ΠΑΝΕΠΙΣΤΗΜΙΟ ΘΕΣΣΑΛΙΑΣ

ΠΟΛΥΤΕΧΝΙΚΗ ΣΧΟΛΗ

ΤΜΗΜΑ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ

Επεξεργασία Χωρικών Ερωτημάτων με χρήση GPU

Διπλωματική Εργασία

Πατέρας Ιωάννης

Επιβλέπων: Βασιλακόπουλος Μιχαήλ

Σεπτέμβριος 2025

ΠΑΝΕΠΙΣΤΗΜΙΟ ΘΕΣΣΑΛΙΑΣ

ΠΟΛΥΤΕΧΝΙΚΗ ΣΧΟΛΗ

ΤΜΗΜΑ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ

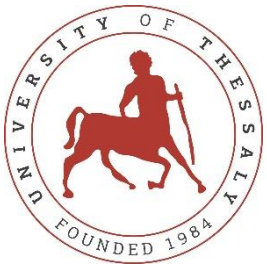
Επεξεργασία Χωρικών Ερωτημάτων με χρήση GPU

Διπλωματική Εργασία

Πατέρας Ιωάννης

Επιβλέπων: Βασιλακόπουλος Μιχαήλ

Σεπτέμβριος 2025



UNIVERSITY OF THESSALY

SCHOOL OF ENGINEERING

DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING

Spatial Queries Processing using GPU

Diploma Thesis

Pateras Ioannis

Supervisor: Vassilakopoulos Michael

September 2025

Εγκρίνεται από την Επιτροπή Εξέτασης:

Επιβλέπων	**Βασιλακόπουλος Μιχαήλ** Καθηγητής, Τμήμα Ηλεκτρολόγων Μηχανικών και Μηχανικών Υπολογιστών, Πανεπιστήμιο Θεσσαλίας
Μέλος	**Φεύγας Αθανάσιος** Ε.ΔΙ.Π., Τμήμα Ηλεκτρολόγων Μηχανικών και Μηχανικών Υπολογιστών, Πανεπιστήμιο Θεσσαλίας
Μέλος	**Θάνος Γεώργιος** Ε.ΔΙ.Π., Τμήμα Ηλεκτρολόγων Μηχανικών και Μηχανικών Υπολογιστών, Πανεπιστήμιο Θεσσαλίας

ΥΠΕΥΘΥΝΗ ΔΗΛΩΣΗ ΠΕΡΙ ΑΚΑΔΗΜΑΪΚΗΣ ΔΕΟΝΤΟΛΟΓΙΑΣ ΚΑΙ ΠΝΕΥΜΑΤΙΚΩΝ ΔΙΚΑΙΩΜΑΤΩΝ

Με πλήρη επίγνωση των συνεπειών του νόμου περί πνευματικών δικαιωμάτων, δηλώνω ρητά ότι η παρούσα διπλωματική εργασία, καθώς και τα ηλεκτρονικά αρχεία και πηγαίοι κώδικες που αναπτύχθηκαν ή τροποποιήθηκαν στα πλαίσια αυτής της εργασίας, αποτελούν αποκλειστικά προϊόν προσωπικής μου εργασίας, δεν προσβάλλουν οποιασδήποτε μορφής δικαιώματα διανοητικής ιδιοκτησίας, προσωπικότητας και προσωπικών δεδομένων τρίτων, δεν περιέχουν έργα/εισφορές τρίτων για τα οποία απαιτείται άδεια των δημιουργών/δικαιούχων και δεν είναι προϊόν μερικής ή ολικής αντιγραφής, οι πηγές δε που χρησιμοποιήθηκαν περιορίζονται στις βιβλιογραφικές αναφορές και μόνον και πληρούν τους κανόνες της επιστημονικής παράθεσης. Τα σημεία όπου έχω χρησιμοποιήσει ιδέες, κείμενο, αρχεία ή/και πηγές άλλων συγγραφέων αναφέρονται ευδιάκριτα στο κείμενο με την κατάλληλη παραπομπή και η σχετική αναφορά περιλαμβάνεται στο τμήμα των βιβλιογραφικών αναφορών με πλήρη περιγραφή. Δηλώνω επίσης ότι τα αποτελέσματα της εργασίας δεν έχουν χρησιμοποιηθεί για την απόκτηση άλλου πτυχίου. Αναλαμβάνω πλήρως, ατομικά και προσωπικά, όλες τις νομικές και διοικητικές συνέπειες που δύναται να προκύψουν στην περίπτωση κατά την οποία αποδειχθεί, διαχρονικά, ότι η εργασία αυτή ή τμήμα της δεν μου ανήκει διότι είναι προϊόν λογοκλοπής.

Ο Δηλών

Πατέρας Ιωάννης

DISCLAIMER ON ACADEMIC ETHICS AND INTELLECTUAL PROPERTY RIGHTS

Being fully aware of the implications of copyright laws, I expressly state that this diploma thesis, as well as the electronic files and source codes developed or modified in the course of this thesis, are solely the product of my personal work and do not infringe any rights of intellectual property, personality and personal data of third parties, do not contain work / contributions of third parties for which the permission of the authors / beneficiaries is required and are not a product of partial or complete plagiarism, while the sources used are limited to the bibliographic references only and meet the rules of scientific citing. The points where I have used ideas, text, files and / or sources of other authors are clearly mentioned in the text with the appropriate citation and the relevant complete reference is included in the bibliographic references section. I also declare that the results of the work have not been used to obtain another degree. I fully, individually and personally undertake all legal and administrative consequences that may arise in the event that it is proven, in the course of time, that this thesis or part of it does not belong to me because it is a product of plagiarism.

The Declarant

Pateras Ioannis

Ευχαριστίες ή Σχόλια

Πρωτίστως θα ήθελα να ευχαριστήσω τον καθηγητή και επιβλέποντα της διπλωματικής μου, κύριο Βασιλακόπουλο Μιχαήλ για τη βοήθεια και καθοδήγηση που μου προσέφερε.

Ευχαριστώ επίσης τους κυρίους Φεύγα Αθανάσιο και Θάνο Γεώργιο, μέλη της Επιτροπής Εξέτασης.

Τέλος, θα ήθελα να ευχαριστήσω τον κύριο Πολυχρόνη Βελέντζα για τη βοήθεια και τις οδηγίες που μου παρείχε στη συγγραφή του κώδικα.

Διπλωματική Εργασία

Επεξεργασία Χωρικών Ερωτημάτων με χρήση GPU

Πατέρας Ιωάννης

Περίληψη

Οι αλγόριθμοι που μελετούν και επεξεργάζονται μεγάλα σύνολα δεδομένων στον χώρο παρουσιάζουν μεγάλο ενδιαφέρον τόσο στην επιστημονική έρευνα, όσο και σε πλήθος εφαρμογών στη βιομηχανία. Η γρήγορη και αποτελεσματική υλοποίηση αυτών των αλγορίθμων είναι μέγιστης σημασίας για την επιστήμη των δεδομένων. Οι κάρτες γραφικών είναι ένα τυπικό παράδειγμα ενός συστήματος μαζικά παράλληλου προγραμματισμού και η χρήση τους είναι συνήθης για την επιτάχυνση του κώδικα. Οι κάρτες γραφικών λειτουργούν καλύτερα σε προγράμματα που είναι δεκτικά σε παραλληλοποίηση και δέχονται ως είσοδο μεγάλα σύνολα δεδομένων. Για αυτόν τον λόγο η επεξεργασία μεγάλων ερωτημάτων με δεδομένα στον χώρο χρησιμοποιώντας κάρτες γραφικών είναι μια ιδιαίτερα δημοφιλής προσέγγιση. Αυτή η εργασία επικεντρώνεται στην ανάπτυξη ενός τέτοιου αλγόριθμου και των παραλλαγών του, όπου λαμβάνοντας ως είσοδο δύο σύνολα δεδομένων στον δισδιάστατο ή τρισδιάστατο χώρο, θα υπολογίζει τα k κοντινότερα ζευγάρια μεταξύ τους, ο οποίος θα εκτελείται σε κάρτα γραφικών, προσπαθώντας να εκμεταλλευτεί όλες τις δυνατότητές τους.

Λέξεις-κλειδιά:

Κάρτα γραφικών, μεγάλα σύνολα δεδομένων, παράλληλος προγραμματισμός, K κοντινότερα ζεύγη, C++.

Diploma Thesis

Spatial Queries Processing using GPU

Pateras Ioannis

Abstract

Algorithms that study and process large data sets in the field are of great interest both in scientific research and in a variety of applications in industry. The fast and efficient implementation of these algorithms is of utmost importance for data science. Graphics cards are a typical example of a massively parallel programming system, and their use is common to speed up code. Graphics cards work best in programs that are receptive to parallelism and accept large data sets as input. That's why processing large queries with data in space using graphics cards is a particularly popular approach. In this paper we will focus on the development of such an algorithm and its variations, where taking two sets of data in two-dimensional or three-dimensional space as input, it will calculate the k closest pairs to each other, which will be executed on a graphics card, trying to exploit all their potential.

Keywords:

GPU, Big Data Sets, parallel programming, K nearest pairs, C++.

Πίνακας περιεχομένων

Κατάλογος εικόνων

Κατάλογος γραφημάτων

Κατάλογος πινάκων

Συντομογραφίες

βλπ	βλέπε
κ.λπ.	και λοιπά
κ.ο.κ	και ούτω καθεξής
GPU	Graphics Processing Unit
KNP-DLB	K Nearest Pairs Distance List Buffer
KNP-MHDLB	K Nearest Pairs Max Heap Distance List Buffer
DBFS	Disk Brute-Force Simple
DBF	Disk Brute-Force
DSPP	Disk Symmetric Progression Partitioning
DSPP+	Improved Disk Symmetric Progression Partitioning
DSPP+P	Improved Disk Symmetric Progression Partitioning with Pinned memory
DSPP+PEA	Improved Disk Symmetric Progression Partitioning with Pinned memory with Enhanced Assignment

Κεφάλαιο 1 Εισαγωγή

Η υλοποίηση αλγορίθμων που μελετούν δεδομένα στον χώρο με χρήση κάρτας γραφικών παρουσιάζει μεγάλο ενδιαφέρον, καθώς σε πολλές περιπτώσεις μπορεί να είναι ιδιαίτερα αποτελεσματική. Οι κάρτες γραφικών περιέχουν πολύ μεγάλο αριθμό πυρήνων σε σχέση με τους επεξεργαστές και για αυτό σε ορισμένες περιπτώσεις προτιμώνται. Επίσης η μνήμη που χρησιμοποιούν οι κάρτες γραφικών είναι πιο γρήγορη από την κύρια μνήμη και αν χρησιμοποιηθεί σωστά ακολουθώντας τα κατάλληλα μοτίβα προσπέλασης μπορεί να επιταχύνει επιπλέον τους υπολογισμούς. Ωστόσο η μεταφορά δεδομένων από την κύρια μονάδα αποθήκευσης προς τη μνήμη της κάρτα γραφικών δεν είναι κάτι που γίνεται αυτόματα, όπως στον παραδοσιακό προγραμματισμό. Τα δεδομένα μεταφέρονται προς σ' αυτή τη μνήμη με ευθύνη του προγραμματιστή την κατάλληλη στιγμή επιχειρώντας να αξιοποιήσουν στο μέγιστο τον διαθέσιμο χώρο. Τα σύγχρονα, μεγάλα σύνολα δεδομένων μπορούν να υπερβούν τη χωρητικότητα της μνήμης. Θα πρέπει επομένως τα δεδομένα να χωριστούν σε κομμάτια και να μεταφέρονται συντονισμένα, ώστε τέτοιες υλοποιήσεις να είναι εφαρμόσιμες, ανεξάρτητα από τον όγκο των δεδομένων.

1.1 Αντικείμενο της διπλωματικής

Η παρούσα εργασία μελετά έναν αλγόριθμο που λαμβάνοντας ως είσοδο δύο σύνολα από σημεία εντοπίζει τα k κοντινότερα ζευγάρια σημείων ανάμεσα σε αυτά τα δύο σύνολα. Διάφορες παραλλαγές του αλγορίθμου προγραμματίστηκαν με τη διεπαφή εφαρμογών Cuda που επιτρέπει τη χρήση συγκεκριμένων μονάδων επεξεργασίας γραφικών για την επιτάχυνση του κώδικα. Οι παραλλαγές υλοποιήθηκαν ώστε κάθε μία να βελτιστοποιεί την προηγούμενη σε αλγοριθμικό ή προγραμματιστικό επίπεδο με κριτήρια την ευελιξία του κώδικα στο μέγεθος της εισόδου και την επίδοση.

Πιο συγκεκριμένα η παρούσα διπλωματική εργασία εστιάζει σε μεγάλα σύνολα δεδομένων στο δισδιάστατο και τρισδιάστατο χώρο και θα χρησιμοποιηθούν διάφορες τεχνικές διαμέρισης (partitioning), ώστε τα μεγάλα αυτά σύνολα δεδομένων να χωρούν στη μνήμη της κάρτας γραφικών και να βελτιωθεί η ευελιξία και η κλιμακωσιμότητα του

κώδικα. Ακόμα, εφόσον εστιάζει σε δισδιάστατα και τρισδιάστατα δεδομένα, εκτός από brute force θα εφαρμοστεί και μία τεχνική χωρικής υποδιαίρεσης, η οποία λειτουργεί αποδοτικά σε τέτοιου είδους δεδομένα, ώστε να αποφευχθούν περιττοί υπολογισμοί και να μειωθεί ο χρόνος εκτέλεσης. Επίσης θα χρησιμοποιηθούν δύο χαρακτηριστικά που προσφέρει η Cuda και ονομάζονται pinned memory και concurrent kernel execution και βοηθούν στην επικάλυψη της μεταφοράς δεδομένων με υπολογισμούς, πράγμα που είναι ιδιαίτερα κρίσιμο στην περίπτωση αυτή, όπου εξαιτίας μίας τεχνικής διαμέρισης που χρησιμοποιείται ο αριθμός των αναγνώσεων (και άρα και των μεταφορών στη μνήμη της κάρτας γραφικών) ενός από τα δύο σύνολα δεδομένων αυξάνεται δραματικά (βλπ ενότητα 4.4 όπου γίνεται αναλυτική επεξήγηση), το οποίο επηρεάζει την επίδοση. Επιπλέον θα παρουσιαστούν δύο διαφορετικές υλοποιήσεις για τις λίστες που χρησιμοποιούν τα νήματα για να αποθηκεύσουν την πληροφορία για τα k κοντινότερα ζευγάρια (μία distance list buffer και μία max-heap distance list buffer). Στο τέλος αξιολογούνται πειραματικά όλες οι μέθοδοι και ερμηνεύονται τα αποτελέσματα.

1.1.1 Συνεισφορά

Στην παρούσα εργασία παρουσιάζονται τα ακόλουθα:

- Μελετώνται μεθοδολογίες επίλυσης παρόμοιων προβλημάτων.
- Προτείνεται μια brute force τεχνική.
- Εφαρμόζεται διαμέριση και στα δύο σύνολα δεδομένων.
- Προτείνεται μία τεχνική χωρικής υποδιαίρεσης.
- Μελετώνται πειραματικά οι επιδόσεις των αλγορίθμων σε συνθετικά τρισδιάστατα δεδομένα και σε μεγάλα, πραγματικά σύνολα δεδομένων στον δισδιάστατο χώρο. Επίσης για τα συνθετικά δεδομένα δοκιμάζονται διάφορα μεγέθη συνόλων δεδομένων, καθώς και τιμές της παραμέτρου k και παρατηρείται πως αυτό επηρεάζει τον χρόνο εκτέλεσης.
- Ερμηνεύονται τα πειραματικά αποτελέσματα και εντοπίζεται ποια από τις παραλλαγές που υλοποιήθηκαν είναι η καλύτερη.

1.2 Οργάνωση τόμου

Η οργάνωση της εργασίας είναι ως εξής:

Το Κεφάλαιο 2 περιέχει το απαραίτητο γνωστικό υπόβαθρο σχετικά με τον προγραμματισμό με κάρτες γραφικών.

Στο Κεφάλαιο 3 παρουσιάζονται σχετικές εργασίες.

Στο Κεφάλαιο 4 παρουσιάζονται οι αλγόριθμοι που χρησιμοποιήθηκαν.

Στο Κεφάλαιο 5 αξιολογούνται πειραματικά οι αλγόριθμοι, ερμηνεύονται τα αποτελέσματα και εντοπίζεται η αποδοτικότερη υλοποίηση.

Τέλος, στο Κεφάλαιο 6 αναφέρονται τα συμπεράσματά και παρουσιάζονται τα μελλοντικά σχέδια.

Κεφάλαιο 2 Γνωστικό υπόβαθρο

Σ' αυτό το κεφάλαιο αναλύονται μερικές βασικές έννοιες και δίνονται κάποιες τεχνικές πληροφορίες σχετικά με τον προγραμματισμό με κάρτες γραφικών.

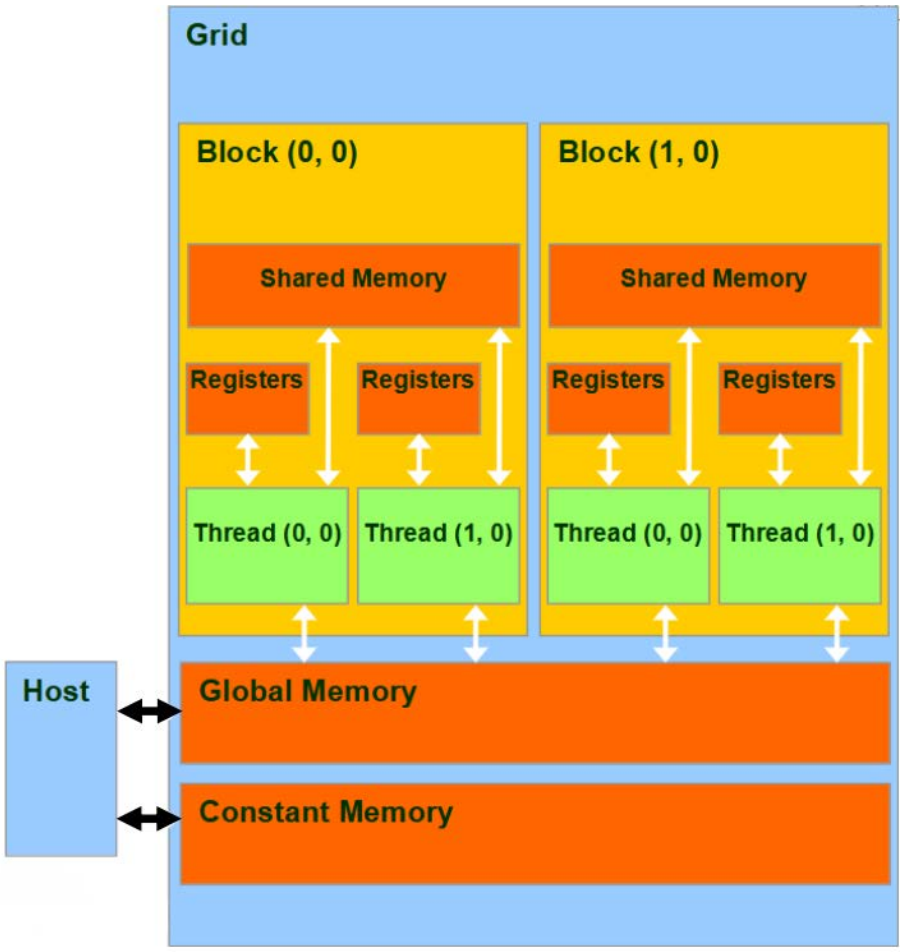
2.1 Παράλληλος προγραμματισμός

Ο παράλληλος προγραμματισμός [14],[15] έχει γνωρίσει μεγάλη ανάπτυξη τα τελευταία χρόνια εξαιτίας των πλεονεκτημάτων που παρουσιάζει στην επίδοση, καθώς και στην ικανότητά του να μειώνει το ενεργειακό κόστος. Όσο τα transistors γίνονται μικρότερα μπορούμε ν' αυξήσουμε τη συχνότητα, στην οποία λειτουργούν οι επεξεργαστές. Ταυτόχρονα όμως αυξάνεται και η παραγόμενη θερμότητα, πράγμα που θέτει κάποιους φυσικούς περιορισμούς. Μπορούμε να μειώσουμε τη συχνότητα και να αυξήσουμε τον αριθμό των transistors επιτυγχάνοντας τις ίδιες επιδόσεις και μειώνοντας σημαντικά το ενεργειακό κόστος. Για να επιτευχθεί αυτό πρέπει να προσθέσουμε επιπλέον επεξεργαστικές μονάδες, οι οποίες θα λειτουργούν παράλληλα. Ευτυχώς πολλοί, δημοφιλείς αλγόριθμοι είναι δεκτικοί σε παραλληλισμούς (με προφανή ή μη τρόπο) και οι σύγχρονες εφαρμογές λαμβάνουν μεγάλες εισόδους και εκτελούν πολλούς υπολογισμούς, που τους κάνει ιδανικούς υποψήφιους για παράλληλη εκτέλεση. Βέβαια ο παράλληλος προγραμματισμός έχει μερικούς περιορισμούς, καθώς πολλές εφαρμογές μπορεί να περιέχουν σειριακά τμήματα, που δεν μπορούν να παραλληλοποιηθούν, η εκτέλεση των οποίων θέτει ένα κάτω όριο στην επίδοση (νόμος του Amdahl) [17],[19]. Επίσης σε πολλές περιπτώσεις η επιτάχυνση που επιτυγχάνεται δεν είναι γραμμική ως προς το πλήθος των επεξεργαστικών μονάδων, που χρησιμοποιούνται. Ωστόσο περιορισμοί τέτοιου είδους δεν είναι ικανοί να υπερνικήσουν τα οφέλη του παράλληλου προγραμματισμού και για αυτό είναι πλέον αποδεκτός στην επιστήμη των υπολογιστών ότι υπολογιστικά απαιτητικές εφαρμογές δεν μπορούν να λειτουργήσουν χωρίς αυτόν.

2.2 Τεχνικές πληροφορίες σχετικά με τον προγραμματισμό με κάρτες γραφικών

Ο προγραμματισμός με Cuda [11],[12] είναι αρκετά διαφορετικός από τον συνηθισμένο όχι μόνο επειδή ο παράλληλος προγραμματισμός είναι διαφορετικός, αλλά και γιατί ο προγραμματισμός με κάρτες γραφικών απαιτεί ορισμένες γνώσεις επάνω στην αρχιτεκτονική τους.

Στην Cuda τα νήματα που εκτελούν τον κώδικα είναι οργανωμένα σε ομάδες που ονομάζονται blocks. Τα νήματα που ανήκουν στο ίδιο block εκτελούνται στον ίδιο streaming multiprocessor, έχουν πρόσβαση σε κοινή shared memory και μπορούν να συγχρονιστούν εύκολα [1]. Ο αριθμός των νημάτων που μπορούν να υπάρχουν σε ένα block είναι περιορισμένος και εξαρτάται από την κάρτα γραφικών.



Εικόνα 2-1: Οργάνωση GPU

Τα block οργανώνονται σε ένα grid. Το grid έχει και αυτό περιορισμένο αριθμό νημάτων.

Όταν ο προγραμματιστής θελήσει να εκτελέσει κώδικα στην κάρτα γραφικών θα πρέπει να δημιουργήσει έναν kernel και να περάσει σε αυτόν, ως παραμέτρους, το πως θέλει να είναι οργανωμένα γεωμετρικά το block και το grid.

Στην Cuda ο κώδικας που εκτελείται στην κάρτα γραφικών (device) χρησιμοποιεί διαφορετικό χώρο διευθύνσεων από αυτόν που εκτελείται στον επεξεργαστή (host) [1]. Ο προγραμματιστής είναι υπεύθυνος για τη μεταφορά των κομματιών που θέλει από τον host στο device και αντίστροφα.

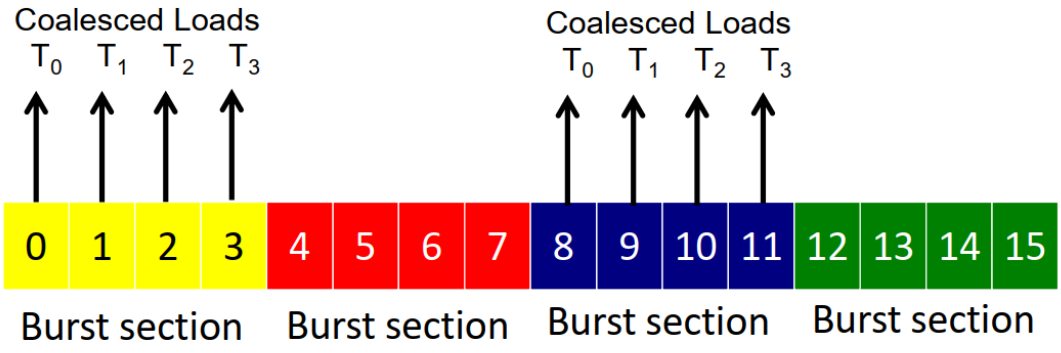
Οι κάρτες γραφικών δεν έχουν κύρια μονάδα αποθήκευσης. Στο χαμηλότερο επίπεδο της ιεραρχίας μνήμης του βρίσκεται μια μνήμη που ονομάζεται global memory και είναι DRAM [2],[5]. Γενικά η Global memory θα μπορούσε να χαρακτηριστεί ως μια γρήγορη και μεγάλου μεγέθους μνήμη. Ωστόσο μπορεί να προκαλέσει μεγάλη καθυστέρηση και στις σύγχρονες εφαρμογές που διαχειρίζονται μεγάλο όγκο δεδομένων δεν επαρκεί.

Για το πρόβλημα της έλλειψης χώρου η λύση είναι τα δεδομένα που υπάρχουν στον host να χωριστούν σε κομμάτια και να μεταφέρονται στο device. Αν τα δεδομένα τροποποιούνται στο device, τότε θα πρέπει να μεταφερθούν πίσω στον host προτού άλλα πάρουν τη θέση τους. Ο προγραμματιστής πρέπει να είναι όσο το δυνατόν πιο συντηρητικός με τις μεταφορές, καθώς είναι ιδιαίτερα χρονοβόρες. Γενικά είναι καλή τακτική οι αντιγραφές δεδομένων να επικαλύπτονται όσο γίνεται με υπολογισμούς χρησιμοποιώντας τη λειτουργικότητα που προσφέρει η Cuda για ασύγχρονες μεταφορές δεδομένων [4],[9].

Για το πρόβλημα της καθυστέρησης στην global memory υπάρχουν δύο γενικές κατευθύνσεις.

