Μια κατανεμημένη υλοποίηση k-NN σε Julia και CUDA για μεγάλα σετ δεδομένων

Αλέξανδρος Καλπακίδης

Τμήμα Ηλεκτρολογών Μηχανικών και Μηχανικών Υπολογιστών
Αριστοτέλειο Πανεπιστήμιο Θεσσαλονίκης, Ελλάδα

Ημερομηνία: _____
Εγκρίθηκε:

_____
Νίκος Πιτσιάνης, Επιβλέπων

_____
<Ονοματεπώνυμο>

_____
<Ονοματεπώνυμο>

Πτυχιακή που υποβλήθηκε για την μερική εκπλήρωση των προϋποθέσεων για το Δίπλωμα Ηλεκτρολόγου Μηχανικού και Μηχανικού Υπολογιστών,
Αριστοτέλειο Πανεπιστήμιο Θεσσαλονίκης, Ελλάδα 2023

Περίληψη

Καθώς μοντέλα μηχανικής μάθησης χρησιμοποιούνται ολοένα και περισσότερο για επίλυση προβλημάτων, η ανάγκη επεξεργασίας μεγάλου όγκου δεδομένων αυξάνεται, κάτι το οποίο απαιτεί διαφορετική προσέγγιση στην φάση της επεξεργασίας τους. Στο πλαίσιο αυτής της διπλωματικής θα εξετάσουμε μια μέθοδο υπολογισμού k-NN σε μεγάλα σετ δεδομένων, με χρήση κατανεμημένου προγραμματισμού, και μονάδων γραφικών, ώστε να μην υπερβούμε τους φυσικούς πόρους ενός μεμονωμένου συστήματος. Επίσης θα εξεταστεί μια προσέγγιση μείωσης της διαστατικότητας ενός σετ δεδομένων με χρήση τυχαίων προβολών σε υπερεπίπεδα.

Λέξεις κλειδιά: k-NN, κατανεμημένος προγραμματισμός, υπολογισμός σε GPU, CUDA, γλώσσα προγραμματισμού Julia, υπολογιστικές συστοιχίες, τυχαία προβολή σε υπερεπίπεδα

Αφιερώνω τη διπλωματική μου εργασία στην οικογένειά μου, που όλο αυτό το διάστημα μου συμπαραστάθηκε και έδειξε κατανόηση στον χρόνο που χρειάστηκε να αφιερώσω για την εκπόνησή της.

Περιεχόμενα

Περίληψη **iii**

Κατάλογος πινάκων **vi**

Κατάλογος σχημάτων **vii**

Ευχαριστίες **ix**

1 Εισαγωγή **1**

2 Θεωρία **9**

3 Μέθοδοι **15**

4 Πειράματα **24**

5 Συμπεράσματα **61**

A Πρώτο παράρτημα **64**

A.0.1 Ορίσματα 65

A.0.2 Μορφή δεδομένων εισόδου και εξόδου 68

A.0.3 Παράδειγμα χρήσης 69

B Δεύτερο παράρτημα **72**

References 75

Κατάλογος πινάκων

Κατάλογος σχημάτων

2.1 Η προβολή των σημείων a, b, c και d στις ευθείες R' και R" 13

3.1 Διαχωρισμός της εργασίας σε p διεργασίες μεγέθους $\frac{N_q}{nw}$, και επιπρόσθετα κατάτμηση του φόρτου εργασίας σε κομμάτια μεγέθους qSize x cSize . . 17

4.1 Η αρχιτεκτονική της υπολογιστικής συστοιχίας "Αριστοτέλης" 25

4.2 Σύγκριση των χρόνων εκτέλεσης με χρήση 2 GPUs και 4 διεργασίες των 16 CPUs 31

4.4 std σφάλματος δεδομένων HIGGS 35

4.5 std σφάλματος συνθετικών δεδομένων 35

4.6 Σύγκριση των χρόνων εκτέλεσης με 4 GPUs 36

4.7 Σύγκριση της ακρίβειας για σενάριο με 4 GPUs 38

4.8 Σύγκριση των χρόνων εκτέλεσης για δεδομένα 2800 διαστάσεων 39

4.9 Σύγκριση της ακρίβειας για δεδομένα 2800 διαστάσεων 40

4.10 Σύγκριση των χρόνων εκτέλεσης για 8 διεργασίες με 8 CPUs η κάθε μία για τα συνθετικά δεδομένα 2800 διαστάσεων 42

4.11 Σύγκριση της ακρίβειας για 8 διεργασίες με 8 CPUs η κάθε μία για τα συνθετικά δεδομένα 2800 διαστάσεων 43

4.12 Σύγκριση των χρόνων εκτέλεσης για k = 5 46

4.13 Σύγκριση της ακρίβειας για σενάριο εκτέλεσης με k = 5 47

4.14 Σύγκριση των χρόνων εκτέλεσης για k = 500 50
4.15 Σύγκριση της ακρίβειας για σενάριο εκτέλεσης με k = 500 51
4.16 Σύγκριση queries per second με recall για τα δεδομένα 2800 διαστάσεων 52
4.17 Σύγκριση της επιρροής των qSize και cSize για 4 διεργασίες των 16 CPUs 55
4.18 Σύγκριση ακρίβειας και QPSRecall για διαφορετικά qSize και cSize για 4 διεργασίες των 16 CPUs 56
4.19 Σύγκριση της επιρροής των qSize και cSize για 8 διεργασίες των 8 CPUs 57
4.20 Σύγκριση ακρίβειας και QPSRecall για διαφορετικά qSize και cSize για 8 διεργασίες των 8 CPUs 58

Ευχαριστίες

Θα ήθελα να ευχαριστήσω τον επιβλέποντά μου, κύριο Νικόλαο Πιτσιάνη, καθώς και τον Δημήτρη Φλώρο για την βοήθεια στα πρώτα βήματά μου στη χρήση της Julia. Επίσης θα ήθελα να ευχαριστήσω την ομάδα της υπολογιστικής συστοιχίας Αριστοτέλης για την υποστήριξη στην ενσωμάτωση της γλώσσας στη συστοιχία.

1

Εισαγωγή

Περιγραφή του προβλήματος

Ο αλγόριθμος k-nearest neighbors, γνωστός και ως KNN ή k-NN, είναι ένας μη παραμετρικός ταξινομητής μάθησης με επίβλεψη, ο οποίος χρησιμοποιεί την εγγύτητα σημείων σε πολυδιάστατο χώρο για να κάνει ταξινομήσεις ή προβλέψεις σχετικά με την ομαδοποίηση τους. Μπορεί να χρησιμοποιηθεί τόσο για προβλήματα παλινδρόμησης [18] όσο για προβλήματα ταξινόμησης [7], αν και συνήθως χρησιμοποιείται περισσότερο στην δεύτερη περίπτωση, βασιζόμενος στην υπόθεση ότι παρόμοια σημεία μπορούν να βρεθούν κοντά το ένα στο άλλο. Αξίζει επίσης να σημειωθεί ότι ο k-NN ανήκει σε μία οικογένεια "τεμπέλικης μάθησης" μιάς και επεξεργάζεται απευθείας το σετ δεδομένων εκμάθησης, χωρίς το τυπικό στάδιο της εκπαίδευσης που υιοθετούν άλλοι αλγόριθμοι.

Παρά το γεγονός αυτό, όσο τα σετ δεδομένων μεγαλώνουν σε μέγεθος, τόσο αυξάνεται και ο χώρος που καταναλώνουν οι ενδιάμεσοι πίνακες αποστάσεως, ιδίως στην περίπτωση που υιοθετείται ένας brute force αλγόριθμος. Αν υποθέσουμε ότι τα σετ δεδομένων μας

είναι τα C και Q, για το corpus και query σετ αντίστοιχα, d ο αριθμός των διαστάσεων των σημείων του, k ο ζητούμενος αριθμός των γειτόνων του κάθε σημείου του Q. Τότε για τον πλήρη υπολογισμό των αποστάσεων μεταξύ των σημείων του Q από το C θέλουμε έναν ενδιάμεσο πίνακα Dist μεγέθους Nq * Nc όπου Nq και Nc το πλήθος των σημείων του κάθε αντίστοιχου σετ. Κάτι τέτοιο μας δίνει υπολογισμούς πολυπλοκότητας $O(Nq * Nc)$ στην περίπτωση που τα σετ Q και C είναι διαφορετικά, ενώ $O(Nq^2)$ σε ένα ALLknn πρόβλημα [23]. Καθώς τα Nq και Nc αυξάνουν, γίνεται ιδιαίτερα δύσκολο να κρατήσουμε τον πίνακα Dist στην μνήμη, ιδίως στην περίπτωση που θέλουμε να εκτελέσουμε τους υπολογισμούς με τη βοήθεια μιας GPU. Αν και ο μαζικός παραλληλισμός που προσφέρουν οι GPU είναι ιδανικός ακόμα και για brute force αλγορίθμους, το γεγονός ότι οι μνήμες τους είναι ιδιαίτερα μικρότερες συγκριτικά σε μέγεθος από την μεγιστη ποσότητα της Ram ενός μοντέρνου υπολογιστή, καθιστά δύσκολη την μεταφορά μεγάλων πινάκων στην ιδιωτική μνήμη τους.

Βέβαια έχουν υλοποιηθεί τελευταία μέθοδοι για τμηματική μεταφορά πινάκων μεταξύ του host συστήματος και της GPU [16], αν και αυτό δεν είναι από μόνο του αρκετό για να επιλύσει το συνολικό πρόβλημα. Σε ιδιαίτερα μεγάλα σετ δεδομένων, οι υπολογιστικοί πόροι μιας μεμονωμένης GPU δεν κρίνονται αρκετοί για να εκτελεστεί ένας k-NN αλγόριθμος εντός ενός εύλογου χρονικού ορίου. Για το λόγο αυτό κρίνεται αναγκαία η χρήση παραπάνω μονάδων GPU ως ένας απλός τρόπος μείωσης του απαιτούμενου χρόνο εκτέλεσης με επιμερισμό της εργασίας μεταξύ τους. Με τον αριθμό των GPU ανά υπολογιστή να είναι περιορισμένος για πρακτικούς συνήθως λόγους, κρίνεται αναγκαία η χρήση μιας κατανεμημένης προσέγγισης, που επιτρέπει την χρήση υπολογιστικών πόρων πολλών συστημάτων, τόσο σε επίπεδο CPU όσο και σε GPU.

Φυσικά εδώ δεν θα πρέπει να υποτιμηθεί η "κατάρα της διαστατικότητας" [19] που δυ-

σχεραίνει τους υπολογισμούς καθώς ο αριθμός των διαστάσεων των σημείων των Q και C αυξάνεται. Σε ορισμένες περιπτώσεις είναι πλέον εντελώς ασύμφορο να εκτελέσουμε τους υπολογισμούς αυτούς, ακόμα και σε μια μεγάλη υπολογιστική συστοιχία. Ιδίως αν το k δεν είναι ιδιαίτερα μεγάλο, μπορούμε να εξετάσουμε και λιγότερο ακριβείς προσεγγιστικές μεθόδους. Μια τέτοια μέθοδος θα παρουσιαστεί και στο πλαίσιο της διπλωματικής αυτής που βασίζεται σε χρήση τυχαίων προβολών των σημείων των Q και C σε υπερεπίπεδα [8], με σκοπό την μείωση των διαστάσεών τους. Μάλιστα στην περίπτωση δεδομένων ιδιαίτερα μεγάλων διαστάσεων, η τυχαία προβολή μπορεί να λειτουργήσει ως μια αξιόλογη προσεγγιστική λύση για την επίλυση ενός kNN προβλήματος, καθώς αποφεύγει τον μαζικό υπολογισμό όλων των αποστάσεων των σημείων των σετ Q και C.

Μια ακόμη προσέγγιση για την αποφυγή των υπολογισμών είναι η εφαρμογή ενός αρχικού clustering των σετ Q και C και έπειτα την εφαρμογή της τριγωνικής ανισότητας μεταξύ των αποστάσεων των κέντρων των clusters του C από τα σημεία του Q. Η εναλλακτική αυτή μέθοδος αποσκοπεί στον αποκλεισμό cluster τα οποία απέχουν αρκετά από τα σημεία query, και ως εκ τούτου δεν υπάρχει κέρδος από το να εκτελέσουμε τους υπολογισμούς των αποστάσεών τους. Η προσέγγιση αυτή περιγράφεται στο άρθρο των Guoyang Chen, Yufei Ding, και Xipeng Shen με τίτλο Sweet KNN [6] στο οποίο παρουσιάζεται αναλυτικά η μεθοδολογία για των υπολογισμό των υποψηφίων clusters.

Σχετικές εργασίες

Ως μέτρο σύγκρισης της απόδοσης των αλγορίθμων που παρουσιάζουμε στο πλαίσιο αυτής της διπλωματικής, χρησιμοποιούμε τέσσερα διαφορετικά προγράμματα kNN. Το πρώτο από αυτά είναι το FLANN [14]το οποίο είναι σχεδιασμένο για να εκτελείται στη CPU, και μάλιστα είναι γραμμένο σε C++ αν και υπάρχει βιβλιοθήκη της Julia [21] που

παρέχει ένα wrapper που επιτρέπει την χρήση του μέσα από κώδικα Julia. Το FLANN περιλαμβάνει μια σειρά από διαφορετικές προσεγγίσεις για τον υπολογισμό ενός kNN προβλήματος, όπως γραμμική αναζήτηση (για μια brute force αναζήτηση), αλλά και χρήση δέντρων αναζήτησης τα οποία μπορούν υπό ορισμένες συνθήκες να ξεπεράσουν σε απόδοση την γραμμική αναζήτηση, με μειωμένη φυσικά ακρίβεια. Το δεύτερο από τα προγράμματα που χρησιμοποιούμε για μέτρο σύγκρισης είναι το πακέτο ParallelNeighbors [13] που μάλιστα είναι εγγενώς γραμμένο σε Julia, και επιπλέον κάνει χρήση της GPU, κάτι που το θέτει ως ένα πολύ καλύτερο μέτρο σύγκρισης για τους αλγορίθμους που παρουσιάζουμε στο πλαίσιο αυτής της διπλωματικής. Το τρίτο πρόγραμμα είναι το Nearest Neighbors [12] που αποτελεί μια υλοποίηση Knn αλγορίθμων σε Julia εγγενώς (σε αντίθεση με το FLANN), και για τον λόγο αυτό Θεωρείται καλύτερο μέτρο σύγκρισης με τον δικούς μας αλγορίθμους σε CPU. Τέλος το τέταρτο πρόγραμμα είναι το Faiss [9] που παρέχει μια προσεγγιστική μέθοδο βάσει στην ομοιότητα των δεδομένων ως διανύσματα στο χώρο.

Πιο συγκεκριμένα, στην περίπτωση του FLANN χρησιμοποιούμε την Linear Search μέθοδο, η οποία πραγματοποιεί μια γραμμική, brute force αναζήτηση, που είναι άμεσα συγκρίσιμη με την brute force υλοποίηση σε CPU που παρουσιάζουμε στο πλαίσιο αυτής της διπλωματικής. Επιπρόσθετα ως μια μέθοδο για προσέγγιση των αποτελεσμάτων, χρησιμοποιούμε την αναζήτηση με χρήση KD trees, έτσι ώστε να συγκρίνουμε τις προσεγγιστικές μεθόδους που επίσης προτείνουμε στην διπλωματική. Με το ίδιο σκεπτικό, στην περίπτωση του Nearest Neighbors χρησιμοποιήσαμε την έκδοση που κάνει αναζήτηση με KD trees, μιας και έχουμε ήδη αρκετούς brute force αλγορίθμους, και έτσι η χρήση ενός ακόμη υλοποίησης KD trees είναι πιο εκλυστική.

Αντίστοιχα, στην περίπτωση της βιβλιοθήκης ParallelNeighbors, χρησιμοποιούμε την

υβριδική μέθοδο υπολογισμών, η οποία πραγματοποιεί τους υπολογισμούς αποστάσεων στην GPU και έπειτα ταξινομεί τα αποτελέσματα στην CPU. Ο βασικός λόγος αυτής της επιλογής έχει να κάνει με την καλύτερη απόδοση των αλγορίθμων ταξινόμησης στις CPU παρά στις GPU, κάτι το οποίο αναφέρεται και από τους δημιουργούς της βιβλιοθήκης ParallelNeighbors.

Σε ότι αφορά το Faiss, αν και το ίδιο το πρόγραμμα είναι υλοποιημένο σε C++, ο κώδικάς του είναι προσβάσιμος μέσω Python, που απαιτεί τόσο ένα περιβάλλον Conda για την εγκατάστασή του, αλλά και τη βιβλιοθήκη wrapper Faiss.jl [11] που παρέχει μία διεπαφή σε Julia μέσω της Python. Για τις ανάγκες της διπλωματικής εστιάσαμε στην GPU έκδοση του Faiss, μιας και είναι η γρηγορότερη εκδοχή του.

Επίσης, για την ανάγκη του διαμοιρασμού των σετ Q και C σε clusters για μια από τις προσεγγίσεις υπολογισμού kNN προβλημάτων, χρησιμοποιούμε το πακέτο Clustering της Julia [10]. Ως αλγόριθμος clustering στην περίπτωση αυτή χρησιμοποιείται ο kmeans, καθώς μας δίνει ένα καλό διαχωρισμό των σετ δεδομένων βάσει της απόστασης των σημείων τους. Δυστυχώς ωστόσο, όπως θα δούμε και στα αποτελέσματα των δοκιμών που έγιναν, ο επιπρόσθετος φόρτος εργασίας για την δημιουργία των clusters είναι ιδιαίτερα μεγάλος, κάτι που αθροιστικά αυξάνει αρκετά τον χρόνο εκτέλεσης καθώς κατά τη διάρκεια του επιμερισμού της εργασίας χωρίζουμε σε πολλά κομμάτια το πρόβλημα. Αυτό δημιουργεί την ανάγκη να επιμερίσουμε κάθε ένα από αυτά τα υποσύνολα σε clusters, με αποτέλεσμα να έχουμε πολύ μεγάλη απώλεια στην συνολική απόδοση. Εδώ θα μπορούσε να γίνει μια επαναχρησιμοποίηση κάποιων από τα clusters, ιδίως για το σετ C, αλλά δεδομένου ότι η προσέγγισή μας αφορά μεγάλα σετ δεδομένων, κάτι τέτοιο θα οδηγούσε σε πολύ μεγάλη κατανάλωση μνήμης στην οποία θα έπρεπε να κρατάμε προσωρινά τα clusters αυτά. Εκτός αυτού, θα χρειαζόμασταν και πάλι να υπολογίσουμε ξανά τις αποστάσεις των σημείων με

τα κέντρα των cluster από τα σύνολα Q, καθώς αυτά τα χρησιμοποιούμε μία φορά στην διάρκεια της εκτέλεσης του αλγορίθμου μας.

Βεβαίως μια καλύτερη προσέγγιση για το clustering θα μπορούσε να βελτιώσει σημαντικά την χρήση του αλγορίθμου που βασίζεται στην τριγωνική ανισότητα, κάτι το οποίο θα μπορούσε να βελτιώσει μελλοντικά την υλοποίηση του αντίστοιχου αλγορίθμου. Ιδιαίτερο ενδιαφέρον θα είχε μια υλοποίηση που θα μπορούσε να εκτελεστεί εξολοκήρου στην GPU, αν και προς το παρόν η ανάγκη για ταξινομήσεις κατά την διάρκεια της επιλογής των υποψηφίων clusters τα οποία έχουν σημεία που μπορεί να είναι αρκετά κοντά στα σημεία query ώστε να ανήκουν στα k κοντινότερα που ζητάμε, οδηγεί σε προβλήματα απόδοσης.

Ένας ακόμη περιοριστικός παράγοντας που επηρεάζει όλους τους GPU αλγορίθμους μας είναι η μη ύπαρξη αποδοτικής υλοποιησης της συνάρτησης μερικής ταξινόμησης partialsortperm της Julia σε CUDA. Αν και υπάρχει υλοποίηση της sort για ταξινόμηση πινάκων που είναι γραμμένη για GPU, κάτι τέτοιο δεν μας καλύπτει στην γενικότερη περίπτωση, καθώς είναι πολύ καλύτερη η απόδοση όταν μπορούμε να σταματήσουμε την εκτέλεση της ταξινόμησης μόλις φτάσουμε τα k κοντινότερα σημεία για το κάθε κομμάτι των αποτελεσμάτων μας.

