο εντός του πυρήνα μέσω της ενσωματωμένης μεταβλητής blockIdx. Η διάσταση του μπλοκ νημάτων είναι προσβάσιμη μέσα στον πυρήνα μέσω της ενσωματωμένης μεταβλητής blockDim.

Επεκτείνοντας το προηγούμενο παράδειγμα MatAdd () για τον χειρισμό πολλών μπλοκ, ο κώδικας γίνεται ως εξής.

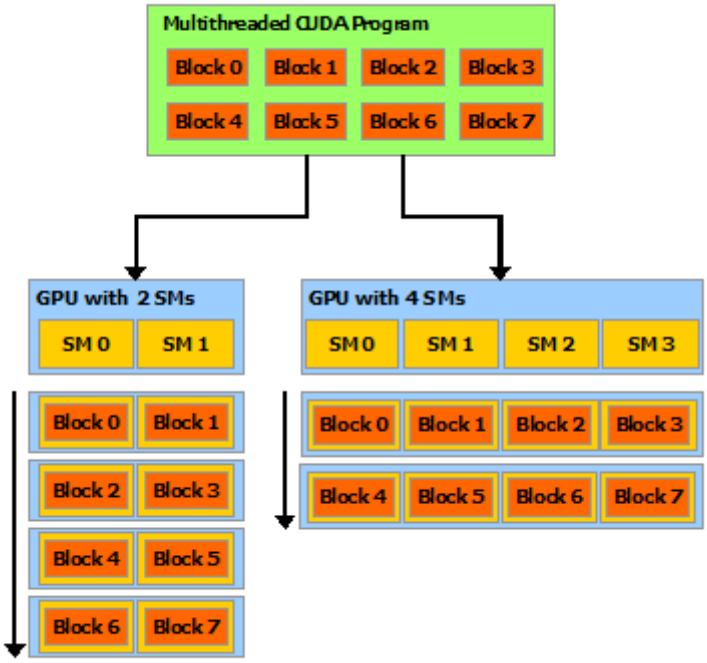
```
// Kernel definition
__global__ void MatAdd(float* A[N][N], float* B[N][N],
                       float* C[N][N])
{
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    int j = blockIdx.y * blockDim.y + threadIdx.y;
    if (i < N && j < N)
        C[i][j] = A[i][j] +B[i][j];
}

int main()
{
    ...
```

```
    // Kernel invocation
    dim3 threadPerBlock(16, 16);
    dim3 numBlocks(N/threadsPerBlock.x, N/threadsPerBlock.y);
    MatAdd<<<numBlocks, threadsPerBlock>>>(A, B, C);
    ...
}
```

Ένα μέγεθος μπλοκ νήματος 16x16 (256 νήματα), αν και αυθαίρετο σε αυτήν την περίπτωση, είναι μια κοινή επιλογή. Το πλέγμα δημιουργείται με αρκετά μπλοκ για να έχει ένα νήμα ανά στοιχείο πίνακα όπως προηγουμένως. Για απλότητα, αυτό το παράδειγμα προϋποθέτει ότι ο αριθμός νημάτων ανά πλέγμα σε κάθε διάσταση διαιρείται ομοιόμορφα από τον αριθμό νημάτων ανά μπλοκ σε αυτήν την διάσταση, αν και αυτό δεν χρειάζεται να ισχύει.

Απαιτούνται μπλοκ νήματος για ανεξάρτητη εκτέλεση: Πρέπει να είναι δυνατή η εκτέλεση τους με οποιαδήποτε σειρά, παράλληλα ή σε σειρά. Αυτή η απαίτηση ανεξαρτησίας επιτρέπει στο μπλοκ νημάτων να προγραμματίζεται με οποιαδήποτε σειρά σε οποιονδήποτε αριθμό πυρήνων, όπως απεικονίζεται στο Σχήμα 3.6, επιτρέποντας στους προγραμματιστές να γράφουν κώδικα που κλιμακώνεται με τον αριθμό των πυρήνων.



Σχήμα 3.6: Αυτόματη Επεκτασιμότητα (πηγή [2])

Τα νήματα μέσα σε ένα μπλοκ μπορούν να συνεργαστούν μοιράζοντας δεδομένα μέσω κάποιας κοινόχρηστης μνήμης και συγχρονίζοντας την εκτέλεσή τους για το συντονισμό των προσβάσεων στη μνήμη. Πιο συγκεκριμένα, μπορεί κανείς να καθορίσει σημεία συγχρονισμού στον πυρήνα καλώντας την εγγενή συνάρτηση __syncthreads () που ενεργεί ως φράγμα στο οποίο όλα τα νήματα στο μπλοκ πρέπει να περιμένουν προτού επιτραπεί να προχωρήσει. Η Κοινή Μνήμη δίνει ένα

παράδειγμα χρήσης κοινόχρηστης μνήμης.

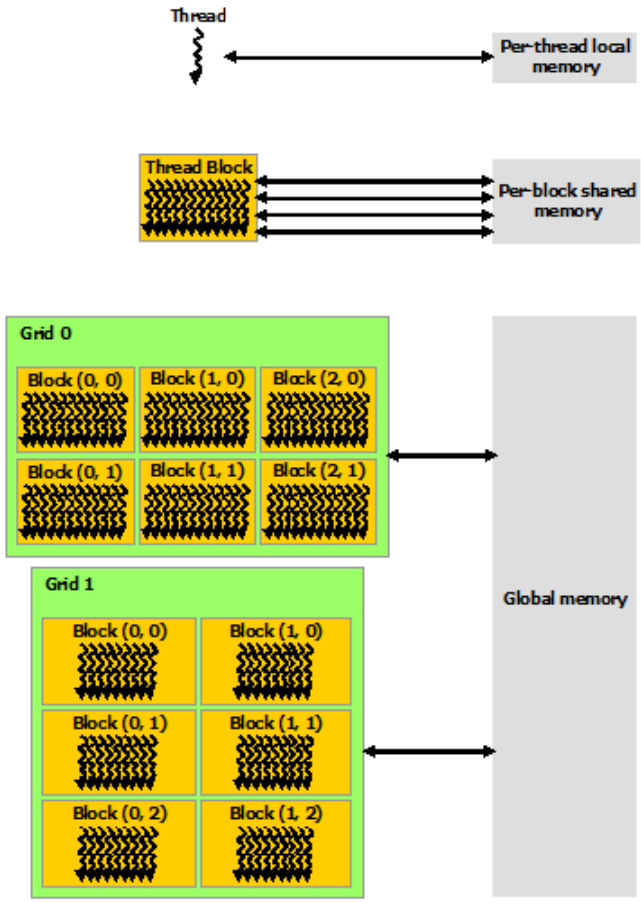
Για αποτελεσματική συνεργασία, η κοινόχρηστη μνήμη αναμένεται να είναι μια μνήμη χαμηλού χρόνου απόκρισης κοντά σε κάθε πυρήνα επεξεργαστή (σαν L1 cache).

Ιεραρχία της μνήμης

Τα νήματα CUDA ενδέχεται να έχουν πρόσβαση σε δεδομένα από πολλούς χώρους μνήμης κατά την εκτέλεση τους, όπως φαίνεται στο Σχήμα 3.7. Κάθε νήμα έχει ιδιωτική τοπική μνήμη. Κάθε μπλοκ νημάτων έχει κοινή μνήμη ορατή σε όλα τα νήματα του μπλοκ και με την ίδια διάρκεια ζωής με το μπλοκ. Όλα τα νήματα έχουν πρόσβαση στην ίδια παγκόσμια μνήμη.

Υπάρχουν επίσης δύο επιπλέον χώροι μνήμης μόνο για ανάγνωση προσβάσιμοι από όλα τα νήματα: οι χώροι μνήμης σταθερού (constant) και υφής (texture). Οι καθολικοί χώροι μνήμης, οι σταθεροί και οι υφής βελτιστοποιούνται για διαφορετικές χρήσεις μνήμης. Η μνήμη υφής προσφέρει επίσης διαφορετικούς τρόπους διευθύνσεων, καθώς και φιλτράρισμα δεδομένων, για ορισμένες συγκεκριμένες μορφές δεδομένων.

Οι καθολικοί, σταθεροί και χώροι μνήμης υφής παραμένουν σε όλες τις εκκινήσεις του πυρήνα από την ίδια εφαρμογή.



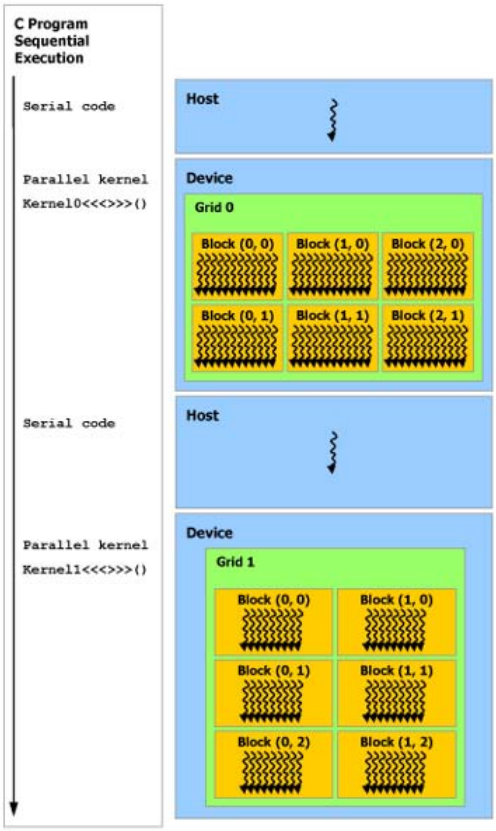
Σχήμα 3.7: Ιεραρχία μνήμης (πηγή [2])

Ετερογεννής Προγραμματισμός

Όπως απεικονίζεται στο Σχήμα 3.8, το μοντέλο προγραμματισμού CUDA υποθέτει ότι τα νήματα CUDA εκτελούνται σε μια φυσικώς ξεχωριστή συσκευή που λειτουργεί ως συνεπεξεργαστής στον κεντρικό υπολογιστή που εκτελεί το πρόγραμμα C++. Αυτό συμβαίνει, για παράδειγμα, όταν οι πυρήνες εκτελούνται σε GPU και το υπόλοιπο πρόγραμμα C++ εκτελείται σε CPU.

Το μοντέλο προγραμματισμού CUDA προϋποθέτει επίσης ότι τόσο ο κεντρικός υπολογιστής όσο και η συσκευή διατηρούν τους δικούς τους ξεχωριστούς χώρους μνήμης στη DRAM, που αναφέρονται ως μνήμη κεντρικού υπολογιστή και μνήμη συσκευής, αντίστοιχα. Επομένως, ένα πρόγραμμα διαχειρίζεται τους χώρους καθολικής, σταθερής και υφής μνήμης που είναι ορατοί στους πυρήνες μέσω κλήσεων προς το χρόνο εκτέλεσης CUDA. Αυτό περιλαμβάνει την κατανομή και την αφαίρεση της μνήμης της συσκευής καθώς και τη μεταφορά δεδομένων μεταξύ της μνήμης του κεντρικού υπολογιστή και της συσκευής.

Η ενοποιημένη μνήμη παρέχει διαχειριζόμενη μνήμη για τη γεφύρωση των κεντρικών χώρων και των χώρων μνήμης της συσκευής. Η διαχειριζόμενη μνήμη είναι προσβάσιμη από όλες τις CPU και GPU του συστήματος ως μία, συνεκτική εικόνα μνήμης με κοινό χώρο διευθύνσεων. Αυτή η δυνατότητα επιτρέπει την υπερβολική εγγραφή της μνήμης της συσκευής και μπορεί να απλοποιήσει σε μεγάλο βαθμό το έργο μεταφοράς εφαρμογών, εξαλείφοντας την ανάγκη να αντικατοπτρίζονται ρητά τα δεδομένα στον κεντρικό υπολογιστή και τη συσκευή.



Σχήμα 3.8: Ετερογεννής Προγραμματισμός (πηγή [2])

Σημείωση: Ο σειριακός κώδικας εκτελείται στον κεντρικό υπολογιστή ενώ παράλληλος κώδικας εκτελείται στη συσκευή.

3.3.4 Παραδείγματα Παραλληλοποίησης με Thrust

Η Thrust είναι μια βιβλιοθήκη προτύπων C++ για CUDA με βάση τη Standard Template Library (STL). Παρέχει μια πλούσια συλλογή πρωτόγονων παραλληλοποίησης δεδομένων όπως σάρωση (scan), ταξινόμηση (sort) και μείωση (reduction), τα οποία μπορούν να συντεθούν μαζί για την εφαρμογή σύνθετων αλγορίθμων με συνοπτικό, αναγνώσιμο πηγαίο κώδικα. Τα παρακάτω αποσπάσματα κώδικα έχουν υιοθετηθεί από το επίσημη ιστοσελίδα του CUDA [5].

Διανύσματα

Το Thrust παρέχει δύο διανυσματικά κοντέινερ, το host_vector και το device_vector. Όπως υποδηλώνουν τα ονόματα, το host_vector αποθηκεύεται στη μνήμη του κεντρικού υπολογιστή ενώ το device_vector ζει στη μνήμη της συσκευής GPU. Τα διανυσματικά δοχεία του Thrust είναι ακριβώς όπως το st::vector στο C++ STL. Όπως το std::vector, το host_vector και το device_vector είναι γενικά κοντέινερ (μπορούν να αποθηκεύσουν οποιονδήποτε τύπο δεδομένων) που μπορούν να αλλάξουν το μέγεθος τους δυναμικά. Ο ακόλουθος πηγαίος κώδικας απεικονίζει τη χρήση των διανυσματικών κοντέινερ του Thrust.

```
#include <thrust/host_vector.h>
#include <thrust/device_vector.h>
#include <thrust/copy.h>
#include <iostream>

int main()
{
    // initialize all ten integers of a host_vector to 1
    thrust::host_vector<int> H(10, 1);

    // allocate a device_vector with 10 integers
    thrust::device_vector<int> D(10);

    // copy all of H to the beginning of D
    thrust::copy(H.begin(), H.end(), D.begin());

    // print D
    for(int i = 0; i < D.size(); i++)
        std::cout << "D[" << i << "]=" << D[i] << std::endl;

    return 0;
}
```

Αλγόριθμοι

Το Thrust παρέχει μεγάλο αριθμό κοινών παράλληλων αλγορίθμων. Πολλοί από αυτούς τους αλγόριθμους έχουν άμεσους ανάλογους στο STL, και όταν υπάρχει μια ισοδύναμη συνάρτηση STL, επιλέγουμε το όνομα (π.χ. thrust::sort and std::sort). Όλοι οι αλγόριθμοι στη Thrust έχουν υλοποιήσεις τόσο για τον κεντρικό υπολογιστή όσο και για τη συσκευή.