Η πρώτη είναι ότι ο κώδικας θα πρέπει να είναι γραμμένος με τέτοιον τρόπο, ώστε όλα τα νήματα όταν χρειάζεται να διαβάσουν δεδομένα από την global memory να διαβάζουν από γειτονικές διευθύνσεις. Πρέπει δηλαδή να υπάρχει coalescing [3]. Ο λόγος που αυτό είναι πιο αποτελεσματικό είναι γιατί κάθε φορά που ζητείται ένα στοιχείο από τη μνήμη διαβάζονται μαζί με αυτό και τα γειτονικά του. Αυτό ονομάζεται bursting. Επομένως αν τα δεδομένα που ζητούνται βρίσκονται σε κοντινές θέσεις στη μνήμη θα

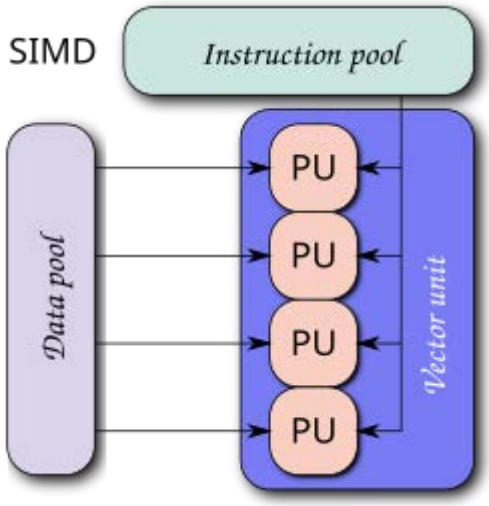
διαβαστούν πιο γρήγορα και επίσης θα χρειαστούν λιγότερα αιτήματα προς τη μνήμη πράγμα που μειώνει την καθυστέρηση επιπλέον.



Εικόνα 2-2: Coalescing

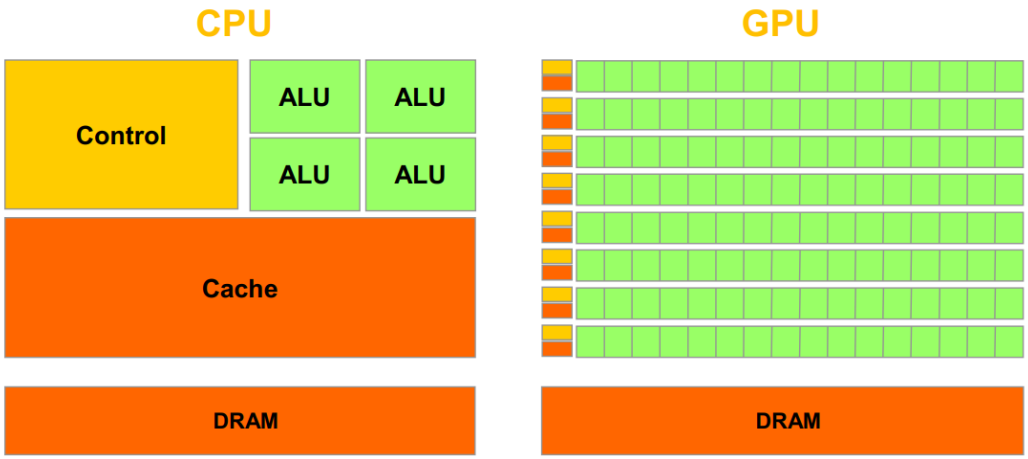
Η δεύτερη κατεύθυνση είναι ο προγραμματιστής να μεταφέρει τα δεδομένα από την global memory σε μία άλλη μνήμη, που ονομάζεται shared memory. Η shared memory [2],[8] είναι μία μνήμη cache που ελέγχεται από το λογισμικό. Είναι πιο γρήγορη από την global memory είναι όμως και πιο μικρή. Επίσης κάθε block έχει πρόσβαση μόνο στο δικό του κομμάτι shared memory και μόνο τα νήματα που ανήκουν σε αυτό μπορούν να διαβάσουν και να τροποποιήσουν δεδομένα από αυτή. Το γεγονός αυτό δημιουργεί πρόβλημα, όταν τα νήματα θέλουν να τροποποιήσουν τα δεδομένα που διαβάζουν, ειδικά αν αυτά τα δεδομένα πρέπει να τροποποιηθούν από νήματα που ανήκουν σε διαφορετικά blocks. Επίσης εάν οι αλλαγές που έγιναν στα δεδομένα πρέπει να συνεχίσουν να υπάρχουν και μετά την εκτέλεση όλων των νημάτων του αντίστοιχου block θα πρέπει να μεταφερθούν πάλι πίσω στην global memory. Η χρήση μεγάλου μέρους shared memory ανά block μπορεί να μειώσει τον αριθμό των νημάτων που τρέχουν ενεργά κάθε στιγμή.

Ένα ακόμα σημαντικό χαρακτηριστικό που έχουν οι κάρτες γραφικών είναι ότι κάνουν πράξεις ως single instruction multiple data (SIMD) [3]. Αυτό σημαίνει ότι πολλά νήματα εκτελούν παράλληλα την ίδια πράξη για διαφορετικά δεδομένα.



Εικόνα 2-3: SIMD

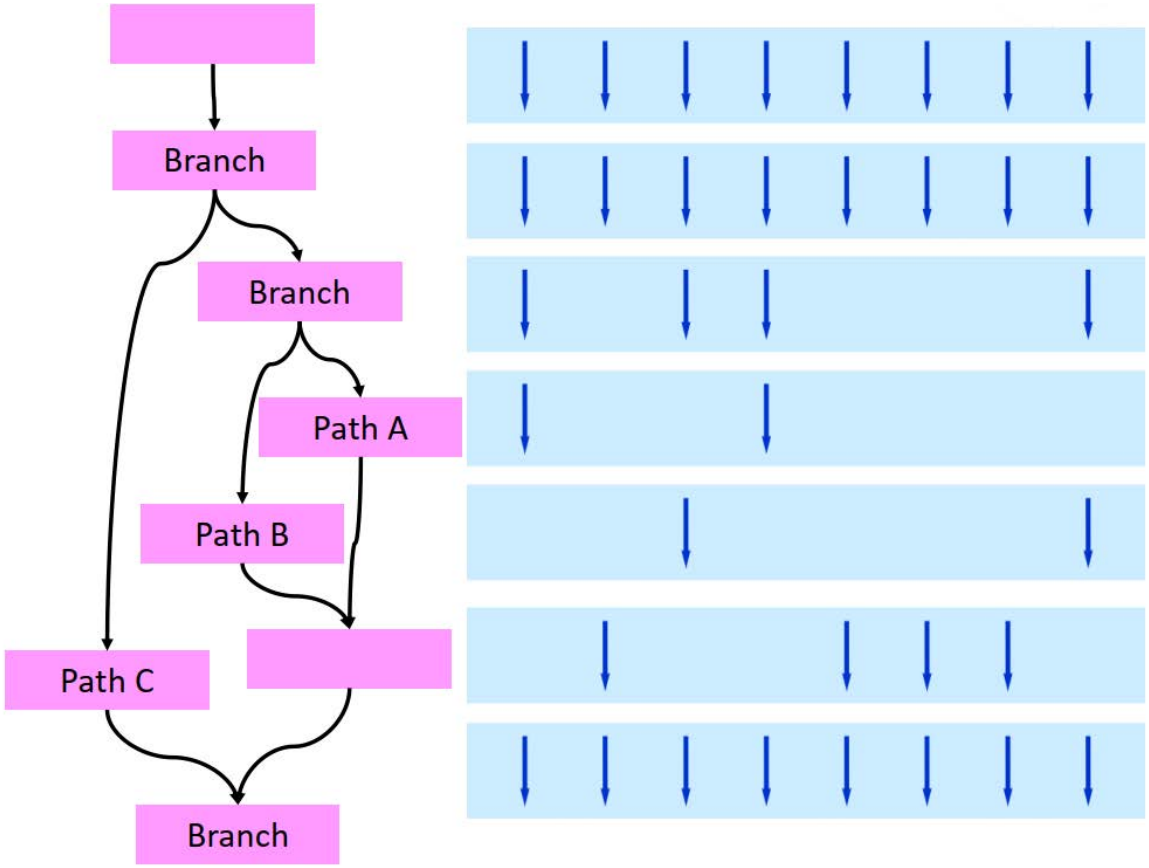
Ο λόγος που λήφθηκε η συγκεκριμένη σχεδιαστική απόφαση είναι για οικονομία χώρου καθώς έτσι δεν χρειάζεται κάθε ALU να έχει τη δική της μονάδα ελέγχου. Ο χώρος που εξοικονομείται γεμίζει με ALUs.



Εικόνα 2-4: Σύγκριση CPU-GPU

Τα νήματα χωρίζονται σε ομάδες που λέγονται warps [3],[7]. Τα νήματα που ανήκουν στο ίδιο warp εκτελούν τις ίδιες πράξεις. Το warp περιέχει έως και 32 νήματα. Εάν τα νήματα που ανήκουν σε ένα warp φτάσουν σε μία εντολή διακλάδωσης και ακολουθήσουν διαφορετικά μονοπάτια εκτέλεσης, τότε το warp θα διασπαστεί. Αυτή η συμπεριφορά δεν είναι επιθυμητή και ονομάζεται divergence. Ο κώδικας πρέπει να

γράφεται ώστε να μην υπάρχει divergence ή να εμφανίζεται σε όσο το δυνατόν μετέπειτα στάδιο [3],[6]. Αυτό όμως δεν είναι πάντα δυνατό.



Εικόνα 2-5: Divergence

Όλα τα παραπάνω λήφθηκαν υπόψιν κατά τη σχεδίαση των προγραμμάτων.

2.3 Εργαλεία που χρησιμοποιούνται στην εργασία

Ο κώδικας των αλγορίθμων σ' αυτή την εργασία είναι γραμμένος σε C++ [13] χρησιμοποιώντας cuda. Επίσης χρησιμοποιείται thrust [16], που περιέχει πολλούς παράλληλους αλγόριθμους και δομές δεδομένων.

Κεφάλαιο 3 Σχετικές εργασίες

Στο κεφάλαιο αυτό μελετώνται σχετικές εργασίες που επιλύουν το πρόβλημα των k κοντινότερων γειτόνων (K nearest neighbors), που είναι παρόμοιο με το θέμα που παρουσιάζεται στη συγκεκριμένη εργασία.

3.1 K nearest neighbors

Στο [10] οι συγγραφείς μελετούν τον αλγόριθμο knn εστιάζοντας σε δισδιάστατα και τρισδιάστατα δεδομένα για μεγάλα σύνολα δεδομένων. Υλοποιούν πέντε διαφορετικές παραλλαγές του αλγορίθμου, όπου χρησιμοποιούν διαμέριση και για το reference και για το query dataset. Επίσης παρουσιάζουν δύο διαφορετικές προτάσεις για τις λίστες που αποθηκεύονται οι knn μία distance list buffer και μία max-heap distance list buffer. Υλοποιούν brute force εκδοχές του αλγορίθμου, αλλά και δύο τεχνικές χωρικής υποδιαίρεσης. Ακόμη χρησιμοποιούν δύο χαρακτηριστικά που λέγονται pinned merory και concurrent kernel execution για να επικαλύψουν τις μεταφορές δεδομένων με υπολογισμούς και να κρύψουν το κόστος της αντιγραφής δεδομένων. Στο τέλος αξιολογούν πειραματικά τις μεθόδους τους για πραγματικά και συνθετικά δεδομένα και για διάφορα μεγέθη και για διάφορες τιμές της παραμέτρου k και καταλήγουν στο συμπέρασμα ότι η παραλλαγή DSPP+P είναι πιο αποδοτική ειδικά όταν χρησιμοποιούνται μεγάλα k και/ή μεγάλα σύνολα δεδομένων.

Στο [20] οι συγγραφείς υλοποιούν μία μέθοδο όπου εστιάζει κυρίως σε δεδομένα πολλών διαστάσεων και χωρίζεται σε τρείς φάσεις. Στην πρώτη φάση υπολογίζονται όλες οι αποστάσεις ανάμεσα στα train και test datasets τα οποία οργανώνονται ως δισδιάστατοι πίνακες A, B με μεγέθη mxd και nxd αντίστοιχα με m και n να είναι ο αριθμός των σημείων που εμπεριέχονται στα train και test datasets και d να είναι ο αριθμός των διαστάσεων. Ακόμα χρησιμοποιείται ένας πίνακας C μεγέθους mxn, όπου αποθηκεύονται οι αποστάσεις που υπολογίζονται. Κάθε νήμα υπολογίζει ένα στοιχείο του C. Ένα block υπολογίζει μια τετράγωνη περιοχή του πίνακα C. Ακόμα για να εκμεταλλευτούν τη shared memory κάθε block φορτώνει σε αυτή σε τετράγωνα κομμάτια τους πίνακες A και B και κάθε νήμα υπολογίζει κάθε φορά ένα κομμάτι της απόστασης, που του έχει ανατεθεί,

μέχρι να υπολογιστεί ολόκληρη η απόσταση. Στη δεύτερη φάση υλοποιείται στην κάρτα γραφικών ένας αλγόριθμός, ο οποίος είναι radix sort για να επιλεγούν οι k κοντινότεροι γείτονες. Στην τρίτη φάση, η οποία υλοποιείται στη CPU, λαμβάνονται οι αποφάσεις για το σε ποια κατηγορία ανήκει το κάθε σημείο. Η πειραματική αξιολόγηση γίνεται με δύο πειράματα, ανάμεσα σε τρεις αλγορίθμους, όπου ο πρώτος είναι αυτός που περιγράφεται στο άρθρο, ο δεύτερος ένας αλγόριθμος που τρέχει στον επεξεργαστή και τρίτος ένας προσεγγιστικός αλγόριθμος που τρέχει στον επεξεργαστή. Το συμπέρασμα που προκύπτει είναι ότι στο πρώτο πείραμα επιτυγχάνεται 32,61 φορές επιτάχυνση σε σχέση με τον αλγόριθμό που τρέχει στον επεξεργαστή και 14,98 φορές επιτάχυνση σε σχέση με τον προσεγγιστικό αλγόριθμό που τρέχει στον επεξεργαστή. Στο δεύτερο πείραμα οι επιταχύνσεις είναι 34,91 φορές και 15,36 φορές αντίστοιχα.

Στο [21] οι συγγραφείς παρουσιάζουν μια μέθοδο υλοποιημένη σε OpenCL κατά την οποία οι πράξεις που εκτελούντα για τον υπολογισμό μιας απόστασης χωρίζονται και εκτελούνται παράλληλα. Για αυτό υπάρχει και ένα επιπλέον στάδιο, κατά το οποίο τα αποτελέσματα αυτών των υπολογισμών αθροίζονται για να υπολογιστούν οι τελικές αποστάσεις. Οι τελικοί υπολογισμοί για την ανάδειξη των k κοντινότερων γειτόνων γίνονται στην CPU. Ο αλγόριθμος μελετάται πειραματικά σε διάφορα σύνολα δεδομένων με διάφορα μεγέθη και το συμπέρασμα που προκύπτει είναι ότι η επίδοση βελτιώνεται σημαντικά σε όλες τις περιπτώσεις.

Στο [22] οι συγγραφείς υλοποιούν μια παραλλαγή του knn που εστιάζει στην κλιμακωσιμότητα του αλγορίθμου ως προς τα train και test datasets. Φορτώνουν στη μνήμη της κάρτας γραφικών ένα κομμάτι του train και ένα κομμάτι του test και για κάθε σημείο του test υπολογίζουν τους k κοντινότερους γείτονες ελέγχοντας για κάθε κομμάτι του test όλα τα κομμάτια του train και κρατώντας κάθε φορά τους μέχρι στιγμής k κοντινότερους γείτονες για κάθε σημείο του test. Έτσι στο τέλος ο πίνακας, που περιέχει την πληροφορία για τους γείτονες, θα έχει πάρει την τελική μορφή του και θα μεταφερθεί στην CPU. Οι συγγραφείς ελέγχουν πειραματικά την επίδοση του αλγορίθμου τους συγκρίνοντάς την με διάφορες άλλες υλοποιήσεις και για πολλά datasets από τα οποία δημιουργούν δείγματα, των οποίων το μέγεθος κλιμακώνουν σταδιακά, ώστε να μελετήσουν καλύτερα τη συμπεριφορά των αλγορίθμων και καταλήγουν στο συμπέρασμα ότι ο αλγόριθμος που υλοποίησαν είναι ο πιο γρήγορος.

Στο [23] οι συγγραφείς προτείνουν μία brute force μέθοδο των k κοντινότερων γειτόνων, η οποία χωρίζει σε κομμάτια τις εισόδους και τον πίνακα των αποστάσεων. Επίσης προτείνουν μία μέθοδο, που τους επιτρέπει να εκτελούν τους αλγορίθμους σε πολλές κάρτες γραφικών χωρίζοντας τα κομμάτια σε ομάδες και χωρίζοντας αυτές τις ομάδες σε υποομάδες, που περιέχουν πολλά κομμάτια. Ακόμα χρησιμοποιούν shared memory για να επιταχύνουν τους υπολογισμούς. Κάθε νήμα υπολογίζει μία απόσταση.

Στο [24] οι συγγραφείς εξηγούν ότι στους Knn αλγόριθμους υπάρχουν δύο βασικές προσεγγίσεις. Η μία προσπαθεί ν' αποκλείσει περιττούς υπολογισμούς, δημιουργώντας όμως ασυνέπειες στους υπολογισμούς. Η άλλη προσπαθεί να εκμεταλλευτεί τη συνέπεια, που υπάρχει στους υπολογισμούς για να εκμεταλλευτεί την υπολογιστική δύναμη της GPU, που όμως εισάγει πολλούς περιττούς υπολογισμούς αποστάσεων. Οι συγγραφείς χρησιμοποιώντας την τριγωνική ανισότητα επιτυγχάνουν να συνδυάσουν τα θετικά των δύο προσεγγίσεων.

Στο [25] οι συγγραφείς υλοποιούν μία μέθοδο Knn κατασκευάζοντας ένα grid και χρησιμοποιώντας shared memory.

Στο [26] οι συγγραφείς υλοποιούν μια μέθοδο του knn, κατά την οποία κάθε νήμα υπολογίζει την απόσταση ανάμεσα σε ένα σημείο του reference και του query και έτσι δημιουργούν έναν δισδιάστατο πίνακα με αποστάσεις μεγέθους mχn, όπου m είναι το μέγεθος του reference και n το μέγεθος του query. Μετά το τέλος αυτής της φάσης και σε διαφορετικό kernel γίνεται η ταξινόμηση του πίνακα με τις αποστάσεις με μια παραλλαγή του insertion sort και υπολογίζονται οι k κοντινότεροι γείτονες. Στην πειραματική αξιολόγηση ο στόχος είναι να επιταχύνουν ένα πρόγραμμα γραμμένο σε Matlab χρησιμοποιώντας εξωτερικές συναρτήσεις της C και της CUDA. Έτσι δημιουργούν τέσσερις παραλλαγές, τις οποίες τις συγκρίνουν πειραματικά. Μία χρησιμοποιεί τον αλγόριθμο που μόλις περιεγράφηκε, μία είναι γραμμένη χρησιμοποιώντας μόνο Matlab, μία που χρησιμοποιεί κώδικα σε C και μία βασισμένη σε kd-tree, η οποία υπάρχει σε μία βιβλιοθήκη της C. Δοκιμάζουν πειραματικά όλους τους αλγορίθμους για διαφορετικό αριθμό διαστάσεων και για διαφορετικά μεγέθη του reference και του query και συμπεραίνουν ότι η χρήση της CUDA μπορεί να επιταχύνει τους υπολογισμούς έως και 120 φορές.

Σε μία επόμενη δημοσίευσή τους στο [27] οι συγγραφείς προτείνουν μια νέα μέθοδο, η οποία χρησιμοποιεί τη βιβλιοθήκη CUBLAS, η οποία είναι μία βιβλιοθήκη που αφορά γραμμική άλγεβρα και εξειδικεύεται στις λειτουργίες με διανύσματα και πίνακες χρησιμοποιώντας κάρτα γραφικών. Μετασχηματίζουν το πρόβλημα του knn, ώστε να γράψουν το κομμάτι που αφορά τον υπολογισμό των αποστάσεων ως πράξεις πινάκων, όπως προσθέσεις και πολλαπλασιασμοί. Έχουν επίσης μια φάση ταξινόμησης, ίδια με αυτή που χρησιμοποιείται στο [26]. Για την πειραματική αξιολόγηση χρησιμοποιήσαν τρεις αλγορίθμους. Αυτόν που μόλις περιεγράφηκε, αυτόν του [26], και τον knn, που υπάρχει υλοποιημένος στην C++ βιβλιοθήκη ANN. Η σύγκριση έγινε με χρήση Matlab για διαφορετικό αριθμό διαστάσεων και για διαφορετικά μεγέθη του reference και του query, καθώς και κάνοντας SIFT matching σε μεγάλο αριθμό διαστάσεων και συμπεραίνουν ότι ο αλγόριθμός που υλοποίησαν, καθώς και ο αλγόριθμός που είχαν υλοποιήσει στο [26], έφτασαν στην πρώτη περίπτωση να είναι έως 189 φορές και 64 φορές αντίστοιχα πιο γρήγοροι από αυτόν της βιβλιοθήκης ANN και στη δεύτερη περίπτωση (SIFT matching) έως και 62 φορές και 25 φορές αντίστοιχα πιο γρήγοροι.

Στο [28] οι συγγραφείς υλοποιούν έναν αλγόριθμο KNN που επικεντρώνεται σε δεδομένα με πολλές διαστάσεις. Για τον υπολογισμό του πίνακα των αποστάσεων χρησιμοποιούν πολλαπλασιασμό πινάκων, χρησιμοποιώντας 3 διαφορετικές υλοποιήσεις, έναν kernel, που δημιούργησαν, την υπορουτίνα SGEMM στο MAGNA και το cuBLAS. Για τη φάση της επιλογής χρησιμοποιούν primitives, που έχουν υλοποιημένο τον αλγόριθμο Merge path, οι οποίοι τους βοηθάνε να βρούνε τους k κοντινότερους γείτονες, χωρίς να χρειάζεται να ταξινομήσουν όλες τις αντίστοιχες αποστάσεις. Ελέγχουν πειραματικά τις μεθόδους τους για διαφορετικές τιμές του k των μεγεθών του reference και query. Συμπεραίνουν ότι ο αλγόριθμος λειτουργεί καλύτερα για μεγάλα k και συγκρίνουν και τις 3 διαφορετικές παραλλαγές, που χρησιμοποίησαν για τον υπολογισμό του πίνακα των αποστάσεων καταλήγοντας στο συμπέρασμα ότι η αποδοτικότερη υλοποίηση εξαρτάται από τις εισόδους.

Στο [29] οι συγγραφείς υλοποιούν μια μέθοδο του knn, η οποία βασίζεται σε μια αποδοτική υλοποίηση του quick sort λαμβάνοντας υπόψη το ότι κάθε warp έχει μέγεθος 32 νήματα και εκτελεί πράξεις ως SIMD. Αρχικά υπολογίζουν τον πίνακα, που περιέχει την πληροφορία για τις αποστάσεις, εκτελώντας πολλαπλασιασμό πινάκων και στη συνέχεια είτε ανά 32 νήματα, είτε ανά block φορτώνουν μικρά κομμάτια από κάθε πίνακα, ελέγχουν

ποια είναι μικρότερα, ποια μεγαλύτερα ή ίσα ενός στοιχείου pivot και με τη χρήση συναρτήσεων, που εκτελούν λειτουργίες επάνω σε bit ατομικά σε κάθε νήμα ή για όλα τα νήματα ενός warp, καθώς και λειτουργίες μετατόπισης bit, βρίσκουν τη θέση σ΄ έναν πίνακα, που βρίσκεται στη shared memory (κι έχει μέγεθος είτε ίσο με το μέγεθος του warp, είτε του block αναλόγως), στην οποία πρέπει να γραφτεί το στοιχείο που περιέχει την πληροφορία για μια απόσταση, έτσι ώστε τα στοιχεία που έχουν τιμές μικρότερες από το pivot να βρίσκονται αριστερά και τα στοιχεία που έχουν τιμές μεγαλύτερες ή ίσες με το pivot να βρίσκονται δεξιά. Στην περίπτωση που χρησιμοποιηθούν block πρέπει να εκτελεστεί ένα prefix sum, ώστε να βρεθεί η θέση στη shared memory, όμως αυτό δεν προτείνεται, παρά μόνο στις περιπτώσεις, όπου τα σημεία που υπάρχουν στο query είναι λιγότερα του 1024 και δεν μπορούν να αξιοποιηθούν πλήρως οι πόροι της κάρτας γραφικών. Μόλις τελειώσει αυτή η φάση, τα στοιχεία γράφονται σ’ έναν πίνακα κάνοντας δύο coalesced εγγραφές. Έπειτα ελέγχουν πειραματικά την υλοποίησή τους συγκρίνοντάς την με διάφορες, άλλες υλοποιήσεις για πολλές τιμές του k και για πολλά μεγέθη των αρχείων εισόδων. Συμπεραίνουν ότι η υλοποίησή τους υπερτερεί των άλλων, στις περισσότερες περιπτώσεις, με το κέρδος να διαφέρει κατά περίπτωση.

Στο [30] οι συγγραφείς προτείνουν μια υλοποίηση με διάφορες παραλλαγές, που μπορεί να χρησιμοποιηθεί για τον ακριβή υπολογισμό του k κοντινότερων γειτόνων ή για τον κατά προσέγγιση υπολογισμό. Αναγάγουν τον υπολογισμό του πίνακα των αποστάσεων σε πράξεις με πίνακα. Επίσης φροντίζουν το query, σε περίπτωση που δε χωράει στην κάρτα γραφικών, να χωρίζεται σε partitions. Αργότερα χρησιμοποιούν primitives, που επιτρέπουν η φάση της επιλογής των αποστάσεων να πραγματοποιηθεί χρησιμοποιώντας registers, επιταχύνοντας έτσι σημαντικά τη διαδικασία. Χρησιμοποιούν διάφορα primitives δημιουργώντας πολλές παραλλαγές του αλγορίθμου τους. Δοκιμάζουν πειραματικά τις παραλλαγές για πλήθος σεναρίων και εφαρμογών και εξάγουν συμπεράσματα σχετικά με την επίδοση για κάθε περίπτωση.

Στο [31] οι συγγραφείς παρουσιάζουν τη βιβλιοθήκη faiss, που περιέχει ένα σύνολο από primitives και indexes, που μπορούν να χρησιμοποιηθούν για τον ακριβή και για τον κατά προσέγγιση υπολογισμό των KNN. Όταν απαιτούνται ακριβείς υπολογισμοί, ο πίνακας των αποστάσεων υπολογίζεται χρησιμοποιώντας πολλαπλασιασμό πινάκων, ενώ όταν επιτρέπονται προσεγγίσεις, χρησιμοποιούνται προσεγγιστικές αναπαραστάσεις διανυσμάτων και σε πολλές περιπτώσεις διάφορα indexes. Οι συγγραφείς εκτελούν πολλά

πειράματα συγκρίνοντας τις μεθόδους, που χρησιμοποιούν, και εξάγοντας συμπεράσματα για την επίδοσή τους σε σχέση με τις εισόδους.