Συνεισφορά

Ένας από τους βασικότερους σκοπούς της διπλωματικής αυτής είναι η παρουσίαση μιας προσέγγισης που αφενός κάνει χρήση κατανεμημένου προγραμματισμού για την επίλυση ενός προβλήματος kNN, αλλά και επίσης μπορεί να περιορίσει τους υπολογισμούς δουλεύοντας σε υποσύνολα των σετ δεδομένων κάθε δεδομένη χρονική στιγμή. Αν και οι περισσότερες εναλλακτικές προσεγγίσεις μπορούν να εκμεταλλευτούν τους πλήρεις πόρους ενός υπολογιστή, αυτοί δεν αρκούν όταν τα σετ Q και C μεγαλώνουν σε τέτοιο βαθμό

είναι πρακτικά αδύνατον να χωρέσει ο πίνακας αποστάσεων Dist στην μνήμη. Αλλά ακόμη και σε περιπτώσεις που κάτι τέτοιο είναι οριακά εφικτό, ο χρόνος εκτέλεσης δεν είναι και ο καλύτερος δυνατός, δεδομένου ότι σε μια κατανεμημένη προσέγγιση μπορούμε να επιμερίσουμε το συνολικό πρόβλημα σε κομμάτια που κάθε ένα εκτελείται τμηματικά σε ένα ξεχωριστό υπολογιστή.

Ιδιαίτερα στην περίπτωση υπολογιστικών συστοιχιών, μια τέτοια προσέγγιση μας δίνει πολύ καλύτερα αποτελέσματα καθώς η επικοινωνία των κόμβων είναι αρκετά αποδοτική ώστε να δίνεται η εντύπωση ότι εργαζόμαστε σε ένα συνολικά μεγαλύτερο υπολογιστή που αθροίζει τους πόρους όλων των κόμβων που χρησιμοποιούνται κατά την εκτέλεση των υπολογισμών. Το βασικότερο μειονέκτημα που συνήθως προκύπτει στις περιπτώσεις κατανεμημένου προγραμματισμού είναι η αυξημένη πολυπλοκότητα στον σχεδιασμό της υλοποίησης, μιας και παραδοσιακές προσεγγίσεις βασίζονται σε πολύ χαμηλότερου επιπέδου προγραμματισμό. Έτσι ένα σημαντικό κομμάτι του κώδικα αφορά την διαχείριση της ανταλλαγής μηνυμάτων μεταξύ των κόμβων, το οποίο όταν μάλιστα συνδυαστεί και με την πολυπλοκότητα του προγραμματισμού σε GPUs, καθιστά το όλο εγχείρημα λιγότερο ελκυστικό.

Σε πλήρη αντίθεση με μια υλοποίηση χαμηλού επιπέδου, η επιλογή μιας γλώσσας ανώτερου επιπέδου όπως η Julia, αλλά και η χρήση πακέτων της που αναλαμβάνουν την επικοινωνία μεταξύ κόμβων σε ένα κατανεμημένο περιβάλλον, αλλά και απλοποιούν τον προγραμματισμό σε GPU, βοηθά αρκετά με το να έχουμε μια εναλλακτική λύση που μπορεί εύκολα να προσαρμοστεί στις ανάγκες ακόμη και μεγαλύτερων σετ δεδομένων. Κάνοντας εύκολη υπόθεση το κομμάτι της επικοινωνίας μεταξύ των υπολογιστικών κόμβων ενός κατανεμημένου συστήματος, μειώνεται επίσης σημαντικά και η απαίτηση για συντήρηση του κώδικα, μιάς και όλες αυτές οι βιβλιοθήκες της Julia ενημερώνονται ανεξάρτητα από

τον δικό μας κώδικα.

Η βιβλιοθήκη μάλιστα CUDA.jl είναι ιδιαίτερα χρήσιμη στο κομμάτι της συντήρησης, μιας και η CUDA είναι μια συνεχώς αναπτυσσόμενη γλώσσα, και επομένως η δυνατότητα να έχουμε αρκετές λειτουργίες χαμηλότερου επιπέδου πίσω από ένα πιο συνεπές περιβάλλον διεπαφής και αφαίρεσης (abstraction) μειώνει τόσο την πολυπλοκότητα του κώδικά μας, αλλά και την ανάγκη για τροποποιήσεις από τη μεριά μας όποτε υπάρχουν σχετικά μεγάλες αλλαγές στις εκδόσεις της γλώσσας.

Η βιβλιοθήκη ClusterManagers.jl με τη σειρά της παρέχει τη δυνατότητα εύκολης διαχείρισης πολλών διαφορετικών συστημάτων ουρών εργασιών (job queues), όπως τα Slurm, Load Sharing Facility (LSF), Sun Grid Engine, SGE (μέσω qrsh), PBS, Scyld, HTCondor, Local manager (με CPU affinity setting), Kubernetes (K8s) (μέσω του K8sClusterManagers.jl). Όλές αυτές οι επιλογές είναι άμεσα διαθέσιμες χωρις σημαντικές αλλαγές στον υπόλοιπο κατανεμημένο κώδικα πέρα από την αρχικοποίηση του cluster manager στην αρχή του προγράμματος.

Τέλος οι προτεινόμενες μέθοδοι υπολογισμού των κοντινότερων γειτόνων καλύπτουν μια ποικιλία διαφορετικών προβλημάτων, χωρίς να περιοριζόμαστε στον αριθμό των διαστάσεων, καθώς έχουμε και τη δυνατότητα προσέγγισης του αποτελέσματος μέσω τυχαίων προβολών που είναι ιδιαίτερα χρήσιμο σε περιπτώσεις που ο αριθμός d των διαστάσεων των δεδομένων είναι πολύ μεγάλος. Επίσης χάρη στην δυνατότητα τμηματικού υπολογισμού των αποστάσεων, είναι εφικτό να προσαρμοστεί η εκτέλεση ενός kNN προβλήματος βάσει των διαθέσιμων πόρων του συστήματος ή της συστοιχίας που έχουμε στη διάθεσή μας, χωρίς να δημιουργούνται περιορισμοί που απλά καθιστούν μια τέτοια προσπάθεια ανέφικτη.

2

Θεωρία

Μια από τις βασικότερες έννοιες για την επίλυση ενός kNN προβλήματος είναι ο υπολογισμός των αποστάσεων των σημείων του συνόλου query (Q) από το σύνολο corpus (C). Για τον υπολογισμό αυτόν μπορούν να χρησιμοποιηθούν διάφορες μετρικές, όπως η Ευκλείδεια απόσταση, η Manhattan, η Chebyshev, η Minkowski, η Mahalanobis, καθώς και άλλες λιγότερο συχνές που είναι πιο χρήσιμες σε συγκεκριμένες περιπτώσεις. Γενικότερα οποιαδήποτε από αυτές θα μπορούσε να χρησιμοποιηθεί ως βάσει των υπολογισμών αποστάσεων, ωστόσο στο πλαίσιο της διπλωματικής αυτής προτιμήθηκε η τετράγωνη Ευκλείδεια απόσταση.

Ως Ευκλείδεια απόσταση ορίζεται η απόσταση δύο σημείων στο χώρο, χρησιμοποιώντας τις καρτεσιανές συντεταγμένες τους και το Πυθαγόρειο θεώρημα. Καθώς ασχολούμαστε με πολυδιάστατα δεδομένα, η πιο γενικευμένη της μορφή για n διαστάσεις και δύο

σετ δεδομένων Q και C (query και corpus αντίστοιχα) μπορεί να περιγραφεί ως:

$$d(q,c) = \sqrt{\sum_{i=1}^{n}(q_i - c_i)^2} \text{όπου τα σημεία} q \in Q, c \in C \tag{2.1}$$

Μερικές από τις βασικές ιδιότητες της Ευκλείδειας απόστασης είναι:

- ◇ Είναι συμμετρική, δηλαδή $d(q,c) = d(c,q)$
- ◇ Είναι ένας θετικός αριθμός, εκτός από την περίπτωση που μετράμε την απόσταση ενός σημείου από τον εαυτό του, που είναι μηδέν.
- ◇ Υπακούει στην τριγωνική ανισότητα. Δηλαδή εάν έχουμε τα σημεία a, b και c τότε θα ισχύει ότι $d(a,b) \leq d(a,c) + d(c,b)$ κάτι που απλά σημαίνει ότι δεν υπάρχει κοντινότερος τρόπος για να πάμε από το σημείο a στο b από μια ευθεία γραμμή.

Το βασικότερο βέβαια πρόβλημα της Ευκλείδειας απόστασης είναι η ανάγκη του υπολογισμού της τετραγωνικής ρίζας του αθροίσματος $\sum_{i=1}^{n}(q_i - c_i)^2$ κάτι που δεν είναι ιδιαίτερα αποδοτικό σε μία GPU, καθώς η τετραγωνική ρίζα είναι ένας σχετικά ακριβός υπολογισμός. Αντίθετα αν προτιμήσουμε να κρατήσουμε ως μετρική την τετράγωνη Ευκλείδεια απόσταση, τότε μπορούμε να κερδίσουμε αρκετά στο χρόνο εκτέλεσης, κάτι το οποίο μάλιστα αποτελεί και την προσέγγιση και άλλων προγραμμάτων, όπως το FLANN και το ParallelNeighbors, αν και το τελευταίο διαθέτει ως εναλλακτική επιλογή και την χρήση της απλής Ευκλείδειας απόστασης.

Έχοντας αυτό υπόψη, η μετρική που χρησιμοποιούμε για τον υπολογισμό των αποστάσεων είναι η τετράγωνη Ευκλείδεια απόσταση που ορίζεται ως:

$$d^2(q,c) = \sum_{i=1}^{n} (q_i - c_i)^2 \tag{2.2}$$

Αν και η μετρική αυτή είναι σχετικά απλή στην υλοποίησή της, ένα από τα βασικότερα προβλήματά της είναι ότι όσο αυξάνεται ο αριθμός των σημείων, αλλά και του αριθμού των διαστάσεων, οι απαιτούμενοι υπολογισμοί γίνονται υπερβολικά πολλοί για να έχουμε μια γρήγορη εκτίμηση των αποστάσεων. Για την αντιμετώπιση αυτού του προβλήματος, προσπαθούμε να μειώσουμε τον αριθμό των διαστάσεων, καθώς κάτι τέτοιο μπορεί να αυξήσει την απόδοση των υπολογισμών και να αποφύγει την "κατάρα της διαστατικότητας" που υπάρχει σε όλα τα πολυδιάστατα σετ δεδομένων. Εδώ φυσικά θα μπορούσαν να χρησιμοποιηθούν πολλές διαφορετικές μέθοδοι για μείωση της διαστατικότητας των δεδομένων, αλλά για να αποφύγουμε την υπερβολικό φόρτο εργασίας που περιλαμβάνουν πολλές από αυτές, προτιμήσαμε την τυχαία προβολή των σημείων σε υπερεπίπεδα. Αυτό είναι εφικτό λόγω του ότι βάσει του λήμματος των Johnson–Lindenstrauss, ένας αριθμός από σημεία σε ένα υψηλής διαστατικότητας Ευκλείδειο χώρο, μπορεί να χαρτογραφηθεί σε ένα χώρο μικρότερων διαστάσεων χωρίς να διαταραχθούν οι σχέσεις των μεταξύ τους αποστάσεων.

Έτσι λοιπόν εάν έχουμε έναν πίνακα A με διαστάσεις d x n, και πάρουμε ένα τυχαίο πίνακα R διαστάσεων r x d, τότε ο πίνακας $A^{RP}_{r\times n} = R_{r\times d}A_{d\times n}$ είναι ο πίνακας που προκύπτει από την τυχαία προβολή των σημείων του A στον r-διάστατο χώρο μέσω του πινακα $R_{r\times d}$. Υποθέτοντας ότι επιλέγουμε ένα κατάλληλο $r \ll d$ τότε ο καινούργιος πίνακας $A^{RP}_{r\times n}$ είναι πολύ πιο κατάλληλος για υπολογισμούς αποστάσεων μιας και α αριθμός των διαστάσεων των σημείων του είναι σημαντικά μικρότερος.

Η επιλογή ενός κατάλληλου πίνακα R είναι σημαντική για την απλοποίηση των υπο-

λογισμών, οποτε αν και μπορούμε να επιλέξουμε έναν πίνακα που ακολουθεί μια Gaussian κατανομή, προτιμήσαμε την μέθοδο που προτείνεται από τον Αχλίοπτα που ορίζεται ως:

$$R_{ij} = \begin{cases} +\sqrt{3}, \text{ με πιθανότητα } \frac{1}{6} \\ 0, \text{ με πιθανότητα } \frac{2}{3} \\ -\sqrt{3}, \text{ με πιθανότητα } \frac{1}{6} \end{cases} \tag{2.3}$$

ως μια υπολογιστικά πιο αποδοτική λύση, καθώς ο πίνακας αυτός έχει αρκετά μηδενικά στοιχεία.

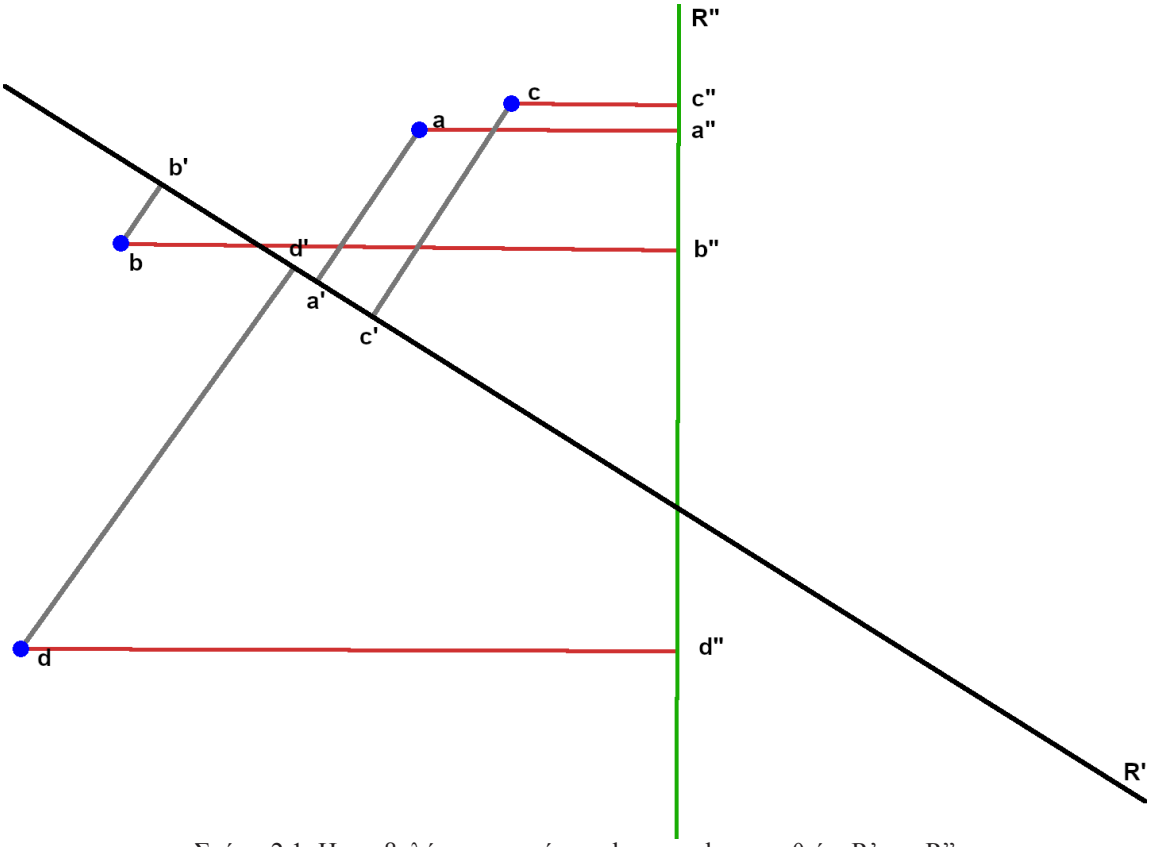
Από το ίδιο το θεώρημα Johnson-Lindenstrauss μπορούμε μάλιστα να εκφράσουμε τη σχέση των σημείων του αρχικού d-διάστατου χώρου με αυτά που προκύπτουν μετά την προβολή σε r διαστάσεις ως:

$$(1-\varepsilon)\|a-b\|^2 \leq \|Ra-Rb\|^2 \leq (1+\varepsilon)\|a-b\|^2 \tag{2.4}$$

όπου το ε είναι ένας θετικός αριθμός που συμβολίζει την "διαστρέβλωση" των δεδομένων λόγω την μείωσης της διαστατικότητας τους μέσω της τυχαίας προβολής, και το r είναι ένας ακέραιος αριθμός για τον οποίο ισχύει ότι $r \geq r_0 = O(\epsilon^{-2} logd)$.

Η "διαστρέβλωση" αυτή των δεδομένων μπορεί να οδηγήσει σε λανθασμένες κατα περίπτωση εκτιμήσεις, κάτι που μπορούμε διαισθητικά να παραστήσουμε στο παρακάτω διάγραμμα, όπου προβάλλουμε τα σημεία a, b, c και d στις γραμμές R' και R" που παίρνουμε τυχαία. Όπως φαίνεται και στο διάγραμμα, η προβολή των σημείων στην R' δημιουργεί την λανθασμένη εντύπωση ότι το d είναι κοντινότερο στο a από ότι πραγματικά είναι, ενώ κάτι τέτοιο δεν ισχύει στην προβολή στην R". Θα μπορούσαμε μάλιστα να πούμε ότι παίρνοντας παραπάνω από μία τυχαίες προβολές, μπορούμε να βελτιώσουμε την ακρίβεια των εκτιμησεων μας χωρίς να χρειαστεί να αυξήσουμε σημαντικά το r ώστε το σφάλμα να

παραμείνει σε πολύ μικρά επίπεδα.



Σχήμα 2.1: Η προβολή των σημείων a, b, c και d στις ευθείες R' και R"

Ως μια μέθοδος για να αποφύγουμε τα παραπάνω προβλήματα είναι να πάρουμε μια σειρά από P τυχαίες προβολές, και για κάθε μία από αυτές να βρούμε τους κοντιμότερους γείτονες των προβολών του συνόλου Q. Έπειτα, μπορούμε να χρησιμοποιήσουμε τη λιστα με τα σημεία που εμφανίστηκαν στα αποτελέσματα ως ένα είδος φιλτραρίσματος των σημείων του συνόλου C, έτσι ώστε να αποκλείσουμε αυτά που δεν φαίνεται να είναι κοντά στα σημεία του συνόλου Q. Επιπρόσθετα για να κρατήσουμε τον αριθμό των προβολών P σχετικά χαμηλό, είναι καλό να αυξήσουμε το πλήθος των γειτόνων k που ψάχνουμε, έτσι ώστε να εξετάσουμε περισσότερα πιθανά σημεία, ιδίως σε περιπτώσεις που οι προβολές

τους είναι πολύ κοντά η μία στην άλλη. Γενικά, ένας ικανοποιητικός αριθμός γειτόνων για την κάθε προβολή κυμαίνεται μεταξύ 2-4k, έτσι ώστε να μη χρειαστούμε μεγαλύτερο αριθμό προβολών για αυξήσουμε την ακρίβεια του τελικού αποτελέσματος.

Με τον τρόπο αυτό, για κατάλληλα r και P μπορούμε να δημιουργήσουμε μια λίστα από υποψήφια σημεία που πρέπει να εξετάσουμε για να βρούμε τους k κοντινότερους γείτονες των σημείων του συνόλου Q από το σύνολο C. Φυσικά σε αυτό το σημείο χρειάζεται να κάνουμε τους υπολογισμούς αποστάσεων στα αρχικά d-διάστατα δεδομένα, καθώς αυτά θα μας δώσουν τα ακριβή αποτελέσματα για τις πραγματικές αποστάσεις μεταξύ των σημείων. Ωστόσο λαμβάνοντας υπόψη ότι τα προηγούμενα βήματα έγιναν με υπολογισμούς σε δεδομένα πολύ μικρότερης διαστατικότητας, και από τα οποία μπορούμε να αποκλείσουμε αρκετά σημεία που δεν ανήκουν στην γειτονιά των σημείων του Q, μπορούμε πάρουμε μια προσέγγιση του αποτελέσματος γλιτώνοντας κάποιο αριθμό υπολογισμών.