Με εξαίρεση το thrust::copy, το οποίο μπορεί να αντιγράψει δεδομένα μεταξύ κεντρικού υπολογιστή και συσκευής, όλα τα επαναληπτικά ορίσματα σε έναν αλγόριθμο Thrust θα πρέπει να ζουν στο ίδιο μέρος: είτε όλα στον κεντρικό υπολογιστή είτε όλα στη συσκευή. Όταν παραβιαστεί αυτή η απαίτηση, ο μεταγλωττιστής θα παράγει ένα μήνυμα σφάλματος.

```
#include <thrust/device_vector.h>
#include <thrust/transform.h>
#include <thrust/sequence.h>
#include <thrust/copy.h>
#include <thrust/fill.h>
#include <thrust/sort.h>
#include <iostream>

int main(void)
{
    // allocate three device_vectors with 10 elements
    thrust::device_vector<int> X(10);
    thrust::device_vector<int> Y(10);

    // initialize X to 0,1,2,3, ....
    thrust::sequence(X.begin(), X.end());

    // compute Y = -X
    thrust::transform(X.begin(), X.end(), Y.begin(),
                      thrust::negate<int>());

    // fill Z with twos
    thrust::fill(X.begin(), X.end(), 2);

    // Sort the Y elements
    thrust::sort(Y.begin(), Y.end());

    // print Y
    thrust::copy(Y.begin(), Y.end(),
                 std::ostream_iterator<int>(std::cout, "\n"));

    return 0;
}
```

Επαναληπτές

Οι επαναληπτές (iterators) εκτελούν μια ποικιλία πολύτιμων σκοπών.

```
#include <thrust/iterator/zip_iterator.h>
...
// initialize vectors
thrust::device_vector<int>  A(3);
thrust::device_vector<char> B(3);
A[0] = 10;  A[1] = 20;  A[2] = 30;
B[0] = 'x'; B[1] = 'y'; B[2] = 'z';

// create iterator (type omitted)
first = thrust::make_zip_iterator
                (thrust::make_tuple(A.begin(), B.begin()));
last  = thrust::make_zip_iterator
                (thrust::make_tuple(A.end(), B.end()));

first[0]   // returns tuple(10, 'x')
first[1]   // returns tuple(20, 'y')
first[2]   // returns tuple(30, 'z')

// maximum of [first, last)
thrust::maximum< tuple<int,char> > binary_op;
thrust::tuple<int,char> init = first[0];

// returns tuple(30, 'z')
thrust::reduce(first, last, init, binary_op);
```

Κεφάλαιο 4

Ακολουθιακοί αλγόριθμοι

4.1 Εισαγωγή

Έγινε χρήση τριών βασικών αλγορίθμων που αποτέλεσαν την βάση της ανάπτυξης των παράλληλων εκδοχών τους:

- Αλγόριθμος Ωμής Βίας (Brute Force)
- Αλγόριθμος Σάρωσης Επιπέδου (Plane Sweep)
- Αλγόριθμος Σάρωσης Επιπέδου με Λωρίδες (Plane Sweep Striped)

Για τον αλγόριθμο με λωρίδες υπολογίζεται προσεγγιστικά ο βέλτιστος αριθμός λωρίδων και εξετάζονται εναλλακτικοί τρόποι διαχωρισμού σε λωρίδες.

Οι ορισμοί και οι υλοποιήσεις των παραπάνω αναφέρονται από τους Χριστοφή και Βασιλακόπουλο (2018), στο [17].

4.2 Αλγόριθμος ωμής βίας

Υπολογίζεται για κάθε σημείο του συνόλου input η τετράγωνική ευκλείδεια απόστασή του από κάθε σημείο του συνόλου training. Οι αποστάσεις αυτές αποθηκεύονται σε σωρούς μεγίστου (max heap) k θέσεων που διατηρούνται ξεχωριστά για κάθε σημείο του συνόλου input, δηλ. έχουμε τόσους σωρούς όσα και τα σημεία του input. Η προσθήκη στους σωρούς μεγίστου γίνεται ως εξής: γίνεται έλεγχος αν η κάθε απόσταση που πρέπει να προστεθεί είναι μικρότερη από την κορυφή του σωρού οπότε αφαιρούμε την κορυφή του σωρού και προσθέτουμε τη νέα απόσταση. Με κάθε νέα προσθήκη, ο σωρός αναδιατάσσεται ώστε να διατηρεί την βασική ιδιότητά του. Στο τέλος του αλγορίθμου ο κάθε σωρός περιέχει τους k πλησιέστερους γείτονες του αντίστοιχου σημείου από την μεγαλύτερη προς την μικρότερη απόσταση. Να σημειωθεί ότι δεν χρειάζεται ο υπολογισμός της τετραγωνικής ρίζας που υπάρχει στον τύπο της ευκλείδειας απόστασης γιατί πάντα συγκρίνουμε μεταξύ τους τετράγωνα αποστάσεων. Ο ψευδοκώδικας του αλγορίθμου δίδεται στο Παράρτημα Ι 2.

4.3 Αλγόριθμος σάρωσης επιπέδου

Αρχικά γίνεται ταξινόμιση των σημείων των δύο συνόλων ως προς τη διεύθυνση x και έπειτα για κάθε σημείο του συνόλου input αναζητώνται γείτονες εναλλάξ δεξιά και αριστερά κατά τον άξονα x, δηλ. γίνεται βαθμιαία απομάκρυνση από το input σημείο κατ' αυτόν τον άξονα και προς τις δύο κατευθύνσεις. Όμοια με τον brute force αλγόριθμο, Τα τετράγωνα ευκλείδειων αποστάσεων των γειτόνων αποθηκεύονται σε σωρούς μεγίστου. Η αναζήτηση γειτόνων σταματά όταν το τετράγωνο της απόστασης κατά τον x άξονα, $(\Delta x)^2$ μεταξύ input και training σημείου γίνει μεγαλύτερο από την κορυφή του σωρού αφού κάθε επόμενο σημείο θα έχει σίγουρα μεγαλύτερο Δx και $(\Delta x)^2$ συνεπώς και το τετράγωνο ευκλείδειας απόστασης θα είναι κι αυτό μεγαλύτερο από την κορυφή του σωρού. Με τον αλγόριθμο αυτό, αποφεύγεται ο υπολογισμός αποστάσεων για όλα τα ζευγάρια σημείων input και training. Ο ψευδοκώδικας του αλγορίθμου δίδεται στο Παράρτημα Ι 2.

4.4 Αλγόριθμος σάρωσης επιπέδου με λωρίδες

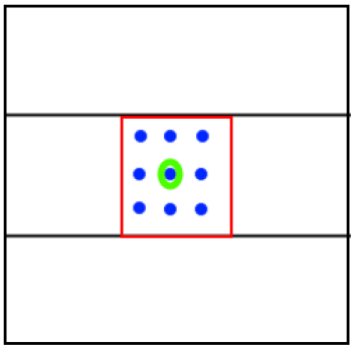
Οι λωρίδες δημιουργούνται σε διεύθυνση κάθετη ως προς τη γραμμή σάρωσης, δηλ. ενώ η σάρωση των σημείων training γίνεται κατά τη διεύθυνση x, ο διαχωρισμός σε λωρίδες γίνεται κατά τη διεύθυνση y. Αφού οριστεί ως παράμετρος το πλήθος των λωρίδων που θα χρησιμοποιήθεί, πραγματοποιείται διαίρεση του πλήθους των σημείων ενός εκ των δύο συνόλων (π.χ. του input) με το πλήθος των λωρίδων για την εύρεση του προσεγγιστικού αριθμού σημείων (έστω M) που θα περιλαμβάνει η κάθε λωρίδα. Έχοντας ταξινομηθεί τα δύο σύνολα input και training κατά y, προσθέτονται σε κάθε λωρίδα τουλάχιστον M σημεία από το input και επιπλέον όσα χρειάζονται αν υπάρχουν ισοτιμίες για το y στο τέλος του διαστήματος. Έτσι υπολογίζονται τα όρια για το y της κάθε λωρίδας και προστίθενται όσα σημεία του συνόλου training βρίσκονται σε αυτή την περιοχή. Αυτό που επιτυγχάνεται είναι η κάθε λωρίδα να έχει περίπου ίσο αριθμό σημείων input ώστε να υπάρχει ισοκατανομή φόρτου εργασίας όταν οι λωρίδες αυτες αναθετηθούν σε threads του λειτουργικού συστήματος. Κατά την εκτέλεση του αλγορίθμου, για κάθε σημείο του συνόλου input αναζητούμε γείτονες αρχικά στην ίδια λωρίδα όπως στον κανονικό Plane Sweep και έπειτα στις γειτονικές λωρίδες εναλλάξ και προς τις δύο κατευθύνσεις του άξονα y. Η αναζήτηση σταματάει σε γειτονικές λωρίδες όταν η τετραγωνική απόσταση κατά y, $(\Delta y)^2$ μεταξύ του input σημείου και του ορίου της λωρίδας γίνει μεγαλύτερο από την κορυφή του σωρού. Με παρόμοια διαδικασία μπορούν να δημιουργηθούν λωρίδες με βάση τα σημεία του συνόλου training, δηλ. να υπάρχει ίδιος αριθμός σημείων training αντί input σε κάθε λωρίδα. Ο ψευδοκώδικας του αλγορίθμου δίδεται στο Παράρτημα Ι 3.

4.5 Προσεγγιστικός υπολογισμός βέλτιστου πλήθους λωρίδων

Για τους αλγορίθμους οι οποίοι χρησιμοποιούν λωρίδες, το πλήθος λωρίδων που θα διαχωρίσουν τον δυσδιάστατο χώρο αποτελεί μεταβλητή εισόδου, δηλ. πρέπει να καθοριστεί η τιμή της από το χρήστη πριν την εκτέλεση του αλγορίθμου.

Όπως συμπεραίνουν οι (Χριστοφής, & Βασιλακόπουλος, 2018 [17]) κατά την πειραματική αξιολόγηση των αλγορίθμων, υπάρχει βέλτιστο πλήθος λωρίδων το οποίο εξαρτάται από το πλήθος σημείων των συνόλων input και training όπως και το πλήθος των αναζητούμενων πλησιέστερων γειτόνων. Όσο αυξάνεται το πλήθος σημείων των συνόλων, τόσο πρέπει να αυξάνεται το πλήθος λωρίδων που θα χωριστούν τα σύνολα ώστε να επιτευχθεί ο βέλτιστος χρόνος εκτέλεσης του αλγορίθμου. Αντίθετα, όσο αυξάνεται το πλήθος των αναζητούμενων γειτόνων (k) πρέπει να μειώνεται το πλήθος λωρίδων. Παρόλο που υπάρχει βέλτιστο πλήθος λωρίδων, αυτό δεν φαίνεται να είναι πολύ ευαίσθητο στο μέγεθος των συνόλων σημείων και στο πλήθος αναζητούμενων γειτόνων. Επίσης η κατανομή των σημείων στο δυσδιάστατο χώρο είναι λογικό ότι θα επηρεάζει και αυτή το βέλτιστο πλήθος λωρίδων.

Το βέλτιστο πλήθος λωρίδων θα πρέπει να δίνει τέτοιο ύψος λωρίδων έτσι ώστε ένα σημείο input που βρίσκεται στο μέσο του ύψους μιας λωρίδας να περικλείεται από ένα τετράγωνο κελί (ή ακριβέστερα κυκλικό αλλά εδώ η διαφορά δεν έχει ιδιαίτερη υπολογιστική σημασία) που εφάπτεται ακριβώς στα όρια της λωρίδας και περιέχει k γείτονες από το σύνολο training. Υποθέτοντας ότι η κατανομή των σημείων είναι απόλυτα ομοιόμορφη τόσο κατά x όσο και κατά y (δηλ. τα σημεία training σχηματίζουν ένα τέλεια τετραγωνισμένο πλέγμα), το τετράγωνο αυτό κελί θα έχει πλευρές με $\sqrt{k}$ σημεία training. Αν για παράδειγμα, όπως φαίνεται στο Σχήμα 4.1, αναζητούνται οι 9 πλησιέστεροι γείτονες για το σημείο input που συμβολίζεται με πράσινο κύκλο, η βέλτιστη λωρίδα θα έπρεπε να έχει τέτοιο ύψος ώστε να «χωράει» 3 σημεία training:



Σχήμα 4.1: Βέλτιστη λωρίδα για εύρεση 9 πλησιέστερων γειτόνων (πηγή [17])

Αν στο σύνολο training με κανονικοποιημένες διαστάσεις 0≤x≤1 , 0≤y≤1 περιέχονται N σημεία, κατά μήκος κάθε διάστασης θα υπάρχουν συνολικά $\sqrt{N}$ σημεία. Προσεγγιστικά, λοιπόν, υπολογίζεται το βέλτιστο πλήθος λωρίδων ως εξής:

$$S_{opt} = \sqrt{\frac{N}{k}}$$

όπου N το πλήθος σημείων του συνόλου training και k το πλήθος των αναζητούμενων πλησιέστερων γειτόνων. Πειραματικά η παραπάνω προσέγγιση φαίνεται να δίνει ικανοποιητικά αποτελέσματα ακόμα και σε μη ομοιόμορφα σύνολα σημείων, δηλαδή ο χρόνος εκτέλεσης του αλγορίθμου που επιτυγχάνεται με αυτό το πλήθος λωρίδων βρίσκεται πολύ κοντά στο ελάχιστο. Πρέπει να σημειωθεί ότι οι λωρίδες πάντοτε χωρίζονται έτσι ώστε να περιλαμβάνουν σταθερό αριθμό σημείων είτε του input είτε του training συνόλου. Έτσι παρόλο που η εύρεση του βέλτιστου αριθμού

λωρίδων έχει βασιστεί σε ομοιόμορφη κατανομή σημείων, σε κάθε περίπτωση λαμβάνεται υπόψη η διαφορετική πυκνότητα σημείων κατά τον άξονα y.

4.6 Εναλλακτικοί τρόποι χωρισμού λωρίδων

4.6.1 Χωρισμός βάσει του συνόλου input ή training

Κατά τη δημιουργία λωρίδων δίνεται η δυνατότητα επιλογής σταθερού αριθμού σημείων κάθε λωρίδας τόσο για το σύνολο input όσο και για το training. Η πρώτη επιλογή ήταν ο δαχωρισμός σε λωρίδες με σταθερού αριθμού σημείων input με τη λογική ότι παράγονται απόλυτα ισοζυγισμένες λωρίδες όσον αφορά τον φόρτο εργασίας (σημ. για κάθε σημείο input αναζητούνται οι k πλησιέστεροι γείτονές του από το σύνολο training επομένως η πολυπλοκότητα χρόνου είναι τουλάχιστον ανάλογη του πλήθους σημείων του συνόλου input). Λαμβάνοντας υπόψη ότι η βιβλιοθήκη Thrust διαθέτει δυναμικό τρόπο ανάθεσης των επαναλήψεων ενός βρόχου σε threads, αξίζει να μελετηθεί ο διαχωρισμός σε λωρίδες χρησιμοποιώντας σταθερό αριθμό σημείων training για κάθε λωρίδα. Με τον τρόπο αυτό δημιουργούνται λωρίδες με μεταβλητό πλήθος σημείων input, όμως, η ανισορροπία αυτή μπορεί να αντιμετωπιστεί με τη δυναμική ανάθεση λωρίδων σε threads (dynamic scheduling).

4.6.2 Χωρισμός με σειριακό ή παράλληλο βρόχο

Ο χωρισμός των λωρίδων μπορεί να γίνει με ένα σειριακό βρόχο όπου υπολογίζεται πόσα σημεία input ή training πρέπει να έχει η κάθε λωρίδα και έπειτα με ένα σειριακό βρόχο while αναθέτονται αυτά τα σημεία σε λωρίδες. Σε κάθε επανάληψη του βρόχου while εκτελείται παράλληλη ταξινόμηση των σημείων της λωρίδας κατά τον x άξονα χρησιμοποιώντας τον παράλληλο αλγόριθμο quicksort. Ο αλγόριθμος τροποποιήθηκε ώστε αντί του σειριακού while να χρησιμοποιείται ένα παράλληλο for. Τα στάδια επεξεργασίας είναι τα ακόλουθα:

1. Ταξινόμιση κατά y των συνόλων input και training με παράλληλο αλγόριθμο quicksort.