Στο [32] οι συγγραφείς παρουσιάζουν έναν προσεγγιστικό αλγόριθμο με τη χρήση ενός γράφου, που εστιάζει σε πολυδιάστατα δεδομένα. Προσπαθούν να πετύχουν καλό χρόνο ανά ερώτημα και αποτελεσματική κατασκευή του index. Δημιουργούν 4 παραλλαγές του αλγορίθμου τους, μία βασική, μία πιο αποτελεσματική, αλλά με μειωμένη ακρίβεια, καθώς και μία μέθοδο, που μπορεί να εκτελεστεί παράλληλα σε πολλές κάρτες γραφικών, στη συγκεκριμένη περίπτωση χρησιμοποιούν 8 κάρτες, την οποία εφαρμόζουν στις δύο προηγούμενες παραλλαγές δημιουργώντας έτσι δύο καινούριες. Στην πειραματική αξιολόγηση συγκρίνουν τις παραλλαγές τους με πολλές υλοποιήσεις σε CPU και σε GPU και για άλλα πειράματα και συμπεραίνουν ότι ο αλγόριθμός τους είναι γρηγορότερος διατηρώντας ακρίβεια μεγαλύτερη του 99%.

Στο [33] οι συγγραφείς προτείνουν έναν προσεγγιστικό αλγόριθμο και κάποιες παραλλαγές του για την εύρεση των k κοντινότερων γειτόνων χρησιμοποιώντας γράφο κι έναν συνδυασμό διάφορων δομών δεδομένων και βελτιστοποιήσεων. Εστιάζουν σε δεδομένα πολλών διαστάσεων και για τον υπολογισμό των αποστάσεων χωρίζουν τα διανύσματα και κάθε νήμα ενός block παίρνει ένα κομμάτι από ένα διάνυσμα του query και το αντίστοιχο κομμάτι από ένα διάνυσμα του reference και υπολογίζει ένα κομμάτι της τελικής απόστασης. Στο τέλος το νήμα συγκεντρώνει κομμάτια των αποστάσεων όλων των warps χρησιμοποιώντας shfl down warp reduction. Μελετούν πειραματικά τις μεθόδους τους σε σύγκριση με άλλους αλγόριθμους και συμπεραίνουν ότι ο αλγόριθμός τους είναι αποδοτικότερος.

Στο [34] οι συγγραφείς υλοποιούν έναν προσεγγιστικό αλγόριθμο εύρεσης των k κοντινότερων γειτόνων χρησιμοποιώντας έναν γράφο εγγύτητας (proximity graph) και χρησιμοποιώντας μια ιδιότητα του λογισμικού να διασπά το warp, ώστε να επωφεληθούν από το γεγονός ότι οι εντολές που φορτώνουν στα 128 bit να είναι πιο αποδοτικές. Επίσης χρησιμοποιούν ένα hash table για να διαχειρίζονται τη λίστα που περιέχει την πληροφορία για τους κόμβους που έχουν επισκεφθεί. Διατηρούν μία internal top-M list με M μεγαλύτερο ή ίσο του k, καθώς και μία λίστα με υποψήφιους κόμβους. Αυτές οι δύο λίστες αποτελούν το buffer. Από τη λίστα με τους υποψήφιους κόμβους υπολογίζονται όλες οι αποστάσεις των κόμβων, που δεν έχουν υπάρξει ξανά υποψήφιοι και συνολικά απ' όλο τον buffer κρατάνε τις M μικρότερες αποστάσεις και εισάγουν τους αντίστοιχους κόμβους

στην internal top-M list. Αυτός ο υπολογισμός γίνεται με τη χρήση μιας single warp-level bitonic sort, όταν το μέγεθος του buffer είναι μικρότερο ή ίσο του 512. Διαφορετικά χρησιμοποιείται μία radix-based sort μέσα σ' ένα μόνο block. Στη συνέχεια από την internal top-M list δημιουργείται η νέα λίστα υποψηφίων παίρνοντας όλους τους κόμβους, που υπάρχουν σ' αυτή και δεν έχουν υπάρξει ξανά γονείς, εισάγοντας τα παιδιά τους σ' αυτή. Αυτή η διαδικασία επαναλαμβάνεται μέχρι η internal top-M list να μην αλλάζει μετά από μία επανάληψη, η οποία θα είναι και η τελευταία. Συγκρίνουν την υλοποίησή τους με πολλούς, άλλους αλγόριθμους και με μια σειρά πειραμάτων συμπεραίνουν ότι είναι πιο αποδοτική.

Στο [35] οι συγγραφείς υλοποιούν έναν αλγόριθμο υπολογισμών των k κοντινότερων γειτόνων, ο οποίος μπορεί να εκτελεί υπολογισμούς σ' έναν σύνολο δεδομένων με δισεκατομμύρια στοιχεία εισάγοντας μια μέθοδο υπολογισμού σε φάσεις, η οποία επιτρέπει να μεταφέρουν δεδομένα ασύγχρονα επικαλύπτοντας το κόστος των μεταφορών με υπολογισμούς. Με αυτόν τον τρόπο ξεπερνούν το βασικό πρόβλημα του ότι σε προσεγγιστικές μεθόδους που χρησιμοποιούν γράφο, ο γράφος πρέπει να βρίσκεται αποθηκευμένος στη μνήμη της κάρτας γραφικών περιορίζοντας τα μεγέθη των συνόλων δεδομένων για τα οποία μπορούν να εκτελεστούν. Επίσης χρησιμοποιούν μια μέθοδο συμπίεσης των διανυσμάτων για γρήγορους υπολογισμούς των αποστάσεων. Συγκρίνουν την υλοποίησή τους με άλλες υλοποιήσεις και συμπεραίνουν ότι ο δικός τους αλγόριθμος είναι πολύ αποδοτικότερος, όταν απαιτείται ακρίβεια.

Στο [36] οι συγγραφείς υλοποιούν έναν προσεγγιστικό αλγόριθμο εύρεσης των k κοντινότερων γειτόνων εστιάζοντας στο πρόβλημα της περιορισμένης μνήμης της κάρτας γραφικών και στην περιορισμένη πυκνότητα υπολογισμών που εμφανίζουν οι προσεγγιστικοί υπολογισμοί με γράφο. Υλοποιούν μια υβριδική μέθοδο, που χρησιμοποιεί και τη GPU και τη CPU και χωρίζουν τους υπολογισμούς σε 3 στάδια, που εκτελούνται σε pipeline. Στο πρώτο στάδιο στη GPU χρησιμοποιούνται συμπιεσμένα διανύσματα και κομμάτια του πλήρους γράφου για να βρουν τους υποψήφιους κόμβους. Το δεύτερο στάδιο εκτελείται στη CPU χρησιμοποιώντας πλήρη διανύσματα, υπολογίζοντας με ακρίβεια τις αποστάσεις, που έχουν προηγουμένως υπολογιστεί στη GPU και στη συνέχεια εκτελεί μια περιορισμένη διάσχιση στο κομμάτι του γράφου που χρησιμοποιήθηκε στη GPU επαναξιολογώντας τους υποψήφιους, βασισμένο στις πλήρεις αποστάσεις τους. Το τελικό στάδιο εκτελείται στη CPU χρησιμοποιώντας πλήρη

διανύσματα και κάνοντας αναζήτηση στον πλήρη γράφο, όπου παράγονται τα τελικά αποτελέσματα. Συγκρίνουν την υλοποίησή τους με άλλες υλοποιήσεις σε CPU και GPU και συμπεραίνουν ότι η μέθοδός τους είναι πολύ πιο αποδοτική από μεθόδους που χρησιμοποιούν μόνο CPU και από μεθόδους που χρησιμοποιούν GPU, όταν τα μεγέθη των συνόλων δεδομένων, που χρησιμοποιούνται στην είσοδο, είναι πολύ μεγάλα.

Στα [37],[38] οι συγγραφείς υλοποιούν αλγόριθμους που χρησιμοποιούν Locality Sensitive Hashing (LSH). Στο [37] οι συγγραφείς χρησιμοποιούν ένα RP-tree για να χωρίσουν το σύνολο των δεδομένων σε υποομάδες που για κάθε μια από αυτές δημιουργούν ένα LSH hash table. Κάθε ερώτημα αποφασίζεται σε ποια υποομάδα ανήκει και στη συνέχεια εκτελείται μια αναζήτηση των k κοντινότερων γειτόνων στα αντίστοιχα LSH table. Ο αλγόριθμος είναι προσεγγιστικός. Στο [38] οι συγγραφείς υλοποιούν ένα winner-take-all (WTA) hash φτιάχνοντας ένα παράλληλο hash table για αναζητήσεις με κώδικες LSH.

Στο [39] οι συγγραφείς παρουσιάζουν μία μέθοδο εύρεσης των k κοντινότερων γειτόνων χρησιμοποιώντας δέντρα, αξιοποιώντας τις δυνατότητες, που δίνει το υλικό (hard wear) των μοντέρνων GPUs, που κάνουν τις παράλληλες πράξεις πάνω σε δέντρα εύκολες. Για να το πετύχουν αυτό χρησιμοποιούν ray-tracing πυρήνες, οι οποίοι έχουν από τη φύση τους το μειονέκτημα ότι μπορούν να χρησιμοποιούν ως μετρική για τις αποστάσεις μόνο την ευκλείδεια απόσταση. Για να ξεπεράσουν αυτό το πρόβλημα και για να επιτρέψουν και τη χρήση και άλλων αποστάσεων προτείνουν ένα φίλτρο, το Arkade Filter-Refine και έναν μετασχηματισμό, τον Arkade Monotone μετασχηματισμό. Καθένα από αυτά τα δύο επιτρέπει τον υπολογισμό μη ευκλείδειων αποστάσεων ως ευκλείδειες.

Κεφάλαιο 4 Αλγόριθμοι

Στο κεφάλαιο αυτό παρουσιάζονται αναλυτικά οι παραλλαγές του αλγορίθμου και επεξηγούνται τα πλεονεκτήματα και τα μειονεκτήματα, που εμφανίζονται.

4.1 Brute Force (DBFS)

Ο αλγόριθμος δέχεται ως εισόδους δύο σύνολα σημείων στον τρισδιάστατο χώρο D_1 και D_2 με m και n σημεία αντίστοιχα και τον αριθμό των κοντινότερων ζευγαριών k. Ο host διαβάζει τα δεδομένα και τα μεταφέρει στο device, όπου εκτελούνται οι υπολογισμοί για την εύρεση των k κοντινότερων ζευγαριών.

Για την εκτέλεση του κώδικα στην κάρτα γραφικών χρησιμοποιήθηκαν μονοδιάστατα blocks μεγέθους N. Κάθε thread υπολογίζει τους k κοντινότερους γείτονες για ένα σημείο του D_2. Για να το πετύχει αυτό υπολογίζει όλες τις αποστάσεις ανάμεσα στο συγκεκριμένο σημείο και τα σημεία του D_1. Οι k πρώτες αποστάσεις εισάγονται σε μία ιδιωτική λίστα που κανένα άλλο νήμα δεν έχει πρόσβαση. Οι επόμενες αποστάσεις συγκρίνονται με τη μεγαλύτερη απόσταση στη λίστα και εάν είναι μικρότερες εισάγονται στη λίστα παίρνοντας τη θέση της.

Στη συνέχεια οι λίστες που περιέχουν την πληροφορία για τους γείτονες μεταφέρονται στον host, από όπου επιλέγονται τα k ζευγάρια με τις μικρότερες αποστάσεις. Για τις λίστες προτείνουμε δύο διαφορετικές υλοποιήσεις τις οποίες και θα συγκρίνουμε πειραματικά.

Παρακάτω δίνεται ο ψευδοκώδικας για τον host και το device.

```
struct point_struct {
   unsigned long long int id;
   double x;
   double y;
   double z;
};
```

```
struct knp_distance_struct {
   unsigned long long int id1;
   unsigned long long int id2;
   double distance;
};
```

Host DBFS

```
input(file data1_file, file data2_file, int k, int N) {
   host_vector data1 = data1_file.read();
   host_vector data2 = data2_file.read();
   int data1_size = data1.size();
   int data2_size = data2.size();

   device_vector d_data1 = data1;
   device_vector d_data2 = data2;
   host_vector distance_list(k * data2_size);
   device_vector d_distance_list = distance_list;
   host_vector knp(k);

   knp_kernel <<<(data2_size - 1) / N + 1, N >>> (d_data1,
       d_data2, data1_size, data2_size, d_distance_list, k);

   cudaDeviceSyncronize();

   distance_list = d_distance_list;
   find_knp(knp, distance_list, k);
}
```

Kernel DBFS

```
input(device_vector d_data1, device_vector d_data2, int
      data1_points, int data2_points, device_vector
      distance_list, int k) {
   unsigned int data2Idx = blockIdx.x * blockDim.x +
         threadIdx.x;

   if (data2Idx < data2_points) {
      unsigned int distance_list_offset = data2Idx * k;
      unsigned int inserted = 0;
      double dist;

      for (unsigned int i=0; i<data1_points; i++) {
         dist = Euclidean_distance(d_data1[i],
            d_data2[data2Idx]);
         insert_to_distance_list(distance_list,
            distance_list_offset, inserted,dist,
            d_data1[i].id, d_data2[data2Idx].id);
      }
   }

}
```

4.2 Brute Force με Partitioning (DBF)

Στον προηγούμενο αλγόριθμο θεωρήθηκε πως και τα δύο σύνολα σημείων χωρούν στη μνήμη του device. Στην πραγματικότητα αυτό μπορεί να μην ισχύει. Επομένως θα πρέπει να χρησιμοποιηθεί μια τεχνική διαμέρισης.

Αυτός ο αλγόριθμος, όπως και ο προηγούμενος, δέχεται ως εισόδους δύο σύνολα σημείων στον τρισδιάστατο χώρο D_1 και D_2 με m και n σημεία αντίστοιχα και τον αριθμό των κοντινότερων ζευγαριών k.

Αρχικά το D_2 διαβάζεται από τον host και μεταφέρεται στο device.

Στη συνέχεια το D_1 χωρίζεται σε p1 partitions μεγέθους p1_size το καθένα. Αν το m δε διαιρείται ακριβώς με το p1_size, τότε το τελευταίο partition θα έχει μέγεθος ίσο με το υπόλοιπο της διαίρεσης του m με το p1_size.

Τα partitions διαβάζονται διαδοχικά από τον host και μεταφέρονται στο device.

Για την εκτέλεση του κώδικα στην κάρτα γραφικών χρησιμοποιήθηκαν μονοδιάστατα blocks μεγέθους N. Κάθε thread υπολογίζει τους k κοντινότερους γείτονες για ένα σημείο του D_2. Για να το πετύχει αυτό υπολογίζει όλες τις αποστάσεις ανάμεσα στο συγκεκριμένο σημείο και τα σημεία του προς εξέταση partition. Οι k πρώτες αποστάσεις εισάγονται σε μία ιδιωτική λίστα που κανένα άλλο νήμα δεν έχει πρόσβαση. Οι επόμενες αποστάσεις συγκρίνονται με τη μεγαλύτερη απόσταση στη λίστα και εάν είναι μικρότερες τότε εισάγονται σε αυτή.

Στη συνέχεια οι λίστες που περιέχουν την πληροφορία για τους γείτονες μεταφέρονται στον host από όπου επιλέγονται τα k ζευγάρια με τις μικρότερες αποστάσεις.

Host DBF

```
input(file data1_file, file data2_file, int k, unsigned int
      data1_partition_size , unsigned int
      original_partitions, int N) {
   host_vector data1_partition(data1_partition_size);
   host_vector data2 = data2_file.read();
   int data1_size = data1_file.size() / sizeof(point_struct);
   int data2_size = data2.size();

   unsigned int partitions = original_partitions;

   device_vector d_data1_partition;
   device_vector d_data2 = data2;
   host_vector distance_list(k * data2_size);
   device_vector d_distance_list = distance_list;
   host_vector<unsigned int> inserted_list(data2_size, 0);
   device_vector<unsigned int> d_inserted_list =
      inserted_list;
```

```
   host_vector knp(k);

   unsigned int data1_backet_size;
   unsigned int data1_partition_loops = (data1_size – 1) /
      data1_partition_size;

   for (unsigned int p = 0; p < data1_partition_loops; p++) {
      if ((p + 1) * data1_partition_size < data1_size) {
         data1_backet_size = data1_partition_size;
      }
      else {
         data1_backet_size = data1_size – p *
            data1_partition_size;
      }

      data1_partition = data1_file.read(data1_backet_size);
      knp_kernel <<<(data2_size - 1) / N + 1, N >>>
          (d_data1_partition, d_data2, data1_backet_size,
          data2_size, d_distance_list, d_inserted_list, k);

      cudaDeviceSyncronize();
   }

   distance_list = d_distance_list;
   find_knp(knp, distance_list, k);
}
```

Kernel DBF

```
input(device_vector d_data1_partition, device_vector d_data2,
      int data1_partition_size, int data2_points,
      device_vector distance_list, device_vector
      d_incerted_list, int k) {
```

```
   unsigned int data2Idx = blockIdx.x * blockDim.x +
        threadIdx.x;

   if (data2Idx < data2_points) {
      unsigned int distance_list_offset = data2Idx * k;
      unsigned int inserted = 0;
      double dist;

      inserted = d_inserted_list[data2Idx];

      for (unsigned int i=0; i<data1_partition_size; i++) {
         dist = Euclidean_distance(d_data1_partition[i],
              d_data2[data2Idx]);
         insert_to_distance_list(distance_list,
              distance_list_offset, inserted, dist,
              d_data1_partition[i].id, d_data2[data2Idx].id);
      }
   }

   d_inserted_list.[data2Idx] = inserted;

}
```

4.3 Disk Symmetric Progression Partitioning (DSPP)

Οι brute force αλγόριθμοι μπορεί να είναι ιδιαίτερα αποτελεσματικοί, ειδικά αν εκτελούνται παράλληλα. Ωστόσο σε πολλές περιπτώσεις εκτελούν περιττούς υπολογισμούς. Σε αυτή την παραλλαγή του αλγορίθμου θα παρουσιαστεί μια μέθοδος χωρικής υποδιαίρεσης που εκμεταλλεύεται τις λίγες διαστάσεις που χρησιμοποιούν τα δεδομένα για να μειώσει το κόστος των υπολογισμών.

Αυτός ο αλγόριθμος δέχεται ως εισόδους δύο σύνολα σημείων στον τρισδιάστατο χώρο D_1 και D_2 με m και n σημεία αντίστοιχα και τον αριθμό των κοντινότερων ζευγαριών k.

Αρχικά το D_2 διαβάζεται από τον host και μεταφέρεται στο device.

Στη συνέχεια το D_1 χωρίζεται σε p1 partitions μεγέθους p1_size το καθένα. Αν το m δε διαιρείται ακριβώς με το p1_size, τότε το τελευταίο partition θα έχει μέγεθος ίσο με το υπόλοιπο της διαίρεσης του m με το p1_size.

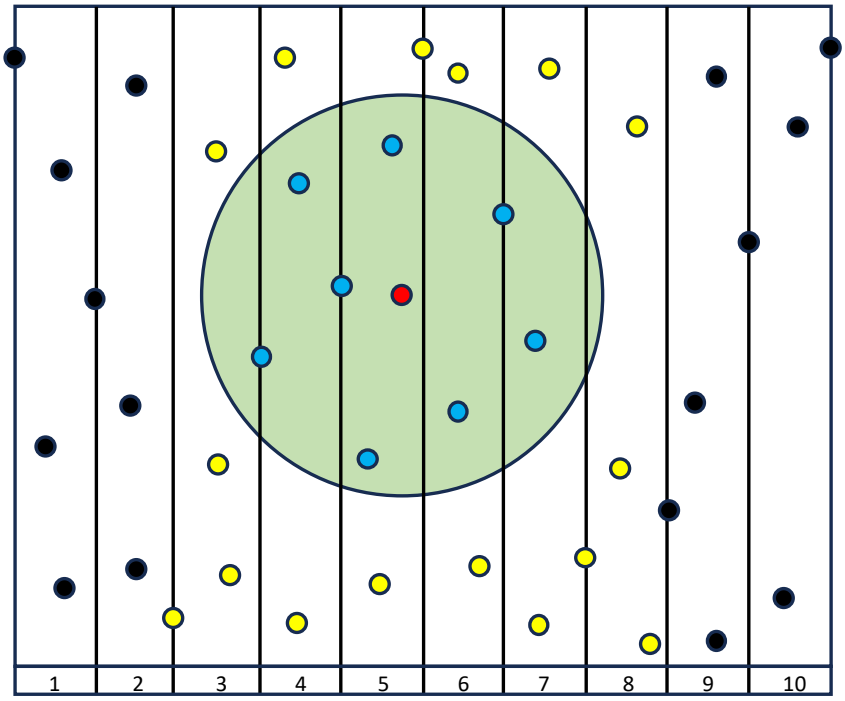
Τα partitions διαβάζονται διαδοχικά από τον host και μεταφέρονται στο device. Εκεί ταξινομούνται με βάση την τιμή της συντεταγμένης χ και στη συνέχεια μεταφέρονται πίσω στον host, όπου χωρίζονται σε μικρότερα sub-partitions και παράλληλα δημιουργείται και ένα sub-partition index, το οποίο περιέχει πληροφορία για τη μικρότερη και τη μεγαλύτερη τιμή χ σε κάθε sub-partition.

Για την εκτέλεση του κώδικα στην κάρτα γραφικών χρησιμοποιήθηκαν μονοδιάστατα blocks μεγέθους N. Κάθε thread υπολογίζει τους k κοντινότερους γείτονες για ένα σημείο του D_2. Για να το πετύχει αυτό υπολογίζει όλες τις αποστάσεις ανάμεσα στο συγκεκριμένο σημείο και τα σημεία του προς εξέταση partition χρησιμοποιώντας το sub-partition index για να μειώσει τον αριθμό των υπολογισμών. Πιο συγκεκριμένα το νήμα διαβάζει το sub-partition index, προσπαθώντας να βρει αν κάποια από τα όρια που υπάρχουν σε αυτό εμπεριέχουν την τιμή χ του προς εξέταση σημείου του D_2. Αν υπάρχουν τέτοια όρια, τότε το αντίστοιχο sub-partition θα είναι το πρώτο που θα επιλεγεί. Αν δεν υπάρχουν όρια που να ικανοποιούν την παραπάνω συνθήκη, τότε ως πρώτο sub-partition επιλέγεται το αριστερότερο ή το δεξιότερο με κριτήριο πιο από τα δύο βρίσκεται πιο κοντά στον άξονα χ στο προς εξέταση σημείο του D_2. Για το πρώτο sub-partition όλοι οι απαραίτητοι υπολογισμοί εκτελούνται όπως περιεγράφηκε στις προηγούμενες παραλλαγές του αλγορίθμου. Δηλαδή οι k πρώτες αποστάσεις που υπολογίζονται εισάγονται σε μία ιδιωτική λίστα που κανένα άλλο νήμα δεν έχει πρόσβαση. Οι επόμενες αποστάσεις συγκρίνονται με τη μεγαλύτερη απόσταση στη λίστα και εάν είναι μικρότερες τότε εισάγονται σε αυτή. Αφού ολοκληρωθούν οι υπολογισμοί το sub-partition με τη μικρότερη απόσταση στον άξονα χ από το προς εξέταση σημείο του D_2, θα επιλεγεί ως το επόμενο υπό τη συνθήκη ότι η μικρότερη από τις αποστάσεις στον άξονα χ των σημείων του sub-partition με το προς εξέταση σημείο του D_2 να είναι μικρότερη από τη μεγαλύτερη απόσταση στη λίστα που χρησιμοποιεί το νήμα. Το να βρεθεί η συγκεκριμένη απόσταση είναι εύκολο, καθώς τα σημεία του sub-partition είναι ταξινομημένα με βάση την τιμή της συντεταγμένης χ. Αν η συνθήκη δεν ισχύει τότε το sub-partition δεν ελέγχεται και δεν αναζητείται άλλο στη θέση του. Διαφορετικά εκτελούνται οι απαραίτητοι υπολογισμοί

ακριβώς όπως και για το πρώτο sub-partition. Το επόμενο sub-partition επιλέγεται ακολουθώντας την ίδια διαδικασία.

Στο τέλος οι λίστες που περιέχουν την πληροφορία για τους γείτονες μεταφέρονται στον host από όπου επιλέγονται τα k ζευγάρια με τις μικρότερες αποστάσεις.

Ένα παράδειγμα εκτέλεσης του κώδικα του kernel φαίνεται στην Εικόνα 4-1. Το κόκκινο σημείο συμβολίζει το σημείο του D2 που εξετάζει το νήμα, η ακτίνα του κύκλου είναι ίση με τη μέγιστη απόσταση που είναι αποθηκευμένη στην αντίστοιχη λίστα, τα μπλε σημεία είναι τα σημεία που θα ελεγχθούν και θα εισαχθούν, τα κίτρινα είναι τα σημεία που θα ελεγχθούν, αλλά δε θα εισαχθούν και τα μαύρα τα σημεία που δε θα ελεγχθούν. Ανάμεσα στα διαστήματα υπάρχει ο ίδιος αριθμός σημείων με τα σημεία που βρίσκονται πάνω στα όρια των διαστημάτων να ανήκουν στο διάστημα που βρίσκεται δεξιά, με εξαίρεση το σημείο που βρίσκεται πάνω στο τελευταίο όριο. Αρχικά ελέγχεται το sub-partitions 5 που εμπεριέχει το κόκκινο σημείο. Στη συνέχεια ελέγχεται το 6 που είναι το αμέσως κοντινότερο και με την ίδια λογική ελέγχονται διαδοχικά τα 4,7,3,8. Τα 1,2,9,10 δεν ελέγχονται καθόλου. Ο ίδιος κανόνας εφαρμόζεται και στον τρισδιάστατο χώρο με τις απαραίτητες μετατροπές. Σε ένα ρεαλιστικό παράδειγμα θα υπήρχαν πολύ περισσότερα σημεία, τα διαστήματα θα συνέχιζαν να εμπεριέχουν τον ίδιο αριθμό σημείων, αλλά δε θα ήταν ισαπέχοντα. Οι αποστάσεις ανάμεσα στα διαστήματα εξαρτώνται από την κατανομή που ακολουθούν τα σημεία του D1 και αυτός είναι και ο λόγος που ο αλγόριθμος εμφανίζει διαφορές στον χρόνο εκτέλεσης ανάλογα με την κατανομή που ακολουθούν τα σημεία, όπως θα δούμε και στην πειραματική μελέτη στο Κεφάλαιο 5 .