3

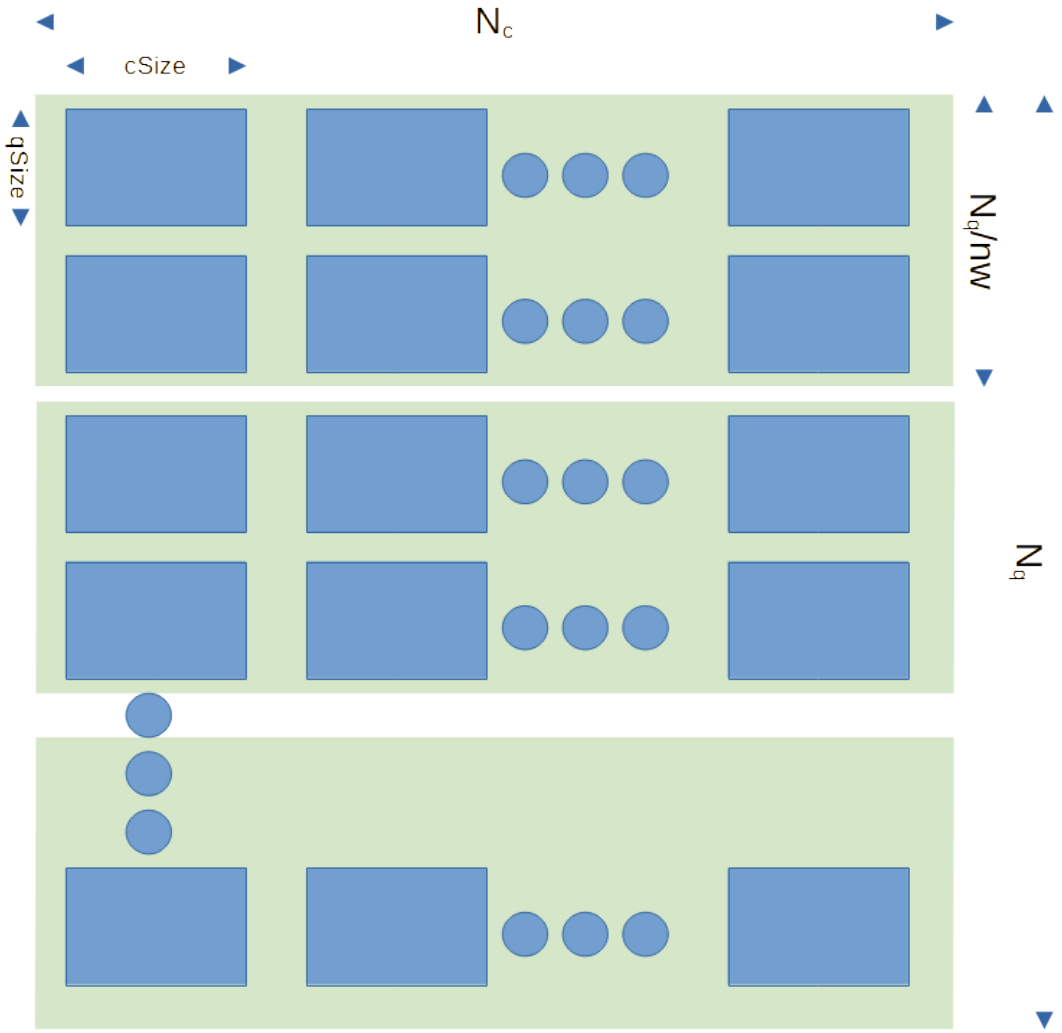
Μέθοδοι

Στην ενότητα αυτή θα παρουσιάσουμε αναλυτικότερα τις μεθόδους που υλοποιήθηκαν για την επίλυση ενός προβλήματος kNN με τη βοήθεια κατανεμημένου προγραμματισμού και χρήσης υπολογισμών σε GPUs. Καταρχάς θα πρέπει να αναφέρουμε ότι κύριος γνώμονας για τη σχεδίαση των μεθόδων ήταν η δυνατότητα διαχείρισης μεγάλων σετ δεδομένων, κάτι που κάνει πολύ πιο αποδοτική μια κατανεμημένη προσέγγιση. Άλλωστε ο επιπρόσθετος φόρτος εργασίας για το διαμοιρασμό ενός μικρού σετ δεδομένων σε ένα σύνολο υπολογιστικών μονάδων, καθιστά μια τέτοια προσπάθεια άσκοπη, καθώς σε όλες τις περιπτώσεις είναι πιο αποδοτικό να εκτελεστούν οι υπολογισμοί σε ένα μεμονωμένο υπολογιστή με τη συνδρομή της GPU του. Ωστόσο, όταν έχουμε να αντιμετωπίσουμε ένα ιδιαίτερα πρόβλημα όπου τα σετ Q και C έχουν ένα αρκετά μεγάλο αριθμό σημείων, τότε τα πλεονεκτήματα από τη χρήση περισσότερων διαθέσιμων πόρων γίνονται εμφανή. Για αυτό τον λόγο ως είσοδο στις μεθόδους υπολογισμού μας χρησιμοποιούμε δυαδικά αρχεία τα οποία μπορούμε εύκολα να δημιουργήσουμε με τη βοήθεια των κατάλληλων συναρτή-

σεων από την συνοδευτική βιβλιοθήκη read_file.jl που μας δίνει τη δυνατότητα να αποθηκεύσουμε ένα μεγάλο όγκο δεδομένων χωρίς να επιβαρύνουμε τη χρήση της μνήμης των συστημάτων μας.

Η βασική συνάρτηση για τον υπολογισμό του προβλήματος kNN είναι η distributedKNN, που έρχεται σε δύο μορφές, αναλόγως με το αν θέλουμε επιστροφή των αποτελεσμάτων σε μορφή πινάκων ή σε μορφή δυαδικών αρχείων εξόδου. Στην πρώτη περίπτωση παίρνουμε ως επιστρεφόμενο αποτέλεσμα δύο πίνακες: τα indexes των k κοντινότερων γειτόνων, και τις αντίστοιχες αποστάσεις τους από τα σημεία του Q. Στην δεύτερη περίπτωση, τα αποτελέσματα αντί να επιστραφούν σε μορφή πινάκων, αποθηκεύονται σε δυαδικά αρχεία εξόδου που όπως και προηγουμένως κρατούν τα indexes των k κοντινότερων γειτόνων, και τις αντίστοιχες αποστάσεις τους από τα σημεία του Q. Η επιλογή της καταλληλότερης μεθόδου βασίζεται στους διαθέσιμους πόρους, καθώς εάν θέλουμε τα αποτελέσματα σε μορφή πινάκων, τότε θα πρέπει να έχουμε αρκετό χώρο στη μνήμη RAM του κόμβου που εκτελεί τον κώδικα της Julia, διαφορετικά η μόνη άλλη λύση είναι να χρησιμοποιήσουμε τα αρχεία εξόδου.

Και στις δύο περιπτώσεις το βασικό κομμάτι της συνάρτησης distributedKNN διαμοιράζει την εργασία σε κομμάτια, λαμβάνοντας υπόψη το μέγεθος N_Q που αντιστοιχεί το πλήθος των σημείων του συνόλου Query) έτσι ώστε η κάθε μία από τις p διεργασίες να αναλάβει όσο το δυνατόν πιο κοντά σε $\frac{1}{p}$ σημεία των οποίων τους k κοντινότερους γείτονες στο C θα πρέπει να βρει. Επίσης αν έχουμε παραπάνω από μία GPUs διαθέσιμες στη συστοιχία μας, η κάθε μία διεργασία ανατείθεται και σε διαφορετική GPU (σε περίπτωση που οι διεργασίες είναι πιο πολλές από τις GPU, αρχίζουμε ξανά από την αρχή τις αναθέσεις μόλις φτάσουμε στην τελευταία). Έπειτα η distributedKNN περιμένει τα αποτελέσματα από όλες τις διεργασίες και ταυτόχρονα τυπώνει την πρόοδό τους στο τερματικό.



Σχήμα 3.1: Διαχωρισμός της εργασίας σε p διεργασίες μεγέθους $\frac{N_q}{nw}$, και επιπρόσθετα κατάτμηση του φόρτου εργασίας σε κομμάτια μεγέθους qSize x cSize

Για τους υπολογισμούς, η distributedKNN διαθέτει διάφορες μεθόδους που μπορούν να χρησιμοποιηθούν βάσει του ορίσματος algorithm που μας δίνει τις ακόλουθες επιλογές:

- Algorithm 0 για FLANN (brute force, linear search)
- Algorithm 1 για FLANN (KD trees, approximated)
- Algorithm 2 για brute force στην GPU
- Algorithm 3 για brute force στην CPU
- Algorithm 4 για ParallelNeighbors
- Algorithm 5 για random projection στην CPU
- Algorithm 6 για random projection στην GPU
- Algorithm 7 για TI (triangular inequality) filtering στην CPU
- Algorithm 8 για TI (triangular inequality) filtering στην GPU
- Algorithm 9 για NearestNeighbors (kd trees)
- Algorithm 10 για Faiss (GPU)

Στην παραπάνω λίστα βρίσκουμε και τα προγράμματα που χρησιμοποιούμε ως μετρικές σύγκρισης, όπως το FLANN, το ParallelNeighbors, το Faiss και το NearestNeighbors μιας και κανένα από αυτά δεν διαθέτει κατανεμημένη υλοποίηση έτσι ώστε να τα συγκρίνουμε απευθείας με τις μεθόδους μας. Από τις υπόλοιπες μεθόδους έχουμε βασικά τρεις επιλογές, που αφορούν την brute force αναζήτηση, την χρήση random projection και τέλος τη χρήση triangular inequality για τους υπολογισμούς των αποστάσεων. Βέβαια κάθε μία

από αυτές τις επιλογές έρχεται σε δύο εκδόσεις ανάλογα με το αν υλοποιούνται οι υπολογισμοί στην CPU ή στην GPU. Αν και η βασική εστίαση αυτής της διπλωματικής έχει να κάνει με τον υπολογισμό ενός κατανεμημένου προβλήματος kNN σε GPUs, παρέχουμε και ως εναλλακτικές επιλογές την χρήση μόνο της CPU ειδικά σε περιπτώσεις που δεν υπάρχει υποστηρίξιμη GPU, ή θέλουμε να συγκρίνουμε την απόδοση μεταξύ της GPU και της CPU.

Για τον υπολογισμό των brute force μεθόδων χρησιμοποιούμε την πιο απλή μορφή της τετράγωνης Ευκλείδειας απόστασης 2.2, μιας και μας δίνει μία καλή λύση για την υλοποίηση των υπολογισμών στην GPU. Καθώς μπορούμε να υπολογίσουμε τις αποστάσεις του κάθε σημείου το Q από το C ανεξάρτητα από τα υπόλοιπα σημεία του ίδιου συνόλου. αυτό μας επιτρέπει να αποφύγουμε τον συγχρονισμό των νημάτων της GPU. Αυτό γίνεται διότι το τετράγωνο του αθροίσματος μπορεί να υπολογιστεί για κάθε σημείο Q και C αντίστοιχα, έχει μόνο ένα νήμα της GPU που γράφει στο περιεχόμενό του. Με αυτό τον τρόπο αποφεύγουμε την ανάγκη ατομικότητας στον κώδικα του kernel που τρέχουμε στη GPU, κάτι που θα καθυστερούσε ιδιαίτερα τους υπολογισμούς. Έτσι ο kernel των υπολογισμών διαμορφώνεται ως εξής:

```
@inbounds function euclDistGPUb(
    Q_d,
    C_d,
    Distance_d,
    nQ::Int,
    nC::Int,
    d::Int
    )
    xIndex = (blockIdx().x - 1) * blockDim().x + threadIdx().x
    yIndex = (blockIdx().y - 1) * blockDim().y + threadIdx().y


    if (xIndex <= nC) && (yIndex <= nQ)
         tempDist = 0.0f0

```

```
        @inbounds for j in 1:d
          tempDist += (Q_d[j,yIndex] - C_d[j,xIndex])^2
        end
        @inbounds Distance_d[xIndex,yIndex] = tempDist
    end
    return
end
```

Στον παραπάνω κώδικα ως είσοδο έχουμε τους πίνακες Q_d και C_d που είναι οι εκδόσεις των πινάκων Q και C στην μνήμη της GPU, ενώ το αποτέλεσμα το λαμβάνουμε από τον πίνακα αποστάσεων Distance_d.

Αντίστοιχα στον υπολογισμό των brute force αποτελεσμάτων στη CPU, κάνουμε μερική αναδίπλωση του βρόχου των υπολογισμών, έτσι ώστε να βοηθήσουμε τον just in time compiler της Julia να δημιουργήσει κώδικα που κάνει καλύτερη χρήση διανυσματικών εντολών. Αν και δεν είναι απαραίτητο, σε αρκετές περιπτώσεις μπορεί να βοηθήσει στην βελτιστοποίηση του παραγόμενου κώδικα. Έτσι ο αντίστοιχος κώδικας υπολογισμού των αποστάσεων διαμορφώνεται ως εξής:

```
@inline function euclDistP(
    Q::AbstractArray{Float32},
    C::AbstractArray{Float32},
    Distance::AbstractArray{Float32},
    pointQ::Int,
    nC::Int,
    d::Int
    )

    if d > 3
        @inbounds for j in 1:nC
            lastj = d

            sum0 = 0.0f0
            sum1 = 0.0f0
            sum2 = 0.0f0
            sum3 = 0.0f0
            sum4 = 0.0f0
            @inbounds @simd for k in 1:4:d-3

                    sum0 += (Q[k,pointQ] - C[k,j])^2
```

```
                    sum1 += (Q[k+1,pointQ] - C[k+1,j])^2
                    sum2 += (Q[k+2,pointQ] - C[k+2,j])^2
                    sum3 += (Q[k+3,pointQ] - C[k+3,j])^2

                    lastj = k+3
            end
            @inbounds for k in lastj+1:d

                    sum4 += (Q[k,pointQ] - C[k,j])^2

            end
            Distance[j,pointQ] = sum0 + sum1 + sum2 + sum3 +sum4
        end
    else
        @inbounds for j in 1:nC

            Distance[j,pointQ] = 0.0f0

            @inbounds @simd for k in 1:d
                Distance[j,pointQ] += (Q[k,pointQ] - C[k,j])^2
            end
        end

    end
    return
end
```

Όπως και στο προηγούμενο παράδειγμα, οι πινακες Q και C περιέχουν τα δεδομένα εισόδου, ενώ τα αποτελέσματα αποθηκεύονται στον πίνακα αποστάσεων Distance. Μια ιδιαιτερότητα σ' αυτή την περίπτωση είναι η χρήση της μεταβλητής pointQ που κρατά το σημείο του Q για το οποίο κάνουμε τους υπολογισμούς μας. Ο λόγος που ακολουθείται μια τέτοια προσέγγιση είναι ότι μπορούμε έτσι να τρέξουμε τους υπολογισμούς μέσα σε ένα παράλληλο βρόγχο για όλα τα σημεία του Q, ενώ με κάθε επιστροφή της euclDistP έχουμε τη δυνατότητα να υπολογίσουμε τους k κοντινότερους γείτονες του σημείου για το οποίο τις αποστάσεις (από τα αντίστοιχα σημεία του C) μόλις βρήκαμε. Η παράλληλη αυτή εκτέλεση τόσο των υπολογισμών αποστάσεων όσο και της ταξινόμησης των αποτελεσμάτων βελτιώνει τον συνολικό χρόνο εκτέλεσης, δίνοντάς μας καλύτερη χρήση των

πόρων της CPU μας.

Για την περίπτωση των μεθόδων που βασίζονται στην τυχαία προβολή, αρχικά σχηματίζουμε τους τυχαίους πίνακες προβολής R και όπως περιγράφεται στην 2.3 και έπειτα εκτελούμε μία σειρά από P προβολές, βάσει του αριθμού των επαναλήψεων που καθορίζει ο χρήστης. Σε κάθε προβολή υπολογίζουμε τους πίνακες C_R και Q_R κάνοντας τους πολλαπλασιασμούς πινάκων ως εξής:

$$C_R = RC$$

$$Q_R = RQ$$

Οι νέοι αυτοί πίνακες είναι διαστάσεων r (όπου $r \ll d$) κάτι που κάνει τους υπολογισμούς των αποστάσεων των σημείων πιο εύκολους. Σε κάθε μία τυχαία προβολή που πραγματοποιούμε, υπολογίζουμε $k_r = max\big(4k, \frac{1}{10}cSize\big)$ γείτονες (όπου k ο ζητούμενος αρχικός αριθμός γειτόνων του προβλήματός μας). Με τον τρόπο αυτό προσπαθούμε να αποφύγουμε την ανάγκη για πολλές προβολές έτσι ώστε τα αποτελέσματά μας να είναι πιο ακριβή. Αφού λοιπόν βρούμε αυτούς τους k_r γείτονες, σημειώνουμε τα αποτελέσματα σε μία λίστα υποψηφίων και επαναλαμβάνουμε τη διαδικασία για τις υπόλοιπες εκτελέσεις. Στο τέλος έχουμε μια λίστα υποψηφίων candidateList την οποία χρησιμοποιούμε στο επόμενο στάδιο όπου χρησιμοποιούμε τα αρχικά δεδομένα μόνο που εκτελούμε όλους τους υπολογισμούς για τα σημεία του C τα οποία ανήκουν στην candidateList, ενώ για τα υπόλοιπα αναθέτουμε ως αποτέλεσμα στην αντίστοιχη θέση του πίνακα αποστάσεων Distance την τιμή typemax(Float32) που πρακτικά τα αποκλείει από το να εμφανιστούν στα τελικά αποτελέσματα.

Η παραπάνω διαδικασία παραμένει η ίδια τόσο στην έκδοση του αλγορίθμου για τη CPU όσο και για τη GPU, με την βασική διαφορά ότι οι πολλαπλασιασμοί των πινάκων

C_R και Q_R καθώς και υπολογισμοί των αποστάσεων των σημείων τους γίνονται στην CPU και GPU αντίστοιχα.

Τέλος, για την περίπτωση των μεθόδων που κάνουν χρήση clustering και της τριγωνικής ανισότητας, ξεκινάμε χωρίζοντας τα δεδομένα των πινάκων Q και C σε ένα αριθμό από clusters (ενδεικτικά επιλεχθηκε ο αριθμός $3\sqrt{qSize}$ και $3\sqrt{cSize}$ αντίστοιχα) και έπειτα κάνοντας χρήση του αλγορίθμου που περιγράφεται λεπτομερώς στο άρθρο SweetKNN των Guoyang Chen, Yufei Ding και Xipeng Shen [6] καταλήγουμε με μια λίστα που μας δίνει τα υποψήφια σημεία για τα οποία θα χρειαστούμε να υπολογίσουμε τις αποστάσεις. Όλα τα υπόλοιπα σημεία ανήκουν σε clusters που είναι πολύ μακριά για αποτελούν υποψήφια αποτελέσματα, και επομένως μπορούμε να τα αποκλείσουμε χωρίς άλλους υπολογισμούς. ‘Οπως και στις προηγούμενες μεθόδους υπάρχουν δύο υλοποιήσεις για CPU και GPU έτσι ώστε οι υπολογισμοί των αποστάσεων να γίνονται στην αντίστοιχη υπολογιστική μονάδα.