2. Παράλληλος βρόχος for για κάθε μία λωρίδα, όπου υπολογίζονται προσεγγιστικά η αρχή και το τέλος της κάθε λωρίδας με βάση τον αριθμό σημείων που πρέπει να περιλαμβάνει η καθεμιά. Κάθε λωρίδα εκτός της τελευταίας περιλαμβάνει αρχικά τον ίδιο αριθμό σημείων input ή training (το αν θα χρησιμοποιηθεί το input ή το training σύνολο αποφασίζεται από παράμετρο του αλγορίθμου).

3. Μέσα στον ίδιο παράλληλο βρόχο for, αν υπάρχουν ισοτιμίες y στα προϋπολογισμένα όρια των λωρίδων, τότε μετακινούμε την αρχή ή/και το τέλος της λωρίδας έτσι ώστε σημεία με ίδιο y να περιλαμβάνονται πάντα στην ίδια λωρίδα. Οι λωρίδες που παράγονται στο στάδιο αυτό ενδέχεται να έχουν διαφορετικό αριθμό σημείων ή ακόμα και να είναι κενές οπότε θα αγνοηθούν στην περαιτέρω επεξεργασία. Μέσα σε κάθε επανάληψη του παράλληλου for και αφού έχουν υπολογιστεί τα πραγματικά όρια κάθε λωρίδας, εκτελείται σειριακή ταξινόμηση των σημείων κατά x.

4. Τα αποτελέσματα του παράλληλου βρόχου τοποθετούνται σε διαφορετική θέση ενός πίνακα λωρίδων οπότε δεν απαιτείται κλείδωμα του πίνακα για πρόσβαση από διαφορετικά threads.

Κεφάλαιο 5

Παραλληλοποίηση αλγορίθμων με τη βιβλιοθήκη Thrust

5.1 Εισαγωγή

Οι σειριακοί αλγόριθμοι που αναφέρθηκαν στο προηγούμενο κεφάλαιο, ο αλγόριθμος διαχωρισμού σε λωρίδες καθώς και οι ταξινομήσεις των δεδομένων εισόδου τροποποιήθηκαν με τη χρήση της βιβλιοθήκης CUDA Thrust, ώστε να εκτελούνται παράλληλα σε Nvidia GPUs. Η CPU και η GPU είναι ξεχωριστές οντότητες. Και οι δύο έχουν το δικό τους χώρο μνήμης. Η CPU δεν μπορεί να έχει άμεση πρόσβαση στη μνήμη GPU και το αντίστροφο. Στην ορολογία της CUDA, η μνήμη CPU ονομάζεται μνήμη κεντρικού υπολογιστή (host memory) και η μνήμη GPU ονομάζεται μνήμη συσκευής (device memory). Οι δείκτες προς τη μνήμη CPU και GPU ονομάζονται δείκτης κεντρικού υπολογιστή (host pointer) και δείκτης συσκευής (device pointer), αντίστοιχα.

Για να είναι προσβάσιμα τα δεδομένα από τη GPU, πρέπει να παρουσιάζονται στη μνήμη της συσκευής. Το CUDA παρέχει API για δέσμευση μνήμης συσκευής και μεταφορά δεδομένων μεταξύ μνήμης κεντρικού υπολογιστή και συσκευής. Ακολουθεί η κοινή ροή εργασίας των προγραμμάτων CUDA.

1. Δέσμευση μνήμης κεντρικού υπολογιστή και αρχικοποίηση δεδομένων κεντρικού υπολογιστή

2. Δέσμευση μνήμης συσκευής

3. Μεταφορά δεδομένων εισόδου από τον κεντρικό υπολογιστή στη μνήμη της συσκευής

4. Εκτέλεση πυρήνα

5. Μεταφορά δεδομένων εξόδου από τη μνήμη της συσκευής στον κεντρικό υπολογιστή

5.2 Διαχείριση δεδομένων

5.2.1 Δέσμευση μνήμης

Η Thrust παρέχει δύο δομές δεδομένων, thrust::host_vector και thrust::device_vector, που χρησιμοποιήθηκαν ως πίνακες για την αποθήκευση και διαχείριση των δεδομένων. Οι δεσμεύσεις και αποδεσμεύσεις χειρίζονται αυτόματα από την Thrust κάνοντας χρήση των CUDA συναρτήσεων cudaMalloc και cudaFree.

5.2.2 Μεταφορά δεδομένων

Η αντιγραφή δεδομένων μπορεί να πραγματοποιηθεί με τρεις τρόπους.

- Με τον τελεστή ”=”
- Με τη συνάρτηση thrust::copy
- Με συγκεκριμένους κατασκευαστές οι οποίοι αντιγράφουν και αρχικοποιούν κατά τη δέσμευση

Όπως και με τα διανύσματα, τη διαχείριση μνήμης αναλαμβάνει η Thrust η οποία χρησιμοποιεί και στις τρεις εκδοχές αντιγραφής την CUDA συνάρτηση cudaMemcpy.

5.3 Διαφορές παράλληλης και σειριακής υλοποίησης

5.3.1 Παράλληλοι βρόχοι

Οι σειριακοί αλγόριθμοι μετατρέπονται σε παράλληλους με μια βασική τροποποίηση. Οι σειριακοί βρόχοι αντικαθίσταται από τη συνάρτηση thrust::for_each η οποία εφαρμόζεται σε ένα διάνυσμα και εκτελεί παράλληλα σε κάθε στοιχείο του στο εύρος [πρώτο, τελευταίο], ένα αντικείμενο τύπου μεμονωμένης συνάρτησης (UnaryFunction). Σε αυτό το αντικείμενο περιέχεται το σώμα του βρόχου.

Τα απαραίτητα δεδομένα για την εκτέλεση του πυρήνα μεταφέρονται στη μνήμη συσκευής. Τα σύνολα input και training καθώς και η δομή που περιλαμβάνει τους σωρούς k γειτόνων όλων των σημείων input μετατρέπονται σε thrust::device_vectors. Το πρώτο σύνολο χρησιμοποιείται ως το διάνυσμα που θα διασπαστεί με παράλληλο τρόπο ώστε για κάθε input στοιχείο του, θα εκτελείται το σώμα της μεμονωμένης συνάρτησης. Στη διάσπαση αυτή συμμετέχουν και οι σωροί των k πλησιέστερων γειτόνων σε αντιστοιχία με τα input σημεία. Στη μεμονωμένη συνάρτηση δίδεται ως παράμετρος ένας δείκτης στο σύνολο training ώστε να πραγματοποιηθεί σειριακή αναζήτηση των k γειτόνων για ένα μεμονωμένο input σημείο. Αυτή η διαδικασία αποτελεί τον κορμό της παραλληλοποίησης τον αλγορίθμων και είναι κοινή για όλους τους αλγορίθμους. Ωστόσο από τον παράλληλο αλγόριθμο ωμής βίας που είναι ο πιο απλός, μέχρι τον παράλληλο αλγόριθμο σάρωσης επιπέδου με λωρίδες, τον πιο σύνθετο, παρατηρούνται αρκετές αλλαγές στο πλήθος παραμέτρων και τις συναρτήσεις που χρησιμοποιούνται εντός της μεμονωμένης συνάρτησης.

Παρακάτω παρουσιάζεται ένα απόσπασμα του κώδικα του παράλληλου αλγορίθμου ωμής βίας:

Παράδειγμα παραλληλοποίησης (Parallel Brute Force)

```
struct CreateNeighborsMaxHeap :
    public thrust::unary_function<knn_algorithm_type_tuple_t, void> {
  KernelArray<Point> trainingDataset;

  CreateNeighborsMaxHeap(KernelArray<Point> _trainingDataset) {
    trainingDataset = _trainingDataset;
  }
                                .
                                .
                                .
  __device__ void operator() (knn_algorithm_type_tuple_t tuple) {
    auto& inputPoint = thrust::get<0>(tuple);
    auto& neighbors = thrust::get<1>(tuple);
    auto& numAdditions = thrust::get<2>(tuple);

    //loop through all training points
    for (int i = 0; i < trainingDataset.size; i++) {

      //check distance and add neighbor to max heap
      AddNeighbor(inputPoint, trainingDataset.array[i], neighbors);
    }

    numAdditions = neighbors.GetNumAdditions();
  }
};

void process() {
                                .
                                .
                                .
  thrust::device_vector<Point> d_inputDataset(inputDataset);
  thrust::device_vector<Point> d_trainingDataset(trainingDataset);

  auto trainingPoints = convertToKernel(d_trainingDataset);

  knn_algorithm_zip_iterator_t first = thrust::make_zip_iterator
    (thrust::make_tuple(d_inputDataset.begin(), d_pNeighborsContainer->begin(),
    data.numAdditions.begin()));

  knn_algorithm_zip_iterator_t last = thrust::make_zip_iterator
    (thrust::make_tuple(d_inputDataset.end(), d_pNeighborsContainer->end(),
    data.numAdditions.end()));

  //parallel loop through all input points
  thrust::for_each(thrust::device, first, last,
    CreateNeighborsMaxHeap(trainingPoints));
                                .
```

.
.
}

Ο εξωτερικός βρόχος διατρέχει είτε το σύνολο των σημείων input για τα οποία αναζητούμε γείτονες είτε το σύνολο των λωρίδων. Όταν χρησιμοποιείται παραλληλισμός του εξωτερικού βρόχου και επειδή η χρήση μιας GPU απαιτεί μεγάλο πλήθος δεδομένων για να δώσει χρονοβελτίωση, δεν έχει νόημα να χρησιμοποιηθεί παραλληλισμός και στους εσωτερικούς βρόχους διότι όλοι οι πυρήνες του επεξεργαστή είναι ήδη απασχολημένοι.Ακολούθως αναφέρουμε σε ποια σημεία τροποποιήθηκαν οι βασικοί αλγόριθμοι με οδηγίες:

Οι βασικοί αλγόριθμοι τροποποιήθηκαν στα εξής σημεία:

1. Αλγόριθμος Brute Force
    i Εξωτερικός βρόχος που διατρέχει τα σημεία input
2. Αλγόριθμος Plane Sweep
    i Ταξινόμηση σημείων κατά άξονα x
    ii Εξωτερικός βρόχος που διατρέχει τα σημεία input
3. Αλγόριθμος Plane Sweep με λωρίδες
    i Ταξινόμηση σημείων κατά άξονα y
    ii Χωρισμός λωρίδων
    iii Ταξινόμηση σημείων κατά άξονα x μέσα σε κάθε λωρίδα
    iv Εξωτερικός βρόχος που διατρέχει τις λωρίδες

5.3.2 Κορυφή μέγιστης ευκλίδειας απόστασης

Στις σειριακές υλοποιήσεις γίνεται η χρήση του σωρού μέγιστης απόστασης των k γειτόνων για κάθε σημείο input. Στη δομή αυτή οι k γείτονες είναι ταξινομημένοι σε αύξουσα σειρά ως προς την ευκλίδεια απόστασή τους από το εκάστοτε σημείο input. Σε κάθε νέα εκχώρηση γείτονα πρέπει να διατηρεί αυτή του την ιδιότητα. Ωστόσο δεν είναι αναγκαίο να είναι ταξινομημένοι όλοι οι k γείτονες παρά μόνο ο γείτονας με την μεγαλύτερη απόσταση να βρίσκεται στην κορυφή. Με τον τρόπο αυτό εκτελούνται αισθητά λιγότερες πράξεις σε εξάρτηση πάντα με το μέγεθος του συνόλου input και τον αριθμό k.

Κεφάλαιο 6

Πειραματική Αξιολόγηση

Πραγματοποιήθηκε μία σειρά πειραμάτων πάνω σε φυσικά αλλά και τυχαία σημειοσύνολα. Διερευνάται κατά πόσο η παραλληλοποίηση επηρεάζει τον υπολογιστικό χρόνο σε σχέση με τους αντίστοιχους σειριακούς αλγορίθμους, καθώς και ο συνδυασμός μεγεθών input και training που ευννοεί τους εν λόγω αλγορίθμους.

Η ορθότητα των αποτελεσμάτων των αλγορίθμων επαληθεύτηκε με τη μέθοδο της σύγκρισης με κάποιο αλγόριθμο αναφοράς. Αλγόριθμο αναφοράς αποτέλεσε ο σειριακός brute force εφόσον ήταν εφικτό να εκτελεστεί σε λογικά χρονικά πλαίσια ή κάποιος άλλος (όσο το δυνατόν πιο απλός) για τις υπόλοιπες περιπτώσεις. Διαφορετικοί αλγόριθμοι είναι πιθανό να καταλήξουν σε διαφορετικούς πλησιέστερους γείτονες , όμως η απόστασή τους από το σημείο αναφοράς είναι η ίδια. Για το λόγο αυτό η σύγκριση γίνεται μεταξύ των ευκλίδειων αποστάσεων των γειτόνων από το σημείο input και όχι μεταξύ των ταυτοτήτων των γειτόνων (δηλ. το id του σημείου training).

6.1 Σύστημα αξιολόγησης

Η πειραματική αξιολόγηση των αλγορίθμων πραγματοποιήθηκε σε σύστημα με επεξεργαστή Intel Core i7-8550U, CPU @ 1.80GHz 2.00 GHz, 8GB RAM, λειτουργικό σύστημα Windows 10 και κάρτα γραφικών GeForce MX130 (πυρήνες CUDA 384), ρολοι γραφικών 1108 MHz, κοινόχρηστη μνήμη 4053MB και αποκλειστική μνήμη 2048 MB GDDR5.

Ο αριθμός των νημάτων CUDA καθώς και η διαδικασία διαμοίρασης εργασιών σε νήματα καθορίζεται αυτόματα από τη Thrust.

6.2 Δομή αρχείων δεδομένων

Τα σύνολα δεδομένων που χρησιμοποιήθηκαν έχουν τη μορφή αρχείων κειμένου ή δυαδικών αρχείων. Η ανάγνωση των δυαδικών αρχείων γίνεται αρκετά ταχύτερα από αυτη των αρχείων κειμένου. Κάθε σύνολο δεδομένων περιέχει τρία πεδία για κάθε σημείο, ένα μοναδικό αναγνωριστικό (Id) με ακέραια τιμή (long integer) από 1 ως το πλήθος των σημείων και τις συντεταγμένες x, y ως αριθμούς κινητής υποδιαστολής διπλής ακρίβειας (double). Οι συντεταγμένες x, y έχουν κανονικοποιηθεί στο διάστημα [0, 1].

Στα αρχεία κειμένου τα σημεία δίνονται ένα ανά γραμμή και οι στήλες χωρίζονται με χαρακτήρα tab. Στα δυαδικά αρχεία δίνεται αρχικά το πλήθος των σημείων με έναν ακέραιο (long integer) και έπειτα ακολουθεί η αλληλουχία των σημείων (Id, x, y) με δυαδική αναπαράσταση.

6.3 Ανάλυση πειραμάτων

Δημιουργήθηκαν τυχαία δεδομένα από χίλια (1K) έως 50 εκατομμύρια (50M) σημεία που ακολουθούν την ομοιόμορφη κατανομή. Εκτελέστηκαν όλοι οι συνδυασμοί πειραμάτων για input 1K, 100K, 500K, 1M και training 100K, 250K, 500K, 750K, 1M, 10M, 50M για τη αξιολόγιση της απόδοσης των αλγορίθμων και την επίδραση των παραμέτρων των αλγορίθμων στο συνολικό χρόνο εκτέλεσης. Έγινε, επίσης, σύγκριση με τις αντίστοιχες παράλληλες υλοποιήσεις των Χριστοφή και Βασιλακόπουλου (2018) σε επεξεργαστή και τους αλγορίθμους των Velentzas, Vassilakopoulos και Corral σε κάρτα γραφικών.