Εικόνα 4-1: Sub-partitions με βάση τον άξονα χ

```
struct partition_range {
   double x1;
   double x2;
};
```

Host DSPP

```
input(file data1_file, file data2_file, int k, unsigned int
      data1_partition_size, unsigned int partitions,int N) {
   host_vector data1_partition(data1_partition_size);
   host_vector data2 = data2_file.read();
   int data1_size = data1_file.size() / sizeof(point_struct);
   int data2_size = data2.size();

   device_vector d_data1_partition;
   device_vector d_data2 = data2;
   host_vector distance_list(k * data2_size);
```

```
device_vector d_distance_list = distance_list;
host_vector<unsigned int> inserted_list(data2_size, 0);
device_vector<unsigned int> d_inserted_list =
       inserted_list;
host_vector knp(k);
host_vectot<partition_range> xpart;
device_vector<partition_range> d_xpart;

unsigned int data1_backet_size;
unsigned int data1_partition_loops = (data1_size – 1) /
        data1_partition_size;

for (unsigned int p = 0; p < data1_partition_loops; p++) {
   if ((p + 1) * data1_partition_size < data1_size) {
      data1_backet_size = data1_partition_size;
   }
   else {
      data1_backet_size = data1_size – p *
           data1_partition_size;

      if (data1_backet_size >= 2*partitions) {
         partitions_capacity_points = data1_backet_size /
              partitions;

         if (data1_backet_size % partitions != 0) {
            partitions_capacity_points++;
         }
      }
      else {
         partitions_capacity_points = data1_backet_size;
         partitions = 1;
      }

   }
```

```
      xpart.clear();
      double prevx = data1[0].x;
      unsigned int xpart_idx;
      unsigned int i = 0;
      for (i = 0; i < partitions; i++) {
         ((i + 1) * partitions_capacity_points - 1 <
               data1_backet_size) ?
         xpart_idx = (i + 1) * partitions_capacity_points - 1
               : xpart_idx =
         data1_backet_size - 1;
         double x = data1[xpart_idx].x;
         xpart.push_back({ prevx,x });
         prevx = x;
      }
      d_xpart = xpart;

      data1_partition = data1_file.read(data1_backet_size);
      knp_kernel <<<(data2_size - 1) / N + 1, N >>>
            (d_data1_partition, d_data2, data1_backet_size,
            data2_size, d_distance_list, d_inserted_list,
            d_xpart, k);

      cudaDeviceSyncronize();
   }

   distance_list = d_distance_list;
   find_knp(knp, distance_list, k);
}
```

Kernel DSPP

```
input(device_vector d_data1_partition, device_vector d_data2,
    int data1_partition_size, int data2_points, device_vector
```

```
 distance_list, device_vector d_incerted_list,
 device_vector<partition_range> d_xpart, int k) {
unsigned int data2Idx = blockIdx.x * blockDim.x +
    threadIdx.x;

if (data2Idx < data2_points) {
   unsigned int distance_list_offset = data2Idx * k;
   unsigned int inserted = 0;
   double maxdist;
   double dist;

   inserted = d_inserted_list[data2Idx];

   for (cpart = 0; (cpart < d_xpart.size) &&
       (d_xpart.array[cpart].x2 <
       d_data2.array[data2Idx].x); cpart++) {
   }

   if (cpart == d_xpart.size) {
      cpart--;
   }

   unsigned int leftIdx, rightIdx;

   leftIdx = cpart;
   rightIdx = cpart;

   do
   {
      unsigned int id1 = cpart *
           partitions_capacity_points;
        unsigned int id2 = (cpart + 1) *
             partitions_capacity_points;
        if (id2 > data1_points) {
```

```
            id2 = data1_points;
         }

         for (unsigned int i=id1; i<id2; i++) {
            dist = Euclidean_distance(d_data1_partition[i],
               d_data2[data2Idx]);
            insert_to_distance_list(distance_list,
                  distance_list_offset, inserted, dist,
                  d_data1_partition[i].id,
                  d_data2[data2Idx].id);
         }

         leftmost_x = d_xpart.[leftIdx].x1;
         rightmost_x = d_xpart.[rightIdx].x2;

         maxdist = find_max_distance(distance_list,
              distance_list_offset, k);
         cpart = find_next_closest_sub_partition
              (d_data2[data2Idx], d_xpart, cpart, leftIdx,
              rightIdx, maxdist);
      } while ((maxdist > (d_data2.[data2Idx].x - leftmost_x)
            * (d_data2.[data2Idx].x - leftmost_x)) ||
            ((rightmost_x – d_data2. [data2Idx].x) *
            (rightmost_x - d_data2. [data2Idx].x) <
            maxdist));
   }

   d_inserted_list.[data2Idx] = inserted;

}
```

4.4 Βελτιωμένος Disk Symmetric Progression Partitioning (DSPP+)

Στις προηγούμενες παραλλαγές του αλγορίθμου έγινε η υπόθεση ότι το D_2 θα χωρούσε ολόκληρο στην global memory. Σε πολλές περιπτώσεις αυτό ισχύει όμως τα σύγχρονα μεγάλα datasets μπορούν να ξεπεράσουν τη χωρητικότητα αυτής της μνήμης.

Επομένως σε αυτή τη παραλλαγή του αλγορίθμου θα χρησιμοποιηθεί μια μέθοδος διαμέρισης για το D_2.

Αυτός ο αλγόριθμος δέχεται ως εισόδους δύο σύνολα σημείων στον τρισδιάστατο χώρο D_1 και D_2 με m και n σημεία αντίστοιχα και τον αριθμό των κοντινότερων ζευγαριών k.

Αρχικά το D_1 χωρίζεται σε p1 partitions μεγέθους p1_size το καθένα. Αν το m δε διαιρείται ακριβώς με το p1_size τότε το τελευταίο partition θα έχει μέγεθος ίσο με το υπόλοιπο της διαίρεσης του m με το p1_size.

Τα partitions διαβάζονται διαδοχικά από τον host και μεταφέρονται στο device. Εκεί ταξινομούνται με βάση την τιμή της συντεταγμένης χ και στη συνέχεια μεταφέρονται πίσω στον host, όπου χωρίζονται σε μικρότερα sub-partitions και παράλληλα δημιουργείται και ένα sub-partition index το οποίο περιέχει πληροφορία για τη μικρότερη και τη μεγαλύτερη τιμή χ σε κάθε sub-partition.

Στη συνέχεια το D_2 χωρίζεται σε p2 partitions μεγέθους p2_size το καθένα ακριβώς όπως και το D_1. Κάθε φορά που διαβάζεται ένα partition του D_1 διαβάζονται όλα τα partitions του D_2. Αυτό σημαίνει ότι το D_2 διαβάζεται και μεταφέρεται στην κάρτα γραφικών τόσες φορές όσα τα partitions του D_1. Αυτό αναμένεται να επηρεάσει την επίδοση, εκτός αν επικαλυφθούν αυτές οι καινούργιες διαδικασίες με χρήσιμους υπολογισμούς, όπως και γίνεται στην ενότητα 4.5.

Για την εκτέλεση του κώδικα στην κάρτα γραφικών χρησιμοποιήθηκαν μονοδιάστατα blocks μεγέθους N. Κάθε thread υπολογίζει τους k κοντινότερους γείτονες για ένα σημείο του D_2. Για να το πετύχει αυτό υπολογίζει όλες τις αποστάσεις ανάμεσα στο συγκεκριμένο σημείο και τα σημεία του προς εξέταση partition χρησιμοποιώντας το sub-partition index για να μειώσει τον αριθμό των υπολογισμών. Πιο συγκεκριμένα το νήμα διαβάζει το sub-partition index προσπαθώντας να βρει αν κάποια από τα όρια που υπάρχουν σε αυτό εμπεριέχουν την τιμή χ του προς εξέταση σημείου του D_2. Αν υπάρχουν τέτοια όρια, τότε το αντίστοιχο sub-partition θα είναι το πρώτο που θα επιλεγεί. Αν δεν

υπάρχουν όρια που να ικανοποιούν την παραπάνω συνθήκη τότε ως πρώτο sub-partition επιλέγεται το αριστερότερο ή το δεξιότερο με κριτήριο ποιο από τα δύο βρίσκεται πιο κοντά στον άξονα χ στο προς εξέταση σημείο του D_2. Για το πρώτο sub-partition όλοι οι απαραίτητοι υπολογισμοί εκτελούνται όπως περιεγράφηκε στις προηγούμενες παραλλαγές του αλγορίθμου. Δηλαδή οι k πρώτες αποστάσεις που υπολογίζονται εισάγονται σε μία ιδιωτική λίστα που κανένα άλλο νήμα δεν έχει πρόσβαση. Οι επόμενες αποστάσεις συγκρίνονται με τη μεγαλύτερη απόσταση στη λίστα και εάν είναι μικρότερες τότε εισάγονται σε αυτή. Αφού ολοκληρωθούν οι υπολογισμοί το sub-partition με τη μικρότερη απόσταση στον άξονα χ από το προς εξέταση σημείο του D_2 θα επιλεγεί ως το επόμενο υπό τη συνθήκη ότι η μικρότερη από τις αποστάσεις στον άξονα χ των σημείων του sub-partition με το προς εξέταση σημείο του D_2 να είναι μικρότερη από τη μεγαλύτερη απόσταση στη λίστα που χρησιμοποιεί το νήμα. Το να βρεθεί η συγκεκριμένη απόσταση είναι εύκολο, καθώς τα σημεία του sub-partition είναι ταξινομημένα με βάση την τιμή της συντεταγμένης χ. Αν η συνθήκη δεν ισχύει, τότε το sub-partition δεν ελέγχεται και δεν αναζητείται άλλο στη θέση του. Διαφορετικά εκτελούνται οι απαραίτητοι υπολογισμοί ακριβώς όπως και για το πρώτο sub-partition. Το επόμενο sub-partition επιλέγεται ακολουθώντας την ίδια διαδικασία.

Μετά την ολοκλήρωση της εκτέλεσης ενός kernel ο επόμενος θα εκτελεστεί για το partition του D_2 που έχει σειρά. Οι λίστες που θα χρησιμοποιηθούν θα είναι οι ίδιες με αυτές που χρησιμοποιήθηκαν στους προηγούμενους kernels. Αυτό σημαίνει ότι μια λίστα δεν περιέχει απαραίτητα πληροφορία μόνο για ένα σημείο του D_2. Το γεγονός αυτό είναι καλό, καθώς μειώνεται πιο γρήγορα η μέγιστη απόσταση που υπάρχει σε μια λίστα

Στο τέλος οι λίστες, που περιέχουν την πληροφορία για τους γείτονες, μεταφέρονται στον host από όπου επιλέγονται τα k ζευγάρια με τις μικρότερες αποστάσεις.

Host DSPP+

```
input(file data1_file, file data2_file, int k, unsigned int
      data1_partition_size, unsigned int
      data2_partition_size, unsigned int partitions, int N) {
  host_vector data1_partition(data1_partition_size);
  host_vector data2_partition = data2_file.read();
```

```
int data1_size = data1_file.size() / sizeof(point_struct);
int data2_size = data2.size();

device_vector d_data1_partition;
device_vector d_data2 = data2;
host_vector distance_list(k * data2_size);
device_vector d_distance_list = distance_list;
host_vector<unsigned int> inserted_list(data2_size, 0);
device_vector<unsigned int> d_inserted_list =
      inserted_list;
host_vector knp(k);
host_vectot<partition_range> xpart;
device_vector<partition_range> d_xpart;

unsigned int data1_backet_size;
unsigned int data2_backet_size;
unsigned int data1_partition_loops = (data1_size – 1) /
      data1_partition_size;

unsigned int data2_PointsOffest;
for (unsigned int p = 0; p < data1_partition_loops; p++) {
   if ((p + 1) * data1_partition_size < data1_size) {
      data1_backet_size = data1_partition_size;
   }
   else {
      data1_backet_size = data1_size – p *
          data1_partition_size;

      if (data1_backet_size >= 2*partitions) {
         partitions_capacity_points = data1_backet_size /
               partitions;

         if (data1_backet_size % partitions != 0) {
            partitions_capacity_points++;
```

```
      }
   }
   else {
      partitions_capacity_points = data1_backet_size;
      partitions = 1;
   }

}

data1_partition = data1_file.read(data1_backet_size);

xpart.clear();
double prevx = data1[0].x;
unsigned int xpart_idx;
unsigned int i = 0;
for (i = 0; i < partitions; i++) {
   ((i + 1) * partitions_capacity_points - 1 <
        data1_backet_size) ? xpart_idx = (i + 1) *
        partitions_capacity_points - 1 : xpart_idx =
        data1_backet_size - 1;
   double x = data1[xpart_idx].x;
   xpart.push_back({ prevx,x });
   prevx = x;
}
d_xpart = xpart;

data2_PointsOffest = 0;
unsigned int q = 0;
while (data2_PointsOffest<data2_points) {
   if ((q+1) * data2_partition_size < data2_points) {
      data2_backet_size = data2_partition_size;
   }
   else {
```

```
            data2_backet_size = data2_points -
                q*data2_partition_size;
        }

        data2_partition =
            data2_file.read(data2_backet_size);
        knp_kernel <<<(data2_size - 1) / N + 1, N >>>
            (d_data1_partition, d_data2, data1_backet_size,
            data2_size, d_distance_list, d_inserted_list,
            d_xpart, k);

        cudaDeviceSyncronize();

        data2_PointsOffest += data2_backet_size;
        q++;
      }
   }

   distance_list = d_distance_list;
   find_knp(knp, distance_list, k);
}
```

4.5 Disk Symmetric Progression Partitioning με Pinned Memory (DSPP+P)

Όπως αναφέρθηκε και στην ενότητα 4.4 η διαμέριση που εφαρμόζεται έχει το πρόβλημα ότι κάνει το D_2 να διαβάζεται τόσες φορές, όσες τα partitions του D_1 αντί για μόνο μία φορά όπως πριν. Αυτές οι επιπλέον αναγνώσεις και μεταφορές δεδομένων στη μνήμη της κάρτας γραφικών μπορεί να είναι ιδιαίτερα χρονοβόρες και καταστροφικές για την επίδοση. Θα πρέπει επομένως να επικαλυφθούν αυτές οι διαδικασίες με χρήσιμους υπολογισμούς, ώστε να περιοριστεί το κόστος αυτό.

Αυτό θα γίνει σε αυτή την παραλλαγή χρησιμοποιώντας τα χαρακτηριστικά που λέγονται pinned memory και concurrent kernel execution.

Αυτός ο αλγόριθμος, όπως και ο προηγούμενος, δέχεται ως εισόδους δύο σύνολα σημείων στον τρισδιάστατο χώρο D_1 και D_2 με m και n σημεία αντίστοιχα και τον αριθμό των κοντινότερων ζευγαριών k.

Αρχικά το D_1 χωρίζεται σε p1 partitions μεγέθους p1_size το καθένα. Αν το m δε διαιρείται ακριβώς με το p1_size, τότε το τελευταίο partition θα έχει μέγεθος ίσο με το υπόλοιπο της διαίρεσης του m με το p1_size.

Τα partitions διαβάζονται διαδοχικά από τον host και μεταφέρονται στο device. Εκεί ταξινομούνται με βάση την τιμή της συντεταγμένης χ και στη συνέχεια μεταφέρονται πίσω στον host, όπου χωρίζονται σε μικρότερα sub-partitions και παράλληλα δημιουργείται και ένα sub-partition index, το οποίο περιέχει πληροφορία για τη μικρότερη και τη μεγαλύτερη τιμή χ σε κάθε sub-partition.

Στη συνέχεια το D_2 χωρίζεται σε p2 partitions μεγέθους p2_size το καθένα, ακριβώς όπως και το D_1. Κάθε φορά που διαβάζεται ένα partition του D_1, διαβάζονται όλα τα partitions του D_2. Η διαφορά σε σχέση με πριν είναι ότι τα partitions του D_2 διαβάζονται και αποθηκεύονται σε ένα διάνυσμα που έχει δεσμευτεί με pinned memory. Ακόμα έχουν δημιουργηθεί S streams, ώστε να επικαλύπτονται οι αναγνώσεις και οι μεταφορές δεδομένων με υπολογισμούς και να μπορούν οι kernels να εκτελούνται ταυτόχρονα και να αξιοποιούν καλύτερα τους πόρους της κάρτας γραφικών. Αυτό σημαίνει ότι οι kernels εκτελούνται σε παρτίδες των S και μετά καλείται η συνάρτηση cudaDeviceSyncronize για να συγχρονιστούν τα streams. Αν η τελευταία παρτίδα δε συμπληρώνει S kernels, τότε η συνάρτηση cudaDeviceSyncronize καλείται μετά την εκτέλεση του τελευταίου kernels.

Για την εκτέλεση του κώδικα στην κάρτα γραφικών χρησιμοποιήθηκαν μονοδιάστατα blocks μεγέθους N. Κάθε thread υπολογίζει τους k κοντινότερους γείτονες για ένα σημείο του D_2. Για να το πετύχει αυτό υπολογίζει όλες τις αποστάσεις ανάμεσα στο συγκεκριμένο σημείο και τα σημεία του προς εξέταση partition, χρησιμοποιώντας το sub-partition index για να μειώσει τον αριθμό των υπολογισμών. Πιο συγκεκριμένα το νήμα διαβάζει το sub-partition index προσπαθώντας να βρει, αν κάποια από τα όρια που υπάρχουν σε αυτό, εμπεριέχουν την τιμή χ του προς εξέταση σημείου του D_2. Αν υπάρχουν τέτοια όρια, τότε το αντίστοιχο sub-partition θα είναι το πρώτο που θα επιλεγεί. Αν δεν υπάρχουν όρια που να ικανοποιούν την παραπάνω συνθήκη, τότε ως πρώτο sub-partition επιλέγεται το αριστερότερο ή το δεξιότερο με κριτήριο πιο από τα δύο βρίσκεται πιο κοντά στον άξονα χ στο προς εξέταση σημείο του D_2. Για το πρώτο sub-partition όλοι οι

απαραίτητοι υπολογισμοί εκτελούνται όπως περιεγράφηκε στις προηγούμενες παραλλαγές του αλγορίθμου. Δηλαδή Οι k πρώτες αποστάσεις που υπολογίζονται εισάγονται σε μία ιδιωτική λίστα, που κανένα άλλο νήμα δεν έχει πρόσβαση. Οι επόμενες αποστάσεις συγκρίνονται με τη μεγαλύτερη απόσταση στη λίστα και εάν είναι μικρότερες τότε εισάγονται σε αυτή. Αφού ολοκληρωθούν οι υπολογισμοί το sub-partition με τη μικρότερη απόσταση στον άξονα χ από το προς εξέταση σημείο του D_2, θα επιλεγεί ως το επόμενο υπό τη συνθήκη ότι η μικρότερη από τις αποστάσεις στον άξονα χ των σημείων του sub-partition με το προς εξέταση σημείο του D_2 να είναι μικρότερη από τη μεγαλύτερη απόσταση στη λίστα που χρησιμοποιεί το νήμα. Το να βρεθεί η συγκεκριμένη απόσταση είναι εύκολο, καθώς τα σημεία του sub-partition είναι ταξινομημένα με βάση την τιμή της συντεταγμένης χ. Αν η συνθήκη δεν ισχύει, τότε το sub-partition δεν ελέγχεται και δεν αναζητείται άλλο στη θέση του. Διαφορετικά εκτελούνται οι απαραίτητοι υπολογισμοί ακριβώς όπως και για το πρώτο sub-partition. Το επόμενο sub-partition επιλέγεται ακολουθώντας την ίδια διαδικασία.

Στο τέλος οι λίστες, που περιέχουν την πληροφορία για τους γείτονες, μεταφέρονται στον host από όπου επιλέγονται τα k ζευγάρια με τις μικρότερες αποστάσεις.

4.6 Βελτιωμένος Disk Symmetric Progression Partitioning με Pinned Memory (DSPP+PEA)

Μια βελτίωση της προηγούμενης παραλλαγής είναι να αυξηθεί ο αριθμός των σημείων του D_2 που αντιστοιχίζονται σε κάθε νήμα και από ένα μόνο σημείο να αντιστοιχίζονται περισσότερα. Αυτό θα λειτουργήσει θετικά, καθώς θα δώσει τη δυνατότητα να μεγαλώσουν τα partitions του D_2, χωρίς να αυξηθεί ο αριθμός των νημάτων που τρέχουν ανά kernel και χωρίς να αυξηθεί ο αριθμός των λιστών που χρησιμοποιούνται. Αυτά είναι θετικά, καθώς κερδίζουμε χρόνο καλώντας λιγότερες φορές τις συναρτήσεις που διαβάζουν από αρχείο και που μεταφέρουν δεδομένα στην κάρτα γραφικών, εκτελούνται λιγότεροι kernels, υπάρχει μικρότερη ανάγκη για συγχρονισμό και άρα εκτελούνται λιγότερες cudaDeviceSyncronize και διατηρείται το μέγεθος των λιστών μικρό.

Αυτός ο αλγόριθμος, όπως και ο προηγούμενος Ο αλγόριθμος, δέχεται ως εισόδους δύο σύνολα σημείων στον τρισδιάστατο χώρο D_1 και D_2 με m και n σημεία αντίστοιχα και τον αριθμό των κοντινότερων ζευγαριών k.

Αρχικά το D_1 χωρίζεται σε p1 partitions μεγέθους p1_size το καθένα. Αν το m δε διαιρείται ακριβώς με το p1_size, τότε το τελευταίο partition θα έχει μέγεθος ίσο με το υπόλοιπο της διαίρεσης του m με το p1_size.

Τα partitions διαβάζονται διαδοχικά από τον host και μεταφέρονται στο device. Εκεί ταξινομούνται με βάση την τιμή της συντεταγμένης χ και στη συνέχεια μεταφέρονται πίσω στον host, όπου χωρίζονται σε μικρότερα sub-partitions και παράλληλα δημιουργείται και ένα sub-partition index, το οποίο περιέχει πληροφορία για τη μικρότερη και τη μεγαλύτερη τιμή χ σε κάθε sub-partition.

Στη συνέχεια το D_2 χωρίζεται σε p2 partitions μεγέθους p2_size το καθένα, ακριβώς όπως και το D_1. Κάθε φορά που διαβάζεται ένα partition του D_1, διαβάζονται όλα τα partitions του D_2. Τα partitions του D_2 διαβάζονται και αποθηκεύονται σε ένα διάνυσμα που έχει δεσμευτεί με pinned memory. Ακόμα έχουμε δημιουργήσει S streams, ώστε να επικαλύπτονται οι αναγνώσεις και οι μεταφορές δεδομένων με υπολογισμούς και να μπορούν οι kernels να εκτελούνται ταυτόχρονα και να αξιοποιούν καλύτερα τους πόρους της κάρτας γραφικών. Αυτό σημαίνει ότι οι kernels εκτελούνται σε παρτίδες των S και μετά καλείται η συνάρτηση cudaDeviceSyncronize για να συγχρονιστούν τα streams. Αν η τελευταία παρτίδα δε συμπληρώνει S kernels, τότε η συνάρτηση cudaDeviceSyncronize καλείται μετά την εκτέλεση του τελευταίου kernels.

Για την εκτέλεση του κώδικα στην κάρτα γραφικών χρησιμοποιήθηκαν μονοδιάστατα blocks μεγέθους N. Κάθε thread υπολογίζει τους k κοντινότερους γείτονες για l σημεία του D_2. Για να το πετύχει αυτό υπολογίζει όλες τις αποστάσεις ανάμεσα σε όλα τα σημεία που του έχουν ανατεθεί διαδοχικά και τα σημεία του προς εξέταση partition, χρησιμοποιώντας το sub-partition index για να μειώσει τον αριθμό των υπολογισμών. Πιο συγκεκριμένα για κάθε ένα από τα σημεία το νήμα διαβάζει το sub-partition index προσπαθώντας να βρει, αν κάποια από τα όρια που υπάρχουν σε αυτό, εμπεριέχουν την τιμή χ του προς εξέταση σημείου του D_2. Αν υπάρχουν τέτοια όρια, τότε το αντίστοιχο sub-partition θα είναι το πρώτο που θα επιλεγεί. Αν δεν υπάρχουν όρια που να ικανοποιούν την παραπάνω συνθήκη, τότε ως πρώτο sub-partition επιλέγεται το αριστερότερο ή το δεξιότερο με κριτήριο ποιο από τα δύο βρίσκεται πιο κοντά στον άξονα

χ στο προς εξέταση σημείο του D_2. Για το πρώτο sub-partition όλοι οι απαραίτητοι υπολογισμοί εκτελούνται όπως περιεγράφηκε στις προηγούμενες παραλλαγές του αλγορίθμου. Δηλαδή οι k πρώτες αποστάσεις που υπολογίζονται, εισάγονται σε μία ιδιωτική λίστα, που κανένα άλλο νήμα δεν έχει πρόσβαση. Οι επόμενες αποστάσεις συγκρίνονται με τη μεγαλύτερη απόσταση στη λίστα και εάν είναι μικρότερες τότε εισάγονται σε αυτή. Αφού ολοκληρωθούν οι υπολογισμοί το sub-partition με τη μικρότερη απόσταση στον άξονα χ από το προς εξέταση σημείο του D_2, θα επιλεγεί ως το επόμενο υπό τη συνθήκη ότι η μικρότερη από τις αποστάσεις στον άξονα χ των σημείων του sub-partition με το προς εξέταση σημείο του D_2 να είναι μικρότερη από τη μεγαλύτερη απόσταση στη λίστα που χρησιμοποιεί το νήμα. Το να βρεθεί η συγκεκριμένη απόσταση είναι εύκολο, καθώς τα σημεία του sub-partition είναι ταξινομημένα με βάση την τιμή της συντεταγμένης χ. Αν η συνθήκη δεν ισχύει, τότε το sub-partition δεν ελέγχεται και δεν αναζητείται άλλο στη θέση του και προχωράμε στην εξέταση του επόμενου σημείου, αν υπάρχει, που έχει ανατεθεί στο νήμα. Διαφορετικά εκτελούνται οι απαραίτητοι υπολογισμοί ακριβώς όπως και για το πρώτο sub-partition. Το επόμενο sub-partition επιλέγεται ακολουθώντας την ίδια διαδικασία.

Στο τέλος οι λίστες, που περιέχουν την πληροφορία για τους γείτονες, μεταφέρονται στον host από όπου επιλέγονται τα k ζευγάρια με τις μικρότερες αποστάσεις.

4.7 kNP Distance List Buffer

Η πρώτη από τις δύο υλοποιήσεις που έχουμε για τις λίστες που αποθηκεύουν την πληροφορία για τα k κοντινότερα ζευγάρια είναι η kNP Distance List Buffer (KNP-DLB για συντομία). Σ’ αυτήν την υλοποίηση, όταν θέλουμε να εισάγουμε ένα στοιχείο, εάν η λίστα δεν είναι γεμάτη, τότε απλά το προσθέτουμε στην πρώτη κοινή θέση. Αν είναι γεμάτη διαβάζουμε σειριακά τη λίστα, εντοπίζουμε το ζευγάρι με τη μεγαλύτερη απόσταση και αν αυτή είναι μεγαλύτερη από την απόσταση του ζευγαριού που θέλουμε να εισάγουμε, τότε το αντικαθιστούμε. Ο χρόνος που απαιτείται για την αναζήτηση της μέγιστης απόστασης και της εισαγωγής ενός καινούριου ζευγαριού είναι O(k).

4.8 kNP Max Heap Distance List Buffer

Η δεύτερη υλοποίηση για τις λίστες ονομάζεται kNP Max Heap Distance List Buffer (KNP-MHDLB για συντομία). Στην ουσία είναι ενός σωρός μεγίστου υλοποιημένος με πίνακα με k+1 θέσεις, καθώς η πρώτη θέση χρησιμοποιείται ως τερματικό και έχει απόσταση ίση με τη μέγιστη τιμή που μπορεί να πάρει ένας πραγματικός αριθμός διπλής ακρίβειας. Ο χρόνος που απαιτείται για την αναζήτηση της μέγιστης απόστασης και της εισαγωγής ενός καινούριου ζευγαριού είναι O(logk) και επομένως περιμένουμε να λειτουργεί καλά για μεγάλες τιμές του k.