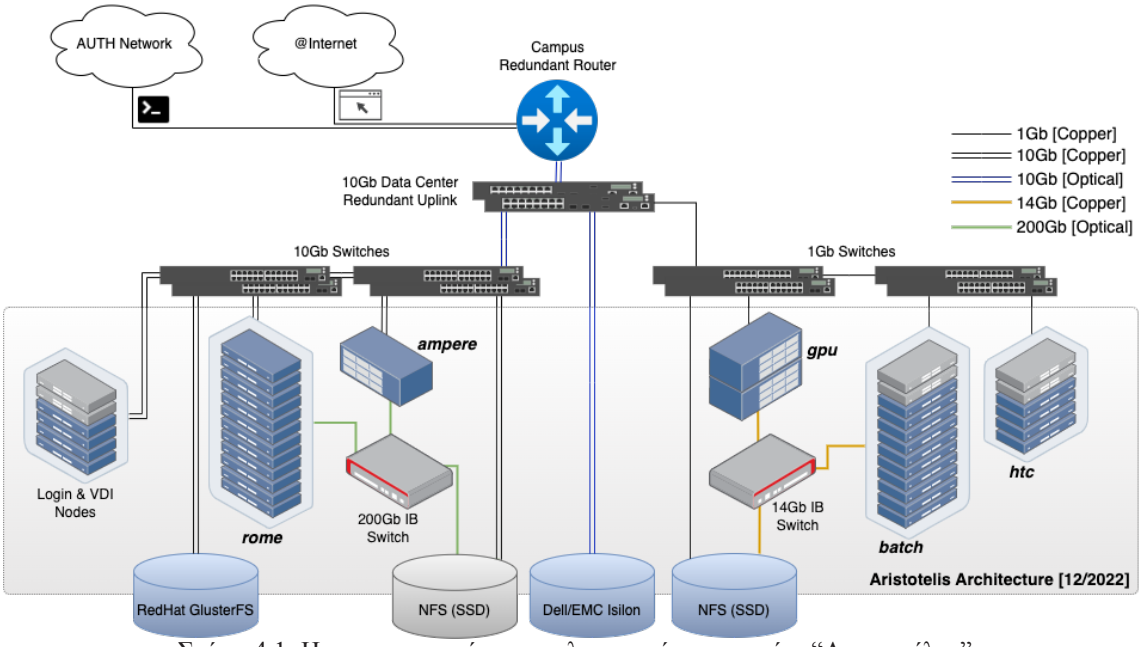
4

Πειράματα

Παράμετροι των πειραμάτων

Για τις ανάγκες της υλοποίησης, δοκιμών και μετρήσεων των αποτελεσμάτων των μεθόδων που καλύπτονται στο πλαίσιο αυτής της διπλωματικής έγινε χρήση των πόρων της υπολογιστικής συστοιχίας "Αριστοτέλης" που αποτελείται από ετερογενείς υπολογιστικούς κόμβους (compute nodes) που ομαδοποιούνται ανά ουρά (partition). Στην παρακάτω εικόνα μπορούμε να δούμε πιο αναλυτικά την αρχιτεκτονική της συστοιχίας:

Στα αρχικά στάδια της ανάπτυξης του κώδικα χρησιμοποιήθηκε η ουρά GPU, καθώς εκείνη τη χρονική περίοδο ήταν η μοναδική ουρά που διέθετε GPUs ικανές για εκτέλεση κώδικα σε CUDA. Ωστόσο με την επέκταση της υποδομής της συστοιχίας στην παρούσα μορφή που περιγράφεται στο παραπάνω διάγραμμα, η ανάπτυξη του κώδικα συνεχίστηκε στην ουρά Ampere. Η δεύτερη διαθέτει τόσο περισσότερους πόρους, αλλά και βασίζεται σε υλικό νεώτερης αρχιτεκτονικής που την καθιστά πολύ πιο κατάλληλη για τις ανάγκες των πειραμάτων μας.



Σχήμα 4.1: Η αρχιτεκτονική της υπολογιστικής συστοιχίας "Αριστοτέλης"

Πιο συγκεκριμένα οι διαθέσιμοι πόροι του κόμβου της (1 διαθέσιμος κόμβος στην ουρά) είναι [2]:

- Επεξεργαστής AMD EPYC 7742 [1] με 128 πυρήνες (1 εργασία ανά πυρήνα για συνολικά 128 job slots).
- 1024 GB RAM
- 200Gb InfiniBand HDR
- 8 κάρτες γραφικών NVIDIA A100 (40GB DDR6 RAM/κάρτα) [15] σε έναν κόμβο

Ως dataset για τα πειράματά μας χρησιμοποιήθηκαν τόσο πραγματικά δεδομένα από το σύνολο HIGGS [20] όσο και συνθετικά δεδομένα, ειδικά για περιπτώσεις πολύ υψηλού αριθμού διαστάσεων. Το συνολικό σετ δεδομένων του HIGGS περιέχει 11000000 κατα-

γραφές, ωστόσο για τις ανάγκες μας χρησιμοποιούμε ένα υποσύνολο 80000 σημείων για το Corpus, και 16000 για το Query. Αντίστοιχα χρησιμοποιήσαμε συνθετικά σετ δεδομένων με ίδιο αριθμό σημείων, αλλά διαφορετικών διαστάσεων, που ξεκινούσαν από 100 και έφτασαν τις 2800 κατά τη διάρκεια των πειραμάτων. Στα αποτελέσματα κρατήσαμε την περίπτωση με τα συνθετικά δεδομένα 2800 διαστάσεων, μιας και έχουν τη μεγαλύτερη διαφορά στις επιδόσεις από αυτά του σετ HIGGS, δίνοντάς μας μια καλύτερη εικόνα της συμπεριφοράς των μεθόδων σε προβλήματα μεγάλου, αλλά και μικρότερου αριθμού διαστάσεων.

Για την σωστή εκτίμηση των αποτελεσμάτων, μια απλή χρονομέτρησή τους μέσω των macros @time και @elapsed της Julia δεν είναι αρκετή. Μάλιστα κάτι τέτοιο θα οδηγούσε σε παραπλανητικά αποτελέσματα, καθώς η just in time φύση του compiler της δημιουργεί ένα σημαντικό σε κάποιες περιπτώσεις επιπλέον αύξηση στον χρόνο εκτέλεσης του κώδικα. Αυτό φυσικά συμβαίνει μόνο κατα την πρώτη εκτέλεση μιας συνάρτησης κατά τη διάρκεια ενός προγράμματος Julia, καθώς μελλοντικές κλήσεις κατά την ίδια συνεδρία χρησιμοποιούν τον ίδιο μεταγλωττισμένο κώδικα. Στην περίπτωση των συναρτήσεων που κάνουν την χρήση κώδικα CUDA, και ιδίως αν έχουν kernels χαμηλότερου επιπέδου, ο χρόνος μεταγλώττισης αυξάνεται ακόμη παραπάνω, καθώς εκτός από το κομμάτι της Julia χρειάζεται να μεταγλωττιστεί και ο κώδικας της CUDA. Έτσι, αν χρησιμοποιήσουμε τα πρώτα αποτελέσματα που πάρουμε από μια απλή χρονομέτρηση, το μόνο σίγουρο είναι ότι θα πάρουμε λανθασμένα αποτελέσματα. Και τα δύο προαναφερθέντα macros (@time και @elapsed) μάλιστα δηλώνουν ρητά το ποσοστό του χρόνου εκτέλεσης μιας εργασίας που αφορά τη μεταγλώττιση, αλλά και πάλι κάτι τέτοιο δεν είναι ιδανικό.

Η πιο δόκιμη μέθοδος για την σωστή χρονομέτρηση των συναρτήσεων (και άλλων εργασιών) στην Julia είναι η χρήση του πακέτου BenchmarkTools [17]. Το πακέτο αυτό

παρέχει μια αρκετά βολική μέθοδο σωστής χρονομέτρησης μέσω του macro @btime που πραγματοποιεί μια επανάληψη της εκτέλεσης του κώδικα που του αναθέτουμε και βάσει μιας δειγματοληψίας των αποτελεσμάτων απορρίπτει τον "θόρυβο" στα δεδομένα.

Εκτός βέβαια από τη χρονομέτρηση των αποτελεσμάτων, σημαντική είναι και η αξιολόγησή τους βάσει της ορθότητάς τους. Ειδικά στην περίπτωση των προσεγγιστικών μεθόδων έχει μεγάλη σημασία να εκτιμήσουμε την ακρίβεια των αποτελεσμάτων, μιας και ο χρόνος εκτέλεσης των υπολογισμών δεν αρκεί από μόνος του για να καταστήσει μια τέτοια λύση ικανοποιητική σε όλες τις περιπτώσεις. Για το λόγο αυτό χρησιμοποιούμε τα αποτελέσματα που παίρνουμε από την brute force μέθοδο στην CPU ως μέτρο σύγκρισης, έτσι ώστε να επαληθεύσουμε τα αποτελέσματα των άλλων μεθόδων. Αν indxs_c είναι ο πίνακας με τα indexes των k κοντινότερων γειτόνων που προκύπτει από την έξοδο της συνάρτησης distributedKNN για τον αλγόριθμο 3 (brute force σε CPU), και indxs_x οποιαδήποτε από τα αντίστοιχα αποτελέσματα των υπόλοιπων αλγορίθμων, τότε παίρνοντας την τομή $indxs\_c \cap indxs\_x$ και εκφράζοντας το πλήθος των σημείων της ως ποσοστό των του μεγέθους του πίνακα indxs_c. Με τη μέθοδο αυτή μπορούμε να πάρουμε μια εκτίμηση για το ποσοστό των κοινών αποτελεσμάτων μεταξύ διαφορετικών αλγορίθμων, αν και πάλι αυτό το μέγεθος από μόνο του δεν είναι αρκετό.

Ως μία ακόμη μετρική μπορούμε να αθροίσουμε τις αποστάσεις για το κάθε σημείο του Q από τον πίνακα dists_c (ο αντίστοιχος πίνακας αποστάσεων του indxs_c) και έπειτα κάνουμε το ίδιο για τους υπόλοιπους πίνακες αποστάσεων από τις άλλες μεθόδους. Παίρνοντας τις διαφορές για κάθε αντίστοιχο σημείο του Q, μπορούμε να υπολογίσουμε την τυπική απόκλισή τους, κάτι που μας βοηθά να δούμε αν υπάρχουν σημαντικές αποκλίσεις από τη μέση τιμή των διαφορών (που κανονικά θα πρέπει να είναι κοντά στο 0) . Αυτό έχει μεγαλύτερη αξία από την ίδια την μέση τιμή των διαφορών, γιατί μας δείχνει αν μια

μέθοδος τείνει να παράγει μεγαλύτερα σφάλματα, τα οποία μάλιστα θα μπορούσαν στην περίπτωση ειδικά των προσεγγιστικών μεθόδων να δίνουν αποστάσεις μικρότερες από τις πραγματικές. Το ίδιο μπορεί να συμβεί και αν χρησιμοποιήσουμε την επιλογή για αποκοπή αντί για στρογγυλοποίηση που συνήθως αποτελεί μέθοδο επιτάχυνσης των μαθηματικών πράξεων. Για τους παραπάνω λόγους είναι πιο σημαντικό να δούμε κατά πόσο μια μέθοδος τείνει να δίνει μεγαλύτερη απόκλιση από τα αποτελέσματα που χρησιμοποιούμε ως σημείο αναφοράς, αντί να συγκρίνουμε τις απόλυτες τιμές αυτών των διαφορών.

Εδώ βέβαια πρέπει να τονιστέι ότι τόσο το FLANN οσο και το πακέτο ParallelNeighbors κάνουν χρήση της τετράγωνης Ευκλείδειας απόστασης 2.2 που κάνουν και οι μέθοδοί μας. Επομένως τα αποτελέσματα που παίρνουμε τόσο στις αποστάσεις όσο και στο ποιοι είναι οι k κοντινότεροι γείτονες των σημείων του Q είναι άμεσα συγκρίσιμα.

Αποτελέσματα

Κατά την διάρκεια των πειραμάτων προσπαθήσαμε να εξομοιώσουμε σενάρια εκτέλεσης των υπολογισμών σε συστοιχίες με διαφορετικό λόγο CPUs ανα GPU. Μιας και μέρος των εργασίας γίνεται στη CPU (ιδίως οι ταξινομήσεις των αποτελεσμάτων από τους υπολογισμούς των αποστάσεων), χρειάζεται να μελετηθεί η συμπεριφορά των αλγορίθμων υπό διαφορετικά σενάρια έτσι ώστε να βρεθεί πότε είναι καταλληλότερη η χρήση του καθένα τους. Για το λόγο αυτό χρησιμοποιούμε τα δύο dataset που αναφέραμε στην προηγούμενη ενότητα (HIGGS και συνθετικά δεδομένα, 28 και 2800 διαστάσεων αντίστοιχα) σε διαφορετικούς συνδυασμούς αριθμών GPUs, αλλά και αριθμό γειτόνων. Ως μία προσπάθεια να κρατήσουμε το χρόνο εκτέλεσης εντός ενός λογικού ορίου, σε κάθε περίπτωση κάναμε χρήση 64 CPUs, τις οποίες μοιράσαμε σε 4 διεργασίες των 16 CPUs η καθεμία. Αυτό προσομοιώνει σε ένα βαθμό τη χρήση 4 υπολογιστικών κόμβων που έχουν επεξερ-

γαστή με 16 νήματα ο καθένας. Αντίστοιχα δοκιμάσαμε και μια διαφορετική κατανομή με 8 διεργασίες των 8 CPUs.

Σε όλες τις παραπάνω περιπτώσεις, για τους υπολογισμούς των τυχαίων προβολών, επιλέξαμε r = 6 και P = 14 (6 διαστάσεις ως το υπερεπίπεδο προβολής, και 14 επαναλήψεις) για το σετ δεδομένων του HIGGS, ενώ για τα συνθετικά δεδομένα επιλέξαμε r = 56 και P = 14. Επίσης σε όλες τις περιπτώσεις επιλέξαμε τα αποτελέσματα να αποθηκευτούν στη μνήμη (και όχι σε δυαδικό αρχείο εξόδου), ενώ ορίσαμε τα cSize και qSize να έχουν τιμή 6400 καθώς είναι ένα ικανοποιητικό μέγεθος για να χωρίσουμε τα επιμέρους τμήματα του συνολικού φόρτου εργασίας που αναλαμβάνει η κάθε διεργασία. Με τον τρόπο αυτό επιτυγχάνουμε την καλύτερη καταμέριση των πόρων μας, μιας και τα αποτελέσματα των μεθόδων GPU χρειάζονται και πάλι να ταξινομηθούν από τη CPU, και επομένως το να στείλουμε στη GPU ακόμη μεγαλύτερη ποσότητα δεδομένων δεν φέρει καλύτερα αποτελέσματα. Ωστόσο θα παρουσιάσουμε και παραδείγματα όπου τα cSize και qSize έχουν τιμή 1600 έτσι ώστε να δούμε την διαφορά στην απόδοση των μεθόδων στην περίπτωση αυτή.

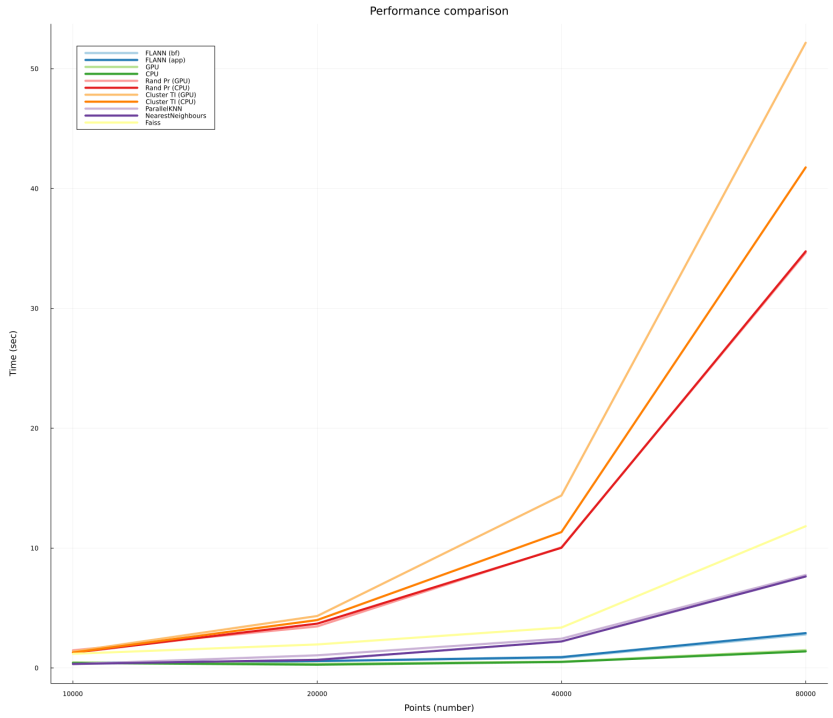
Έτσι συνολικά έχουμε τους εξής συνδυασμούς παραμέτρων:

- δεδομένα διαστάσεων d = 28 ή d = 2800
- διακριτά σετ Q και C ή ένα πρόβλημα allknn για το C
- αριθμός γειτόνων k = 5, k = 150 ή k = 500
- 2 ή 4 GPUs για τους υπολογισμούς
- μεγέθη cSize και qSize 1600 ή 6400
- 4 διεργασίες των 16 CPUs η 8 διεργασίες των 8 CPUs

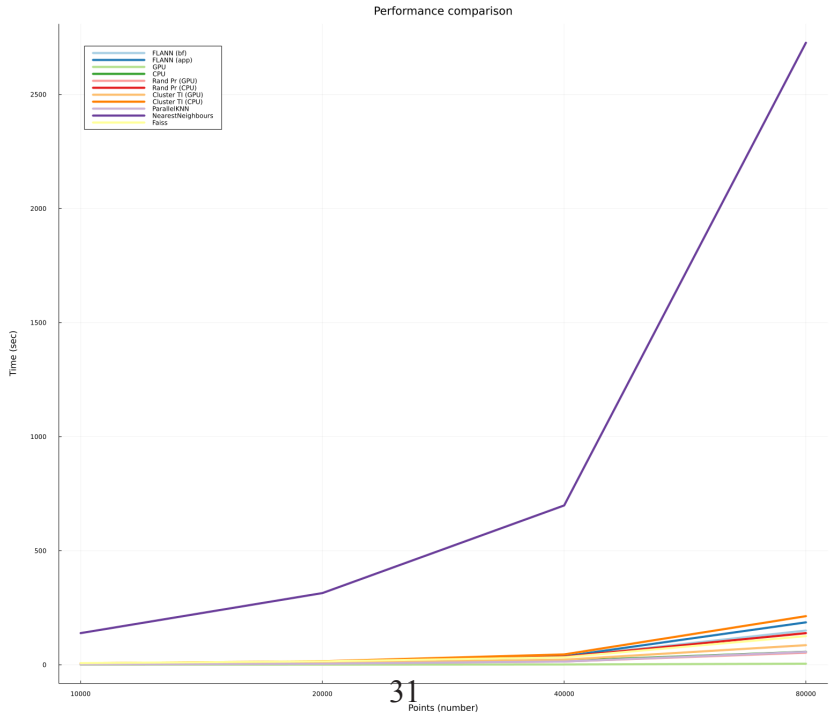
Ξεκινώντας θα παρουσιάσουμε μια σύγκριση των αποτελεσμάτων μεταξύ των δεδομένων του HIGGS και των συνθετικών δεδομένων για 2 GPUs και 4 διεργασίες των 16 CPUs η καθεμία. Για τα σύνολα δεδομένων παίρνουμε C = 80000 σημεία, ενώ το Q = 16000. Επαναλαμβάνουμε πέντε φορές τους υπολογισμούς για υποσύνολα των C και Q έτσι ώστε να δούμε την κλιμάκωση της απόδοσης για διαφορετικά μεγέθη προβλημάτων. Τα υποσύνολα αυτά έχουν τους εξής αριθμούς σημείων: 10000, 20000, 40000 και 80000 για το C. Αντιστοίχως για το Q έχουμε: 2000, 4000, 8000 και 16000 σημεία.

Παρατηρώντας τα αποτελέσματα είναι εμφανές ότι οι αλγόριθμοι που χρησιμοποιούν την GPU τείνουν να έχουν καλύτερη απόδοση από τις αντίστοιχες που κάνουν αποκλειστική χρήση της CPU. Η απόδοση βέβαια των διαφορετικών αλγορίθμων ποικίλει, ανάλογα με την πολυπλοκότητά τους. Χαρακτηριστικά η μέθοδος φιλτραρίσματος μέσω clustering παρουσιάζει σημαντικά χαμηλότερη απόδοση και στις δύο περιπτώσεις, μιας και η απαιτούμενη προεργασία φαίνεται να αναιρεί τη μείωση των απαιτούμενων πράξεων για τον υπολογισμό αποστάσεων. Επίσης η τυχαία προβολή δεν είναι ιδιαίτερα κατάλληλη για δεδομένα μικρού αριθμού διαστάσεων (όπως αυτά του HIGGS), μιας και η μείωση των διαστάσεων είναι μόλις της τάξης των 22 διαστάσεων (από τις αρχικές 28 σε 6) σε αντίθεση με την περίπτωση των συνθετικών δεδομένων που η διάστασή τους μειώνεται από 2800 σε μόλις 56. Παρά το γεγονός αυτό για να επιτύχουμε ακρίβεια πάνω από 90% χρειαζόμαστε 14 επαναλήψεις (ειδικά στην περίπτωση των συνθετικών δεδομένων) κάτι που μειώνει αισθητά την αποδοτικότητα της μεθόδου.