Στην πρώτη σύγκριση χρησιμοποιούνται δύο φυσικά σύνολα που αντιπροσωπεύουν υδατικούς πόρους της Βόρειας Αμερικής (Water Dataset) που αποτελούνται από 5.836.360 γραμμικά τμήματα και παγκόσμια πάρκα ή περιοχές πρασίνου (Parks Dataset) 11,503,925 πολυγόνων όπου αναζητούνται οι k=20 κοντινότεροι γείτονες για σύνολα input τυχαίων δεδομέων μεγέθους 250, 500, 750 και 1000 σημείων. Η δεύτερη χρησιμοποιεί τις σιδηροδρομικές γραμμές 191.637 σημείων (input) και εθνικές οδούς 569.120 σημείων (training) της Βόρειας Αμερικής και πλήθος αναζητούμενων γειτόνων k=10.

Για τη δημιουργία συνόλων σημείων, χρησιμοποιήθηκαν τα κέντρα των γραμμικών τμημάτων από το νερό και τα κεντροειδή πολυγώνων από τα πάρκα. Για όλα τα σύνολα δεδομένων, ο δυσδιαστατος χώρος δεδομένων είναι κανονικοποιημένος ώστε να έχει μήκος μονάδας (τιμές [0, 1] σε κάθε άξονα). Για όλα τα πειράματα χρησιμοποιήθηκε ο αλγόριθμος υπολογισμού του βέλτιστου πλήθους λωρίδων.

6.4 Αξιολόγηση με σύνολα τυχαίων δεδομένων

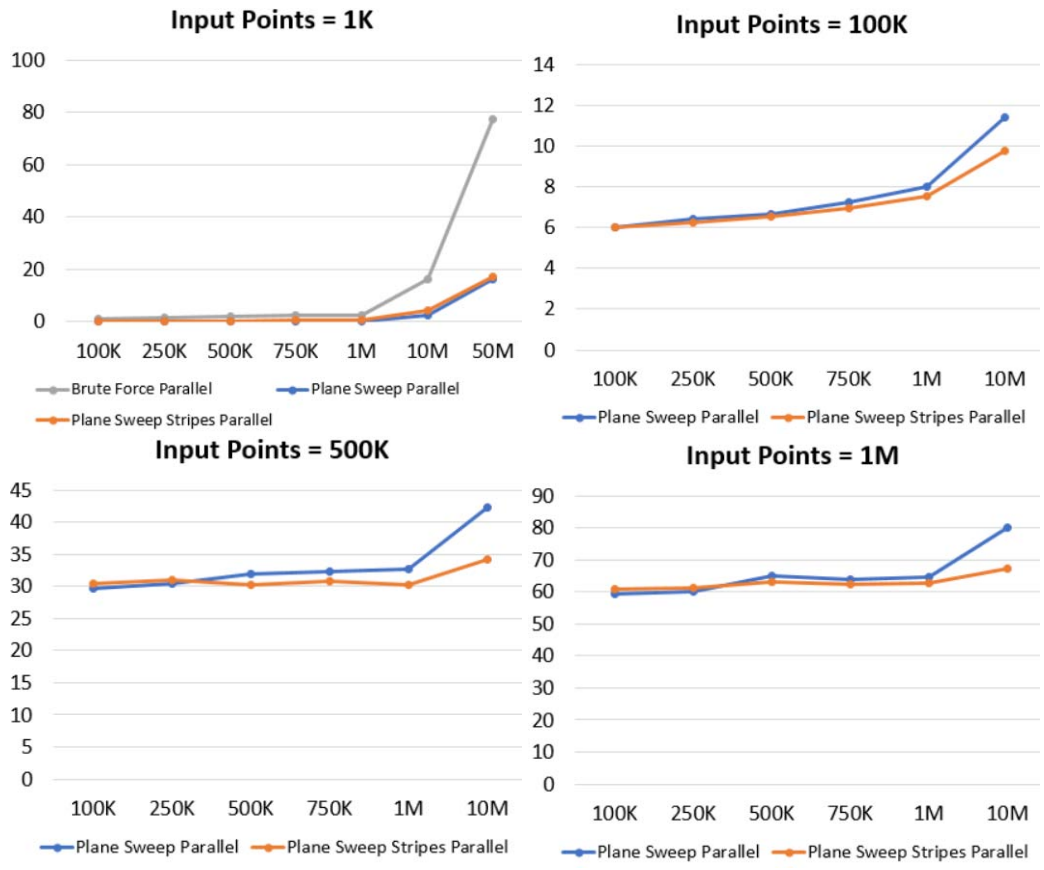
6.4.1 Επίδραση πλήθους σημείων των συνόλων

Διατηρείται σταθερό το πλήθος σημείων του συνόλου input στα 1.000, 100.000, 500.000 και 1.000.000 και μεταβάλλεται το πλήθος των σημείων του συνόλου training από 100.000 έως και 50.000.000. Οι αλγόριθμοι που εξετάζονται είναι:

- ο παράλληλος Brute Force
- ο παράλληλος Plane Sweep με αντιγραφή στοιχείων και παράλληλη ταξινόμηση
- ο παράλληλος Plane Sweep με λωρίδες με παράλληλο διαχωρισμό σε λωρίδες και παράλληλη ταξινόμηση

Οι γραφικές των αποτελεσμάτων φαίνονται στο Σχήμα 6.1.

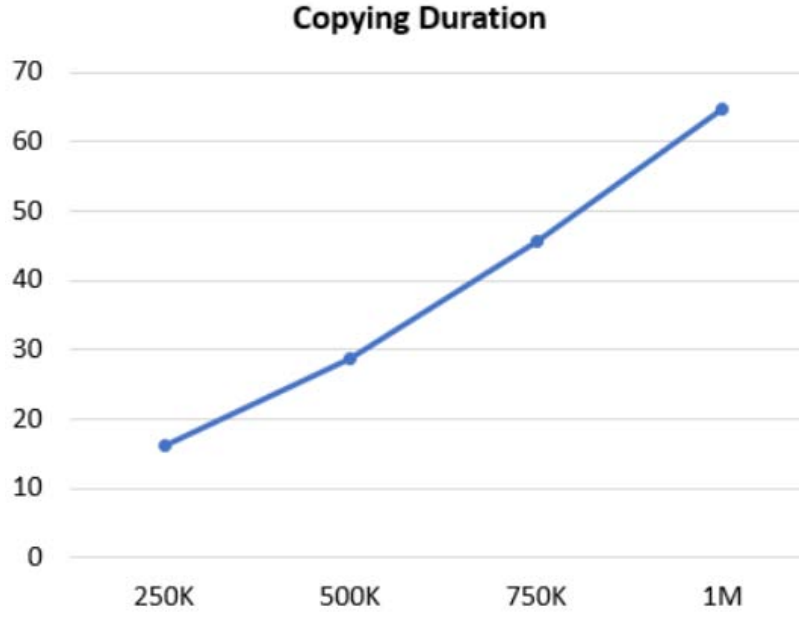
Παρατηρείται ότι πιο ευαίσθητος στη μεταβολή του μεγέθους του συνόλου training είναι ο Brute Force ο οποίος παρουσιάζει τετραγωνική πολυπλοκότητα $O(N^2)$. Αντίθετα, ο Plane Sweep



Σχήμα 6.1: Επίδραση πλήθους σημείων συνόλων στο χρόνο εκτέλεσης

και ο Plane Sweep με λωρίδες παρουσιάζουν γραμμική πολυπλοκότητα $O(N)$. Ωστόσο η κλίση των γραμμικων συνάρτησεων είναι τόσο μικρή που για αύξηση σημείων training κατά 100.000 φορές (100K έως 10M) ο χρόνος εκτέλεσης αυξάνεται λιγότερο από 2 φορές.

Αντιγραφή αποτελεσμάτων Φαίνεται, επίσης, ότι όσο μεγαλύτερο είναι το σύνολο input τόσο μικρότερη είναι η επίδραση του training. Στην πραγματικότητα, η επίδρασή του είναι σταθερή σε αναλογία, όμως, λιγότερο αισθητή σε σχέση με το συνολικό χρόνο εκτέλεσης. Όσο μεγαλώνει το μέγεθος του συνόλου input τόσο μεγαλώνει ο όγκος αντιγραφής των αποτελεσμάτων από τη συσκευή (device) στον κεντρικό υπολογιστή (host). Αν συμβολίσουμε το πλήθος σημείων συνόλου input με N, τότε ο όγκος αυτός είναι της τάξης N x k. Όπως δείχνει το Σχήμα 6.2, ο χρόνος αντιγραφής αυξάνεται γραμμικά με το πλήθος σημείων του input.

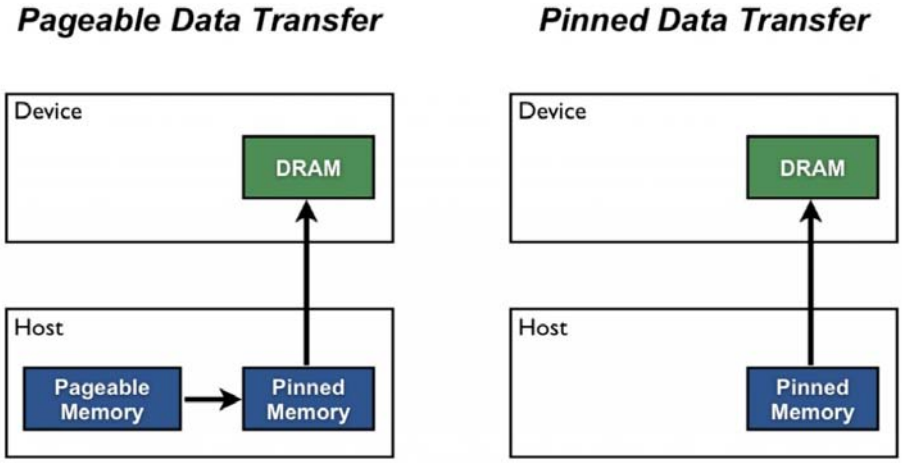


Σχήμα 6.2: Επίδραση πλήθους σημείων input στο χρόνο αντιγραφής αποτελεσμάτων

Δέσμευση κι αρχικοποίηση μνήμης συσκευής Είναι σημαντικό να αναφερθεί ότι από τους συνολικούς χρόνους εκτέλεσης έχει παραληφθεί ο χρόνος δέσμευσης και αρχικοποίησης μνήμης, δηλαδή η μεταφορά όλων των απαραίτητων δεδομένων για την εκτέλεση στη μνήμη της κάρτας γραφικών. Ο λόγος αυτής της παράλειψης είναι ότι η διαδικασία αρχικοποίησης δεν έχει βελτιστοποιηθεί και έτσι θα "θόλωνε" την πραγματική επίδοση των αλγορίθμων.

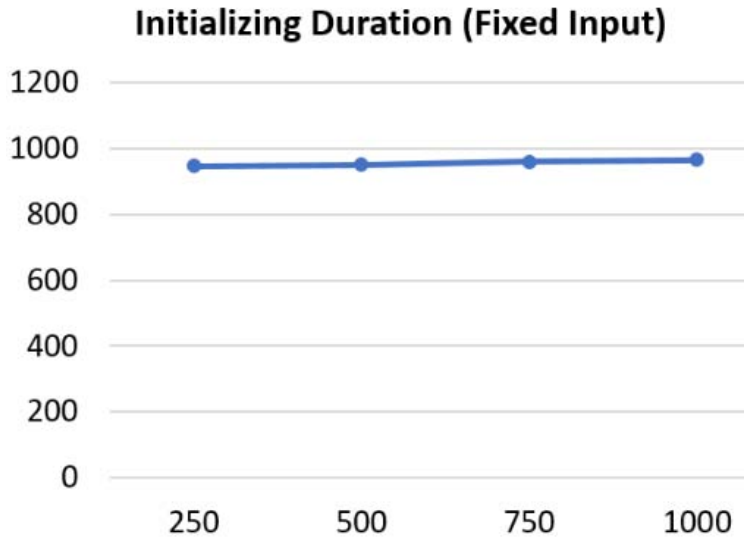
Τα δεδομένα αποθηκεύονται αρχικά σε διανύσματα κύριας μνήμης μέσω της STL βιβλιοθήκης και στη συνέχεια πραγματοποιείται αντιγραφή στοιχείων από την κύρια μνήμη στην μνήμη της συσκευής. Ωστόσο σε αυτό το σημείο πραγματοποιούνται δύο αντιγραφές. Στον κεντρικό υπολογιστή (CPU) τα δεδομένα αποθηκεύονται απο προεπιλογή σε μνήμη όπου πραγματοποιείται σελιδοποίηση (pageable host memory). Η GPU δεν έχει απευθείας πρόσβαση στη μνήμη με σελιδοποίηση

οπότε όταν πραγματοποιείται μεταφορά από αυτή τη μνήμη στη μνήμη συσκευής, ο δρομολογιτής (driver) του CUDA πρέπει πρώτα να δεσμεύσει ένα προσωρινό κλειδωμένων σελίδων (page-locked) ή αλλιώς "καρφιτσωμένο" ("pinned"), πίνακα κεντρικού υπολογιστη, να μεταφέρει τα δεδομένα στον καρφιτσωμένο πίνακα κι έπειτα, από εκεί να γίνει μεταφορά των δεδομένων στην μνήμη συσκευής όπως φαίνεται στο Σχήμα 6.3. Όπως είναι φυσικό όλη αυτή η διαδικασία κοστίζει σε χρόνο. Ωστόσο, αποθηκεύοντας τα απαραίτητα δεδομένα απευθείας στην καρφιτσωμένη μνήμη ή και στην μνήμη συσκευής όπου αυτό είναι δυνατό, το κόστος αυτό μειώνεται αισθητά.



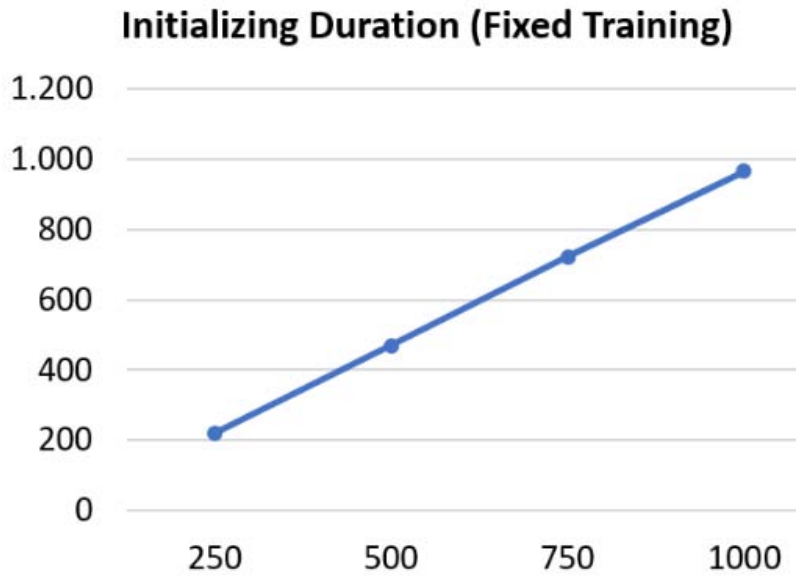
Σχήμα 6.3: Μεταφορά δεδομένων με δυνατότητα σελιδοποίησης και καρφιτσωμένων δεδομένων (πηγή [4])

Στο Σχήμα 6.4 παρατηρείται ότι η επίδραση του πλήθος σημείων του συνόλου training όταν αυτό μεταβάλλεται από 250.000 έως και 10.000.000 σημεία, με σταθερά 1.000.000 σημεία συνόλου input και k=10, είναι απειροελάχιστη στο συνολικό χρόνο αρχικοποίησης και δέσμευσης.