Κεφάλαιο 5 Πειραματική μελέτη

Εκτελείται ένα μεγάλο πλήθος πειραμάτων, ώστε να εξεταστεί η επίδοση των αλγορίθμων. Γίνονται δοκιμές για μεγάλο πλήθος και για διάφορα μεγέθη συνόλων δεδομένων, καθώς και για πολλές τιμές του k. Για την αποθήκευση των αποστάσεων χρησιμοποιούνται πραγματικοί αριθμοί διπλής ακρίβειας για να γίνεται διάκριση των τιμών, όταν αυτές είναι κοντά στο μηδέν.

Εκτελούνται πειράματα για να μελετηθούν οι αλγορίθμοι σχετικά με την επίδοση και τη χρήση της μνήμης. Χρησιμοποιούνται συνολικά 12 αλγόριθμοι.

1. DBFS, Disk Brute-force Simple χρησιμοποιώντας KNP-DLB
2. DBFS Heap, Disk Brute-force Simple χρησιμοποιώντας max-Heap buffer
3. DBF, Disk Brute-force χρησιμοποιώντας KNP-DLB
4. DBF Heap, Disk Brute-force χρησιμοποιώντας max-Heap buffer
5. DSPP, Disk Symmetric Progression Partitioning χρησιμοποιώντας KNP-DLB
6. DSPP Heap, Disk Symmetric Progression Partitioning χρησιμοποιώντας max-Heap buffer
7. DSPP+, Improved Disk Symmetric Progression Partitioning χρησιμοποιώντας KNP-DLB
8. DSPP+ Heap, Improved Disk Symmetric Progression Partitioning χρησιμοποιώντας max- Heap buffer
9. DSPP+P, Improved Disk Symmetric Progression Partitioning with pinned memory χρησιμοποιώντας KNP-DLB
10. DSPP+P Heap, Improved Disk Symmetric Progression Partitioning with pinned memory χρησιμοποιώντας max-Heap buffer
11. DSPP+PEA, Improved Disk Symmetric Progression Partitioning with pinned memory with Enhanced Assignment χρησιμοποιώντας KNP-DLB
12. DSPP+PEA Heap, Improved Disk Symmetric Progression Partitioning with pinned memory Enhanced Assignment χρησιμοποιώντας max-Heap buffer

Οι κώδικες 1 και 2 δε θα εξεταστούν, καθώς δεν είναι φτιαγμένοι για μεγάλα σύνολα δεδομένων.

Τα πειράματα εκτελέστηκαν στην πλατφόρμα google colab, που παρέχει δωρεάν πρόσβαση σε κάρτα γραφικών.

5.1 Πειράματα με συνθετικά δεδομένα

Σε αυτή την ενότητα θα μελετήσουμε τους αλγορίθμους χρησιμοποιώντας μόνο συνθετικά δεδομένα. Τα δεδομένα δημιουργήθηκαν χρησιμοποιώντας τον SpiderWeb generator.

Ο SpiderWeb generator παράγει αρχεία που περιέχουν δισδιάστατα σημεία, που οι συντεταγμένες τους είναι κανονικοποιημένες και παίρνουν τιμές από 0 έως 1. Επίσης δίνεται η δυνατότητα επιλογής της κατανομής που ακολουθούν τα σημεία ανάμεσα σε διάφορες κατανομές. Τα σημεία δημιουργούνται τυχαία με βάση έναν κώδικα (seed). Για τα πειράματα χρησιμοποιήθηκαν 3 διαφορετικές κατανομές σημείων στον τρισδιάστατο χώρο. Ο τρόπος με τον οποίο δημιουργήθηκαν τα αρχεία για κάθε κατανομή ακολουθεί τα παρακάτω βήματα. Αρχικά παράγονται 4 αρχεία με την ίδια κατανομή, 2 με 5M σημεία, που θα χρησιμοποιηθούν για τη δημιουργία D_1 αρχείων και 2 με 500K σημεία για τη δημιουργία D_2 αρχείων. Όλα τα αρχεία δημιουργούνται με διαφορετικό κωδικό. Κάθε φορά που θα πρέπει να δημιουργηθεί ένα D_1 ή D_2 αρχείο επιλέγονται τα αντίστοιχα 2 αρχεία. Από το πρώτο από τα 2 αρχεία και για τον επιθυμητό αριθμό σημείων επιλέγονται οι συντεταγμένες τους, που θα αποτελούν τις συντεταγμένες x, y των νέων σημείων και από το δεύτερο αρχείο και για τον επιθυμητό αριθμό σημείων επιλέγεται η συντεταγμένη x, που θα αποτελέσει τη συντεταγμένη z των σημείων.

Τα D_1 αρχεία περιέχουν από 1M έως 5M τρισδιάστατα σημεία, που ακολουθούν τις κατανομές bit, κανονική και ομοιόμορφη κι έχουν μεγέθη από 32 MB έως 160 MB. Τα D_2 αρχεία περιέχουν από 100K έως 500K τρισδιάστατα σημεία, που ακολουθούν και αυτά τις ίδιες κατανομές κι έχουν μεγέθη από 3.2 MB έως 16 MB. Ο Πίνακας 5-1 περιέχει πληροφορίες για τα αρχεία που ακολουθούν bit κατανομή, ο Πίνακας 5-2 για τα αρχεία που ακολουθούν κανονική κατανομή και ο Πίνακας 5-3 για τα αρχεία που ακολουθούν ομοιόμορφη κατανομή.

Πίνακας 5-1: Συνθετικά δεδομένα bit κατανομή

Κατανομή	Μέγεθος	Μέγεθος αρχείου	Χρήση συνόλου δεδομένων
Bit	1 M	32 MB	D_1
Bit	2 M	64 MB	D_1
Bit	3 M	96 MB	D_1
Bit	4 M	128 MB	D_1
Bit	5 M	160 MB	D_1
Bit	100 K	3.2 MB	D_2
Bit	200 K	6.4 MB	D_2
Bit	300 K	9.6 MB	D_2
Bit	400 K	12.8 MB	D_2
Bit	500 K	16 MB	D_2

Πίνακας 5-2: Συνθετικά δεδομένα κανονική κατανομή

Κατανομή	Μέγεθος	Μέγεθος αρχείου	Χρήση συνόλου δεδομένων
Κανονική	1 M	32 MB	D_1
Κανονική	2 M	64 MB	D_1
Κανονική	3 M	96 MB	D_1
Κανονική	4 M	128 MB	D_1
Κανονική	5 M	160 MB	D_1
Κανονική	100 K	3.2 MB	D_2
Κανονική	200 K	6.4 MB	D_2
Κανονική	300 K	9.6 MB	D_2
Κανονική	400 K	12.8 MB	D_2
Κανονική	500 K	16 MB	D_2

Πίνακας 5-3: Συνθετικά δεδομένα ομοιόμορφη κατανομή

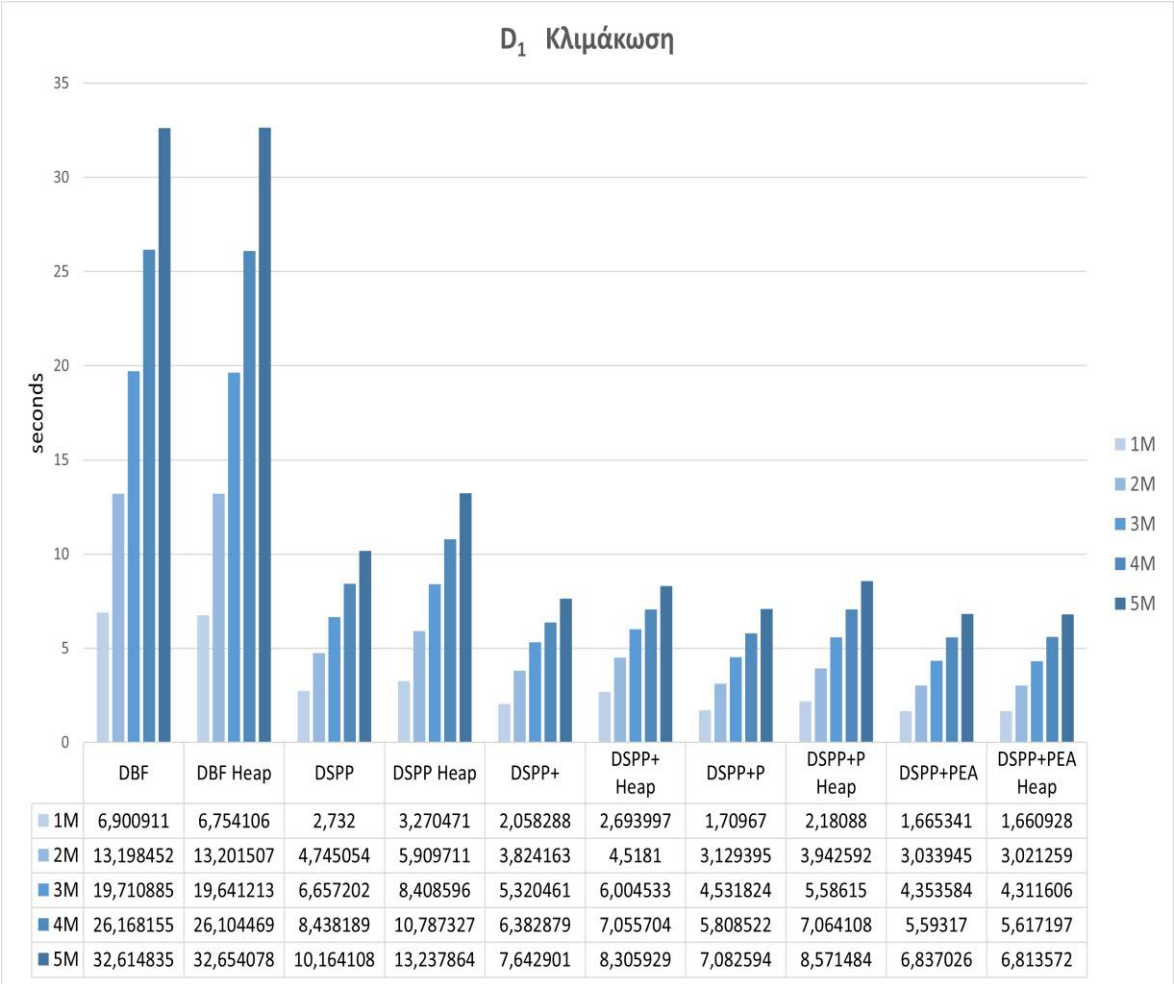
Κατανομή	Μέγεθος	Μέγεθος αρχείου	Χρήση συνόλου δεδομένων
Ομοιόμορφη	1 M	32 MB	D_1
Ομοιόμορφη	2 M	64 MB	D_1
Ομοιόμορφη	3 M	96 MB	D_1
Ομοιόμορφη	4 M	128 MB	D_1
Ομοιόμορφη	5 M	160 MB	D_1
Ομοιόμορφη	100 K	3.2 MB	D_2
Ομοιόμορφη	200 K	6.4 MB	D_2
Ομοιόμορφη	300 K	9.6 MB	D_2
Ομοιόμορφη	400 K	12.8 MB	D_2
Ομοιόμορφη	500 K	16 MB	D_2

Εκτελέστηκαν τρία διαφορετικά πειράματα με συνθετικά δεδομένα κλιμακώνοντας το μέγεθος του D_1, κλιμακώνοντας το μέγεθος του D_2 και κλιμακώνοντας την τιμή K.

5.1.1 Κλιμάκωση του D_1 με bit κατανομή

Σ’ αυτό το πείραμα κλιμακώνεται το μέγεθος του D_1, που ο αριθμός των σημείων που περιέχει κυμαίνεται από 1M έως 5M με το D_2 να έχει σταθερό αριθμό σημείων ίσο με 100K και την τιμή k να είναι ίση με 20.

Κρατήθηκε σταθερό το μέγεθος D_2 και η τιμή του k για να εστιάσουμε στην κλιμάκωση του D_1.



	DBF	DBF Heap	DSPP	DSPP Heap	DSPP+	DSPP+ Heap	DSPP+P	DSPP+P Heap	DSPP+PEA	DSPP+PEA Heap
1M	6,900911	6,754106	2,732	3,270471	2,058288	2,693997	1,70967	2,18088	1,665341	1,660928
2M	13,198452	13,201507	4,745054	5,909711	3,824163	4,5181	3,129395	3,942592	3,033945	3,021259
3M	19,710885	19,641213	6,657202	8,408596	5,320461	6,004533	4,531824	5,58615	4,353584	4,311606
4M	26,168155	26,104469	8,438189	10,787327	6,382879	7,055704	5,808522	7,064108	5,59317	5,617197
5M	32,614835	32,654078	10,164108	13,237864	7,642901	8,305929	7,082594	8,571484	6,837026	6,813572

Γράφημα 5-1: Κλιμάκωση του D_1 με bit κατανομή

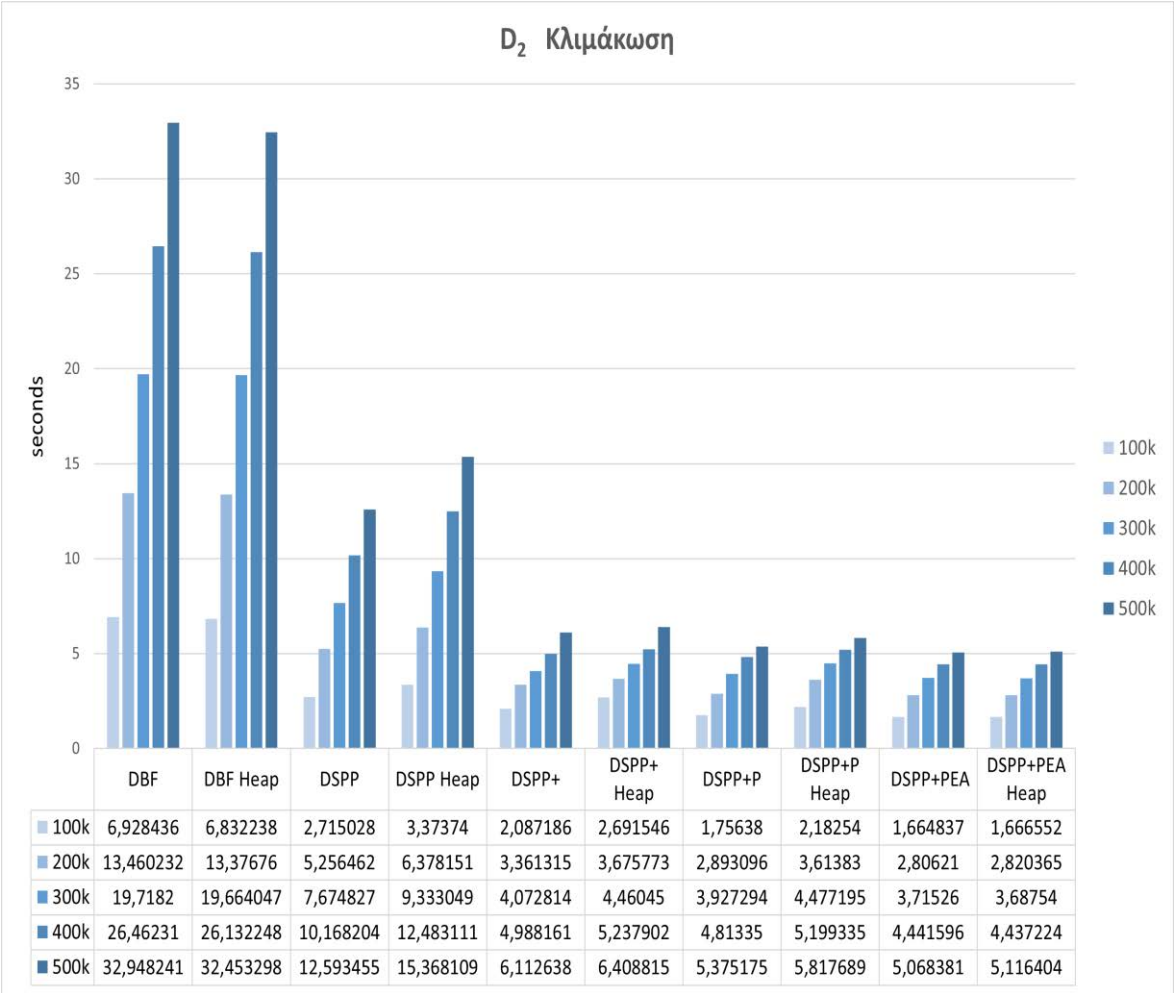
Στο Γράφημα 5-1 φαίνονται τα πειραματικά αποτελέσματα. Παρατηρούμε ότι ο πιο αποδοτικός αλγόριθμος είναι ο DSPP+PEA. Επίσης παρατηρούμε ότι η μέθοδος χωρικής υποδιαίρεσης που χρησιμοποιήθηκε ήταν απόλυτα επιτυχημένη, καθώς μείωσε σημαντικά τον χρόνο εκτέλεσης. Ακόμα παρατηρούμε ότι η χρήση του pinned memory και του concurrent kernel execution βελτίωσε την επίδοση στις περιπτώσεις, όπου χρησιμοποιήθηκε η KNP-DLB λίστα, ενώ όταν χρησιμοποιήθηκε η max-Heap buffer λίστα οι επιδόσεις ήταν παρόμοιες. Αυτό συνέβη καθώς ο DSPP+ και ο DSPP αποδίδουν παρόμοια, πράγμα εντελώς απρόσμενο (αυτό το συμπέρασμα δε θα ισχύει αργότερα, όταν θα κλιμακώσουμε το μέγεθος του D_2 και την τιμή k). Καθώς όπως αναφέρθηκε προηγουμένως εξαιτίας της διαμέρισης που γίνεται στο D_2 το σύνολο αυτό διαβάζεται τόσες φορές όσα τα partitions του D_1. Ωστόσο ο DSPP+ έχει και κάποια πλεονεκτήματα σε σχέση με τον DSPP. Ο αλγόριθμος αυτός χρησιμοποιεί πολύ λιγότερα νήματα ανά

εκτέλεση kernel, πράγμα που βοηθά την καλύτερη χρήση των μνημών cache, καθώς και στην επίτευξη coalescing στην προσπέλαση των στοιχείων D_2. Ακόμη μειώνεται ο αριθμός των νημάτων που τρέχουν ενεργά ανά εκτέλεση και kernel και άρα μειώνεται ο αριθμός των λιστών που αποθηκεύουν την πληροφορία για τα ζευγάρια και επομένως μειώνεται ο χρόνος που απαιτείται για να αρχικοποιήθουν αυτές οι λίστες και να μεταφερθούν πίσω στον host, μειώνεται ο χρόνος επεξεργασίας για την εύρεση των k κοντινότερων ζευγαριών και μειώνεται και ο συνολικός αριθμός εισαγωγών στις λίστες, όπως αναλύθηκε στην ενότητα 4.4. Επίσης όταν χρησιμοποιείται max-Heap buffer ο DSPP+P σε σχέση με τον DSPP+ αποδίδουν παρόμοια, καθώς το k είναι πολύ μικρό και άρα η max-Heap buffer δε λειτουργεί τόσο αποδοτικά και ο μικρότερος αριθμός λιστών (και άρα και μικρότερος αριθμός εισαγωγών) στον DSPP+ λειτουργεί καλύτερα.

5.1.2 Κλιμάκωση του D_2 με bit κατανομή

Σ’ αυτό το πείραμα κλιμακώνεται το μέγεθος του D_2, που ο αριθμός των σημείων που περιέχει κυμαίνεται από 100K έως 500K με το D_1 να έχει σταθερό αριθμό σημείων ίσο με 1M και την τιμή k να είναι ίση με 20.

Κρατήθηκε σταθερό το μέγεθος D_1 και η τιμή του k για να εστιάσουμε στην κλιμάκωση του D_2.



	DBF	DBF Heap	DSPP	DSPP Heap	DSPP+	DSPP+ Heap	DSPP+P	DSPP+P Heap	DSPP+PEA	DSPP+PEA Heap
100k	6,928436	6,832238	2,715028	3,37374	2,087186	2,691546	1,75638	2,18254	1,664837	1,666552
200k	13,460232	13,37676	5,256462	6,378151	3,361315	3,675773	2,893096	3,61383	2,80621	2,820365
300k	19,7182	19,664047	7,674827	9,333049	4,072814	4,46045	3,927294	4,477195	3,71526	3,68754
400k	26,46231	26,132248	10,168204	12,483111	4,988161	5,237902	4,81335	5,199335	4,441596	4,437224
500k	32,948241	32,453298	12,593455	15,368109	6,112638	6,408815	5,375175	5,817689	5,068381	5,116404

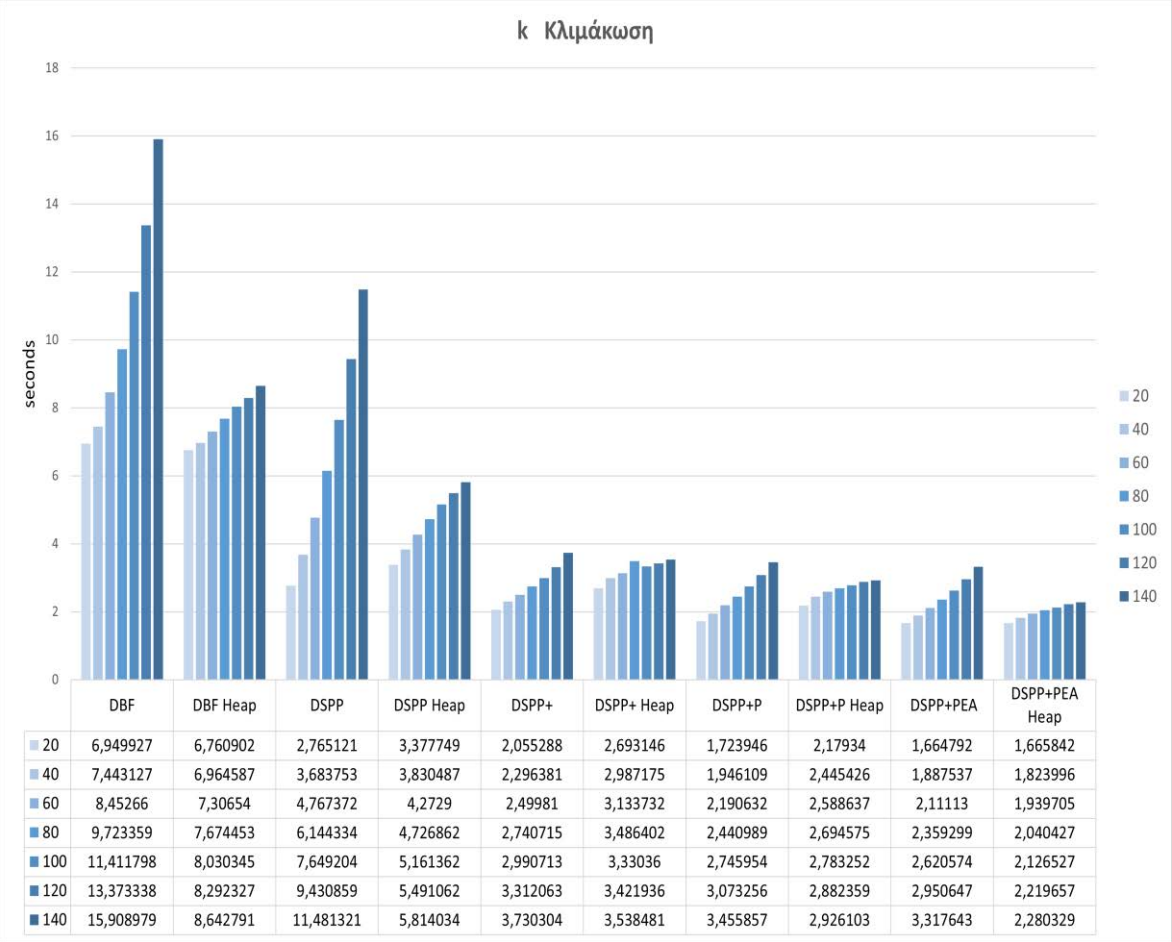
Γράφημα 5-2: Κλιμάκωση του D_2 με bit κατανομή

Στο Γράφημα 5-2 φαίνονται τα πειραματικά αποτελέσματα. Παρατηρούμε ότι ο πιο αποδοτικός αλγόριθμος είναι ο DSPP+PEA. Επίσης παρατηρούμε ότι η μέθοδος χωρικής υποδιαίρεσης που χρησιμοποιήθηκε ήταν απόλυτα επιτυχημένη, καθώς μείωσε σημαντικά τον χρόνο εκτέλεσης. Ακόμα παρατηρούμε ότι η χρήση του pinned memory και του concurrent kernel execution βελτίωσε την επίδοση στις περιπτώσεις, όπου χρησιμοποιήθηκε η KNP-DLB buffer λίστα, ενώ όταν χρησιμοποιήθηκε η max-Heap buffer λίστα οι επιδόσεις ήταν παρόμοιες για μικρά μεγέθη του D_2 και καλύτερη όταν το D_2 είχε μέγεθος 500 K. Αυτό συμβαίνει καθώς ο DSPP+ και ο DSPP αποδίδουν παρόμοια και το K είναι μικρό και ισχύουν αυτά που εξηγήθηκε πριν στο 5.1.1.

5.1.3 Κλιμάκωση του k με bit κατανομή

Σ’ αυτό το πείραμα κλιμακώνεται το k, που παίρνει τιμές που κυμαίνονται από 20 έως 140 με το D_1 και το D_2 να έχουν σταθερό αριθμό σημείων ίσο με 1M και 100K αντίστοιχα.

Κρατήθηκαν σταθερά τα μεγέθη του D_1 και του D_2 για να εστιάσουμε στην κλιμάκωση του k.