Σε πλήρη αντίθεση η GPU έκδοση του brute force αλγορίθμου μας φαίνεται να είναι η γρηγορότερη συνολικά, ειδικά στην περίπτωση των δεδομένων με 2800 διαστάσεις. Αντιθέτως στα δεδομένα του HIGGS, μερικές από τις CPU μεθόδους όπως η brute force μέθοδός μας, αλλά και το brute force FLANN (linear search) καταφέρνουν να φτάσουν



(α□) Δεδομένα HIGGS



(β□) Δεδομένα 2800 διαστάσεων

Σχήμα 4.2: Σύγκριση των χρόνων εκτέλεσης με χρήση 2 GPUs και 4 διεργασίες των 16 CPUs

σε απόδοση την GPU έκδοση του brute force αλγορίθμου μας. Αυτό κατά κύριο λόγο συμβαίνει λόγω του επιπλέον χρόνου που χρειάζεται για να σταλούν οι πίνακες των δεδομένων από και προς την GPU, καθώς και οι καθυστερήσεις κατά την κλήση των διεργασιών CUDA. Έτσι, αν και θεωρητικά η GPU έχει το υπολογιστικό πλεονέκτημα, η CPU μπορεί να την φτάσει σε απόδοση για προβλήματα μικρότερων διαστάσεων. Πρέπει ωστόσο να τονιστεί ότι στις δύο αυτές περιπτώσεις είχαμε μόλις 2 GPU για τις 4 διεργασίες, που κάθε μία τους είχε 16 CPUs η καθεμία. Κάτι τέτοιο επέτρεψε τόσο το brute force FLANN, όσο και τον δικό μας brute force αλγόριθμο σε CPU να φτάσουν σε απόδοση τον brute force αλγόριθμο σε GPU.

Ιδιαίτερο ενδιαφέρον έχει η συμπεριφορά του προσεγγιστικού Nearest Neighbors (kd trees) που στην περίπτωση των δεδομένων του HIGGS είναι σχεδόν παρόμοια σε απόδοση με τις άλλες brute force μεθόδους σε CPU, αλλά στα συνθετικά δεδομένα των 2800 διαστάσεων η απόδοσή της μειώνεται σε πολύ μεγάλο βαθμό. Μάλιστα στην τελευταία εκτέλεση με C = 80000 και Q = 16000 σημεία, το Nearest Neighbors (kd trees) γίνεται η πιο αργή μέθοδος από το σύνολο των υπολοίπων. Εδώ φαίνεται η επιδραση της "κατάρας της διαστατικότητας" σε μία μέθοδο που κανονικά θα έπρεπε να βελτιώνει σημαντικά την απόδοση σε σχέση με brute force μεθόδους, αλλά στην πράξη παρατηρούμε το αντίθετο. Αυτό κατά κύριο λόγο συμβαίνει γιατί οι μέθοδοι που ακολουθούνται για να μειωθούν οι απαιτούμενοι υπολογισμοί από αυτούς τους αλγορίθμους λειτουργούν καλύτερα για δεδομένα με μικρότερο αριθμό διαστάσεων.

Πρέπει βέβαια να τονιστεί εδώ ότι ειδικά στην περίπτωση των προσεγγιστικών μεθόδων όπως το Nearest Neighbors (kd trees) και η τυχαία προβολή (τόσο σε GPU όσο και σε CPU), μπορούν να γίνουν αρκετά πιο αποδοτικές αν μειώσουμε την ακρίβειά τους. Το ζητούμενο είναι κατά πόσο αποδεκτό είναι κάτι τέτοιο, και σε ποιο βαθμό είμαστε διατε-

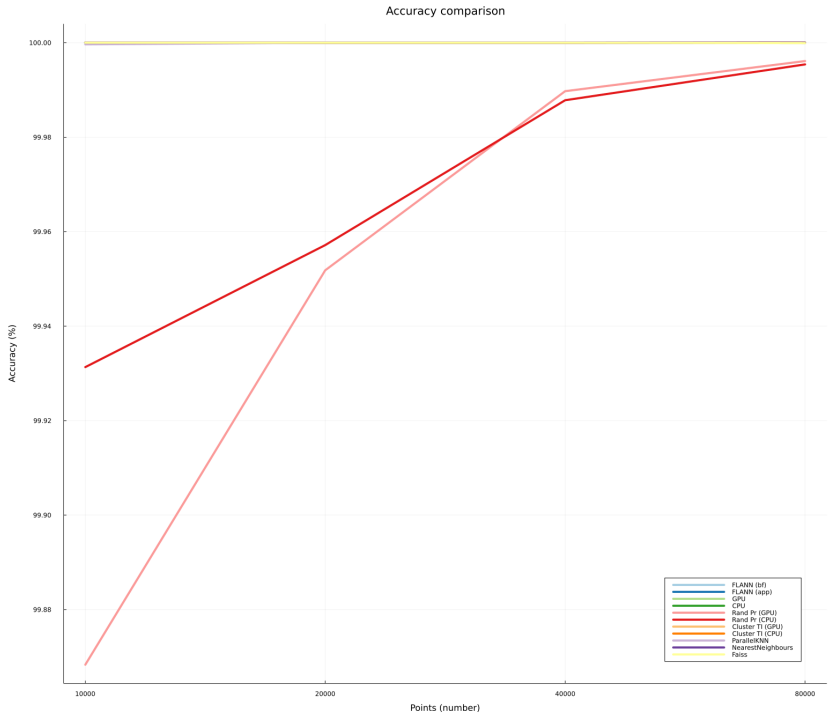
θειμένοι να μειώσουμε την ακρίβεια έτσι ώστε να επιταχύνουμε τις μεθόδους αυτές. Για την ανάγκη των πειραμάτων αυτών επιδιώξαμε να δούμε πόσο συγκρίσιμες είναι σε σχέση με τους υπόλοιπους αλγορίθμους όταν η ακρίβεια είναι αρκετά μεγάλη (κοντά στο 90%).

Παρακάτω μπορούμε να δούμε ότι η ακρίβεια και στις δύο περιπτώσεις είναι πρακτικά 100% ή τουλάχιστον πάρα πολύ κοντά στην τιμή αυτή για όλες τις μεθόδους εκτός από την τυχαία προβολή.

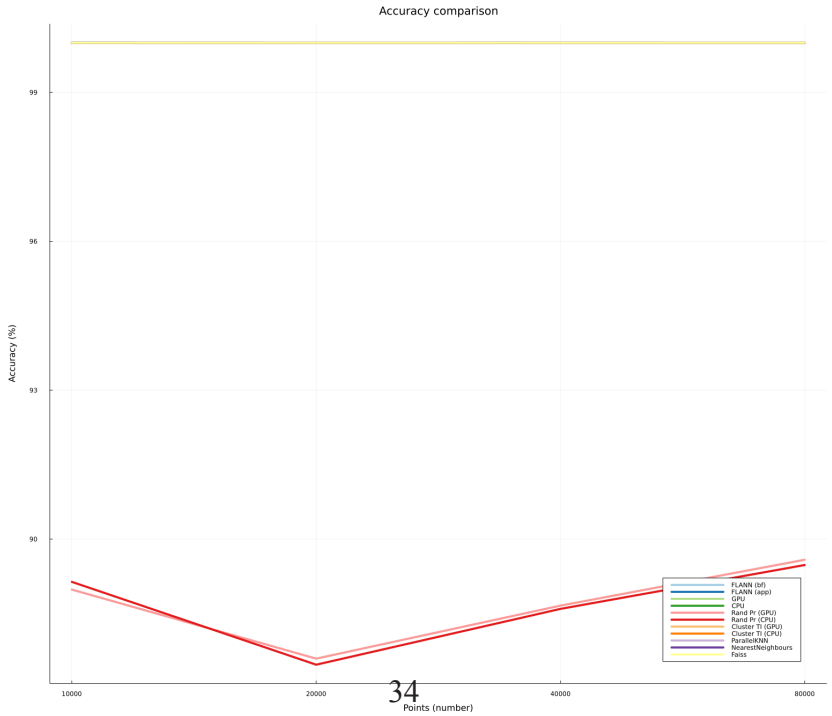
Αντίστοιχα έχουμε και την εκτίμηση του σφάλματος που η τυπική απόκλιση είναι σχεδόν 0 στις περισσότερες περιπτώσεις (εκτός από το Nearest Neighbors. Οι τιμές των ίδιων των σφαλμάτων είναι πρακτικά πολύ μικρές, και στην πλειονότητά τους αφορούν κυρίως σφάλματα στρογγυλοποιήσεων, ιδίως λόγω της χρήσης απλής ακρίβειας στους υπολογισμούς.

Ως επόμενο βήμα στα πειράματά μας, θα συγκρίνουμε τα αποτελέσματα από την εκτέλεση με την χρήση περισσότερων GPUs. Έτσι στην επόμενη εκτέλεση χρησιμοποιήσαμε 4 GPUs για ίσο αριθμό διεργασιών, που μας δίνει μια από τις καλύτερες κατανομές πόρων.

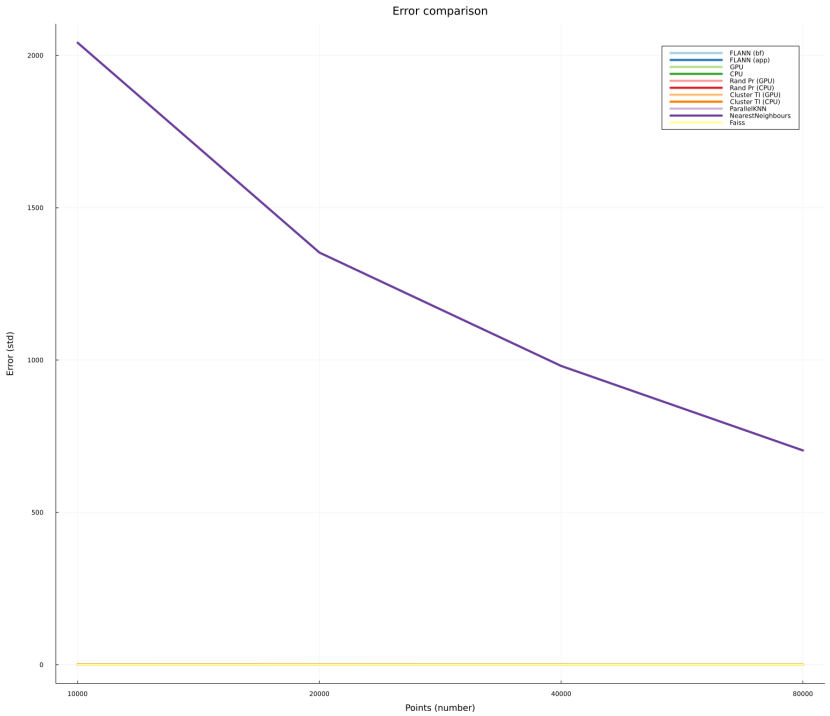
Σε πρώτη όψη τα αποτελέσματα των 4.6α□ και 4.6β□ είναι αρκετά όμοια με αυτά των 4.2α□ και 4.2β□. Ωστόσο μπορούμε να παρατηρήσουμε μια σχετικά μικρή αλλαγή ως προς την απόδοση των μεθόδων σε GPUs, που φαίνονται σχετικά πιο αποδοτικές σε σχέση με την περίπτωση που χρησιμοποιήσαμε μόνο 2 για τους υπολογισμούς μας. Αν εξαιρέσουμε τις μεθόδους που κάνουν χρήση clustering και την τυχαία προβολή, υπάρχει μια συνολική βελτίωση στην απόδωση των άλλων GPU μεθόδων. Αντίστοιχα οι υπόλοιπες μέθοδοι σε CPUs δεν φαίνεται να διαφοροποιούνται ιδιαίτερα, όπως είναι και το αναμενόμενο, καθώς ο αριθμός των CPUs παρέμεινε ο ίδιος. Δεδομένης της ικανοποιητικής υπολογιστικής ισχύος των A100 καρτών της συστοιχίας, θα μπορούσαμε να είχαμε και καλύτερα αποτελέσματα αν υπήρχε η δυνατότητα για χρήση ακόμη περισσότερων CPUs,



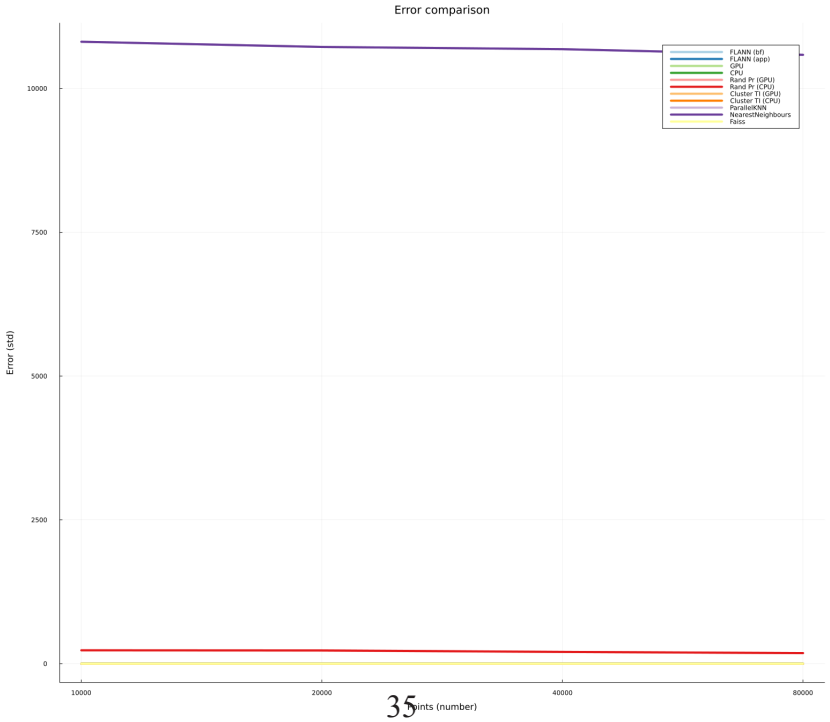
(α□) Ακρίβεια δεδομένων HIGGS



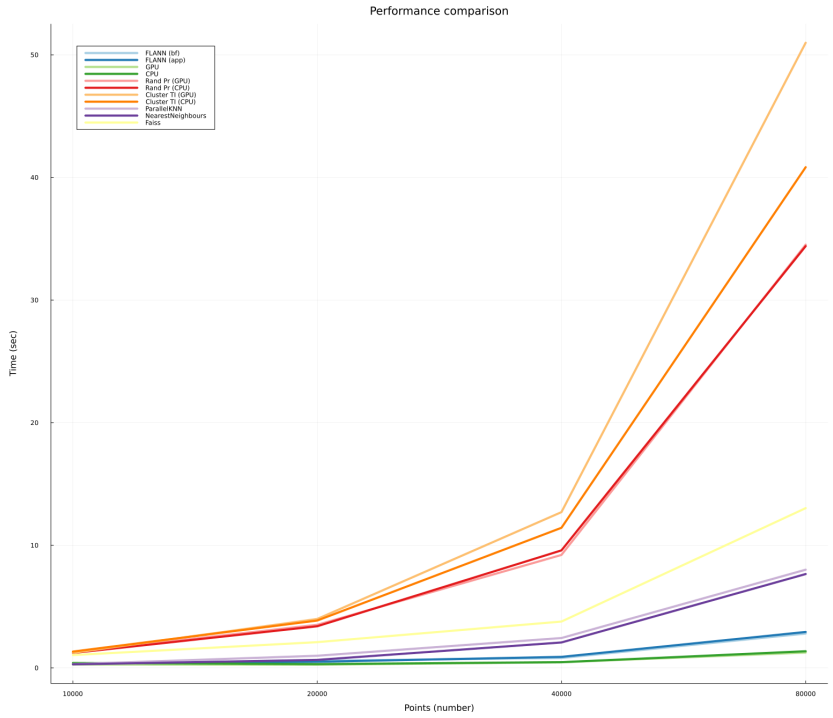
(β□) Ακρίβεια συνθετικών δεδομένων



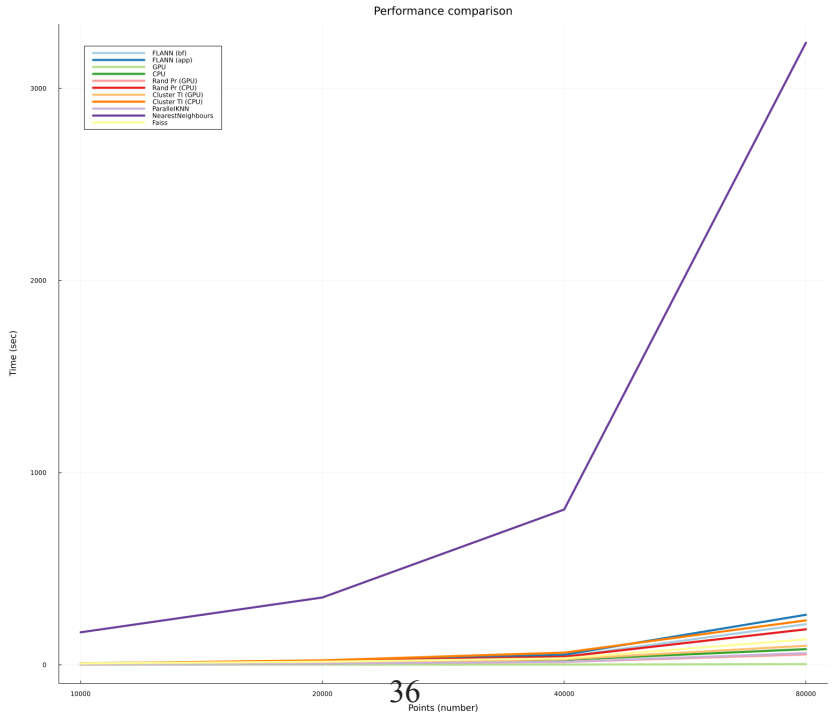
Σχήμα 4.4: std σφάλματος δεδομένων HIGGS



Σχήμα 4.5: std σφάλματος συνθετικών δεδομένων



(α□) Δεδομένα HIGGS



(β□) Συνθετικά δεδομένα

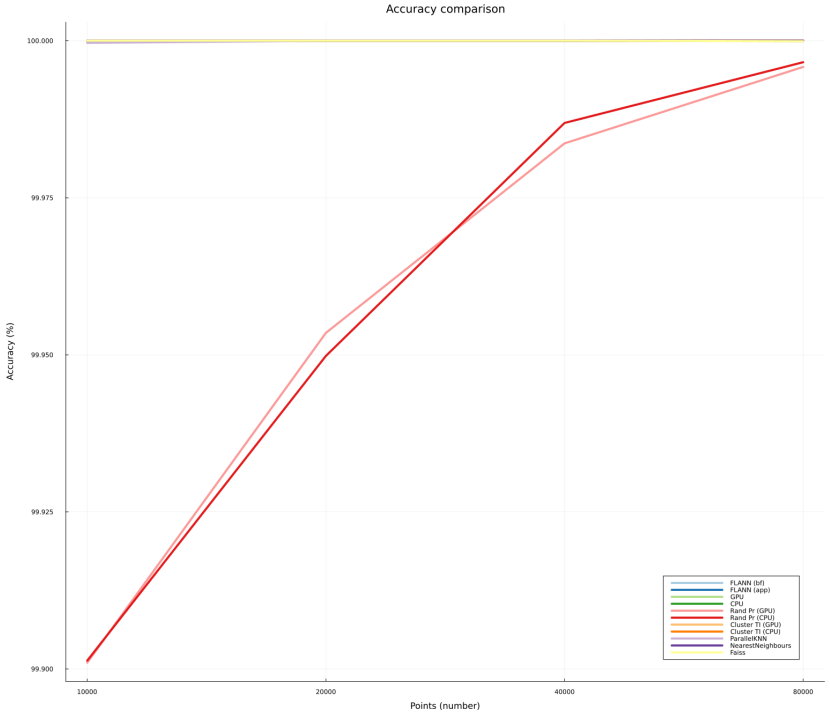
Σχήμα 4.6: Σύγκριση των χρόνων εκτέλεσης με 4 GPUs

έτσι ώστε οι ταξινομήσεις των αποτελεσμάτων να πραγματοποιούνται γρηγορότερα. Με αυτό το δεδομένο, δεν βλέπουμε μία μείωση του χρόνου εκτέλεσης των GPU μεθόδων στη μέση από τον διπλασιασμό των GPUs, αν και πάλι υπάρχει μία παρατηρήσημη βελτίωση του χρόνου εκτέλεσης.