Σχήμα 6.4: Επίδραση μεγέθους σημειοσυνόλου training στο χρόνο αρχικοποίησης

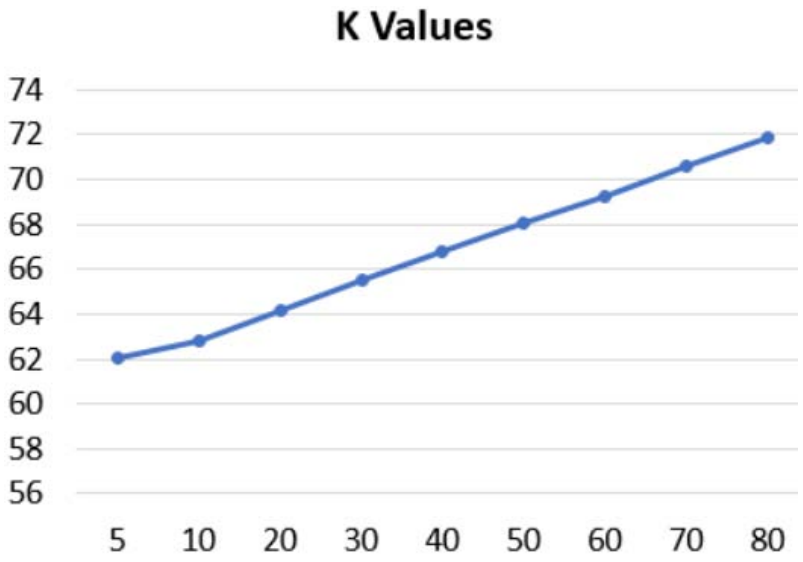
Στο Σχήμα 6.5 φαίνεται η επίδραση του πλήθος σημείων του συνόλου input όταν αυτό μεταβάλλεται από 250.000 έως και 1.000.000 σημεία, με σταθερά 10.000.000 σημεία συνόλου training και k=10 στο συνολικό χρόνο αρχικοποίησης και δέσμευσης. Ο χρόνος αυξάνεται γραμμικά με το μέγεθος του συνόλου input. Ωστόσο, όπως και με το σύνολο training, ο αποκλειστικός χρόνος αντιγραφής των σημείων input παραμένει σχεδόν ανεπηρέαστος από την μεταβολή του μεγέθους του σημειοσυνόλου. Εκείνο που καταλαμβάνει σχεδόν το απόλυτο ποσοστό του χρόνου αρχικοποίησης είναι η δημιουργία του δοχείου που περιέχει τους σωρους μεγίστου για κάθε σημείο input. Εξ' ου και η γραμμική αύξηση, λόγω της συσχέτισης του πλήθους σημείων input με το δοχείο σωρών.



Σχήμα 6.5: Επίδραση μεγέθους σημειοσυνόλου input στο χρόνο αρχικοποίησης

6.4.2 Επίδραση πλήθους αναζητούμενων γειτόνων k

Για να αξιολογηθεί η επίδραση του αριθμού πλησιέστερων γειτόνων k εξετάστηκε ο παράλληλος Plane Sweep με λωρίδες με παράλληλη ταξινόμηση και διαχωρισμό σε λωρίδες για 1.000.000 σημεία συνόλου input και training και k από 5 έως 80. Τα αποτελέσματα φαίνοται στο Σχήμα 6.6 όπου παρατηρείται ότι ο αλγόριθμος παρουσιάζει γραμμική πολυπλοκότητα (N) όσο το k αυξάνεται.

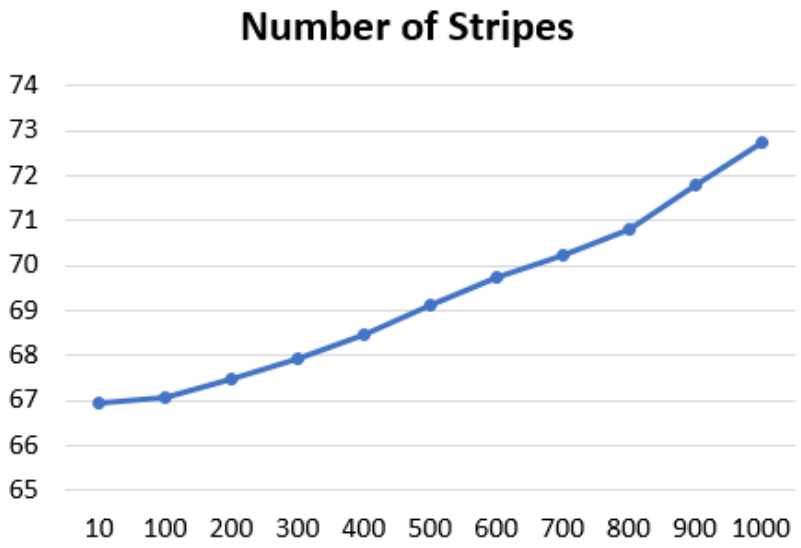


Σχήμα 6.6: Επίδραση πλήθους αναζητούμενων γειτόνων k στο χρόνο εκτέλεσης

6.4.3 Επίδραση του πλήθους λωρίδων

Η επίδραση του πλήθους λωρίδων αξιολογήθηκε χρησιμοποιώντας τον παράλληλο Plane Sweep με λωρίδες και παράλληλη ταξινόμηση και διαχωρισμό σε λωρίδες για σημειοσύνολα input και training μεγέθους 1.000.000 σημείων και k=10. Το πλήθος λωρίδων που μελετήθηκε κυμαίνεται από 10 έως και 1000. Τα αποτελέσματα φαίνονται στο Σχήμα 6.7 όπου ο άξονας X αναπαριστά το πλήθος λωρίδων και ο άξονας Y το συνολικό χρόνο εκτέλεσης.

Παρατηρείται ότι ο συνολικός χρόνος εκτέλεσης αυξάνεται γραμμικά με το πλήθος λωρίδων. Τα αποτελέσματα αυτά διαφέρουν από την έρευνα [17] όπου η επίδραση του αριθμού λωρίδων δίνει χρονοβελτίωση μέχρι ενός σημείου μετά από αυτό ο συνολικός χρόνος εκτέλεσης παραμένει σχεδόν αμετάβλητος. Αυτό συμβαίνει διότι η μεταφορά των σημείων του συνόλου γίνεται σταδιακά το οποίο κοστίζει περισσότερο σε χρόνο απότι μία ενιαία μεταφορά. Έτσι όσο πιο μεγάλο είναι το πλήθος λωρίδων τόσο περισσότερο κοστίζει χρονικά. Σύμφωνα με τον προσεγγιστικό τύπο της παραγράφου 4.5 ο βέλτιστος αριθμός λωρίδων είναι 317 που όπως φαίνεται δεν ανταποκρίνεται στην πραγματικότητα για τα συγκεκριμένα τυχαία σύνολα δεδομένων. Δεν αποτελεί τη βέλτιστη επιλογή, είναι ωστόσο ικανοποιητική.



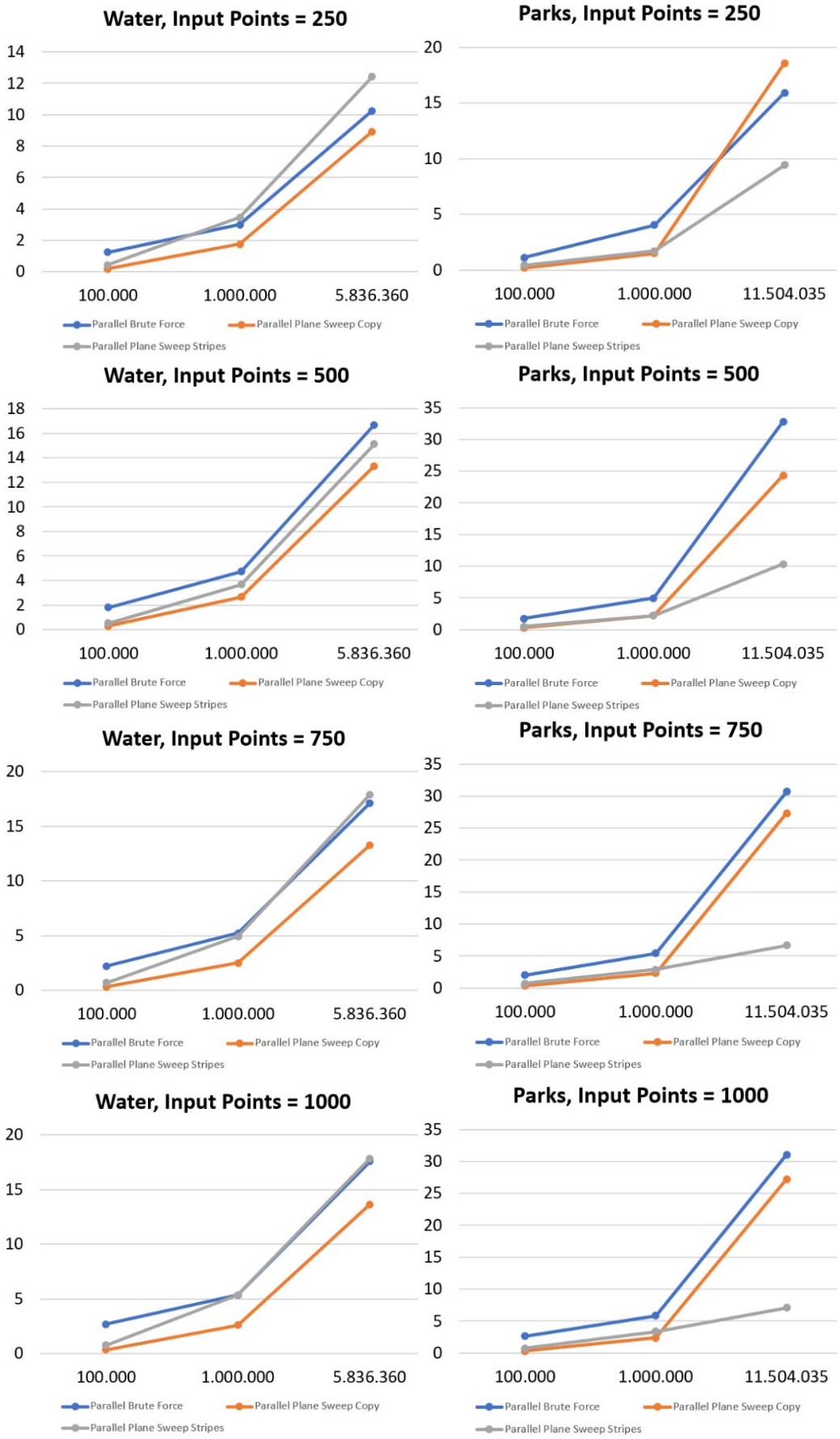
Σχήμα 6.7: Επίδραση του πλήθους λωρίδων στο συνολικό χρόνο εκτέλεσης

6.5 Αξιολόγηση με σύνολα φυσικών δεδομένων

6.5.1 Σύγκριση παράλληλων υλοποιήσεων σε GPU

Εκτελέστηκαν 8 πειράματα χρησιμοποιώντας τα φυσικά σύνολα δεδομένων υδατικών πόρων της Βόρειας Αμερικής με 5.836.360 σημεία και παγκόσμιων πάρκων ή περιοχών πρασίνου 11,503,925 σημείων για αριθμό πλησιέστερων γειτόνων k=20. Το γκρουπ συνόλων input αποτελείται από 250, 500, 750 και 1000 σημεία. Για την κλιμάκωση του χρόνου εκτέλεσης δημιουργήθηκαν υποσύνολα training των 100.000 και 1.000.000 σημείων για καθένα από τα δύο αυτά πραγματικά σύνολα. Έτσι τελικά δημιουργήθηκαν γκρουπ συνόλων training σημείων 100.000, 1.000.000 και πλήθους εγγραφών ολόκληρου του εκάστοτε αρχείου. Οι αλγόριθμοι που εξετάστηκαν είναι ο παράλληλος Brute Force, ο παράλληλος Plane Sweep με αντιγραφή στοιχείων και παράλληλη ταξινόμηση και ο παράλληλος Plane Sweep με λωρίδες με παράλληλο διαχωρισμό σε λωρίδες με βάση το σύνολο training και παράλληλη ταξινόμηση.

Παρατηρείται ότι για το σύνολο των υδατικών πόρων ταχύτερος είναι ο Plane Sweep με αντιγραφή στοιχείων ενώ για το σύνολο των πάρκων ο Plane Sweep με λωρίδες.



Σχήμα 6.8: Χρόνοι εκτέλεσης για φυσικά σύνολα training και k=20

Οι επιδόσεις αυτών των αλγορίθμων θα συγκριθούν με τον T-DS για κάθε σύνολο από την έρευνα [13]. Στον Πίνακα 6.1 και 6.2 φαίνονται οι επιδόσεις των ταχύτερων αλγορίθμων κάθε εργασίας για ολόκληρο το σύνολο δεδομένων των υδατικών πόρων και πάρκων αντίστοιχα.

Παρατηρείται ότι ο T-DS υστερεί σε όλα τα πειράματα εκτός του πρώτου. Ο Plane Sweep Copy φαίνεται να διατηρεί μια σταθερή απόκριση όσο αυξάνεται το πλήθος σημείων του συνόλου input, σε αντίθεση με τον T-DS ο οποίος αυξάνεται γραμμικά. Αυτό συμβαίνει διότι ο T-DS είναι αλγόριθμος ωμής βίας. Από την άλλη οι Plane Sweep αλγόριθμοι ελέγχουν υποσύνολα του συνόλου input βασισμένοι στην ευκλίδεια απόσταση και εκτελώντας, έτσι, λιγότερους υπολογισμούς.

Για το πρώτο σύνολο οι χρόνοι εκτέλεσης των δύο αλγορίθμων είναι κοντά. Ο T-DS για 250 σημεία input δίνει 1,2 ταχύτερη απόκριση όμως όσο αυξάνεται το μέγεθος του input συνόλου, ο Plane Sweep Copy γίνεται ταχύτερος, με 2 φορές ταχύτερη απόκριση στα 1000 σημεία input.

Πλήθος Σημείων	**Plane Sweep Copy**	**T-DS**	**Πηλίκο Χρόνου Εκτέλεσης Plane Sweep Copy/T-DS**
Water, 250	8,911	7,629	1,168
Water, 500	13,326	15,556	0,857
Water, 750	13,271	21,158	0,627
Water, 1000	13,642	27,505	0,496

Πίνακας 6.1: Σύγκριση χρόνων εκτέλεσης Plane Sweep Copy και T-DS για το σύνολο υδατικών πόρων

Στο δεύτερο σύνολο οι διαφορές στο χρόνο εκτέλεσης είναι μεγαλύτερες. Ο Plane Sweep Stripes ξεκινάει με 1,25 ταχύτερη απόκριση και φτάνει να είναι έως και 7 φορες γρηγορότερος του T-DS.