	DBF	DBF Heap	DSPP	DSPP Heap	DSPP+	DSPP+ Heap	DSPP+P	DSPP+P Heap	DSPP+PEA	DSPP+PEA Heap
20	6,949927	6,760902	2,765121	3,377749	2,055288	2,693146	1,723946	2,17934	1,664792	1,665842
40	7,443127	6,964587	3,683753	3,830487	2,296381	2,987175	1,946109	2,445426	1,887537	1,823996
60	8,45266	7,30654	4,767372	4,2729	2,49981	3,133732	2,190632	2,588637	2,11113	1,939705
80	9,723359	7,674453	6,144334	4,726862	2,740715	3,486402	2,440989	2,694575	2,359299	2,040427
100	11,411798	8,030345	7,649204	5,161362	2,990713	3,33036	2,745954	2,783252	2,620574	2,126527
120	13,373338	8,292327	9,430859	5,491062	3,312063	3,421936	3,073256	2,882359	2,950647	2,219657
140	15,908979	8,642791	11,481321	5,814034	3,730304	3,538481	3,455857	2,926103	3,317643	2,280329

Γράφημα 5-3: Κλιμάκωση του k με bit κατανομή

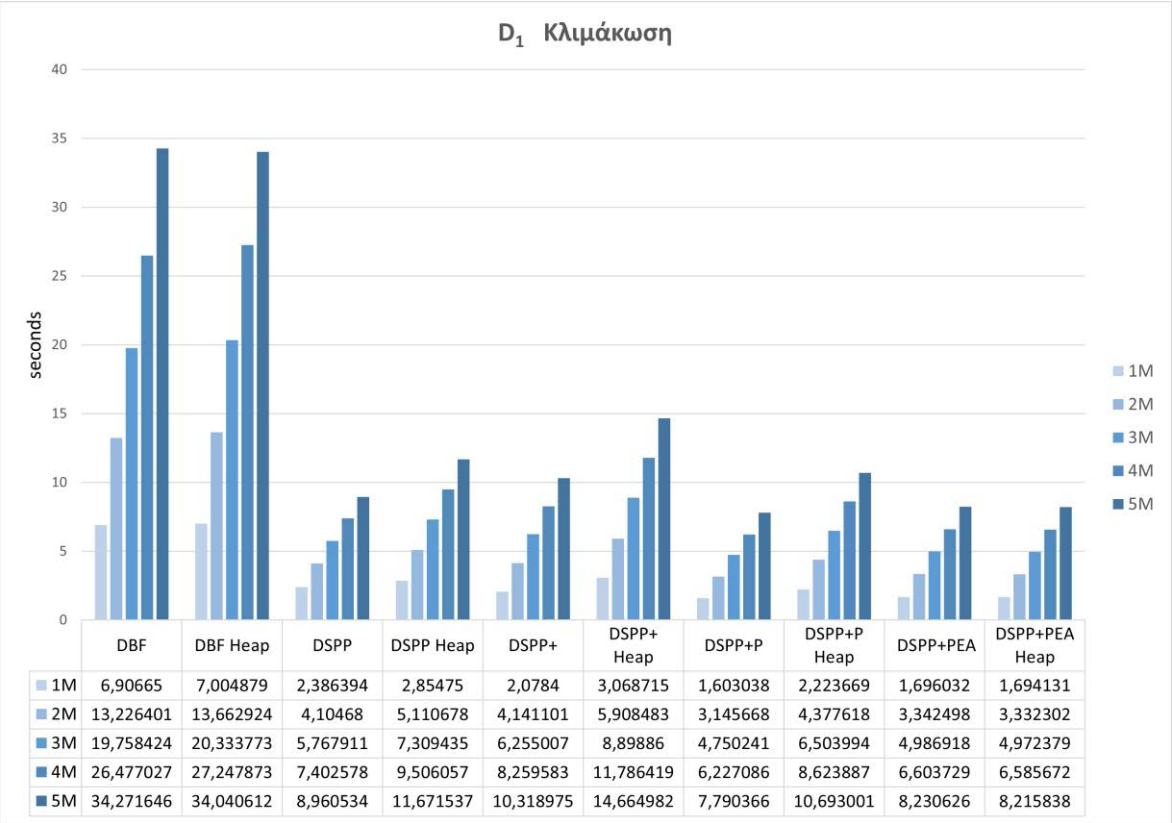
Στο Γράφημα 5-3 φαίνονται τα πειραματικά αποτελέσματα. Παρατηρούμε ότι ο πιο αποδοτικός αλγόριθμος είναι ο DSPP+PEA. Επίσης παρατηρούμε ότι η μέθοδος χωρικής υποδιαίρεσης που χρησιμοποιήθηκε ήταν απόλυτα επιτυχημένη, καθώς μείωσε σημαντικά τον χρόνο εκτέλεσης. Ακόμα παρατηρούμε ότι η χρήση του pinned memory και του concurrent kernel execution βελτίωσε την επίδοση στην περίπτωση, που χρησιμοποιήθηκε η KNP-DLB buffer λίστα και όταν χρησιμοποιήθηκε η max-Heap buffer λίστα και ειδικά για μεγάλα K. Αυτό συμβαίνει καθώς με την πάροδο του χρόνου ο αριθμός

των εισαγωγών στις λίστες μειώνεται. Επομένως μειώνονται και οι υπολογισμοί μέσα σε κάθε kernel, που εκτελείται. Άρα μειώνεται και ο βαθμός επικάλυψης διαβάσματος και μεταφοράς δεδομένων στη μνήμη της κάρτας γραφικών, το οποίο κάνει τον DSPP+P και τον DSPP+ να λειτουργούν σε αναλογικά πιο κοντινούς χρόνους. Αλλά όταν το k μεγαλώνει ο αριθμός των υπολογισμών αυξάνεται και ο βαθμός επικάλυψης μπορεί να διατηρηθεί μεγάλος ακόμα και στα πιο μετέπειτα στάδια εκτέλεσης. Άρα η διαφορά ανάμεσα στον DSPP+P και στον DSPP+ γίνεται πιο αισθητή. Τέλος παρατηρούμε ότι η max-Heap buffer λίστα ήταν πιο αποδοτική από την KNP-DLB buffer λίστα, ειδικά όταν χρησιμοποιήθηκαν μεγάλα k, με εξαίρεση τις περιπτώσεις DSPP+P και του DSPP+. Στην περίπτωση του DSPP+P η max-Heap buffer λίστα άρχισε να λειτουργεί καλύτερα για k μεγαλύτερο ή ίσο του 100 και στην περίπτωση του DSPP+ λειτουργούσε καλύτερα μόνο για k 120.

5.1.4 Κλιμάκωση του D_1 με κανονική κατανομή

Σ’ αυτό το πείραμα κλιμακώνεται το μέγεθος του D_1, που ο αριθμός των σημείων που περιέχει κυμαίνεται από 1M έως 5M με το D_2 να έχει σταθερό αριθμό σημείων ίσο με 100K και την τιμή k να είναι ίση με 20.

Κρατήθηκε σταθερό το μέγεθος D_2 και η τιμή του k για να εστιάσουμε στην κλιμάκωση του D_1.



	DBF	DBF Heap	DSPP	DSPP Heap	DSPP+	DSPP+ Heap	DSPP+P	DSPP+P Heap	DSPP+PEA	DSPP+PEA Heap
1M	6,90665	7,004879	2,386394	2,85475	2,0784	3,068715	1,603038	2,223669	1,696032	1,694131
2M	13,226401	13,662924	4,10468	5,110678	4,141101	5,908483	3,145668	4,377618	3,342498	3,332302
3M	19,758424	20,333773	5,767911	7,309435	6,255007	8,89886	4,750241	6,503994	4,986918	4,972379
4M	26,477027	27,247873	7,402578	9,506057	8,259583	11,786419	6,227086	8,623887	6,603729	6,585672
5M	34,271646	34,040612	8,960534	11,671537	10,318975	14,664982	7,790366	10,693001	8,230626	8,215838

Γράφημα 5-4: Κλιμάκωση του D_1 με κανονική κατανομή

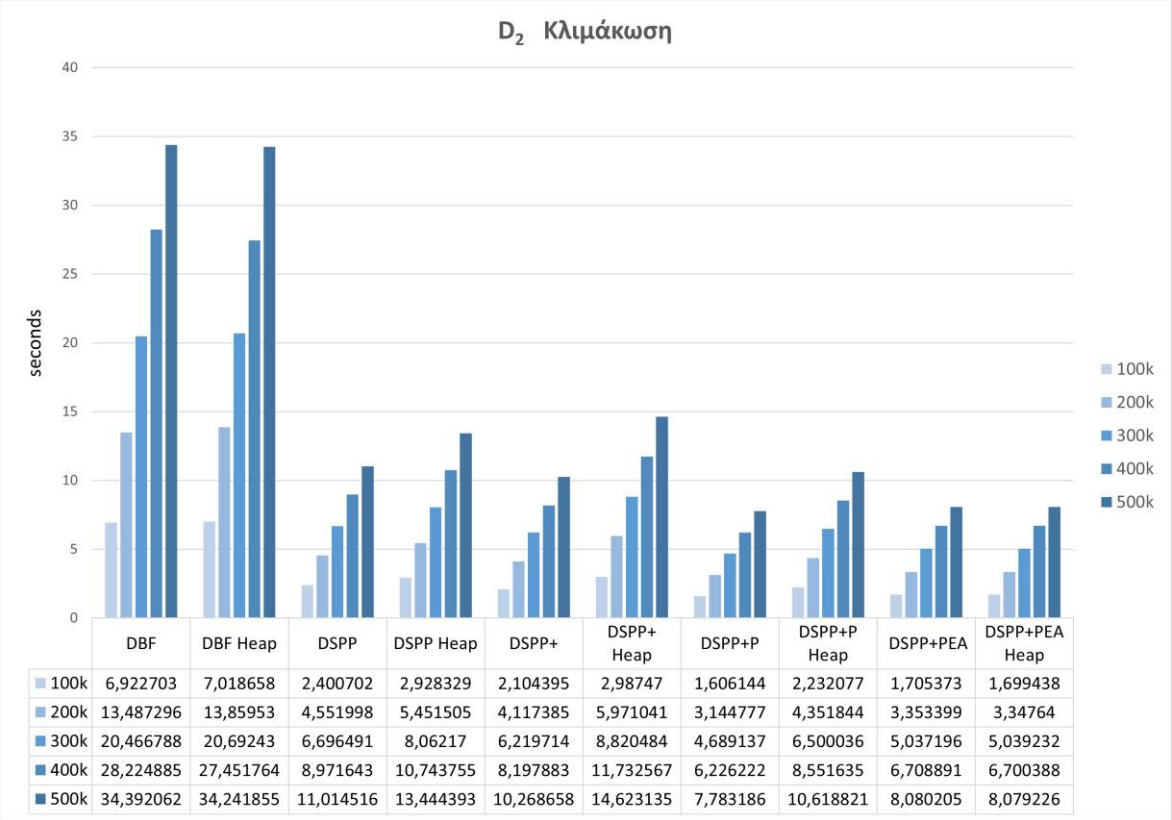
Στο Γράφημα 5-4 φαίνονται τα πειραματικά αποτελέσματα. Παρατηρούμε ότι όταν χρησιμοποιείται max-Heap buffer είναι ο DSPP+PEA είναι πιο αποδοτικός από τον αντίστοιχο DSPP+P ενώ όταν χρησιμοποιείται KNP-DLB ο DSPP+P είναι πιο αποδοτικός από τον αντίστοιχο DSPP+PEA και είναι και ο πιο γρήγορος από όλους τους άλλους. Επίσης παρατηρούμε ότι η μέθοδος χωρικής υποδιαίρεσης που χρησιμοποιήθηκε ήταν απόλυτα επιτυχημένη, καθώς μείωσε σημαντικά τον χρόνο εκτέλεσης. Ακόμα παρατηρούμε ότι η χρήση του pinned memory και του concurrent kernel execution βελτίωσε την επίδοση.

Παρατηρείται ότι η DSPP+PEA είναι λίγο πιο αργή σε σχέση με την αντίστοιχη περίπτωση στη bit κατανομή. Επίσης παρατηρείται ότι η DSPP+ είναι πιο γρήγορη σε σχέση με την αντίστοιχη bit περίπτωση. Αυτό συμβαίνει, καθώς όπως αναφέρθηκε στην ενότητα 4.3, η μέθοδος χωρικής υποδιαίρεσης, που χρησιμοποιείται, μπορεί να επηρεαστεί από την κατανομή των σημείων και δεδομένου ότι οι παράμετροι ανάμεσα στην DSPP+ και στην DSPP+PEA είναι ελαφρώς διαφορετικές, μπορούν να επηρεάσουν μια σειρά πραγμάτων, όπως τα μοτίβα προσπέλασης μνήμης, τη χρήση των caches και των warp divergence. Έτσι μπορούν και προκαλούν αυτές τις αποκλίσεις του χρόνου.

5.1.5 Κλιμάκωση του D_2 με κανονική κατανομή

Σ’ αυτό το πείραμα κλιμακώνεται το μέγεθος του D_2, που ο αριθμός των σημείων που περιέχει κυμαίνεται από 100K έως 500K με το D_1 να έχει σταθερό αριθμό σημείων ίσο με 1M και την τιμή k να είναι ίση με 20.

Κρατήθηκε σταθερό το μέγεθος D_1 και η τιμή του k για να εστιάσουμε στην κλιμάκωση του D_2.



	DBF	DBF Heap	DSPP	DSPP Heap	DSPP+	DSPP+ Heap	DSPP+P	DSPP+P Heap	DSPP+PEA	DSPP+PEA Heap
100k	6,922703	7,018658	2,400702	2,928329	2,104395	2,98747	1,606144	2,232077	1,705373	1,699438
200k	13,487296	13,85953	4,551998	5,451505	4,117385	5,971041	3,144777	4,351844	3,353399	3,34764
300k	20,466788	20,69243	6,696491	8,06217	6,219714	8,820484	4,689137	6,500036	5,037196	5,039232
400k	28,224885	27,451764	8,971643	10,743755	8,197883	11,732567	6,226222	8,551635	6,708891	6,700388
500k	34,392062	34,241855	11,014516	13,444393	10,268658	14,623135	7,783186	10,618821	8,080205	8,079226

Γράφημα 5-5: Κλιμάκωση του D_2 με κανονική κατανομή

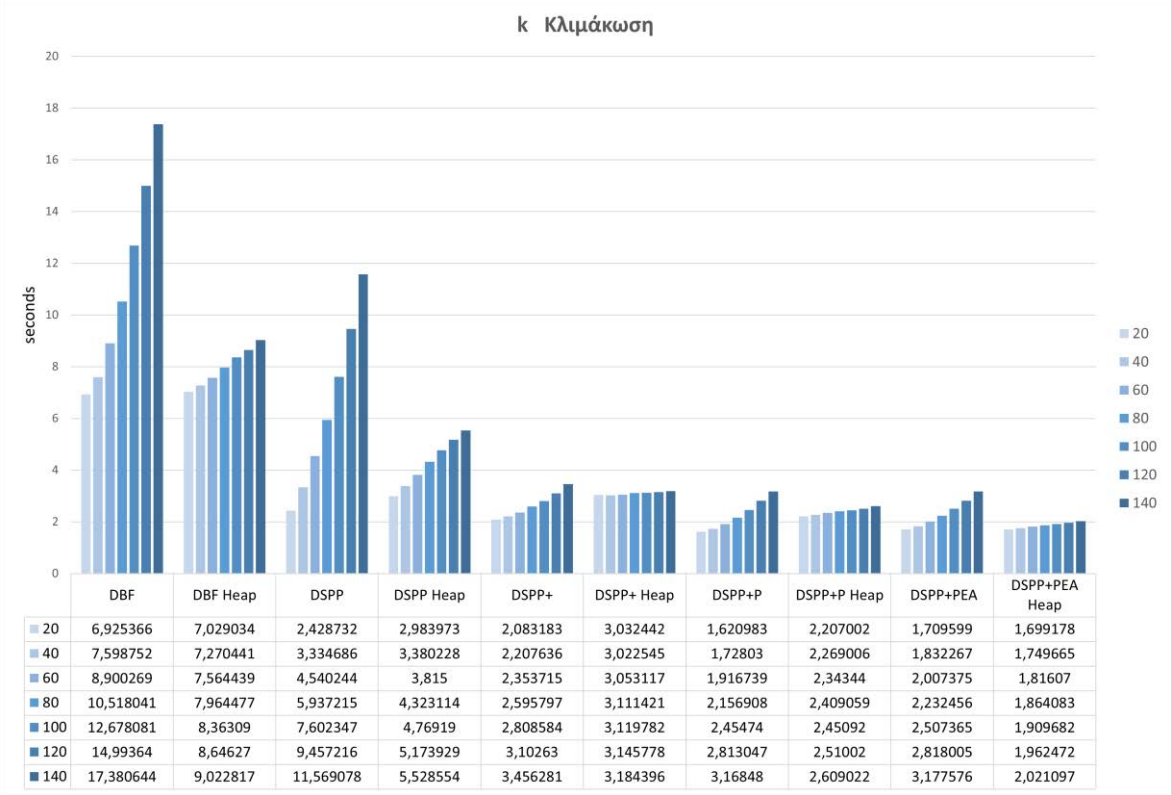
Στο Γράφημα 5-5 φαίνονται τα πειραματικά αποτελέσματα, τα οποία είναι παρόμοια με του 5.1.4. Παρατηρούμε ότι όταν χρησιμοποιείται max-Heap buffer είναι ο DSPP+PEA είναι πιο αποδοτικός από τον αντίστοιχο DSPP+P ενώ όταν χρησιμοποιείται KNP-DLB ο DSPP+P είναι πιο αποδοτικός από τον αντίστοιχο DSPP+PEA και είναι και ο πιο γρήγορος από όλους τους άλλους. Επίσης παρατηρούμε ότι η μέθοδος χωρικής υποδιαίρεσης που χρησιμοποιήθηκε ήταν απόλυτα επιτυχημένη, καθώς μείωσε σημαντικά τον χρόνο εκτέλεσης. Ακόμα παρατηρούμε ότι η χρήση του pinned memory και του concurrent kernel execution βελτίωσε την επίδοση.

Παρατηρείται και σε αυτή την περίπτωση ότι οι χρόνοι του DSPP+PEA αυξάνονται, ενώ οι χρόνοι του DSPP+ μειώνονται, σε σχέση με το αντίστοιχο πείραμα της bit κατανομής. Αυτό συμβαίνει για τους ίδιους λόγους που αναφέρονται στην ενότητα 5.1.4, δηλαδή η αλλαγή στο μοτίβο προσπέλασης στη μνήμη, τη χρήση των caches και των word divergence, που μπορούν να προκαλούν αυτές τις αποκλίσεις του χρόνου.

5.1.6 Κλιμάκωση του k με κανονική κατανομή

Σ' αυτό το πείραμα κλιμακώνεται το k, που παίρνει τιμές που κυμαίνονται από 20 έως 140 με το D_1 και το D_2 να έχουν σταθερό αριθμό σημείων ίσο με 1M και 100K αντίστοιχα.

Κρατήθηκαν σταθερά τα μεγέθη του D_1 και του D_2 για να εστιάσουμε στην κλιμάκωση του k.



	DBF	DBF Heap	DSPP	DSPP Heap	DSPP+	DSPP+ Heap	DSPP+P	DSPP+P Heap	DSPP+PEA	DSPP+PEA Heap
20	6,925366	7,029034	2,428732	2,983973	2,083183	3,032442	1,620983	2,207002	1,709599	1,699178
40	7,598752	7,270441	3,334686	3,380228	2,207636	3,022545	1,72803	2,269006	1,832267	1,749665
60	8,900269	7,564439	4,540244	3,815	2,353715	3,053117	1,916739	2,34344	2,007375	1,81607
80	10,518041	7,964477	5,937215	4,323114	2,595797	3,111421	2,156908	2,409059	2,232456	1,864083
100	12,678081	8,36309	7,602347	4,76919	2,808584	3,119782	2,45474	2,45092	2,507365	1,909682
120	14,99364	8,64627	9,457216	5,173929	3,10263	3,145778	2,813047	2,51002	2,818005	1,962472
140	17,380644	9,022817	11,569078	5,528554	3,456281	3,184396	3,16848	2,609022	3,177576	2,021097

Γράφημα 5-6: Κλιμάκωση του k με κανονική κατανομή

Στο Γράφημα 5-6 φαίνονται τα πειραματικά αποτελέσματα. Παρατηρούμε ότι για μικρά k, μέχρι και 40, ο πιο αποδοτικός αλγόριθμος είναι ο DSPP+P με KNP-DLB, ενώ για k, από 60 μέχρι και 140 ο πιο αποδοτικός είναι ο DSPP+PEA με max-Heap buffer. Επίσης παρατηρούμε ότι η μέθοδος χωρικής υποδιαίρεσης που χρησιμοποιήσαμε ήταν απόλυτα

επιτυχημένη, καθώς μείωσε σημαντικά τον χρόνο εκτέλεσης. Ακόμα παρατηρούμε ότι η χρήση του pinned memory και του concurrent kernel execution βελτίωσε την επίδοση.

Και σ' αυτή την περίπτωση παρατηρούμε ότι η αλλαγή της κατανομής επηρεάζει ελαφρώς τα αποτελέσματα σε σχέση με τα αντίστοιχα πειράματα της bit κατανομής, αλλά για μεγάλα k αυτό δεν είναι ικανό να επηρεάσει το ποια υλοποίηση είναι πιο αποδοτική.

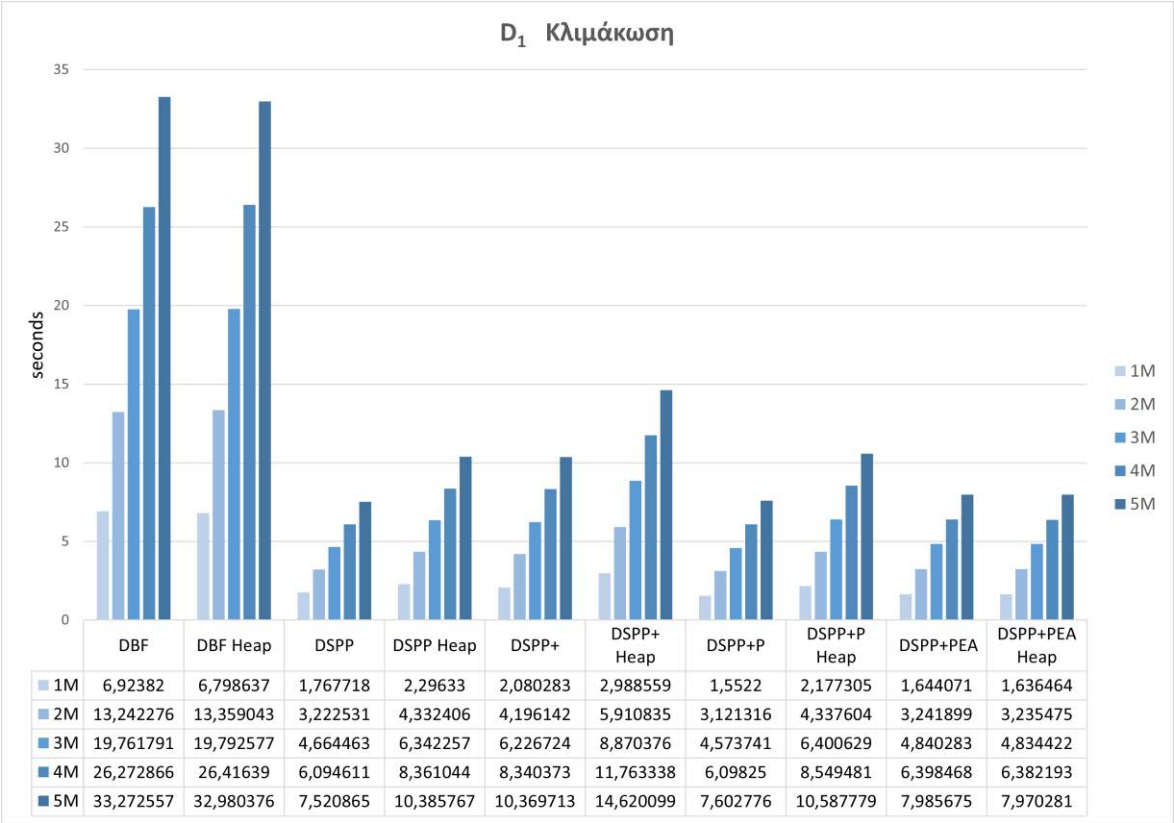
5.1.7 Κλιμάκωση του D_1 με ομοιόμορφη κατανομή

Σ' αυτό το πείραμα κλιμακώνεται το μέγεθος του D_1, που ο αριθμός των σημείων που περιέχει κυμαίνεται από 1M έως 5M με το D_2 να έχει σταθερό αριθμό σημείων ίσο με 100K και την τιμή k να είναι ίση με 20.

Κρατήθηκε σταθερό το μέγεθος D_2 και η τιμή του k για να εστιάσουμε στην κλιμάκωση του D_1.

Στο Γράφημα 5-7 φαίνονται τα πειραματικά αποτελέσματα. Παρατηρούμε ότι όταν χρησιμοποιείται max-Heap buffer είναι ο DSPP+PEA είναι πιο αποδοτικός από τον αντίστοιχο DSPP+P ενώ όταν χρησιμοποιείται KNP-DLB ο DSPP+P είναι πιο αποδοτικός από τον αντίστοιχο DSPP+PEA και είναι και ο πιο γρήγορος από όλους τους άλλους. Επίσης παρατηρούμε ότι η μέθοδος χωρικής υποδιαίρεσης που χρησιμοποιήθηκε ήταν απόλυτα επιτυχημένη, καθώς μείωσε σημαντικά τον χρόνο εκτέλεσης. Ακόμα παρατηρούμε ότι η χρήση του pinned memory και του concurrent kernel execution βελτίωσε την επίδοση.

Παρατηρείται και σ' αυτήν την περίπτωση ότι η DSPP+PEA και η DSPP+ εμφανίζουν ελαφρώς διαφορετικά αποτελέσματα σε σχέση με τα αντίστοιχα πειράματα της bit και της κανονικής (στους τέσσερις αυτούς αλγόριθμους κάποιες τιμές αυξάνονται και κάποιες μειώνονται). Αυτό συμβαίνει, καθώς όπως αναφέρθηκε στην ενότητα 4.3, η μέθοδος χωρικής υποδιαίρεσης, που χρησιμοποιείται, μπορεί να επηρεαστεί από την κατανομή των σημείων και δεδομένου ότι οι παράμετροι ανάμεσα στην DSPP+ και στην DSPP+PEA είναι ελαφρώς διαφορετικές, μπορούν να επηρεάσουν μια σειρά πραγμάτων, όπως τα μοτίβα προσπέλασης μνήμης, τη χρήση των caches και των warp divergence. Έτσι μπορούν και προκαλούν αυτές τις αποκλίσεις του χρόνου.