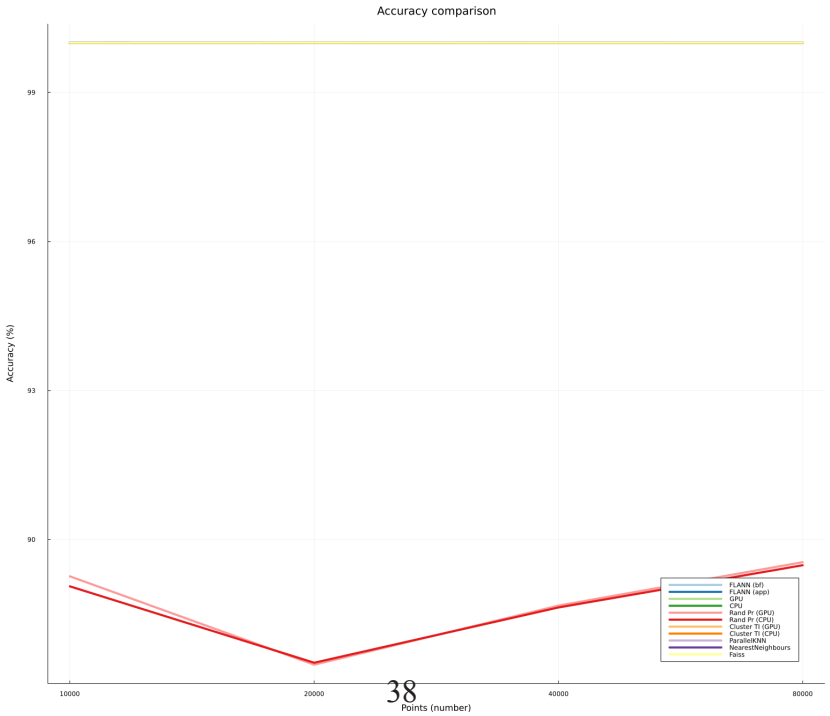
Εδώ επίσης πρέπει να αναφέρουμε ότι ο αριθμός των γειτόνων παρέμεινε σε 150 όπως και στο προηγούμενο σετ πειραμάτων. Επίσης, και στις δύο περιπτώσεις των 4.6α και 4.6β η ακρίβεια και το σφάλμα παραμένουν στα ίδια γενικά επίπεδα 4.7 με πριν (σταθερά άνω του 90 %, εκτός από την τυχαία προβολή).

Προσπαθώντας να μελετήσουμε περαιτέρω την συμπεριφορά των αλγορίθμων σε διαφορετικά μεγέθη προβλημάτων, επιχειρήσαμε να τρέξουμε μια allKNN εκδοχή των πειραμάτων μας και να την συγρίνουμε με την αντίστοιχη απλή έκδοση για το ίδιο σετ δεδομένων. Με τον τρόπο αυτό έχουμε πολύ μεγαλύτερο φόρτο εργασίας στην allKNN έκδοση μιας και στην τελευταία επανάληψη του βρόχου εκτέλεσης των πειραμάτων έχουμε ένα $80000 * 80000$ πίνακα αποστάσεων, τον οποίο βέβαια υπολογίζουμε τμηματικά, αλλα και πάλι αποτελεί μια πρόκληση ειδικά όταν ο αριθμός των διαστάσεων των δεδομένων αυξάνεται. Έτσι λοιπόν, ακολουθώντας τα συμπεράσματα από τα προηγούμενα πειράματα, χρησιμοποιήσαμε τις ίδιες παραμέτρους για την τυχαία προβολή και τα μεγέθη qSize και cSize, καθώς και 4 GPUs. Επίσης επικεντρωθήκαμε στο σετ δεδομένων των 2800 διαστάσεων, καθώς αποτελεί υπολογιστικά μια σημαντικότερη πρόκληση.

Συγκρίνοντας το γράφημα 4.8β με το 4.6β μπορούμε να δούμε ότι πρακτικά είναι παρόμοια ως προς την απόδοση των αλγορίθμων, παρά τις διαφορετικές απόλυτες τιμές του χρόνου εκτέλεσης της κάθε μεθόδου. Δεδομένου ότι μεταξύ των δύο αυτών εκτελέσεων αυξήσαμε τον συνολικό αριθμό των σημείων κατά 5 φορές, μπορούμε να παρατηρήσουμε ότι οι χρόνοι εκτέλεσης των μεθόδων στο 4.8β σε γενικές γραμμές κυμαίνονται

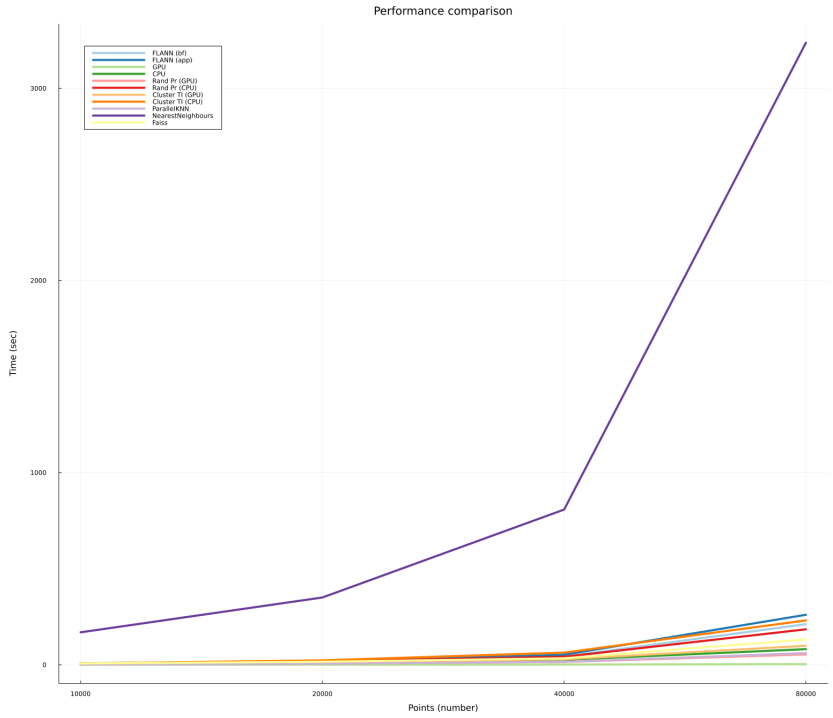


(α□) Δεδομένα HIGGS

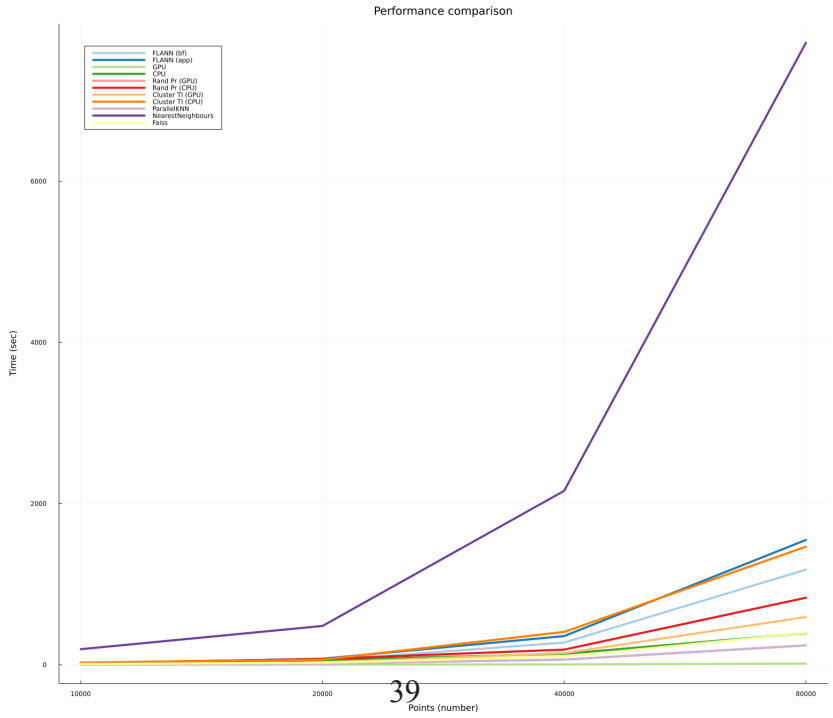


(β□) Συνθετικά δεδομένα

Σχήμα 4.7: Σύγκριση της ακρίβειας για σενάριο με 4 GPUs

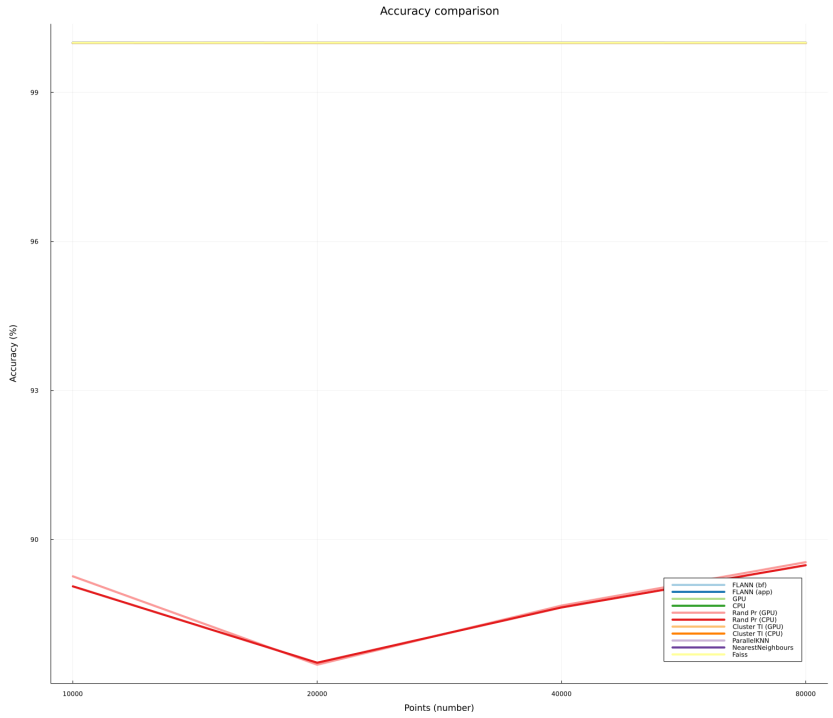


(α□) C = 80000 και Q = 16000

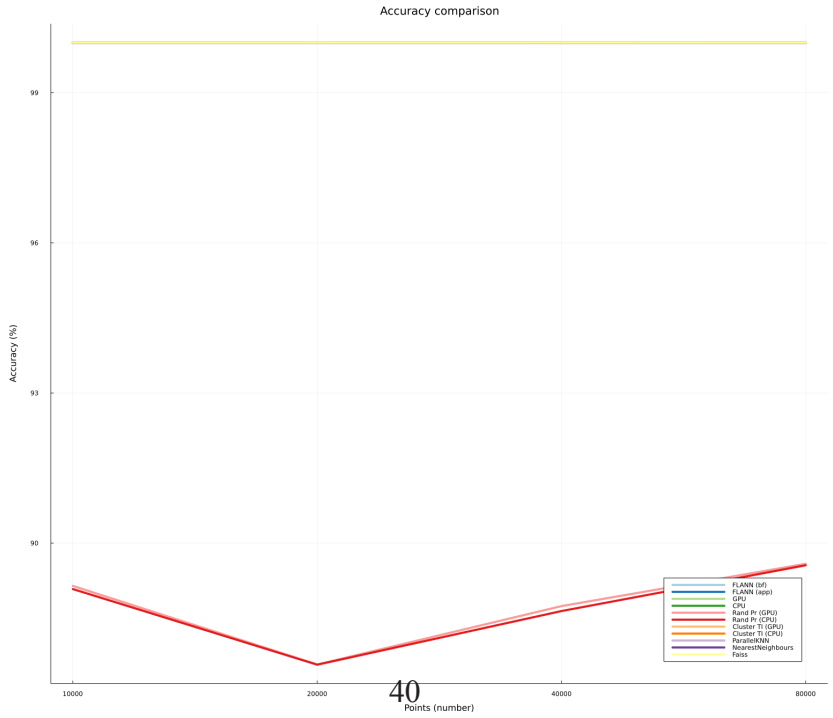


(β□) C = Q= 80000

Σχήμα 4.8: Σύγκριση των χρόνων εκτέλεσης για δεδομένα 2800 διαστάσεων



(α□) C = 80000 και Q = 16000



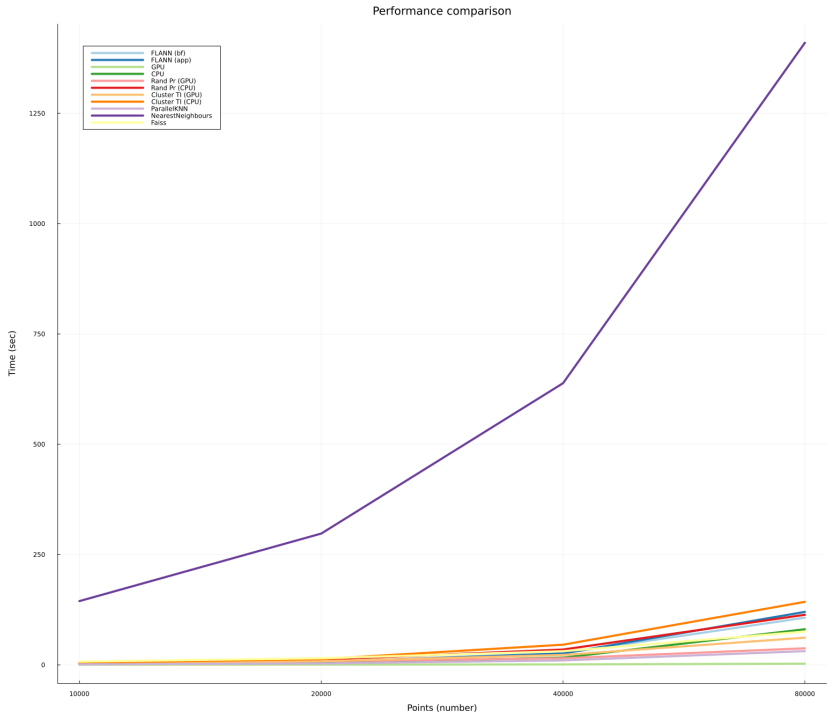
(β□) C = Q= 80000

Σχήμα 4.9: Σύγκριση της ακρίβειας για δεδομένα 2800 διαστάσεων

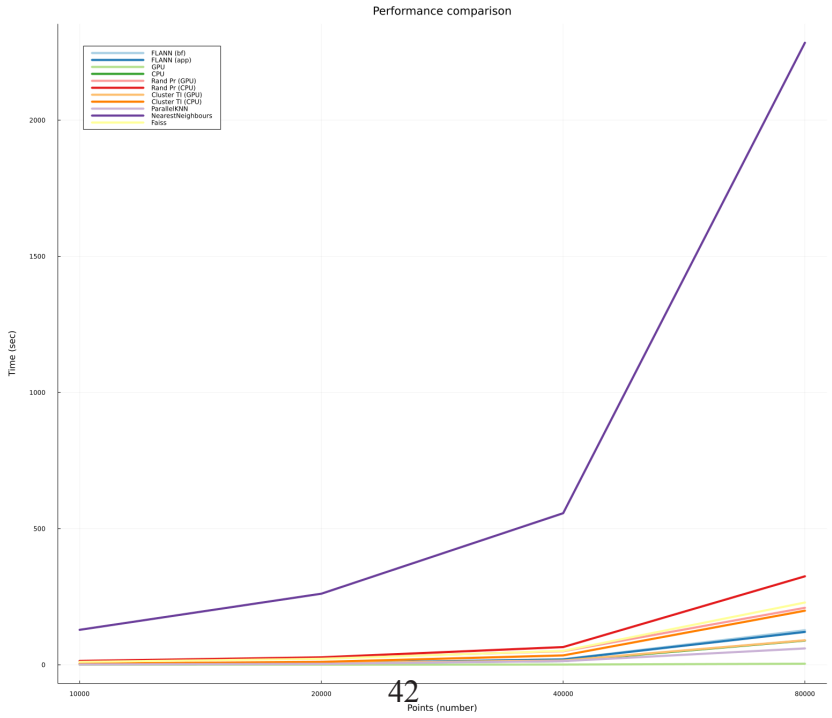
μεταξύ 4 με 5 φορές τις αντίστοιχες στο 4.6β□, κάτι που υποδηλώνει ότι ο χρόνος εκτέλεσης αυξάνεται γραμμικά με το μέγεθος του προβλήματος. Φυσικά προαπαιτούμενο για να συμβεί αυτό είναι να κρατήσουμε όλες τις υπόλοιπες παραμέτρους της εκτέλεσης των πειραμάτων ίδιες μεταξύ των δύο αυτών εκτελέσεων, γιατί όπως θα δούμε και παρακάτω, κάτι τέτοιο μπορεί να επιφέρει σημαντικές αλλαγές στα αποτελέσματα. Το σημαντικότερο από όλα είναι ότι η συμπεριφορά των μεθόδων για δεδομένες παραμέτρους εκτέλεσης παραμένει γενικά σταθερή (εκτός από ακραίες περιπτώσεις όπου πλησιάζουμε τα όρια των υπολογιστικών κόμβων μας). Έτσι είναι δυνατόν να παρέχουμε κάποιες εκτιμήσεις για την απόδοση των μεθόδων μας, εκτελώντας πρώτα μικρότερες αποδοχές των προβλημάτων μας. Μάλιστα μιας και οι χρόνοι εκτέλεσης για μικρότερα προβλήματα είναι αντιστοίχως μικρότεροι, μπορούμε να εκμεταλλευτούμε το γεγονός αυτό για να πραγματοποιήσουμε διερευνητικές δοκιμές σε ένα υποσύνολο των δεδομένων μας, έτσι ώστε να βρούμε τις κατάλληλες ρυθμίσεις για να βελτιστοποιήσουμε τον χρόνο εκτέλεσης.

Ως επόμενο στάδιο στα πειράματά μας δοκιμάσαμε μια διαφορετική κατανομή των CPUs οργανώνοντάς τες σε 8 διεργασίες των 8 CPUs η κάθε μία. Επίσης δοκιμάσαμε δύο διαφορετικές τιμές τιμές για τα qSize και cSize, χρησιμοποιώντας τόσο την συνήθη τιμή 6400 από όλα τα παραπάνω παραδείγματα, αλλά και την μικρότερη τιμή 1600 που θεωρητικά θα πρέπει να δίνει ένα μικρό πλεονέκτημα στις μεθόδους που κάνουν χρήση της CPU. Και στις δύο περιπτώσεις εστιάζουμε την προσοχή μας στα συνθετικά δεδομένα, καθώς αποτελούν ένα πιο σύνθετο υπολογιστικά πρόβλημα, και ως αποτέλεσμα περιμένουμε μεγαλύτερη διαφοροποίηση στους χρόνους εκτέλεσης των πειραμάτων. Σταθερός επίσης παραμένει ο αριθμός των ζητούμενων γειτόνων (k = 150) αλλά και οι υπόλοιποι παράμετροι εκτέλεσης.