Παρατηρείται ότι για το πρώτο σύνολο ο T-DS υπερτερεί για όλα τα μεγέθη του συνόλου input με έως και 3 φορές ταχύτερη απόκριση. Όσο αυξάνεται το πλήθος σημείων input αυξάνεται και ο χρόνος εκτέλεσης του T-DS ενώ ο Plane Sweep Copy φαίνεται να διατηρεί μια σταθερή χρονική απόκριση. Το γεγονός αυτό παρατηρείται και στο δεύτερο σύνολο όπου ο T-DS ξεκινάει με 2,5 φορές ταχύτερη απόκριση για 250 σημεία και για 1000 σημεία καταλήγει να είναι 2 φορές αργότερος του Plane Sweep Stripes.

Πλήθος Σημείων	**Plane Sweep Striped**	**T-DS**	**Πηλίκο Χρόνου Εκτέλεσης Plane Sweep Copy/T-DS**
Parks, 250	9,42	12,592	0,748
Parks, 500	10,37	23,849	0,435
Parks, 750	6,667	37,92	0,176
Parks, 1000	7,096	48,198	0,147

Πίνακας 6.2: Σύγκριση χρόνων εκτέλεσης Plane Sweep Stripes και T-DS για το σύνολο πάρκων

6.5.2 Σύγκριση παράλληλων υλοποιήσεων σε GPU και CPU

Για τη σύγκριση των παράλληλων υλοποιήσεων σε GPU και CPU χρησιμοποιήθηκαν τα 191.637 σημεία σιδηροδρομικών γραμμών (input) και 569.120 σημεία εθνικών οδών (training) της Βόρειας Αμερικής με αριθμό πλησιέστερων γειτόνων k=10. Για τους αλγορίθμους που χρησιμοποιούν λωρίδες χρησιμοποιήθηκε ο αυτόματος υπολογισμός του βέλτιστου αριθμού λωρίδων που για τα συγκεκριμένα δεδομένα είναι 240.

Για εξοικονόμηση χώρου θα χρησιμοποιηθούν συντομογραφίες των αλγορίθμων καθώς και των διάφορων μετρήσεων των πειραμάτων όπως φαίνονται στον Πίνακα 6.3. Τα αποτελέσματα των πειραμάτων φαίνονται στον Πίνακα 6.4.

Αλγόριθμος	**ΣΧΑ**	**ΧΥ**	**ΧΤ**	**ΧΑ**	**ΣΠΤ**	**ΜΟΠΤ**
BF	237,242	237,242	0,000	0,000	380.857.703	1987
PBF	322,967	305,507	0,000	17,46	380.857.703	1987
PS	2,713	2,604	0,109	0,000	11.464.091	60
PSC	2,259	2,151	0,109	0,000	11.464.091	60
PPSC	18,104	0,769	0,11	17,226	11.464.175	60
PPSC-PS	18,098	0,763	0,18	17,156	11.464.175	60
PSS	0,494	0,299	0,195	0,000	3.646138	19
PPSSI	21,113	0,956	0,202	17,954	3.646.138	19
PPSSI-PD	19,042	0,944	0,271	17,828	3.632.995	19
PPSSI-PS	18,308	0,938	0,274	17,095	3.646.104	19
PPSSI-PS-PD	18,663	0,94	0,336	17,386	3.633.022	19
PPSST	18,895	0,807	0,199	17,888	3.444.681	18
PPSST-PD	18,058	0,788	0,269	17	3.442.250	18
PPSST-PS	18,27	0,785	0,314	17,171	3.444.724	18
PPSST-PS-PD	18,626	0,79	0,354	17,482	3.442.264	18

Πίνακας 6.4: Χρόνοι εκτέλεσης αλγορίθμων για τα φυσικά σύνολα σιδηρόδρομων και εθνικών οδών

Συντομογραφία	**Αλγόριθμος**
BF	Σειριακός Brute Force
PBF	Παράλληλος Brute Force
PS	Plane Sweep με σειριακή ταξινόμηση δεικτών
PSC	Plane Sweep με αντιγραφή στοιχείων και σειριακή ταξινόμηση
PPSC	Παράλληλος Plane Sweep με αντιγραφή στοιχείων και σειριακή ταξινόμηση
PPSC-PS	Παράλληλος Plane Sweep με αντιγραφή στοιχείων και παράλληλη ταξινόμηση
PSS	Plane Sweep με λωρίδες, σειριακή ταξινόμηση και σειριακό διαχωρισμό σε λωρίδες βάση input
PPSSI	Παράλληλος Plane Sweep με λωρίδες, σειριακή ταξινόμηση και σειριακό διαχωρισμό σε λωρίδες βάση input
PPSSI-PD	Παράλληλος Plane Sweep με λωρίδες, σειριακή ταξινόμηση και παράλληλο διαχωρισμό σε λωρίδες βάση input
PSSI-PS	Παράλληλος Plane Sweep με λωρίδες, παράλληλη ταξινόμηση και σειριακό διαχωρισμό σε λωρίδες βάση input
PSSI-PS-PD	Παράλληλος Plane Sweep με λωρίδες, παράλληλη ταξινόμηση και παράλληλο διαχωρισμό σε λωρίδες βάση input
PSST	Παράλληλος Plane Sweep με λωρίδες, σειριακή ταξινόμηση και σειριακό διαχωρισμό σε λωρίδες βάση training
PSST-PD	Παράλληλος Plane Sweep με λωρίδες, σειριακή ταξινόμηση και παράλληλο διαχωρισμό σε λωρίδες βάση training
PSST-PS	Παράλληλος Plane Sweep με λωρίδες, παράλληλη ταξινόμηση και σειριακό διαχωρισμό σε λωρίδες βάση training
PSST-PS-PD	Παράλληλος Plane Sweep με λωρίδες, παράλληλη ταξινόμηση και παράλληλο διαχωρισμό σε λωρίδες βάση training
	Μετρήσεις
ΣΧΑ	Συνολικός χρόνος εκτέλεσης (sec)
ΧΥ	Χρόνος υπολογισμών (sec)
ΧΤ	Χρόνος ταξινόμησης (sec)
ΧΑ	Χρόνος αντιγραφής (sec)
ΣΠΤ	Πλήθος τοποθετήσεων σε σωρό (σύνολο)
ΜΟΠΤ	Πλήθος τοποθετήσεων σε σωρό (μέσος όρος ανά σημείο input)

Πίνακας 6.3: Συντομογραφίες αλγορίθμων

Τα αποτελέσματα αποδείχθηκε να είναι εκτός των προσδοκίων για χρονοβελτίωση. Σημαντικός παράγοντας αυτού είναι η ανεξαρτησία τής μνήμης της κάρτας γραφικών από εκείνη του κεντρικού υπολογιστή. Αυτό δημιουργεί την ανάγκη για μεταφορά δεδομένων προς και από την κάρτα γραφικών η οποία κλιμακώνεται σε σχέση με το πλήθος σημείων του συνόλου input και το k. Αυτή η καθυστέρηση φαίνεται να παίζει καθοριστικό ρόλο στη διαμόρφωση του τελικού χρόνου, κυρίως για τους Plane Sweep αλγορίθμους. Αξίζει, επίσης, να σημειωθεί ότι ούτε ο καθαρός χρόνος υπολογισμών υπερτερεί των αντίστοιχων σειριακών, παρά μόνο αυτός του παράλληλου Plane Sweep με αντιγραφή στοιχείων με και χωρίς παράλληλη ταξινόμηση. Απογοητευτική είναι και η επίδοση της παράλληλης ταξινόμησης η οποία σε όλες τις περιπτώσεις είναι πιο αργή από την αντίστοιχη σειριακή.

Στον Πίνακα 6.5 γίνεται σύγκριση των ομοειδών αλγορίθμων στη σειριακή και παράλληλη έκδοσή τους. Παρατηρείται ότι ο παράλληλος Brute Force είναι πιο αργός κατά 1/3 του σειριακού, ο παράλληλος Plane Sweep 8 φορές πιο αργός, ενώ ο Plane Sweep με λωρίδες 42 φορές αργότερος. Τέλος, η παράλληλη ταξινόμηση είναι κατα 2/5 αργότερη της σειριακής.

Σειριακός Αλγόριθμος	**Παράλληλος Αλγόριθμος**	**Χρόνος Εκτέλεσης Σειριακού**	**Χρόνος Εκτέλεσης Παράλληλου**	**Πηλίκο Χρόνου Εκτέλεσης Σειριακού/Παράλληλου**
BF	PBF	237,242	322,967	0,735
PSC	PPSC-PS	2,259	18,098	0,125
PSS	PPSSI-PD-PS	0,494	21,113	0,024
Σειριακή ταξινόμηση	Παράλληλη ταξινόμηση	0,11	0,18	0,611

Πίνακας 6.5: Σύγκριση χρόνων εκτέλεσης σειριακών και παράλληλων αλγορίθμων

Στον Πίνακα 6.6 γίνεται σύγκριση του καθαρού υπολογιστικού χρόνου. Παρατηρείται ότι οι αποκλίσεις είναι πολύ μικρότερες σε σχέση με το συνολικό χρόνο διότι δεν συνυπολογίζεται το κόστος αντιγραφής δεδομένων. Ο παράλληλος Brute Force και Plane Sweep με λωρίδες είναι πιο αργοί κατά 1/4 και 2/3 αντίστοιχα ενώ ο παράλληλος Plane Sweep δίνει περίπου 3 φορές ταχύτερο χρόνο.

Σειριακός Αλγόριθμος	**Παράλληλος Αλγόριθμος**	**Χρόνος Υπολογισμών Σειριακού**	**Χρόνος Υπολογισμών Παράλληλου**	**Πηλίκο Χρόνου Υπολογισμών Σειριακού/Παράλληλου**
BF	PBF	237,242	305,507	0,776
PSC	PPSC-PS	2,151	0,763	2,819
PSS	PPSSI-PD-PS	0,299	0,956	0,313

Πίνακας 6.6: Σύγκριση χρόνων υπολογισμών σειριακών και παράλληλων αλγορίθμων

Στον Πίνακα 6.7 φαίνονται οι συγκρίσεις συνολικής διάρκειας εκτέλεσης των παράλληλων αλγορίθμων σε κάρτα γραφικών και σε επεξεργαστή των (Χριστοφής, & Βασιλακόπουλος, 2018 [17]). Παρατηρείται ότι η υλοποίηση σε επεξεργαστή είναι εξαιρετικά ταχύτερη, κατά 7 φορές ο Brute Force, 50 φορές ο Plane Sweep και 333 φορές ο Plane Sweep με λωρίδες.

Αλγόριθμος	**Χρόνος Εκτέλεσης Επεξεργαστή**	**Χρόνος Εκτέλεσης Κάρτας Γραφικών**	**Πηλίκο Χρόνου Εκτέλεσης Επεξεργαστή/Κάρτας Γραφικών**
PBF	44,970	322,967	0,140
PSC-PS	0,359	18,098	0,02
PPSSI-PD-PS	0,062	21,113	0,003

Πίνακας 6.7: Σύγκριση χρόνων εκτέλεσης παράλληλων αλγορίθμων σε επεξεργαστή και κάρτα γραφικών

Κεφάλαιο 7

Οδηγίες εγκατάστασης

7.1 Πλατφόρμες και προγραμματιστικά εργαλεία

7.1.1 Visual Studio

Το Microsoft Visual Studio [6] είναι ένα ολοκληρωμένο περιβάλλον ανάπτυξης (IDE) από τη Microsoft. Χρησιμοποιείται για την ανάπτυξη προγραμμάτων υπολογιστών, καθώς και ιστότοπων (websites), εφαρμογών ιστού (web apps), υπηρεσιών ιστού (web services) και εφαρμογών για κινητά (mobile apps). Το Visual Studio χρησιμοποιεί πλατφόρμες ανάπτυξης λογισμικού της Microsoft όπως Windows API, Windows Forms, Windows Presentation Foundation, Windows Store και Microsoft Silverlight. Μπορεί να παράγει τόσο εγγενή κώδικα (native code) όσο και διαχειριζόμενο κώδικα (managed code).

Το Visual Studio περιλαμβάνει έναν επεξεργαστή κώδικα που υποστηρίζει το IntelliSense (το στοιχείο ολοκλήρωσης κώδικα) καθώς και την αναδιαμόρφωση κώδικα. Το ενσωματωμένο πρόγραμμα εντοπισμού σφαλμάτων (integrated debugger) λειτουργεί τόσο ως πρόγραμμα εντοπισμού σφαλμάτων σε επίπεδο πηγής όσο και ως εργαλείο εντοπισμού σφαλμάτων σε επίπεδο μηχανής. Άλλα ενσωματωμένα εργαλεία περιλαμβάνουν έναν αναλυτή κώδικα (code profiler), σχεδιαστή για την κατασκευή εφαρμογών GUI (designer for building GUI applications), σχεδιαστές ιστοσελίδων (web designer), σχεδιαστής τάξεων (class designer) και σχεδιαστής σχήματος βάσης δεδομένων (database schema designer). Δέχεται προσθήκες που επεκτείνουν τη λειτουργικότητα σχεδόν σε κάθε επίπεδο - συμπεριλαμβανομένης της προσθήκης υποστήριξης για συστήματα διαχείρισης πηγαίου κώδικα (όπως το Subversion και το Git) και την προσθήκη νέων συνόλων εργαλείων, όπως οι συντάκτες (editors) και οι οπτικοί σχεδιαστές (visual designers) για συγκεκριμένου τομέα γλώσσες (domain-specific languages) ή εργαλείων άλλες πτυχές του κύκλου ζωής ανάπτυξης λογισμικού.

Το Visual Studio υποστηρίζει 36 διαφορετικές γλώσσες προγραμματισμού και επιτρέπει στον συντάκτη κώδικα και το πρόγραμμα εντοπισμού σφαλμάτων να υποστηρίζει (σε διαφορετικούς βαθμούς) σχεδόν οποιαδήποτε γλώσσα προγραμματισμού. Οι ενσωματωμένες γλώσσες περιλαμβάνουν C, C++, C++ / CLI, Visual Basic .NET, C#, F#, JavaScript, TypeScript, XML, XSLT, HTML και CSS. Υποστήριξη για άλλες γλώσσες, όπως Python, Ruby, Node.js και M είναι, μεταξύ άλλων, διαθέσιμη μέσω προσθηκών. Java (και J#) υποστηρίχθηκαν στο παρελθόν.

Η πιο βασική έκδοση του Visual Studio, η Κοινοτική έκδοση (Community Edition), διατίθεται δωρεάν. Το σύνθημα για την έκδοση Visual Studio Community είναι ”Δωρεάν, πλήρως εξοπλισμένο IDE για μαθητές, ανοιχτό κώδικα και μεμονωμένους προγραμματιστές”.

7.1.2 CUDA

Το CUDA (Compute Unified Device Architecture) [1] είναι μια παράλληλη πλατφόρμα υπολογιστών και μοντέλο διεπαφής προγραμματισμού εφαρμογών (API) που δημιουργήθηκε από τη Nvidia. Επιτρέπει στους προγραμματιστές λογισμικού και τους μηχανικούς λογισμικού να χρησιμοποιούν μια μονάδα επεξεργασίας γραφικών με δυνατότητα CUDA (GPU) για επεξεργασία γενικού σκοπού - μια προσέγγιση που ονομάζεται GPGPU (υπολογισμός γενικής χρήσης για μονάδες επεξεργασίας γραφικών). Η πλατφόρμα CUDA είναι ένα επίπεδο λογισμικού που παρέχει άμεση πρόσβαση στο εικονικό σύνολο εντολών της GPU και στα παράλληλα υπολογιστικά στοιχεία, για την εκτέλεση υπολογιστικών πυρήνων.