	DBF	DBF Heap	DSPP	DSPP Heap	DSPP+	DSPP+ Heap	DSPP+P	DSPP+P Heap	DSPP+PEA	DSPP+PEA Heap
1M	6,92382	6,798637	1,767718	2,29633	2,080283	2,988559	1,5522	2,177305	1,644071	1,636464
2M	13,242276	13,359043	3,222531	4,332406	4,196142	5,910835	3,121316	4,337604	3,241899	3,235475
3M	19,761791	19,792577	4,664463	6,342257	6,226724	8,870376	4,573741	6,400629	4,840283	4,834422
4M	26,272866	26,41639	6,094611	8,361044	8,340373	11,763338	6,09825	8,549481	6,398468	6,382193
5M	33,272557	32,980376	7,520865	10,385767	10,369713	14,620099	7,602776	10,587779	7,985675	7,970281

Γράφημα 5-7: Κλιμάκωση του D_1 με ομοιόμορφη κατανομή

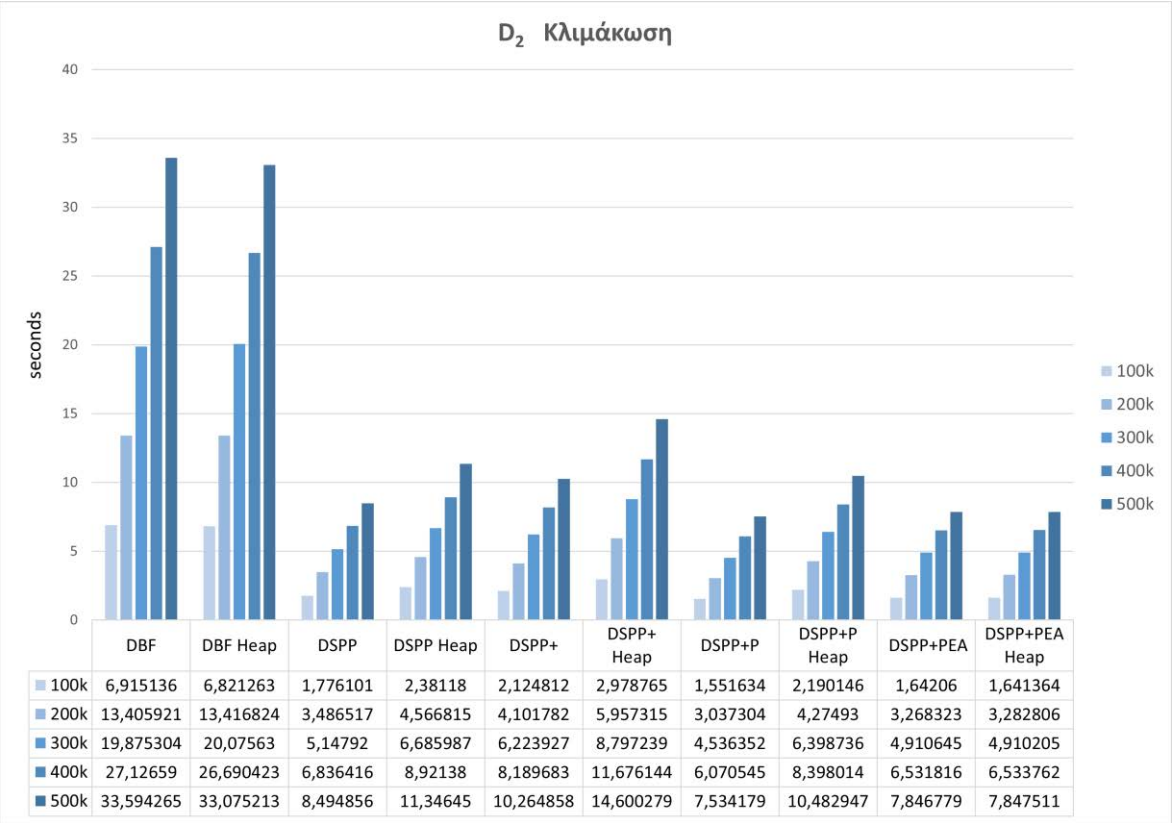
5.1.8 Κλιμάκωση του D_2 με ομοιόμορφη κατανομή

Σ' αυτό το πείραμα κλιμακώνεται το μέγεθος του D_2, που ο αριθμός των σημείων που περιέχει κυμαίνεται από 100K έως 500K με το D_1 να έχει σταθερό αριθμό σημείων ίσο με 1M και την τιμή k να είναι ίση με 20.

Κρατήθηκε σταθερό το μέγεθος D_1 και η τιμή του k για να εστιάσουμε στην κλιμάκωση του D_2.

Στο Γράφημα 5-8 φαίνονται τα πειραματικά αποτελέσματα, τα οποία είναι παρόμοια με του 5.1.7. Παρατηρούμε ότι όταν χρησιμοποιείται max-Heap buffer είναι ο DSPP+PEA είναι πιο αποδοτικός από τον αντίστοιχο DSPP+P ενώ όταν χρησιμοποιείται KNP-DLB ο DSPP+P είναι πιο αποδοτικός από τον αντίστοιχο DSPP+PEA και είναι και ο πιο γρήγορος από όλους τους άλλους. Επίσης παρατηρούμε ότι η μέθοδος χωρικής υποδιαίρεσης που χρησιμοποιήθηκε ήταν απόλυτα επιτυχημένη, καθώς μείωσε σημαντικά τον χρόνο εκτέλεσης. Ακόμα παρατηρούμε ότι η χρήση του pinned memory και του concurrent kernel execution βελτίωσε την επίδοση.

Παρατηρείται, όπως και στην ενότητα 5.1.7, ότι η DSPP+PEA και η DSPP+ εμφανίζουν ελαφρώς διαφορετικά αποτελέσματα σε σχέση με τα αντίστοιχα πειράματα της bit και της κανονικής κατανομής (στους τέσσερις αυτούς αλγόριθμους κάποιες τιμές αυξάνονται και κάποιες μειώνονται). Αυτό συμβαίνει, καθώς όπως αναφέρθηκε στην ενότητα 4.3, η μέθοδος χωρικής υποδιαίρεσης, που χρησιμοποιείται, μπορεί να επηρεαστεί από την κατανομή των σημείων και δεδομένου ότι οι παράμετροι ανάμεσα στην DSPP+ και στην DSPP+PEA είναι ελαφρώς διαφορετικές, μπορούν να επηρεάσουν μια σειρά πραγμάτων, όπως τα μοτίβα προσπέλασης μνήμης, τη χρήση των caches και των warp divergence. Έτσι μπορούν και προκαλούν αυτές τις αποκλίσεις του χρόνου.



	DBF	DBF Heap	DSPP	DSPP Heap	DSPP+	DSPP+ Heap	DSPP+P	DSPP+P Heap	DSPP+PEA	DSPP+PEA Heap
100k	6,915136	6,821263	1,776101	2,38118	2,124812	2,978765	1,551634	2,190146	1,64206	1,641364
200k	13,405921	13,416824	3,486517	4,566815	4,101782	5,957315	3,037304	4,27493	3,268323	3,282806
300k	19,875304	20,07563	5,14792	6,685987	6,223927	8,797239	4,536352	6,398736	4,910645	4,910205
400k	27,12659	26,690423	6,836416	8,92138	8,189683	11,676144	6,070545	8,398014	6,531816	6,533762
500k	33,594265	33,075213	8,494856	11,34645	10,264858	14,600279	7,534179	10,482947	7,846779	7,847511

Γράφημα 5-8: Κλιμάκωση του D_2 με ομοιόμορφη κατανομή

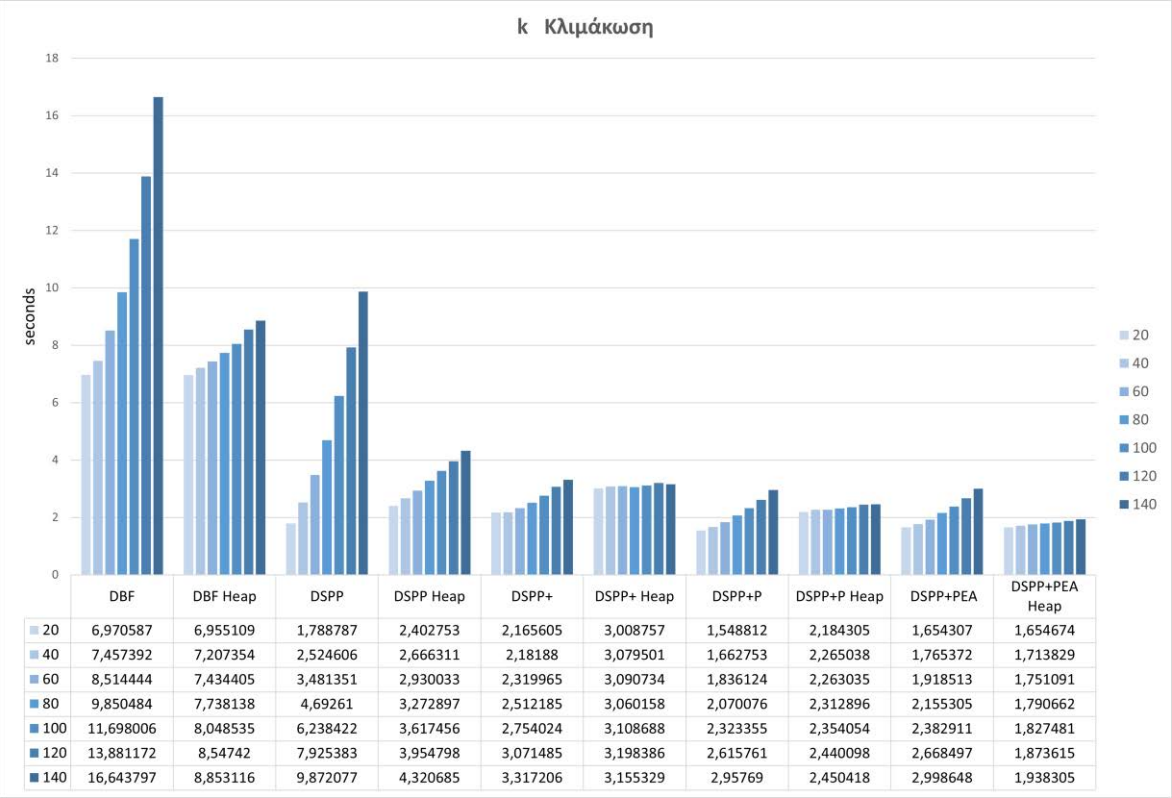
5.1.9 Κλιμάκωση του k με ομοιόμορφη κατανομή

Σ' αυτό το πείραμα κλιμακώνεται το k, που παίρνει τιμές που κυμαίνονται από 20 έως 140 με το D_1 και το D_2 να έχουν σταθερό αριθμό σημείων ίσο με 1M και 100K αντίστοιχα.

Κρατήθηκαν σταθερά τα μεγέθη του D_1 και του D_2 για να εστιάσουμε στην κλιμάκωση του k.

Στο Γράφημα 5-9 φαίνονται τα πειραματικά αποτελέσματα. Παρατηρούμε ότι για μικρά k, μέχρι και 40, ο πιο αποδοτικός αλγόριθμος είναι ο DSPP+P με KNP-DLB, ενώ για k, από 60 μέχρι και 140 ο πιο αποδοτικός είναι ο DSPP+PEA με max-Heap buffer. Επίσης παρατηρούμε ότι η μέθοδος χωρικής υποδιαίρεσης που χρησιμοποιήθηκε ήταν απόλυτα επιτυχημένη, καθώς μείωσε σημαντικά τον χρόνο εκτέλεσης. Ακόμα παρατηρούμε ότι η χρήση του pinned memory και του concurrent kernel execution βελτίωσε την επίδοση.

Παρατηρείται και σ' αυτήν την περίπτωση ότι η DSPP+PEA και η DSPP+ διαφέρουν ελαφρώς από τα αντίστοιχα πειράματα με την bit και την κανονική κατανομή. Αυτό συμβαίνει, καθώς όπως αναφέρθηκε στην ενότητα 4.3, η μέθοδος χωρικής υποδιαίρεσης, που χρησιμοποιείται, μπορεί να επηρεαστεί από την κατανομή των σημείων και δεδομένου ότι οι παράμετροι ανάμεσα στην DSPP+ και στην DSPP+PEA είναι ελαφρώς διαφορετικές, μπορούν να επηρεάσουν μια σειρά πραγμάτων, όπως τα μοτίβα προσπέλασης μνήμης, τη χρήση των caches και των warp divergence. Έτσι μπορούν και προκαλούν αυτές τις αποκλίσεις του χρόνου. Ωστόσο σ' αυτήν την περίπτωση αυτό δεν είναι ικανό να επηρεάσει το ποια μέθοδος είναι αποδοτικότερη.



	DBF	DBF Heap	DSPP	DSPP Heap	DSPP+	DSPP+ Heap	DSPP+P	DSPP+P Heap	DSPP+PEA	DSPP+PEA Heap
20	6,970587	6,955109	1,788787	2,402753	2,165605	3,008757	1,548812	2,184305	1,654307	1,654674
40	7,457392	7,207354	2,524606	2,666311	2,18188	3,079501	1,662753	2,265038	1,765372	1,713829
60	8,514444	7,434405	3,481351	2,930033	2,319965	3,090734	1,836124	2,263035	1,918513	1,751091
80	9,850484	7,738138	4,69261	3,272897	2,512185	3,060158	2,070076	2,312896	2,155305	1,790662
100	11,698006	8,048535	6,238422	3,617456	2,754024	3,108688	2,323355	2,354054	2,382911	1,827481
120	13,881172	8,54742	7,925383	3,954798	3,071485	3,198386	2,615761	2,440098	2,668497	1,873615
140	16,643797	8,853116	9,872077	4,320685	3,317206	3,155329	2,95769	2,450418	2,998648	1,938305

Γράφημα 5-9: Κλιμάκωση του k με ομοιόμορφη κατανομή

5.2 Πειράματα με πραγματικά δεδομένα

Σε αυτή την ενότητα παρουσιάζονται τα αποτελέσματα με πραγματικά δεδομένα. Χρησιμοποιήθηκαν τέσσερα διαφορετικά αρχεία, δύο με συντεταγμένες και δύο με πολύγωνα, από τα οποία χρησιμοποιήθηκαν τα κεντροειδή σημεία. Το πρώτο αρχείο συντεταγμένων περιέχει 2091602 σημεία, το δεύτερο αρχείο περιέχει 11438677 σημεία, το πρώτο αρχείο κεντροειδών περιέχει 562302 σημεία και το δεύτερο αρχείο κεντροειδών περιέχει 244760 σημεία. Οι συντεταγμένες x και y έχουν κανονικοποιηθεί, ώστε να παίρνουν τιμές από 0 μέχρι 1 και οι συντεταγμένες z έχουν την τιμή 0. Ο Πίνακας 5-4 περιέχει πληροφορίες για τα μεγέθη και την προέλευση των αρχείων. Σε όλα τα πειράματα κλιμακώθηκε η τιμή k από 20 ως 140 και εξετάστηκαν οι παρακάτω παραλλαγές:

1. DSPP+, Improved Disk Symmetric Progression Partitioning χρησιμοποιώντας KNP-DLB
2. DSPP+ Heap, Improved Disk Symmetric Progression Partitioning χρησιμοποιώντας max- Heap buffer
3. DSPP+P, Improved Disk Symmetric Progression Partitioning with pinned memory χρησιμοποιώντας KNP-DLB
4. DSPP+P Heap, Improved Disk Symmetric Progression Partitioning with pinned memory χρησιμοποιώντας max-Heap buffer
5. DSPP+PEA, Improved Disk Symmetric Progression Partitioning with pinned memory with Enhanced Assignment χρησιμοποιώντας KNP-DLB
6. DSPP+PEA Heap, Improved Disk Symmetric Progression Partitioning with pinned memory Enhanced Assignment χρησιμοποιώντας max-Heap buffer

Πίνακας 5-4: Πραγματικά δεδομένα

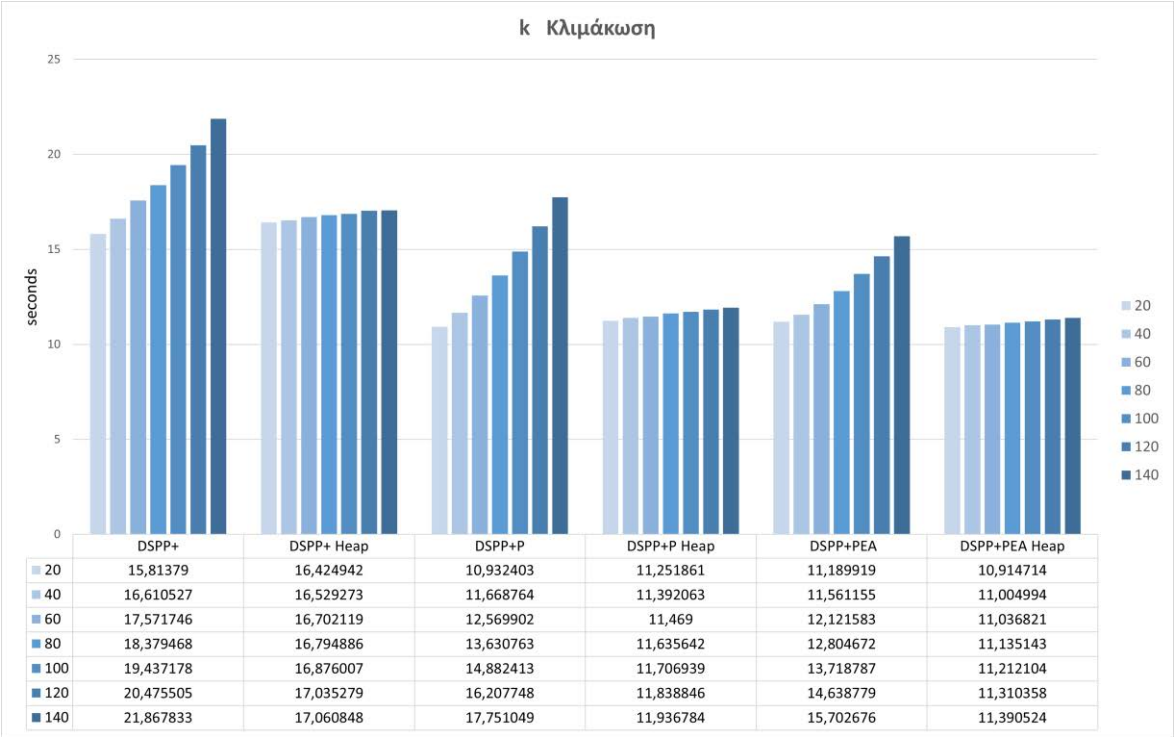
Όνομα	Αριθμός σημείων	Μέγεθος	Διευθύνσεις
Συντεταγμένες1	2091602	65363KB	https://planet.openstreetmap.org/redaction-period/day-replicate/000/000/153.osc.gz
Συντεταγμένες2	1438677	44959KB	https://planet.openstreetmap.org/redaction-period/day-replicate/000/000/158.osc.gz

Κεντροειδή1	562302	17572KB	https://planet.openstreetmap.org/historical-shapefiles/processed_p.tar.bz2
Κεντροειδή2	244760	7649KB	https://planet.openstreetmap.org/historical-shapefiles/shoreline_300.tar.bz2

5.2.1 Κλιμάκωση του k με αρχείο συντεταγμένων1 και αρχείο κεντροειδών1

Στο Γράφημα 5-10 φαίνονται τα πειραματικά αποτελέσματα. Παρατηρούμε ότι ο πιο αποδοτικός αλγόριθμος είναι ο DSPP+PEA Heap, ειδικά όσο μεγαλώνει το k. Ακόμα παρατηρούμε ότι η χρήση του pinned memory και του concurrent kernel execution βελτίωσε την επίδοση σε όλες τις περιπτώσεις.

Παρατηρείται ότι σ' αυτήν την περίπτωση η DSPP+PEA είναι πιο γρήγορη από την DSPP+. Αυτό οφείλεται σε δύο παράγοντες, στην κατανομή που ακολουθούν τα αρχεία και στο γεγονός του ότι ο αριθμός των εισαγωγών στις λίστες μειώνεται με την πάροδο του χρόνου. Αυτό σημαίνει ότι το υπολογιστικό κόστος μικραίνει σε σχέση με τα υπόλοιπα, ειδικά όσο τα μεγέθη αρχείων εισόδου αυξάνονται. Στη συγκεκριμένη περίπτωση, όπου το γινόμενο του μεγέθους του D1 και του D2 είναι μεγάλο, τα πλεονεκτήματα της DSPP+PEA, όπως αναφέρονται στην ενότητα 4.6, φαίνονται πιο έντονα, δηλαδή ο μειωμένος αριθμός κλήσεων kernels, το μειωμένο κόστος συγχρονισμού και ο μικρότερος αριθμός κλήσης συναρτήσεων, που μεταφέρουν δεδομένα (μεταφέρουν ωστόσο τον ίδιο αριθμό δεδομένων συνολικά). Μεγαλώνουν έτσι αναλογικά το κέρδος σε σχέση με τα μεγέθη του D1 και του D2.



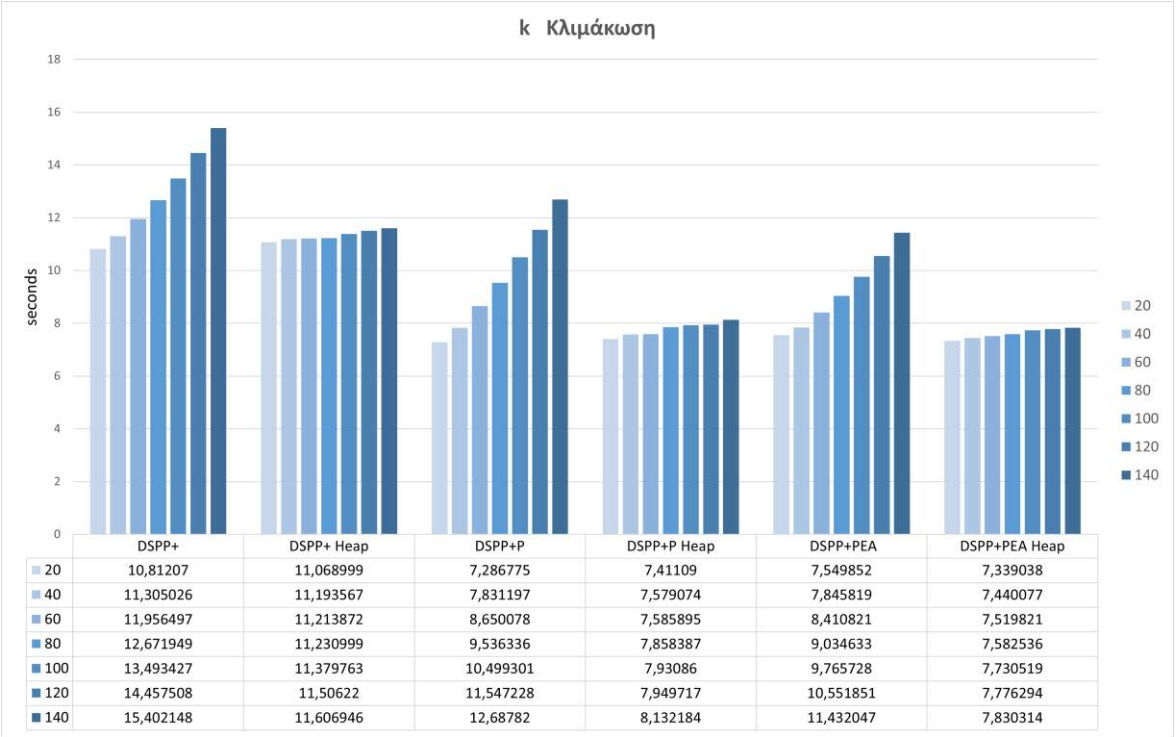
	DSPP+	DSPP+ Heap	DSPP+P	DSPP+P Heap	DSPP+PEA	DSPP+PEA Heap
20	15,81379	16,424942	10,932403	11,251861	11,189919	10,914714
40	16,610527	16,529273	11,668764	11,392063	11,561155	11,004994
60	17,571746	16,702119	12,569902	11,469	12,121583	11,036821
80	18,379468	16,794886	13,630763	11,635642	12,804672	11,135143
100	19,437178	16,876007	14,882413	11,706939	13,718787	11,212104
120	20,475505	17,035279	16,207748	11,838846	14,638779	11,310358
140	21,867833	17,060848	17,751049	11,936784	15,702676	11,390524

Γράφημα 5-10: Κλιμάκωση του k με αρχείο συντεταγμένων1 και αρχείο κεντροειδών1

5.2.2 Κλιμάκωση του k με αρχείο συντεταγμένων2 και αρχείο κεντροειδών1

Στο Γράφημα 5-11 φαίνονται τα πειραματικά αποτελέσματα. Παρατηρούμε ότι για μικρά k ίσο με 20 ο πιο αποδοτικός αλγόριθμος είναι ο DSPP+P με KNP-DLB, ενώ για k, από 40 μέχρι και 140 ο πιο αποδοτικός είναι ο DSPP+PEA με max-Heap buffer. Ακόμα παρατηρούμε ότι η χρήση του pinned memory και του concurrent kernel execution βελτίωσε την επίδοση σε όλες τις περιπτώσεις.

Και σε αυτήν την περίπτωση παρατηρείται ότι οι μέθοδοι που χρησιμοποιούν max-Heap buffer λειτουργούν καλύτερα για μεγάλα k (πράγμα αναμενόμενο) και αποτελούν καθοριστικό παράγοντα στο ποια μέθοδος είναι πιο αποδοτική.



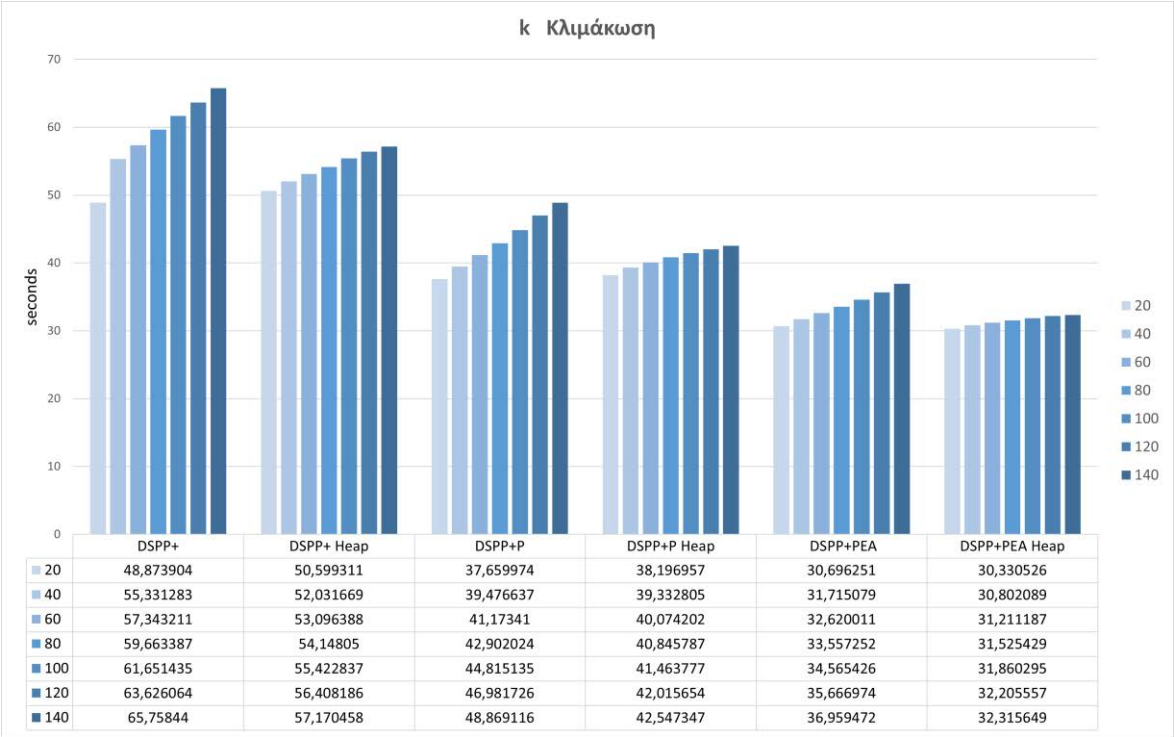
	DSPP+	DSPP+ Heap	DSPP+P	DSPP+P Heap	DSPP+PEA	DSPP+PEA Heap
20	10,81207	11,068999	7,286775	7,41109	7,549852	7,339038
40	11,305026	11,193567	7,831197	7,579074	7,845819	7,440077
60	11,956497	11,213872	8,650078	7,585895	8,410821	7,519821
80	12,671949	11,230999	9,536336	7,858387	9,034633	7,582536
100	13,493427	11,379763	10,499301	7,93086	9,765728	7,730519
120	14,457508	11,50622	11,547228	7,949717	10,551851	7,776294
140	15,402148	11,606946	12,68782	8,132184	11,432047	7,830314

Γράφημα 5-11: Κλιμάκωση του k με αρχείο συντεταγμένων2 και αρχείο κεντροειδών1

5.2.3 Κλιμάκωση του k με αρχείο συντεταγμένων1 και αρχείο συντεταγμένων2

Στο Γράφημα 5-12 φαίνονται τα πειραματικά αποτελέσματα. Παρατηρούμε ότι ο πιο αποδοτικός αλγόριθμος είναι ο DSPP+PEA Heap, ειδικά όσο μεγαλώνει το k. Ακόμα παρατηρούμε ότι η χρήση του pinned memory και του concurrent kernel execution βελτίωσε την επίδοση σε όλες τις περιπτώσεις.