Αρχικά αναφέρουμε ότι και στις δύο περιπτώσεις η ακρίβεια των αποτελεσμάτων 4.11

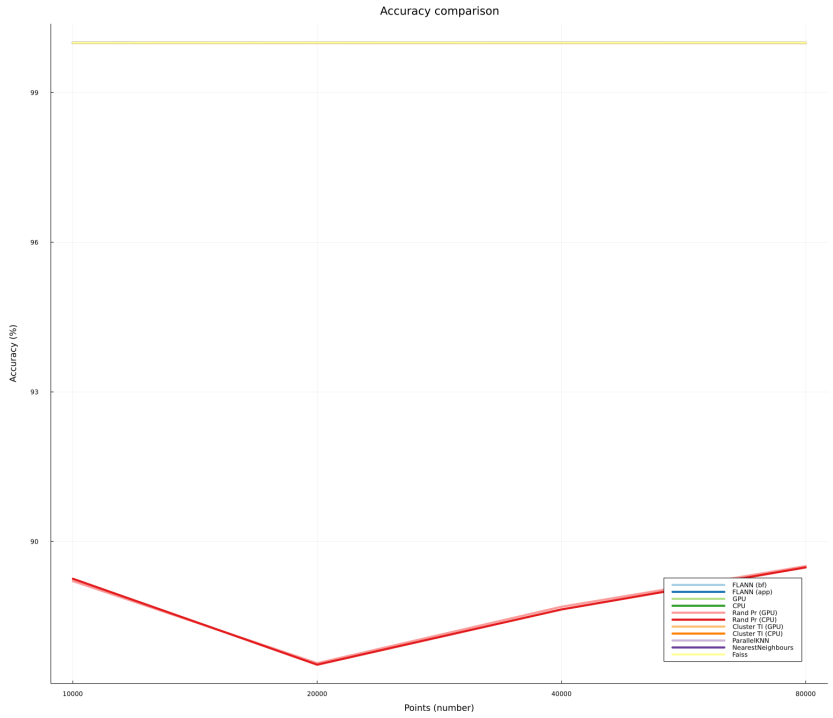


(α□) cSize = qSize = 6400

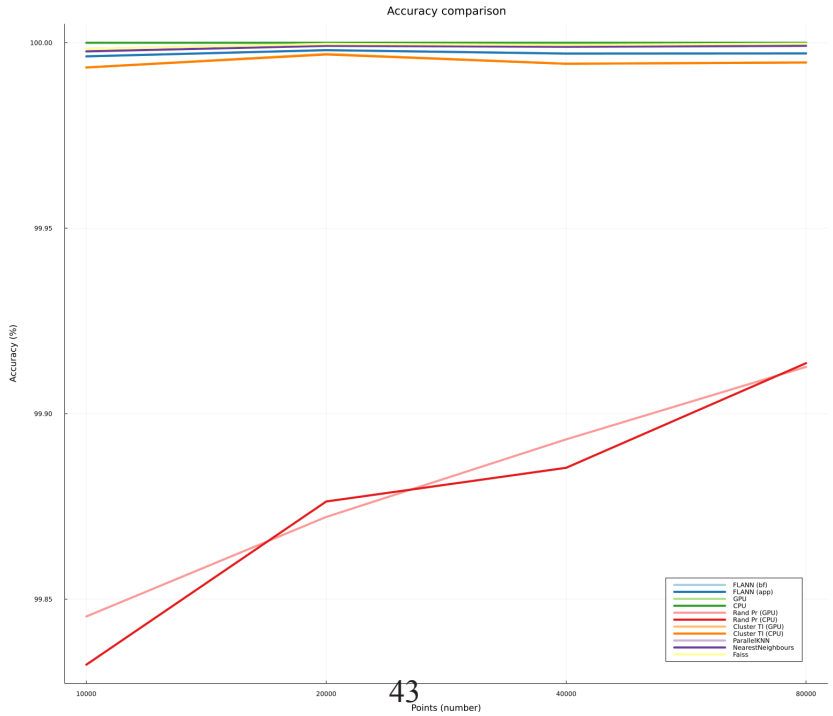


(β□) cSize = qSize = 1600

Σχήμα 4.10: Σύγκριση των χρόνων εκτέλεσης για 8 διεργασίες με 8 CPUs η κάθε μία για τα συνθετικά δεδομένα 2800 διαστάσεων



(α□) cSize = qSize = 6400



(β□) cSize = qSize = 1600

Σχήμα 4.11: Σύγκριση της ακρίβειας για 8 διεργασίες με 8 CPUs η κάθε μία για τα συνθετικά δεδομένα 2800 διαστάσεων

παραμένει πάνω από 90 % εκτός από την τυχαία προβολή, ενώ για τις υπόλοιπες μεθόδους κοντά στο 100 % (τουλάχιστον 99 %. Ωστόσο στην περίπτωση όπου τα qSize και cSize ορίστικαν ως 1600, παρατηρούμε μια σημαντική διαφοροποίηση στην ακρίβεια της τυχαίας προβολής που με τις ίδιες ρυθμίσεις (r = 56 και P = 14) φτάνει το 99 %. Αυτό πρακτικά σημαίνει ότι όσο μικραίνουμε τα κομμάτια που υποδιαιρούμε τα δεδομένα μας, τόσο ευκολότερο γίνεται να πετύχουμε υψηλότερη ακρίβεια στην τυχαία προβολή με τις ίδιες ρυθμίσεις. Οι υπόλοιπες μέθοδοι δεν φαίνεται να διαφοροποιούνται ως προς την ακρίβειά τους σε σχέση με τα μεγέθη qSize και cSize, τουλάχιστον στο παράδειγμα αυτό.

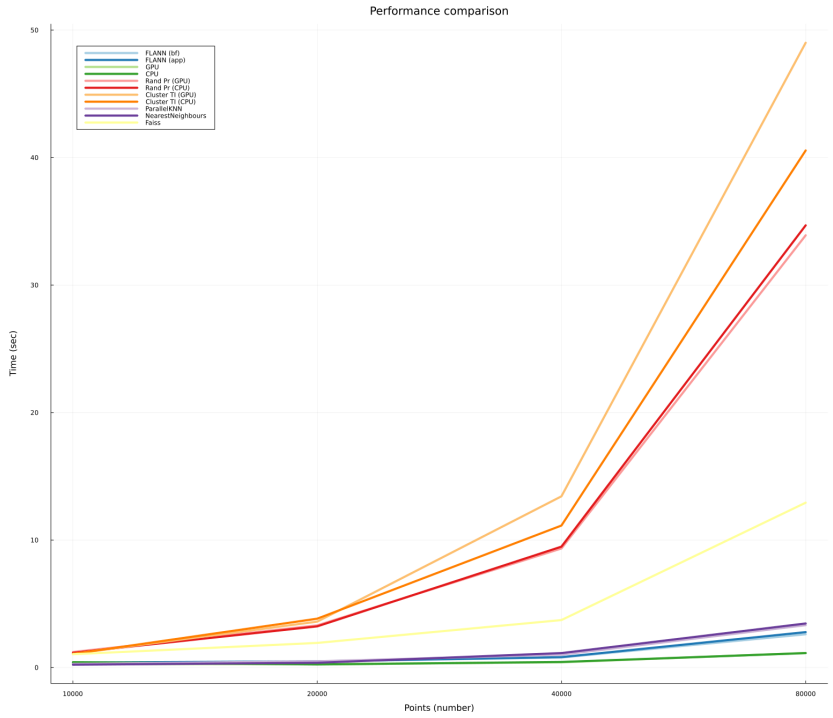
Παρατηρώντας τώρα τα αποτελέσματα φαίνεται ότι οι χρόνοι εκτέλεσης σχεδόν διπλασιάστηκαν στην περίπτωση που επιλέξαμε μεγέθη qSize και cSize ως 1600. Η μεγαλύτερη διαφορά φαίνεται να είναι στην περίπτωση των μεθόδων GPU που αρκετές από αυτές φαίνεται να χάνουν την αποδοτικότητά τους σε σχέση με τις μεθόδους CPU. Χαρακτηριστικά το Faiss φαίνεται να επηρεάζεται περισσότερο από τις υπόλοιπες μεθόδους GPU όταν μικραίνουμε τα qSize και cSize. Βέβαια μια τέτοια επιλογή μικρότερου μεγέθους qSize και cSize μικραίνει αντίστοιχα και τα μεγέθη των πινάκων που χρησιμοποιούμε για τους υπολογισμούς, κάτι που μας βοηθά να κρατήσουμε την χρήση τόσο της κεντρικής RAM αλλά και της VRAM σε επίπεδα που είναι εντός των προδιαγραφών του υλικού μας. Δεδομένου των πόρων της υπολογιστικής συστοιχίας Αριστοτέλης, κάτι τέτοιο δεν αποφέρει κάποιο πρακτικό πλεονέκτημα, αλλά σε περίπτωση που έχουμε περιορισμούς στους διαθέσιμους πόρους, ιδίως αν εργαζόμαστε με δεδομένα πολύ μεγάλου αριθμού διαστάσεων, τότε κάτι τέτοιο είναι απαραίτητο.

Κάτι άλλο το οποίο μπορούμε να κάνουμε με την περίπτωση του πειράματος 4.10α□ είναι να το συγκρίνουμε με το 4.6β□ καθώς και τα δύο έχουν σχεδόν όλες τις παραμέτρους τους ίδιες, εκτός από την κατανομή των CPUs. Ενώ στο 4.6β□ έχουμε 4 διεργασίες

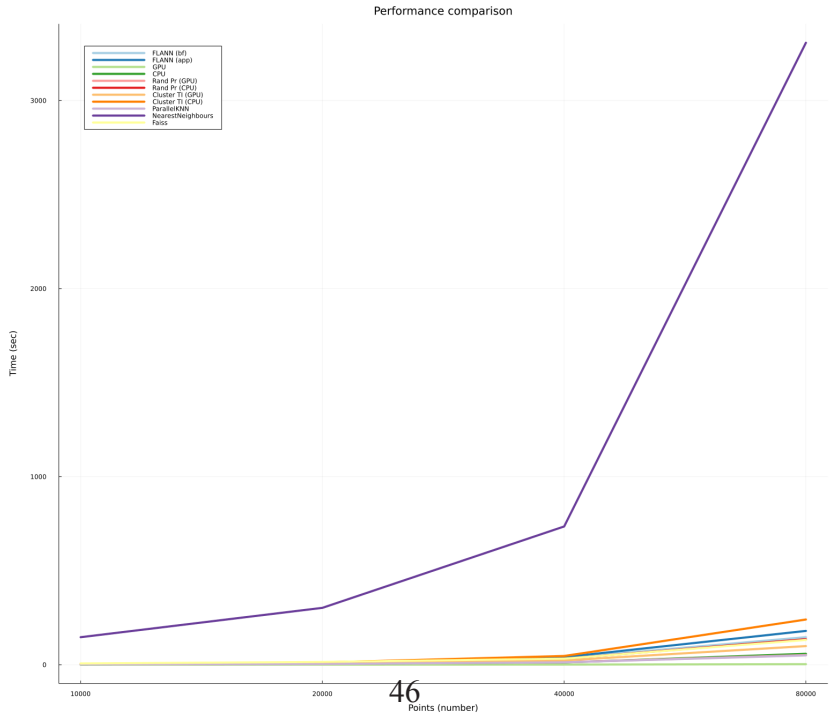
των 16 CPUs η κάθε μία, στο 4.10α□ έχουμε 8 διεργασίες των 8 CPUs. Και στις δύο περιπτώσεις έχουμε το ίδιο σύνολο CPUs, αλλά και κάνουμε χρήση 4 GPUs, με το πρώτο παράδειγμα να έχει ένα λόγο διεργασιών προς GPUs 1:1 ενώ στο δεύτερο ο λόγος γίνεται 2:1. Έτσι ενώ συνολικά έχουμε τους ίδιους διαθέσιμους πόρους, στην περίπτωση που χρησιμοποιούμε 8 διεργασίες των 8 CPUs οι χρόνοι εκτέλεσης βελτιώνονται σε όλες τις μεθόδους. Αυτό μπορούμε να το εξηγήσουμε με το γεγονός ότι ενώ ο μεγαλύτερος παραλληλισμός ανά διεργασία στην περίπτωση του 4.6β□ διευκολύνει τους υπολογισμούς στην περίπτωση των CPU μεθόδων, αλλά και τις ταξινομήσεις των αποτελεσμάτων στις περισσότερες μεθόδους, υπάρχουν ακόμη κομμάτια του κώδικα που είναι γραμμικά. Για το λόγο αυτό, χρησιμοποιώντας περισσότερες διεργασίες μπορούμε να αυξήσουμε την απόδοση, καθώς εξακολουθούμε να έχουμε ένα ικανοποιητικό αριθμό CPUs ανά διεργασία. Αντίστοιχα, οι διεργασίες που τρέχουν μεθόδους GPU έχουν τη δυνατότητα να εκτελούν περισσότερες κλήσεις στη GPU ταυτόχρονα, και δεδομένου των προδιαγραφών των A100 της συστοιχίας, δεν δημιουργούνται προβλήματα στην απόδοση. Κάθε άλλο, μας δίνεται η δυνατότητα να κάνουμε καλύτερη χρήση των πόρων μας, με το να κρατάμε τόσο τις CPUs όσο και τις GPUs απασχολημένες για περισσότερο ποσοστό του χρόνου εκτέλεσης.

Στο επόμενο πείραμα θα κάνουμε μια σύγκριση των μεθόδων για μικρότερο αριθμό ζητούμενων γειτόνων k που στην περίπτωση αυτή θα περιοριστούν σε μόλις 5 (σε αντίθεση με 150 που ήταν σε όλα τα προηγούμενα παραδείγματα). Για να κρατήσουμε τις περισσότερες παραμέτρους κοινές με τα προηγούμενα πειράματα, επιλέγουμε τις ίδιες τιμές σε όλες τις παραμέτρους εκτέλεσης με τα παραδείγματα των 4.6α□ και 4.6β□ με μοναδική διαφορά την τιμή του k.

Συγκρίνοντας τα 4.6α□ και 4.12α□ παρατηρούμε ότι δεν υπάρχει καμία ουσιαστική διαφορά στις περισσότερες μεθόδους εκτός από αυτή του φιλτραρίσματος με clustering

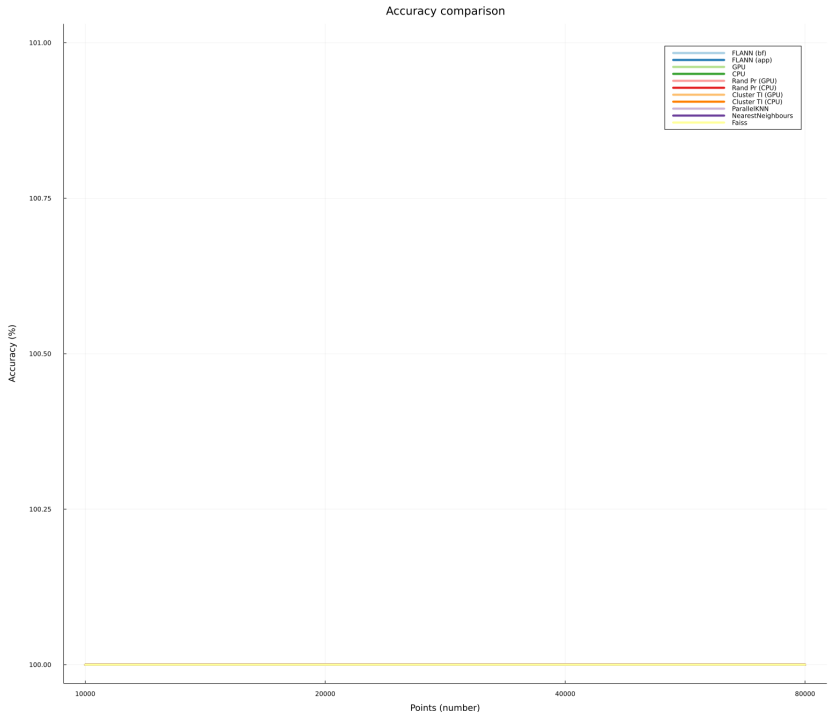


(α□) Δεδομένα HIGGS

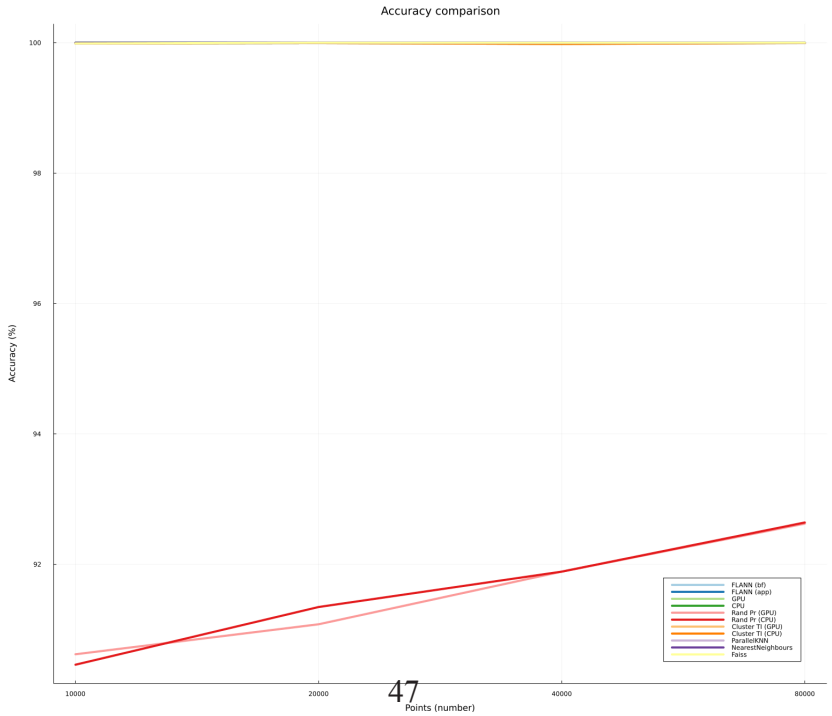


(β□) Συνθετικά δεδομένα

Σχήμα 4.12: Σύγκριση των χρόνων εκτέλεσης για k = 5



(α□) Δεδομένα HIGGS



(β□) Συνθετικά δεδομένα

Σχήμα 4.13: Σύγκριση της ακρίβειας για σενάριο εκτέλεσης με k = 5

(και στις 2 εκδόσεις της). Δεδομένου ότι το φιλτράρισμα λαμβάνει υπόψη τον αριθμό των κοντινότερων γειτόνων k κατά τον υπολογισμό των Upper Bounds, μειώνοντας το k μπορεί να αποκλείσει περισσότερα clusters τα οποία είναι πλέον πολύ μακριά από τα σημεία του Q για να έχουν τους κοντινότερους γείτονες που αναζητάμε. Από τις υπόλοιπες μεθόδους βλέπουμε ότι τόσο η Nearest Neighbors όσο και η ParallelNeighbours φαίνονται να γίνονται πιο γρήγορες για μικρότερα k. Αντιθέτως όλες οι άλλες μέθοδοι δεν επηρεάζονται τόσο σημαντικά, μιας και η μεγαλύτερη αλλαγή έχει να κάνει με τον χρόνο ταξινόμησης των αποτελεσμάτων (καθώς η partialsortperm σταματά μετά τα k πρώτα αποτελέσματα) που είναι συγκριτικά πολύ μικρότερος από το χρόνο εκτέλεσης του υπολογισμού των αποστάσεων.