Η πλατφόρμα CUDA έχει σχεδιαστεί για να λειτουργεί με γλώσσες προγραμματισμού όπως C, C++ και Fortran. Αυτή η προσβασιμότητα διευκολύνει τους ειδικούς στον παράλληλο προγραμματισμό να χρησιμοποιούν πόρους GPU, σε αντίθεση με προηγούμενα API όπως το Direct3D και το OpenGL, τα οποία απαιτούσαν προηγμένες δεξιότητες στον προγραμματισμό γραφικών. Οι GPU που υποστηρίζονται από CUDA υποστηρίζουν επίσης πλαίσια προγραμματισμού όπως OpenACC και OpenCL, και HIP με τη σύνταξη τέτοιου κώδικα στο CUDA. Όταν το CUDA παρουσιάστηκε για πρώτη φορά από τη Nvidia, το όνομα ήταν ένα αρκτικόλεξο για την Compute Unified Device Architecture , αλλά στη συνέχεια η Nvidia έριξε την κοινή χρήση του ακρωνύμιου.

7.2 Εγκατάσταση και εκτέλεση κώδικα

Η ανάπτυξη και εκτέλεση κώδικα πραγματοποιήθηκε στο προγραμματιστικό περιβάλλον Microsoft Visual Studio 2019 Community Edition. Για τον προγραμματισμό της GPU χρησιμοποιήθηκε η CUDA εργαλειοθήκη (CUDA toolkit) της έκδοσης 10.2 για λειτουργικό σύστημα Windows 10. Η μόνη απαίτηση υλικού ώστε να υπάρχει δινατότητα αναπαραγωγής της εργασίας είναι η GPU να είναι κατασκευασμένη από την εταιρεία Nvidia.

Η αναπαραγωγή της εργασίας είναι εξαιρετικά απλή. Αρχικά κατεβάζουμε το Visual Studio από την ιστοσελίδα https://visualstudio.microsoft.com/ . Επιλέγουμε το λειτουργικό σύστημα που χρησιμοποιεί ο υπολογιστής μας. Η συγκεκριμένη εργασία χρησιμοποιεί την Community έκδοση και υποστηρίζεται από τις εκδόσεις των χρονολογιών 2017 και 2019. Στη συνέχεια, κατεβάζουμε κι εγκαθιστούμε την τελευταία έκδοση CUDA toolkit από την επίσημη ιστοσελίδα της Nvidia https://developer.nvidia.com/cuda-downloads με βάση τα χαρακτηριστικά του συστήματός μας. Τέλος, ανοίγουμε το Solution/PlaneSweepParallel.sln αρχείο μέσω του Visual Studio όπου περιέχονται όλες οι απαραίτητες ρυθμίσεις του περιβάλλοντος.

Η εκτέλεση του κώδικα μπορεί να πραγματοποιηθεί με δύο τρόπους. Ο πρώτος πραγματοποιείται μέσω του Visual Studio. Ακολουθούμε το μονοπάτι επιλογών από τη γραμμή μενού:

Debug → *PlaneSweepParallelProperties..* → *ConfigurationProperties* → *Debugging* → *CommandArguments*

Στο σημείο αυτό μπορούμε να ρυθμίσουμε τις παραμέτρους εκτέλεσης. Ένα παράδειγμα παραμέτρων είναι το παρακάτω:

10 ../Data/randomData/1k.bin ../Data/randomData/1M.bin 0 0 0 0 01011001

Εφόσον επιλεγούν η παράμετροι, επιλέγουμε το κουμπί Local Windows Debugger στο πάνω μέρος του περιβάλλοντος και η εκτέλεση πραγματοποιείται σε τερματικό του Visual Studio.

Ο δεύτερος τρόπος περιλαμβάνει τη γραμμή εντολών. Κατευθυνόμαστε στο φάκελο που περιέχει τα αρχεία της εργασίας μέσω της γραμμής εντολών και εκτελούμε το εκτελέσιμο (.exe) αρχείο. Μια ενδεικτική εντολή εκτέλεσης είναι η παρακάτω:

Solution/x64/Release/PlaneSweepParallel.exe 10 Data/randomData/1k.bin
Data/randomData/1M.bin 0 0 0 0 01011001

Τα ορίσματα των αλγόρίθμων αναλύονται ως εξής:

1. Το πλήθος k των κοντινότερων γειτόνων προς εύρεση
2. Το αρχείο του σημειοσυνόλου input
3. Το αρχείο του σημειοσυνόλου training
4. Η ακρίβεια σύγκρισης των αποτελεσμάτων (προεραιτικό)
5. Το πλήθος λωρίδων (προεραιτικό)
6. Αποθήκευση των αποτελεσμάτων σε αρχείο κειμένου (0/1, προεραιτικό)
7. Σύγκριση αποτελεσμάτων με τα αποτελέσματα του πρώτου αλγορίθμου (0/1, προεραιτικό)
8. Ενεργοποίηση/Απενεργοποίηση αλγορίθμων (ακολουθία bit 15 ψηφίων 0 ή 1, π.χ. 011001100111100, προεραιτικό)

Κεφάλαιο 8

Επίλογος

8.1 Περιορισμοί

8.1.1 Απουσία διανυσμάτων στους πυρήνες

Μία εμφανής έλλειψη της CUDA είναι η απουσία διανυσμάτων στους πυρήνες της συσκευής (device kernels). Για την αντιμετώπισή του δημιουργήθηκε δομή struct όπου αποθηκεύεται ο δείκτης στην αρχή του πίνακα και το μέγεθος του σε έναν ακέραιο χρησιμοποιώντας αριθμητική δεικτών μνήμης για την προσπέλαση των στοιχείων. Αυτό όμως απαιτεί τα στοιχεία να βρίσκονται σε συνεχόμενες θέσεις μνήμης.

8.1.2 Διάνυσμα που περιέχει δείκτες στη μνήμη συσκευής

Για την αποθήκευση του σωρού γειτόνων για κάθε σημείο input ήταν απαραίτητη η χρήση διανύσματος συσκευής από δείκτες σε μνήμη συσκευής (διανύσματα συσκευής). Αυτό όμως δημιουργούσε προβλήματα κατά τη διάρκεια της αντιγραφής διότι δεν ήταν επιτρεπτό διάνυσμα του κεντρικού υπολογιστή (host vector) να περιέχει δείκτη στη μνήμη της συσκευής. Το πρόβλημα αυτό αντιμετωπίστηκε αποθηκεύοντας τις σωρούς συσκευής σε ξεχωριστό διάνυσμα στον κεντρικό υπολογιστή. Έτσι οι σωροι συσκευής αντιγράφονται αρχικά σε διανύσματα κεντρικού υπολογιστή και έπειτα δημιουργείται νέο διάνυσμα κεντρικού υπολογιστή από τους σωρούς αυτούς. Αυτό όμως έχει σχεδόν διαπλάσιες απαιτήσεις σε μνήμη.

8.2 Σύνοψη και συμπεράσματα

Τα αποτελέσματα της έρευνας απέρριψαν τις προσδοκίες για βελτίωση του χρόνου απόκρισης των παράλληλων αλγορίθμων με τη χρήση της CUDA Thrust για το χωρικό ερώτημα εύρεσης ”Όλων των k πλησιέστερων γειτόνων”. Η παραλληλοποίηση όχι μόνο δε δίνει χρονοβελτίωση σε σχέση με τους αντίστοιχους σειριακούς αλλά επιβραδύνει σημαντικά την εκτέλεση. Γι αυτήν την καθυστέρηση κυρίως ευθύνεται η ανεξαρτησία της μνήμης της κάρτας γραφικών από αυτή του κεντρικού υπολογιστή. Το γεγονός αυτό καθιστά απαραίτητη τη μεταφορά όλων των απαραίτητων δεδομένων για εκτέλεση στη μνήμη της κάρτας γραφικών και μετά το πέρας της, την αντιγραφή

των αποτελεσμάτων στον κεντρικό υπολογιστή. Για τη διαδικασία της πειραματικής αξιολόγησης κρίθηκε πρέπον να παραλειφθεί ο χρόνος δέσμευσης και αρχικοποίησης μνήμης συσκευής εξαιτίας της απουσίας ενεργειών βελτιστοποίησης. Το κόστος αρχικοποίησης κλιμακώνεται γραμμικά με το μέγεθος του σημειοσυνόλου input για το οποίο αποκλειστικά ευθύνεται η δέσμευση της απαραίτητης μνήμης για το δοχείο που συγκρατεί τους k πλησιέστερους γείτονες για κάθε σημείο input. Ο χρόνος αντιγραφής των αποτελεσμάτων αυξάνεται γραμμικά με το πλήθος σημείων του συνόλου input και το πλήθος των πλησιέστερων γειτόνων προς αναζήτηση k. Για περίπου 200.000 σημεία στο σύνολο input και k=10 η αντιγραφή αποτελεσμάτων για τον Plane Sweep με λωρίδες καταλαμβάνει το 95% του χρόνου εκτέλεσης. Εξετάζοντας μόνο τον καθαρό χρόνο υπολογισμών οι παράλληλος Brute Force και Plane sweep με λωρίδες υστερούν και πάλι ένταντι των σειριακών, ενώ ο παράλληλος Plane Sweep με αντιγραφή στοιχείων δίνει 3 φορές καλύτερη χρονική απόκριση. Οι παράλληλοι αλγόριθμοι σε επεξεργαστή υπερισχύουν σε πολύ μεγάλο βαθμό των αντίστοιχων υλοποιήσεων σε κάρτα γραφικών που φτάνει και τις 300 φορές για το Plane Sweep με λωρίδες. Ωστόσο, οι επιδόσεις τους είναι συγκρίσιμες με παρόμοιους αλγορίθμους σε κάρτες γραφικών για μικρά μεγέθη συνόλου input.

8.3 Μελλοντική έρευνα

Η εν λόγω εργασία έχει περιθώρια εξέλιξης όσων αφορά την χρονική απόκριση των αλγορίθμων. Μια σημαντική ανάγκη για βελτίωση αποτελεί η βελτιστοποίηση της διαδικασίας αποθήκευσης των δεδομένων. Στη συγκεκριμένη υλοποίηση γίνεται αποθήκευση των δεδομέων στη κύρια μνήμη με συνέπεια να απαιτούνται πολλές περισσότερες είσοδοι/έξοδοι στη μνήμη προκειμένου τα δεδομένα να μεταφερθούν πρώτα στη διαμοιραζόμενη μνήμη κι έπειτα στη μνήμη συσκευής. Μπορεί να αποφευχθεί το κόστος μεταφοράς μεταξύ πινάκων κεντρικού υπολογιστή με σελιδοποίηση (pageable) και καρφιτσωμένων (pinned) με την απευθείας δέσμευση pinned μνήμης για την αποθήκευση των πινάκων. Η pinned μνήμη κεντρικού υπολογιστή μπορεί να δεσμευτεί στο CUDA C/C++ χρησιμοποιώντας το *cudaMallocHost*() ή το *cudaHostAlloc*() και αποδεσμεύεται με το *cudaFreeHost*().

Όσον αφορά τις μεταφορές δεδομένων, επίσης, μπορεί να βελτιστοποιηθεί ο αλγόριθμος Σάρωσης Επιπέδου με λωρίδες (Plane Sweep Striped) με μια υλοποίηση όπου το πλήθος λωρίδων θα επηρεάζει το πρόβλημα σε μία αναλογία με την έρευνα [17]. Αν αυτό δεν είναι εφικτό τότε αξίζει να ερευνηθεί αν ο προσεγγιστικός υπολογισμός βέλτιστου πλήθους λωρίδων αποτελεί αξιόπιστη επιλογή.

Στην κατεύθυνση της μείωσης του χρόνου απόκρισης θα μπορούσε να γίνει η χρήση CUDA αντί για Thrust όπου θα υπάρχει μεγαλύτερος έλεγχος των νημάτων CUDA και της διαμοίρασης των εργασιών από τον προγραμματιστή. Η αυτοματοποίηση απλοποιεί σε μεγάλο βαθμό το έργο του προγραμματιστή αφήνοντας τον να επικεντρωθεί στην υλοποίηση, γράφωντας πιο ευανάγνωστο και συμμαζεμένο κώδικα. Ωστόσο οι επιλογές της βιβλιοθήκης είναι προσεγγιστικές αφήνοντας ανοιχτά ενδεχόμενα για μη βέλτιστη αξιοποίηση των πόρων της κάρτας γραφικών. Συνεπώς, για να βελτιστοποιηθεί η χρονική απόκριση μιας υλοποίησης το πλήθος νημάτων CUDA και ο διαμοιρασμός εργασιών σε αυτά θα πρέπει να είναι προσαρμοσμένα στις ανάγκες του εκάστοτε

προβλήματος, γεγονός που δεν μπορεί να επιτευχθεί με τη Thrust.

Υπάρχει, επίσης, η δυνατότητα επέκτασης για την αντιμετώπιση μεγάλων συνόλων δεδομένων που δεν χωράν εξ' ολοκλήρου στη μνήμη των καρτών γραφικών κατά αντιστοιχία με την υλοποίηση εξωτερικής μνήμης της έρευνας [17]. Μία τέτοια υλοποίηση και με τη χρήση μικρού μεγέθους συνόλου input για να αποφεύγεται το κόστος αντιγραφής αποτελεσμάτων ίσως εκμεταλευόταν τις υπολογιστικές δυνάμεις της GPU σε μεγαλύτερο βαθμό.

Η εκμετάλλευση της πληροφορίας γειτνίασης μεταξύ των σημείων του συνόλου input είναι μία προσέγγιση που δεν έχει εξεταστεί. Η αναζήτηση γειτόνων εκτελείται ανεξάρτητα για κάθε σημείο input. Με βάση τη λογική, όμως, δύο γειτονικά σημεία του συνόλου input θα γειτνιάζουν περίπου με τους ίδιους γείτονες από το training. Συνεπώς η εύρεση γειτόνων για τα σημεία input θα μπορούσε να βασίζεται στο σωρό γειτονικών input σημείων χρησιμοποιώντας τον ως αρχική λύση και αντικαθιστώντας ένα μικρό υποσύνολο αυτού, με νέους γείτονες. Για να είναι ωφέλιμη μια τέτοια υλοποίηση θα πρέπει ο χρόνος υπολογισμών αποδοχής/απόρριψης προηγούμενων γειτόνων να είναι μικρότερος από το χρόνο αναζήτησης.