Σε αυτήν την περίπτωση οι αλγόριθμοι DSPP+PEA είναι ξεκάθαρα οι καλύτεροι, διότι όπως αναφέρθηκε και στην ενότητα 5.2.1, όσο μεγαλώνουν τα μεγέθη αρχείων εισόδου, τόσο πιο έντονα εμφανίζονται τα αποτελέσματα. Αυτό οφείλεται στο ότι το κόστος των υπολογισμών μειώνεται σε σχέση με τα άλλα κόστη με την πάροδο του χρόνου και καθώς το κέρδος από τον μειωμένο αριθμό κλήσεων kernels, το μειωμένο κόστος συγχρονισμού και τον μειωμένο αριθμό κλήσης συναρτήσεων, που μεταφέρουν δεδομένα. Μεγαλώνουν έτσι αναλογικά το κέρδος σε σχέση με τα μεγέθη του D1 και του D2.



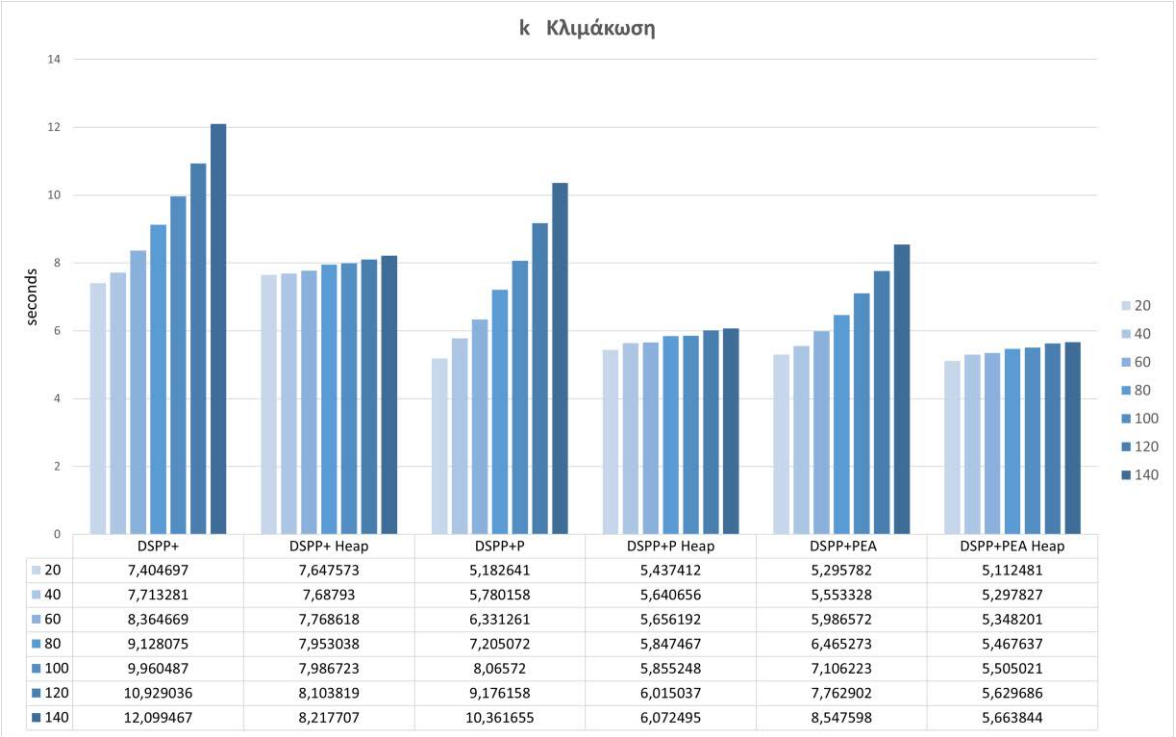
	DSPP+	DSPP+ Heap	DSPP+P	DSPP+P Heap	DSPP+PEA	DSPP+PEA Heap
20	48,873904	50,599311	37,659974	38,196957	30,696251	30,330526
40	55,331283	52,031669	39,476637	39,332805	31,715079	30,802089
60	57,343211	53,096388	41,17341	40,074202	32,620011	31,211187
80	59,663387	54,14805	42,902024	40,845787	33,557252	31,525429
100	61,651435	55,422837	44,815135	41,463777	34,565426	31,860295
120	63,626064	56,408186	46,981726	42,015654	35,666974	32,205557
140	65,75844	57,170458	48,869116	42,547347	36,959472	32,315649

Γράφημα 5-12: Κλιμάκωση του k με αρχείο συντεταγμένων1 και αρχείο συντεταγμένων2

5.2.4 Κλιμάκωση του k με αρχείο συντεταγμένων1 και αρχείο κεντροειδών2

Στο Γράφημα 5-13 φαίνονται τα πειραματικά αποτελέσματα. Παρατηρούμε ότι ο πιο αποδοτικός αλγόριθμος είναι ο DSPP+PEA Heap, ειδικά όσο μεγαλώνει το k. Ακόμα παρατηρούμε ότι η χρήση του pinned memory και του concurrent kernel execution βελτίωσε την επίδοση σε όλες τις περιπτώσεις.

Παρατηρείται και σ' αυτήν την περίπτωση ότι η DSPP+PEA είναι πιο γρήγορη από την DSPP+. Αυτό οφείλεται σε δύο παράγοντες, στην κατανομή που ακολουθούν τα αρχεία και στο γεγονός του ότι ο αριθμός των εισαγωγών στις λίστες μειώνεται με την πάροδο του χρόνου. Αυτό σημαίνει ότι το υπολογιστικό κόστος μικραίνει σε σχέση με τα υπόλοιπα, ειδικά όσο τα μεγέθη αρχείων εισόδου αυξάνονται. Στη συγκεκριμένη περίπτωση, όπου το γινόμενο του μεγέθους του D1 και του D2 είναι μεγάλο, τα πλεονεκτήματα της DSPP+PEA, όπως αναφέρονται στην ενότητα 4.6, φαίνονται πιο έντονα, δηλαδή ο μειωμένος αριθμός κλήσεων kernels, το μειωμένο κόστος συγχρονισμού και ο μικρότερος αριθμός κλήσης συναρτήσεων, που μεταφέρουν δεδομένα (μεταφέρουν ωστόσο τον ίδιο αριθμό δεδομένων συνολικά). Μεγαλώνουν έτσι αναλογικά το κέρδος σε σχέση με τα μεγέθη του D1 και του D2.



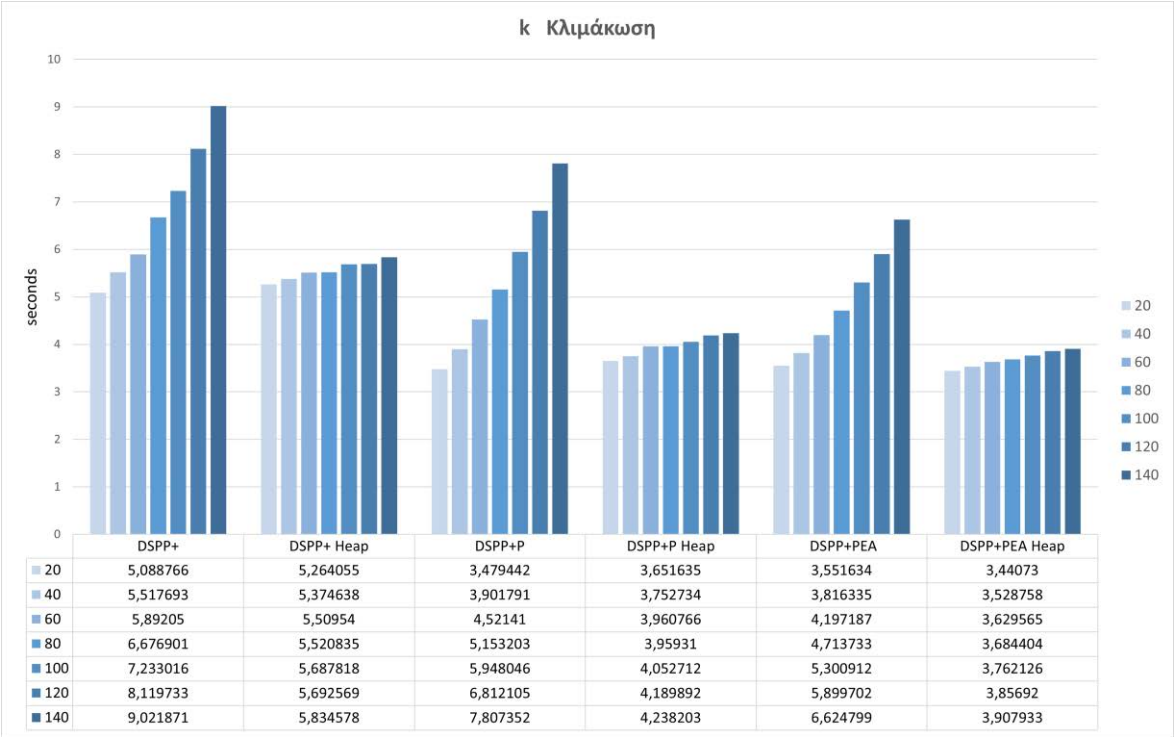
	DSPP+	DSPP+ Heap	DSPP+P	DSPP+P Heap	DSPP+PEA	DSPP+PEA Heap
20	7,404697	7,647573	5,182641	5,437412	5,295782	5,112481
40	7,713281	7,68793	5,780158	5,640656	5,553328	5,297827
60	8,364669	7,768618	6,331261	5,656192	5,986572	5,348201
80	9,128075	7,953038	7,205072	5,847467	6,465273	5,467637
100	9,960487	7,986723	8,06572	5,855248	7,106223	5,505021
120	10,929036	8,103819	9,176158	6,015037	7,762902	5,629686
140	12,099467	8,217707	10,361655	6,072495	8,547598	5,663844

Γράφημα 5-13: Κλιμάκωση του k με αρχείο συντεταγμένων1 και αρχείο κεντροειδών2

5.2.5 Κλιμάκωση του k με αρχείο συντεταγμένων2 και αρχείο κεντροειδών2

Στο Γράφημα 5-14 φαίνονται τα πειραματικά αποτελέσματα. Παρατηρούμε ότι ο πιο αποδοτικός αλγόριθμος είναι ο DSPP+PEA Heap, ειδικά όσο μεγαλώνει το k. Ακόμα παρατηρούμε ότι η χρήση του pinned memory και του concurrent kernel execution βελτίωσε την επίδοση σε όλες τις περιπτώσεις.

Παρατηρείται και σ' αυτήν την περίπτωση ότι η DSPP+PEA είναι πιο γρήγορη από την DSPP+. Αυτό οφείλεται σε δύο παράγοντες, στην κατανομή που ακολουθούν τα αρχεία και στο γεγονός του ότι ο αριθμός των εισαγωγών στις λίστες μειώνεται με την πάροδο του χρόνου. Αυτό σημαίνει ότι το υπολογιστικό κόστος μικραίνει σε σχέση με τα υπόλοιπα, ειδικά όσο τα μεγέθη αρχείων εισόδου αυξάνονται. Στη συγκεκριμένη περίπτωση, όπου το γινόμενο του μεγέθους του D1 και του D2 είναι μεγάλο, τα πλεονεκτήματα της DSPP+PEA, όπως αναφέρονται στην ενότητα 4.6, φαίνονται πιο έντονα, δηλαδή ο μειωμένος αριθμός κλήσεων kernels, το μειωμένο κόστος συγχρονισμού και ο μικρότερος αριθμός κλήσης συναρτήσεων, που μεταφέρουν δεδομένα (μεταφέρουν ωστόσο τον ίδιο αριθμό δεδομένων συνολικά). Μεγαλώνουν έτσι αναλογικά το κέρδος σε σχέση με τα μεγέθη του D1 και του D2.



	DSPP+	DSPP+ Heap	DSPP+P	DSPP+P Heap	DSPP+PEA	DSPP+PEA Heap
20	5,088766	5,264055	3,479442	3,651635	3,551634	3,44073
40	5,517693	5,374638	3,901791	3,752734	3,816335	3,528758
60	5,89205	5,50954	4,52141	3,960766	4,197187	3,629565
80	6,676901	5,520835	5,153203	3,95931	4,713733	3,684404
100	7,233016	5,687818	5,948046	4,052712	5,300912	3,762126
120	8,119733	5,692569	6,812105	4,189892	5,899702	3,85692
140	9,021871	5,834578	7,807352	4,238203	6,624799	3,907933

Γράφημα 5-14: Κλιμάκωση του k με αρχείο συντεταγμένων2 και αρχείο κεντροειδών2

Κεφάλαιο 6 Συμπεράσματα

Στο κεφάλαιο αυτό συνοψίζονται τα συμπεράσματα που προκύπτουν από την πειραματική αξιολόγηση και παρουσιάζονται μελλοντικές επεκτάσεις για την επιπλέον βελτίωση των αλγορίθμων.

6.1 Σύνοψη και συμπεράσματα

Διαπιστώθηκε ότι η πιο αποδοτική μέθοδος είναι η DSPP+PEA Heap ειδικά για μεγάλα k. Σε κάποιες περιπτώσεις για μικρά k η DSPP+P με KNP-DLB μπορεί να είναι πιο αποδοτική. Επίσης παρατηρούμε ότι η μέθοδος χωρικής υποδιαίρεσης που χρησιμοποιήθηκε ήταν απόλυτα επιτυχημένη, καθώς μείωσε σημαντικά τον χρόνο εκτέλεσης. Ακόμα παρατηρούμε ότι η χρήση του pinned memory και του concurrent kernel execution βελτίωσε την επίδοση σημαντικά, σχεδόν σε όλες τις περιπτώσεις.

6.2 Μελλοντικές επεκτάσεις

Σαν μελλοντικές επεκτάσεις για τη βελτίωση των αλγορίθμων προτείνονται:

- Μια υλοποίηση του αλγορίθμου με shared memory
- Συλλογή στατιστικών δεδομένων στοιχείων, που θα βοηθήσουν στην επιλογή του βέλτιστου αριθμού sub – partitions
- Επέκταση του αλγορίθμου, ώστε να λειτουργεί σε πολλαπλές κάρτες γραφικών

Βιβλιογραφία

[1] [Χ. Δ. Αντωνόπουλος. Introduction to CUDA. Διάλεξη μαθήματος Συστημάτων υπολογισμού υψηλών επιδόσεων Πανεπιστήμιο Θεσσαλίας Τμήμα Ηλεκτρολόγων μηχανικών και μηχανικών Η/Υ. https://www.e-ce.uth.gr/studies/undergraduate/courses/ece415/

[2] Χ. Δ. Αντωνόπουλος. GPU / CUDA Memory Hierarchy. Διάλεξη μαθήματος Συστημάτων υπολογισμού υψηλών επιδόσεων Πανεπιστήμιο Θεσσαλίας Τμήμα Ηλεκτρολόγων μηχανικών και μηχανικών Η/Υ. https://www.e-ce.uth.gr/studies/undergraduate/courses/ece415/

[3] Χ. Δ. Αντωνόπουλος. GPU Code Optimization. Διάλεξη μαθήματος Συστημάτων υπολογισμού υψηλών επιδόσεων Πανεπιστήμιο Θεσσαλίας Τμήμα Ηλεκτρολόγων μηχανικών και μηχανικών Η/Υ. https://www.e-ce.uth.gr/studies/undergraduate/courses/ece415/

[4] Χ. Δ. Αντωνόπουλος. CUDA Unified Memory – CUDA Streams. Διάλεξη μαθήματος Συστημάτων υπολογισμού υψηλών επιδόσεων Πανεπιστήμιο Θεσσαλίας Τμήμα Ηλεκτρολόγων μηχανικών και μηχανικών Η/Υ. https://www.e-ce.uth.gr/studies/undergraduate/courses/ece415/

[5] Memory Statistics – Global. https://docs.nvidia.com/gameworks/content/developertools/desktop/analysis/report/cudaexperiments/kernellevel/memorystatisticsglobal.htm

[6] CUDA C++ Best Practices Guide. https://docs.nvidia.com/cuda/cuda-c-best-practices-guide/#control-flow

[7] CUDA C++ Programming Guide. https://docs.nvidia.com/cuda/cuda-c-programming-guide/#cooperative-groups

[8] Using Shared Memory in CUDA C/C++. https://developer.nvidia.com/blog/using-shared-memory-cuda-cc/

[9] Steve Rennich. CUDA C/C++ Streams and Concurrency. https://developer.download.nvidia.com/CUDA/training/StreamsAndConcurrencyWebinar.pdf

[10] P. Velentzas, M. Vassilakopoulos, A. Corral, and C. Antonopoulos. GPU-Based Algorithms for Processing the k Nearest-Neighbor Query on Spatial Data Using Partitioning and Concurrent Kernel Execution. International Journal of Parallel Programming (2023)

[11] https://developer.nvidia.com/cuda-toolkit

[12] https://en.wikipedia.org/wiki/CUDA

[13] https://en.wikipedia.org/wiki/C%2B%2B

[14] https://www.astro.auth.gr/~niksterg/courses/progtools/1-OpenMP-tutorial.pdf

[15] https://el.wikipedia.org/wiki/%CE%A0%CE%B1%CF%81%CE%AC%CE%BB%CE%BB%CE%B7%CE%BB%CE%BF%CF%82_%CF%80%CF%81%CE%BF%CE%B3%CF%81%CE%B1%CE%BC%CE%BC%CE%B1%CF%84%CE%B9%CF%83%CE%BC%CF%8C%CF%82

[16] https://developer.nvidia.com/thrust

[17] https://en.wikipedia.org/wiki/Amdahl%27s_law

[18] Χ. Δ. Αντωνόπουλος. Η/Υ Why Program for Parallelism?. Διάλεξη μαθήματος Συστημάτων υπολογισμού υψηλών επιδόσεων. Πανεπιστήμιο Θεσσαλίας Τμήμα Ηλεκτρολόγων μηχανικών και μηχανικών. https://www.e-ce.uth.gr/studies/undergraduate/courses/ece415/

[19] Χ. Δ. Αντωνόπουλος. Writing parallel programs – Why is it hard? Introduction to parallel architectures. Διάλεξη μαθήματος Συστημάτων υπολογισμού υψηλών επιδόσεων. Πανεπιστήμιο Θεσσαλίας Τμήμα Ηλεκτρολόγων μηχανικών και μηχανικών. https://www.e-ce.uth.gr/studies/undergraduate/courses/ece415/

[20] Quansheng Kuang, and Lei Zhao. A Practical GPU Based KNN Algorithm. Second Symposium International Computer Science and Computational Technology (ISCSCT 2009). https://readingxtra.github.io/docs/ml-gpu/practical-knn-gpu.pdf

[21] V.B.Nikam, B.B.Meshram. PARALLEL kNN ON GPU ARCHITECTURE USING Open CL. IJRET: International Journal of Research in Engineering and Technology eISSN: 2319-1163 | pISSN: 2321-7308 Volume: 03 Issue: 10 | Oct-2014, Available https://d1wqtxts1xzle7.cloudfront.net/41708999/PARALLEL_KNN_ON_GPU_ARCHITECTURE_USING_OPENCL-libre.pdf?1454045544=&response-content-disposition=inline%3B+filename%3DPARALLEL_kNN_ON_GPU_ARCHITECTURE_USING_O.pdf&Expires=1755869289&Signature=ZUNu-B-hQ0rX3~61Tfx0Jygm8oFXIg61n2aVwBKAPPp7CEVxDsean6uekBzNcQFsAIB7WA3~ipDsScjdcm6WeFazcoKp7a7jsJspNxlENiC8~ExGvIqdtao44PQAHRFIeAmHxOh~h6yf1yRYFUm1ICFMlkwLzRCGwkDw6dWP45pGwl0NMQSxSSyNYHpUX~R0wF5nodnmbiNe7YrBGclMWfFhontfLvrrR7NNtn0lralKQi5wN1NgMgG-BkxKJK2tvJeNXmC6-hHcxYIkzppFJLnPdtBrlRYaHFZj4-jTedR4aHIUAjDSkLnceWnPV80HegfXRM4xDxT2yAN81OLqSg__&Key-Pair-Id=APKAJLOHF5GGSLRBV4ZA

[22] Pablo David Gutierrez Perez. Identificacion y Clasificacion Biometrica a Gran Escala Basada en Huellas Dactilares y GPU. Programa de Doctorado en Tecnologıas de la Informacion y la Comunicacion. Granada Universidad de Granada Marzo de 2017 DIRECTORES Francisco Herrera Triguero Miguel Lastra Leidinger https://produccioncientifica.ugr.es/documentos/618f4faa9ff8c939aaca5509

[23] Guoyang Chen, Yufei Ding, and Xipeng Shen. KNN: An Efficient KNN on GPU through Reconciliation between Redundancy Removal and Regularity. Computer Science Department, North Carolina State University Raleigh, NC, USA 27519 2017, IEEE 33rd International Conference on Data Engineering https://sites.cs.ucsb.edu/~yufeiding/publication/icde17.pdf

[24] Ahmed Shamsul Arefin, Carlos Riveros, Regina Berretta, Pablo Moscato. GPU-FS-kNN: A Software Tool for Fast and Scalable kNN Computation Using GPUs. https://journals.plos.org/plosone/article/file?id=10.1371/journal.pone.0044000&type=printable

[25] Gang Mei, Nengxiong Xu, Liangliang Xu. Improving GPU-accelerated adaptive IDW interpolation algorithm using fast kNN search. Mei et al. SpringerPlus (2016) 5:1389 DOI 10.1186/s40064-016-3035-2. https://link.springer.com/content/pdf/10.1186/s40064-016-3035-2.pdf

[26] V. Garcia, E. Debreuve, M. Barlaud. Fast k Nearest Neighbor Search using GPU. Computer Vision on GPU (CVPR Workshop) (2008). https://arxiv.org/abs/0804.1448

[27] V. Garcia, E. Debreuve, M. Barlaud. k-nearest neighbor search: Fast GPU-based implementations and application to high-dimensional feature matching. IEEE International Conference on Image Processing (ICIP) — extended technical report (2010). https://vincentfpgarcia.github.io/data/Garcia_2010_ICIP.pdf

[28] S. Li, N. Amenta. Brute-Force k-Nearest Neighbors Search on the GPU. Similarity Search and Applications (Lecture Notes in Computer Science) (2015). https://web.cs.ucdavis.edu/~amenta/pubs/bfknn.pdf

[29] I. Komarov, A. Dashti, R. D'Souza. Fast k-NNG Construction with GPU-Based Quick Multi-Select. Journal: PLOS ONE (article) (2014) (arXiv preprint 2013). https://journals.plos.org/plosone/article?id=10.1371/journal.pone.0092409

[30] J. Johnson, M. Douze, H. Jégou. Billion-scale similarity search with GPUs. arXiv / associated IEEE Transactions on Big Data (FAISS GPU work) (2017) (paper) / final journal: 2019. https://arxiv.org/abs/1702.08734

[31] M. Douze, A. Guzhva, C. Deng, J. Johnson, G. Szilvasy, P.-E. Mazaré, M. Lomeli, L. Hosseini, H. Jégou. The Faiss library arXiv (software paper describing Faiss) (2024). https://arxiv.org/abs/2401.08281

[32] F. Groh, L. Ruppert, P. Wieschollek, H. P. A. Lensch. GGNN: Graph-based GPU Nearest Neighbor Search. arXiv (and conference presentation) (2019). https://arxiv.org/abs/1912.01059

[33] W. Zhao, S. Tan, P. Li. SONG: Approximate Nearest Neighbor Search on GPU. IEEE International Conference on Data Engineering (ICDE) (2020). https://www.cs.rit.edu/~wjz/papers/conference/2020-icde-song.pdf

[34] H. Ootomo, A. Naruse, C. Nolet, R. Wang, T. Fehér, Y. Wang. CAGRA: Highly Parallel Graph Construction and Approximate Nearest Neighbor Search for GPUs. arXiv / preprint (2023). https://arxiv.org/abs/2308.15136

[35] K. V., S. Khan, S. Singh, H. V. Simhadri, J. Vedurada. BANG: Billion-Scale Approximate Nearest Neighbor Search using a Single GPU. arXiv, 2024. https://arxiv.org/abs/2401.11324

[36] Y. Gui, P. Yin, X. Yan, C. Zhang, W. Zhang, J. Cheng. Pilot ANN: Memory-Bounded GPU Acceleration for Vector Search. arXiv, 2025. https://arxiv.org/abs/2503.21206

[37] J. Pan, D. Manocha. Fast GPU-based Locality Sensitive Hashing for K-Nearest Neighbor Computation. ACM / conference proceedings (paper page / GAMMA project) (2010). http://gamma.cs.unc.edu/KNN/

[38] X. Shi, S. Xu, K. Knight. Fast Locality Sensitive Hashing for Beam Search on GPU (WTA hashing). arXiv / conference technical report (2018). https://arxiv.org/abs/1806.00588

[39] D. Mandarapu, V. Nagarajan, A. Pelenitsyn, M. Kulkarni. Massively Parallel Nearest Neighbor Queries for Dynamic Point Clouds on the GPU. (dissertation / conference work) and Arkade: k-Nearest Neighbor Search With Non-Euclidean Distances using GPU Ray Tracing. thesis / conference (dynamic point clouds work, 2010) and ACM International Conference on Supercomputing (ICS), 2024 (Arkade). URLs: dynamic point clouds thesis / project page — (example). https://attena.ufpe.br/bitstream/123456789/2356/1/arquivo3157_1.pdf