Παράρτημα Ι

Σειριακοί Αλγόριθμοι

Algorithm 1 Brute Force AkNN

```
1:  Procedure Process (array of M points input, array of M points training, array of M max heaps result)
2:  for i: 0..M-1 do
3:      for j: 0..N-1 do
4:          distanceSquared = dist(Input[i], Training[j])^2;
5:          InputId = Input[i].Id;          //Id του σημείου input (1..M)
6:          heap = Result[InputId-1];   //σωρός μεγίστου για το σημείο Input[i]
7:          if distanceSquared < heap.Top.distanceSquared then
8:              heap.pop();   //αφαίρεση κορυφής
9:              neighbor = (&Training[j], distanceSquared);   //διεύθυνση και απόσταση γείτονα
10:             heap.push(neighbor);   //προσθήκη γείτονα και αναδιοργάνωση σωρού
11:         end if
12:     end for
13: end for
14: End Procedure
```

Algorithm 2 Plane Sweep AkNN

```
1:  Procedure Process (array of M points input, array of M points training, array of M max heaps result)
2:  InputCopy = Copy(Input);   //αντιγραφή Input
3:  TrainingCopy = Copy(Training);   //αντιγραφή Training
4:  SortX(InputCopy);   //ταξινόμηση κατά x
5:  SortX(TrainingCopy);
6:  for InputCopy[i] (i: 0..M-1) do
7:     InputId = InputCopy[i].Id;   //Id του σημείου input (1..M)
8:     heap = Result[InputId-1];   //σωρός μεγίστου για το σημείο Input[i]
          //βρες σημείο training με x >= InputCopy[i].x
9:     tnext = FindGreaterEqualX(InputCopy[i], TrainingCopy);   //Id του σημείου input (1..M)
10:    tprev = Previous(tnext) ;   // προηγούμενο του tnext κατά x
11:    stopLow = (tprev == tnext);   //έλεγχος αν βρισκόμαστε στην αρχή του training
12:    stopHigh = (tnext == N);   //έλεγχος αν βρισκόμαστε στο τέλος του training
13:    while (stopLow = false OR stopHigh = false) do
14:       if (stopLow = false) then
15:          distanceSquared = dist(InputCopy[i], tprev)^2;
16:          dx = InputCopy[i].x – tprev.x;   //διαφορά κατά x
17:          if (distanceSquared < heap.Top.distanceSquared) then
18:             heap.pop();   //αφαιρούμε την κορυφή και προσθέτουμε νέο γείτονα
19:             neighbor = (&tprev, distanceSquared);
20:             heap.push(neighbor);
21:          else if (dx2 >= heap.Top.distanceSquared) then
22:             stopLow = true; //αν αυτό ισχύει, μπορούμε να διακόψουμε τη σάρωση προς χαμηλότερο x
23:          end if
24:          if (stopLow = false) then
25:             tprev = Previous(tprev);   //βρες το σημείο training με χαμηλότερο x και κάνε έλεγχο ορίων
26:             if (tprev < 0) then
27:                stopLow= true;   //περάσαμε το πρώτο σημείο κατά x
28:             end if
29:          end if
30:       end if
31:       if (stopHigh = false) then
32:          distanceSquared = dist(InputCopy[i], tnext)^2;
33:          dx = InputCopy[i].x – tnext.x;   //διαφορά κατά x
34:          if (distanceSquared < heap.Top.distanceSquared) then
35:             heap.pop();   //αφαιρούμε την κορυφή και προσθέτουμε νέο γείτονα
36:             neighbor = (&tnext, distanceSquared);
37:             heap.push(neighbor);
38:          else if (dx2 >= heap.Top.distanceSquared) then
39:             stopHigh = true; //σταματούμε τη σάρωση προς υψηλότερο x
40:          end if
41:          if (stopHigh = false) then
42:             tnext = Previous(tnext);
43:             if (tnext >= N) then
44:                stopHigh= true;   //περάσαμε το τελευταίο σημείο κατά x
45:             end if
46:          end if
47:       end if
48:    end while
49: end for
50: End Procedure
```

Algorithm 3 Striped Plane Sweep AkNN

```
1: Procedure Process (array of M points input, array of M points training, approximate number of stripes NSEst,
   dynamic array of stripes Stripe, array of M max heaps result)
        //αρχή δημιουργίας λωρίδων
2: InputCopy = Copy(Input);  //αντιγραφή Input
3: TrainingCopy = Copy(Training);  //αντιγραφή Training
4: SortY(InputCopy);  //ταξινόμηση κατά y
5: SortY(TrainingCopy);
6: InputStripeSize = M/NSEst +1;  //αριθμός σημείων input που ιδανικά θα περιλαμβάνει η κάθε λωρίδα
7: InputStart = 0;  //πρώτο σημείο input για τρέχουσα λωρίδα
8: InputEnd = InputStart + InputStripeSize);  //πρώτο σημείο input για επόμενη λωρίδα
9: TrainingStart = 0;  //πρώτο σημείο training για τρέχουσα λωρίδα
10: exit = false;
11: iStripe = 0;  //αριθμός λωρίδας
12: repeat
13:    while (InputEnd.y = Previous(InputEnd).y) do
14:       InputEnd = Next(InputEnd);  //προχώρησε το InputEnd ώστε να έχει διαφορετικό y από το προηγούμενό του
15:    end while
16:    Stripe[iStripe].Input = [InputStart, InputEnd-1];  //έχουμε τα οριακά σημεία του input για την τρέχουσα λωρίδα
17:    SortX(Stripe[iStripe].Input);  //ταξινόμησε το input της τρέχουσας λωρίδας κατά x
18:    minY = min(InputStart.y, TrainingStart.y));  //βρες το ελάχιστο y για την τρέχουσα λωρίδα
19:    maxY = minY);
20:    if (TrainingStart < N) then
21:       TrainingEnd = FindGreaterY(Previous(InputEnd).y, TrainingCopy);  //βρες ποιο θα είναι το πρώτο σημείο
   training της επόμενης λωρίδας
22:       Stripe[iStripe].Training = [TrainingStart, TrainingEnd-1];  //έχουμε τα οριακά σημεία του training για την
   τρέχουσα λωρίδα
23:       SortX(Stripe[iStripe].Training);  //ταξινόμησε το training της τρέχουσας λωρίδας κατά x
24:       maxY = max(Previous(TrainingEnd).y, Previous(InputEnd).y);  //το άνω όριο y της τρέχουσας λωρίδας
25:       TrainingStart = TrainingEnd;  //αυτό θα είναι το πρώτο σημείο training για την επόμενη λωρίδα
26:    else
27:       Stripe[iStripe].Training = [];  //κενό training για αυτή τη λωρίδα
28:       maxY = Previous(InputEnd).y;
29:    end if
30:    Stripe[iStripe].Boundaries = (minY, maxY);  //αποθήκευσε τα όρια y της τρέχουσας λωρίδας
31:    if (InputEnd < M) then
32:       InputStart = InputEnd;  //πρώτο σημείο input της επόμενης λωρίδας
33:       InputEnd = min(M, InputStart + InputStripeSize);  //πρώτο σημείο input της μεθεπόμενης λωρίδας
34:    else
35:       exit = true;  //η δημιουργία λωρίδων ολοκληρώθηκε
36:    end if
37:    iStripe = iStripe + 1;
38: until (exit = false)
```

```
39: NS = Length(Stripes);   //τελικό Πλήθος λωρίδων που δημιουργήσαμε
        //τέλος δημιουργίας λωρίδων
40: for Stripe[i] (i: 0..NS-1) do
41:   for Stripe[i].Input[j] do
42:     TrainingStripe = Stripe[i];   // ξεκινάμε την αναζήτηση πάντα από την τρέχουσα λωρίδα
43:     InputId = Stripe[i].Input[j].Id;
44:     heap = Result[InputId − 1];
45:     PlaneSweepStripe(Stripe[i].Input[j], TrainingStripe, heap);  // εκτελούμε τον αλγόριθμο PlaneSweepStripe για
    τη λωρίδα TrainingStripe
46:     TrainingStripePrev = Previous(TrainingStripe);   //προηγούμενη λωρίδα κατά y
47:     TrainingStripeNext = Next(TrainingStripe);   //επόμενη λωρίδα κατά y
48:     stopLow = (TrainingStripePrev < 0);   //έλεγχος ορίων
49:     stopHigh = (TrainigStripeNext >= NS); //αρχίζουμε και ψάχνουμε στις γειτονικές λωρίδες απομακρυνόμενοι
    κατά y
50:     while (stopLow = false OR stopHigh = false) do
51:       if (stopLow = false) then
52:         dyLow = Stripe[i].Input[j].y – TrainingStripePrev.MaxY;   //διαφορά κατά y από το άνω όριο της προη-
    γούμενης λωρίδας
        //ελέγχουμε αν έχει νόημα η αναζήτηση σε αυτή τη λωρίδα
53:         if (dyLow2 < heap.Top.distanceSquared) then
54:           PlaneSweepStripe(Stripe[i].Input[j], TrainingStripePrev, heap);
55:           TrainingStripePrev = Previous(TrainingStripePrev);
56:           stopLow = (TrainingStripePrev < 0);
57:         else
58:           stopLow = true;
59:         end if
60:       end if
61:       if (stopHigh = false) then
62:         dyHigh = TrainingStripeNext.MinY - Stripe[i].Input[j].y;   //διαφορά κατά y από το κάτω όριο της επό-
    μενης λωρίδας
        //ελέγχουμε αν έχει νόημα η αναζήτηση σε αυτή τη λωρίδα
63:         if (dyLow2 < heap.Top.distanceSquared) then
64:           PlaneSweepStripe(Stripe[i].Input[j], TrainingStripeNext, heap );
65:           TrainingStripeNext = Next(TrainingStripeNext);
66:           stopHigh = (TrainingStripeNext >= NS);
67:         else
68:           stopHigh = true;
69:         end if
70:       end if
71:     end while
72:   end for
73: end for
74: End Procedure
```

Βιβλιογραφία

[1] About Cuda | NVIDIA Developer. https://developer.nvidia.com/about-cuda. Ημερομηνία πρόσβασης Ιούνιος 24, 2020.

[2] CUDA C++ Programming Guide. https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#introduction. Ημερομηνία πρόσβασης Ιούνιος 10, 2020.

[3] Exploring the GPU Architecture. https://nielshagoort.com/2019/03/12/exploring-the-gpu-architecture/. Ημερομηνία πρόσβασης Μάιος 4, 2020.

[4] How to Optimize Data Transfers in CUDA C/C++. https://developer.nvidia.com/blog/how-optimize-data-transfers-cuda-cc/. Ημερομηνία πρόσβασης Ιούνιος 25, 2020.

[5] Thrust::CUDA Toolkit Documentation. https://docs.nvidia.com/cuda/thrust/index.html. Ημερομηνία πρόσβασης Ιούνιος 10, 2020.

[6] Visual Studio IDE. https://visualstudio.microsoft.com. Ημερομηνία πρόσβασης Ιούνιος 24, 2020.

[7] Zhenhong Du, Xianwei Zhao, Xinyue Ye, Jingwei Zhou, Feng Zhang και Renyi Liu. An Effective High-Performance Multiway Spatial Join Algorithm with Spark. *ISPRS Int. J. Geo-Information*, 6(4):96, 2017.

[8] Shenshen Liang, Ying Liu, Cheng Wang και Liheng Jian. A CUDA-based Parallel Implementation of K-Nearest Neighbor Algorithm.

[9] Gang Mei, Liangliang Xu και Nengxiong Xu. Accelerating Adaptive IDW Interpolation Algorithm on a Single GPU. *CoRR*, abs/1511.02186, 2015.

[10] Panagiotis Moutafis, George Mavrommatis, Michael Vassilakopoulos και Spyros Sioutas. Efficient processing of all-k-nearest-neighbor queries in the MapReduce programming framework. *Data Knowl. Eng.*, 121:42–70, 2019.

[11] Peter Ogden, David B. Thomas και Peter R. Pietzuch. AT-GIS: Highly Parallel Spatial Query Processing with Associative Transducers. Στο *Proceedings of the 2016 International Conference on Management of Data, SIGMOD Conference 2016, San Francisco, CA, USA, June*

*26 - July 01, 2016*Fatma Özcan, Georgia Koutrika και Sam Madden, επιμελητές, σελίδες 1041–1054. ACM, 2016.

[12] George Roumelis, Antonio Corral, Michael Vassilakopoulos και Yannis Manolopoulos. New plane-sweep algorithms for distance-based join queries in spatial databases. *GeoInformatica*, 20(4):571–628, 2016.

[13] Polychronis Velentzas, Michael Vassilakopoulos και Antonio Corral. A Partitioning GPU-based Algorithm for Processing the k Nearest-Neighbor Query.

[14] Jianting Zhang και Simin You. Large-Scale Geospatial Processing on Multi-Core and Many-Core Processors: Evaluations on CPUs, GPUs and MICs. *CoRR*, abs/1403.0802, 2014.

[15] Jianting Zhang, Simin You και Le Gruenwald. Parallel online spatial and temporal aggregations on multi-core CPUs and many-core GPUs. *Inf. Syst.*, 44:134–154, 2014.

[16] Jianting Zhang, Simin You και Le Gruenwald. Towards GPU-Accelerated Web-GIS for Query-Driven Visual Exploration. Στο *Web and Wireless Geographical Information Systems - 15th International Symposium, W2GIS 2017, Shanghai, China, May 8-9, 2017, Proceedings*David Brosset, Christophe Claramunt, Xiang Li και Tianzhen Wang, επιμελητές, τόμος 10181 στο *Lecture Notes in Computer Science*, σελίδες 119–136, 2017.

[17] Θεόδωρος Χριστοφής και Μιχαήλ Βασιλακόπουλος. Παράλληλη Επεξεργασία Απαιτητικών Ερωτημάτων σε Χωρικές Βάσεις Δεδομένων. 2018.

Συντομογραφίες

ΚΕΜ	Κεντρική Μονάδα Επεξεργασίας
ΜΕΓ	Μονάδα Επεξεργασίας Γραφικών
ΟkΠΓ	Όλοι οι k Πλησιέστεροι Γείτονες
kΠΓ	k Πλησιέστεροι Γείτονες
AIDW	Adaptive Inverse Distance Weighting
AkNN	All k nearest neighbors
ALU	Arithmetic/Logic Unit
AoaS	Array of aligned Structures
API	Application Program Interface
CPU	Central Processing Unit
CUDA	Compute Unified Device Architecture
DDR	Double Data Rate
GIS	Geographic Information system
GDDR	Graphics Double Data Rate
GPU	Graphics Processing Unit
GPGPU	General Purpose Graphics Processing Unit
HPC	High-Performance Computing
IoT	Internet of Things
kNN	k nearest neighbors
OLAP	Online Analytical Processing
OpenMP	Open Multi-Processing
SIMD	Single Instruction Multiple Data
SM	Streaming Multiprocessor
SoA	Structure of Arrays
STL	Standard Template Library
STXXL	Standard Template Library for Extra Large Data Sets
TBB	Thread Building Blocks
VPU	Vector Processing Unit

Ορολογία - Γλωσσάρι

Ελληνικός όρος	Αγγλικός όρος
δείκτης	pointer
διάνυσμα	vector
είσοδος	input
εκπαίδευση	training
επαναλήπτης	iterator
επίπεδο	layer
κάρτα γραφικών	device
καρφιτσωμένη μνήμη	pinned memory
κεντρικός υπολογιστής	host
ΚΜΕ	CPU
κρυφή μνήμη	cache
ΜΕΓ	GPU
μείωση	reduction
μεμονωμένη συνάρτηση	unary function
μνήμη δυνατότητα σελιδοποίησης	pageable memory
νήμα	thread
πίνακας	array
πολυεπεγεργαστές ροής	sreaming multiprocessors
πυρήνας	kernel
ροή εντολών	instruction throughput
σάρωση	scan
σάρωση επιπέδου	plane sweep
σάρωση επιπέδου με αντιγραφή στοιχείων	plane sweep copy
σάρωση επιπέδουμε λωρίδες	plane sweep stripes
σύνδεση	join
συστάδα	cluster
ταχύτητα μετάδοσης δεδομένων από τη μνήμη	memory bandwidth
ταξινόμηση	sorting
ωμή βία	brute force