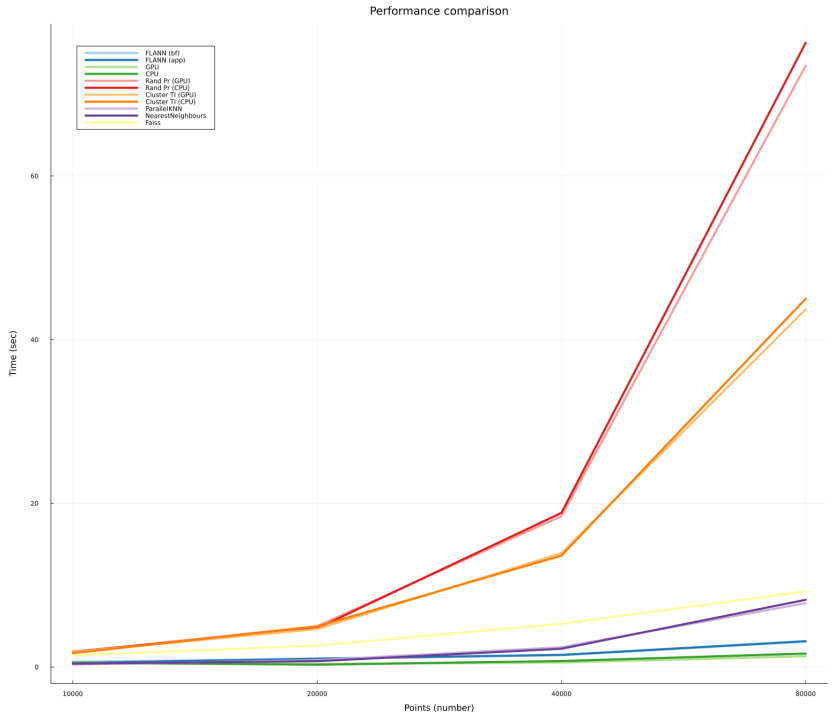
Αντιθέτως στην περίπτωση των 4.6β□ και 4.12β□ που αφορά τα δεδομένα των 2800 διαστάσεων, παρατηρούμε μια πιο μικρή διαφορά στα αποτελέσματα. Αυτό έχει να κάνει με το γεγονός ότι πολύ μεγαλύτερη σημασία έχει ο αριθμός των διαστάσεων σε σχέση με τον αριθμό των κοντινότερων γειτόνων k. Ακόμη και η περίπτωση του φιλτραρίσματος μέσω clustering που θα έπρεπε να μας δώσει σημαντικά πλεονεκτήματα στην περίπτωση αυτή, παρουσιάζει σημαντικές καθυστερήσεις κατά την δημιουργία των clusters λόγω του υψηλού αριθμού των διαστάσεων των δεδομένων. Η μέθοδος με τις τυχαίες παρουσιάζει μια σχετική βελτίωση καθώς μειώνεται ο αριθμός των απαιτούμενων γειτόνων που πρέπει να βρούμε είναι μικρότερος, αλλά και πάλι δεν διαφοροποιείται σε τόσο μεγάλο βαθμό, μιας και ο αριθμός των διαστάσεων παίζει πολύ σημαντικότερο ρόλο από το k. Η Nearest Neighbors εξακολουθεί να είναι η πιο αργή μέθοδος με διαφορά για τα δεδομένα 2800 διαστάσεων, χωρίς το k να επηρρεάζει την απόδοσή της.

Και στις δύο παραπάνω περιπτώσεις η μείωση του αριθμού των κοντινότερων γειτόνων k δεν διαφοροποίησε σε σημαντικό βαθμό τα αποτελέσματα, τουλάχιστον όχι σε σύγκριση

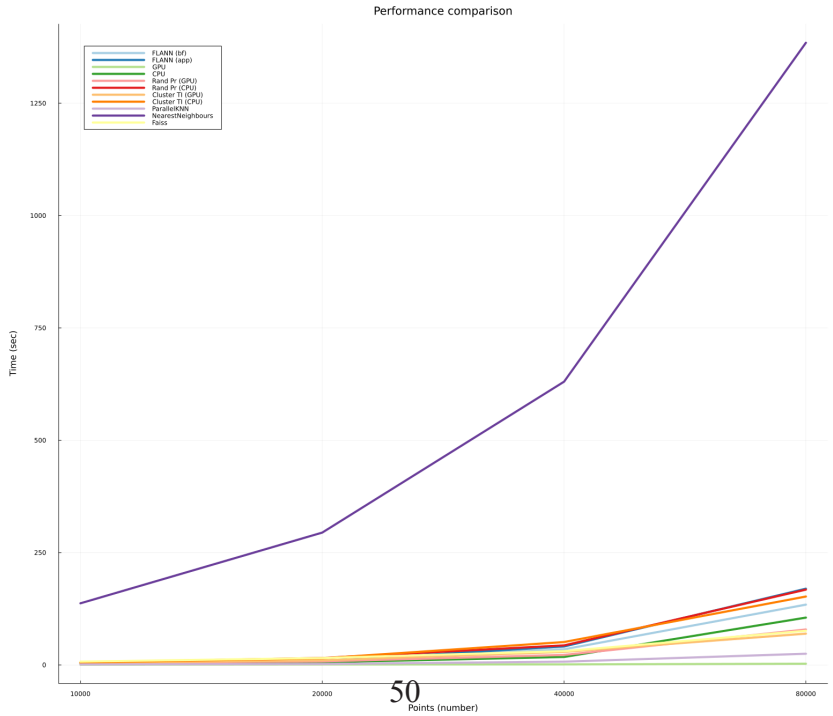
με άλλες παραμέτρους όπως τα μεγέθη των qSize και cSize, αλλά και η κατανομή των διαθέσιμων CPUs. Βέβαια θέλοντας να καλύψουμε και την περίπτωση όπου το k αποκτά μια αρκετά μεγαλύτερη τιμή, επαναλάβαμε το πείραμα για k = 500. Επίσης βάσει των παραπάνω παρατηρήσεων, χρησιμοποιήσαμε την διαφορετική κατανομή πόρων με 8 διεργασίες των 8 CPUs η κάθε μία, καθώς παρουσιάζει καλύτερα αποτελέσματα.

Βλέποντας τόσο τα αποτελέσματα από το 4.15α□ και 4.15β□ παρατηρούμε ότι το Flann και στις δύο εκδοχές του (brute force και approximated) είναι σημαντικά λιγότερο ακριβές. Μάλιστα, η ακρίβεια πέφτει κάτω από το 20 % στην περίπτωση που C = 80000 σημεία, και Q = 16000 σημεία. Έχοντας ως δεδομένο ότι σε κανένα από τα παραπάνω πειράματα δεν παρατηρείται αντίστοιχη συμπεριφορά, είναι προφανές ότι η αύξηση του k σε 500 από την προηγούμενη πιο συνηθισμένη τιμή (150) ήταν αυτό που οδήγησε στην μείωση της ακρίβειας σε τέτοιο βαθμό. Σε ότι αφορά τους χρόνους εκτέλεσης, συγκρίνοντας τα αποτελέσματα με αυτά από προηγούμενα πειράματα, ιδίως στην περίπτωση του 4.10α□ με το 4.14β□ παρατηρούμε ότι οι χρόνοι είναι γενικά παρόμοιοι, αν και οι GPU μέθοδοι παρουσιάζουν μια σχετικά καλύτερη απόδοση. Ωστόσο δεν μπορούμε να πούμε ότι η διαφοροποίηση στην απόδοση είναι τέτοια που να ξεπερνά την επίδραση του αριθμού των διαστάσεων, που συνεχίζει να είναι ο καθοριστικός παράγοντας σε ότι αφορά τους χρόνους εκτέλεσης.

Ως ένα ακόμη μέτρο της απόδοσης των αλγορίθμων μπορούμε να εκφράσουμε τα αποτελέσματα ως queries per second σε σχέση με το recall. Και λαμβάνοντας υπόψη τα παραπάνω αποτελέσματα, επικεντρωθήκαμε στην περίπτωση των 2800 διαστάσεων τόσο για k = 150 όσο και για k = 500. Και στις δύο περιπτώσεις χρησιμοποιήσαμε 8 διεργασίες των 8 CPUs, ενώ ο αριθμός των GPUs παρέμεινε ίδιος με τα προηγούμενα παραδείγματα (4 GPUs), και επίσης το ίδιο ισχύει και με τα qSize και cSize που παρέμειναν 6400.

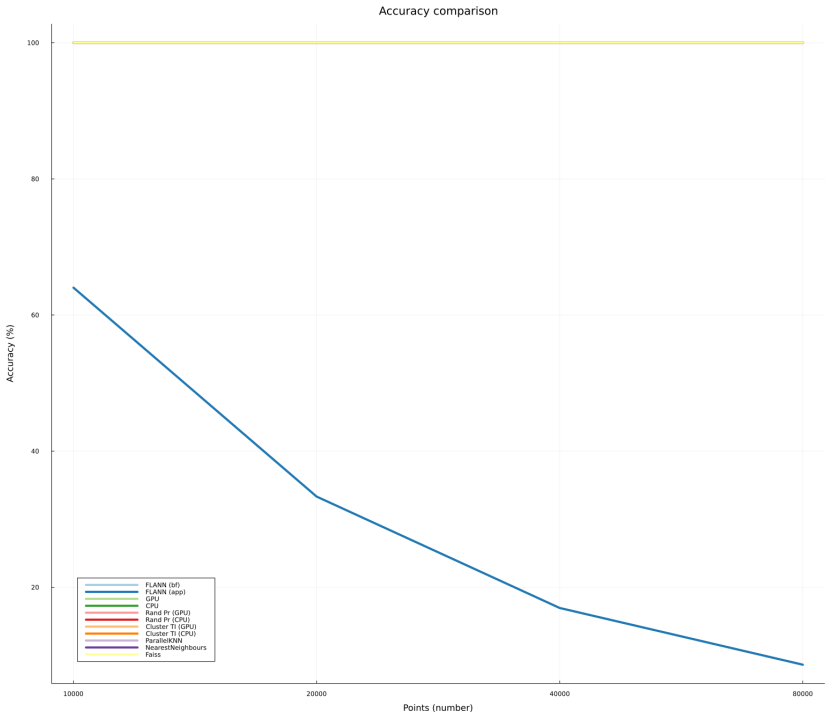


(α□) Δεδομένα HIGGS

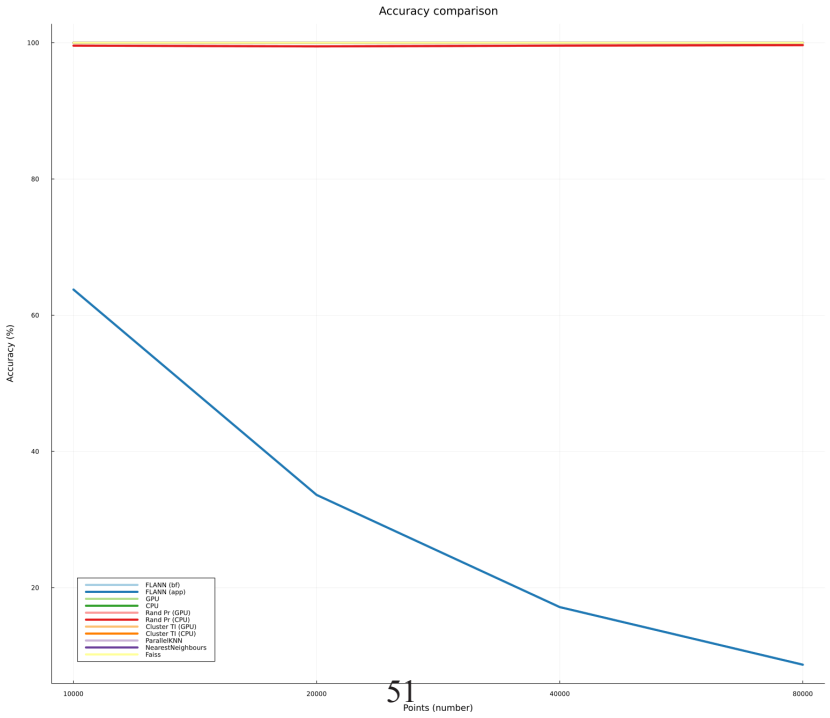


(β□) Συνθετικά δεδομένα

Σχήμα 4.14: Σύγκριση των χρόνων εκτέλεσης για k = 500

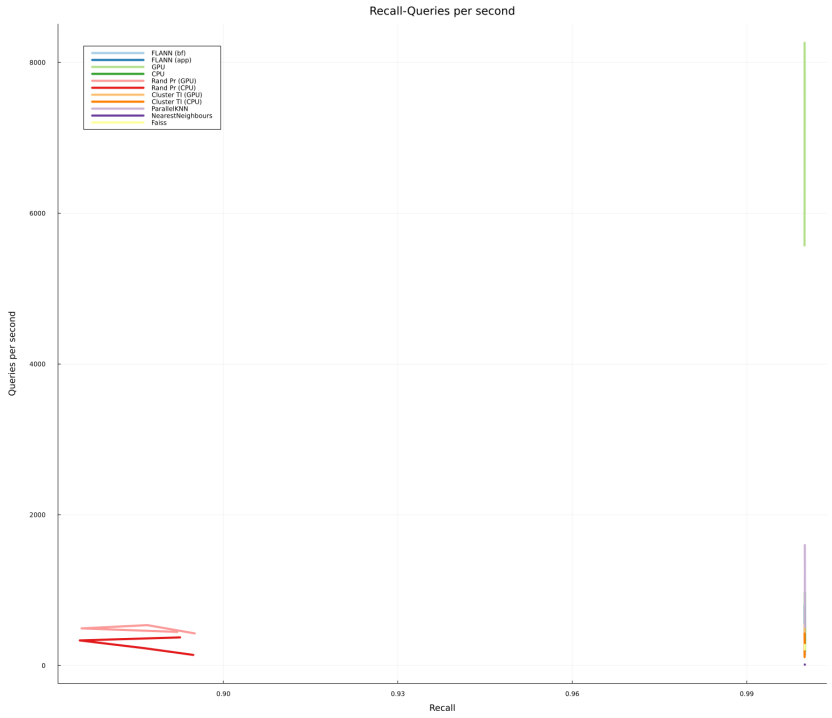


(α□) Δεδομένα HIGGS

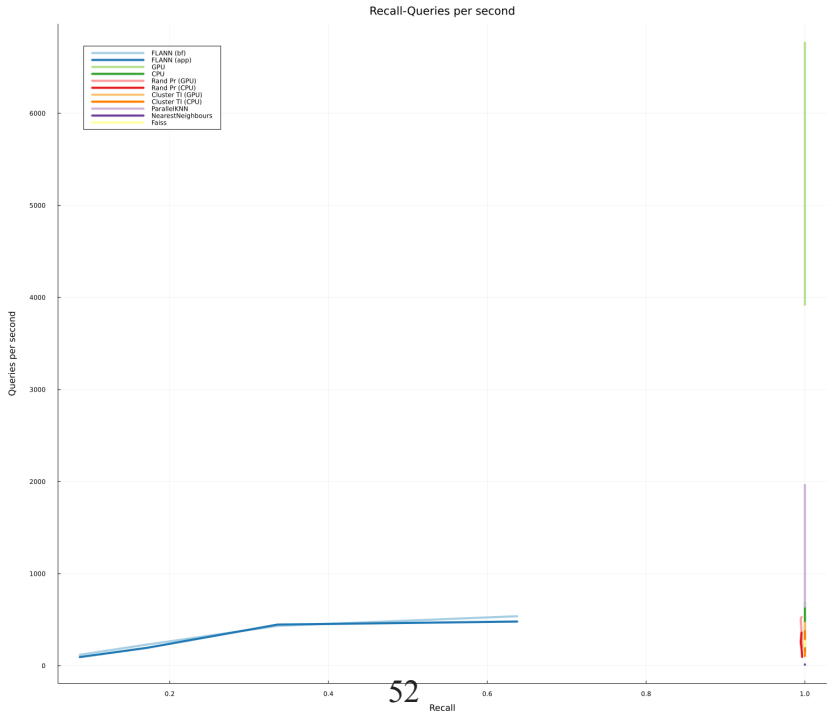


(β□) Συνθετικά δεδομένα

Σχήμα 4.15: Σύγκριση της ακρίβειας για σενάριο εκτέλεσης με k = 500



(α□) k = 150



(β□) k = 500

Σχήμα 4.16: Σύγκριση queries per second με recall για τα δεδομένα 2800 διαστάσεων

Από τα 4.16α και 4.16β παρατηρούμε ότι στην γενικότερη περίπτωση η πλειονότητα των μεθόδων μας δίνει recall = 1, κάτι αναμενόμενο μιας και έχουμε να κάνουμε με brute force αλγορίθμους κατά κύριο λόγο. Παρόλα αυτά φαίνεται ξεκάθαρα ότι η brute force έκδοση του GPU αλγορίθμου μας έχει σχεδόν τέσσερις φορές τον αριθμό των queries per second σε σχέση με το αμέσως πιο γρήγορο αλγόριθμο, που δεν είναι άλλος από τον ParallelNeighbours. Στην περίπτωση των 150 κοντινότερων γειτόνων βλέπουμε ότι μόνο οι δύο εκδόσεις της τυχαίας προβολής (CPU και GPU) έχουν recall κάτω από 0,90 ενώ στην περίπτωση των 500 κοντινότερων γειτόνων αν και ισχύει το ίδιο, το FLANN παρουσιάζει τιμές recall κατα πολύ μικρότερες από αυτές των τυχαίων προβολών.

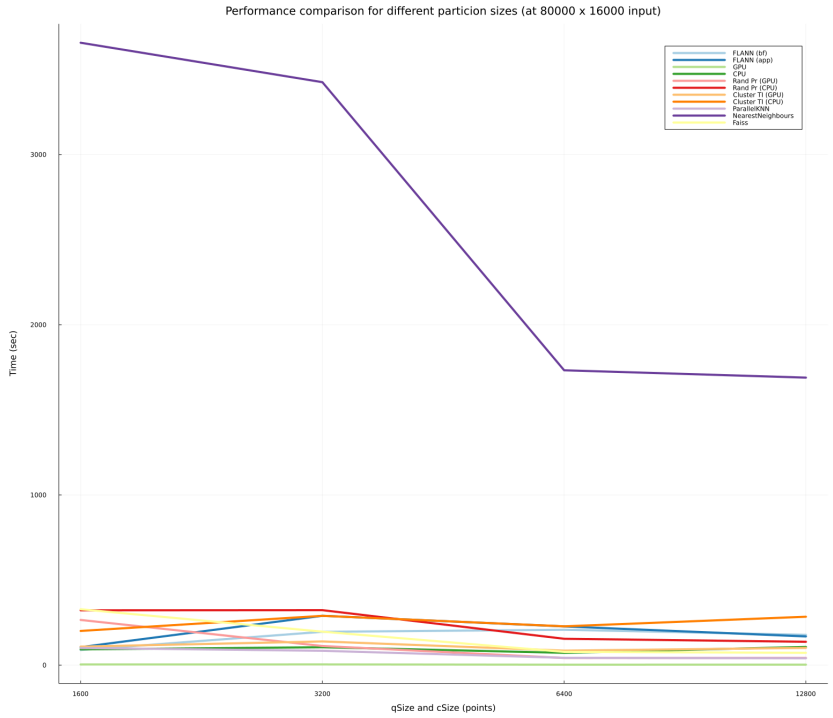
Παρατηρώντας την σχέση μεταξύ του μεγέθους των qSize και cSize με τον χρόνο εκτέλεσης των μεθόδων, προσπαθήσαμε να μελετήσουμε περεταίρω τη συμπεριφορά των μεθόδων για διαφορετικές τιμές των μεταβλητών αυτών, κρατώντας όλες τις υπόλοιπες παραμέτρους εκτέλεσης σταθερές. Μάλιστα για να εξασφαλίσουμε ότι τα αποτελέσματα είναι πιο χρήσιμα, επιλέξαμε ένα μεγάλο αριθμό σημείων για τα σύνολα C και Q (80000 και 16000 αντίστοιχα). Σε κάθε περίπτωση επιλέχθηκε το συνθετικό σετ δεδομένων 2800 διαστάσεων για να εξασφαλίσουμε ότι τα προβλήματα είναι όσο το δυνατόν πιο απαιτητικά σε υπολογισμούς. Εκτός από τους χρόνους εκτέλεσης εξετάσαμε και την ακρίβεια των υπολογισμών κατά την διαφοροποίηση των qSize και cSize, όπως επίσης και τον αριθμό των queries per second σε σχέση με το recall. Οι παράμετροι που κρατήσαμε σταθερές είναι ο αριθμός των διαστάσεων (d = 2800), ο αριθμός των γειτόνων (k = 150), ο αριθμός των μειωμένων διαστάσεων για την τυχαία προβολή (r = 56) και ο αντίστοιχος αριθμός επαναλήψεων των τυχαίων προβολών (P = 14). Επίσης δοκιμάσαμε και για δύο διαφορετικές περιπτώσεις με 4 διεργασίες των 16 CPUs, και 8 διεργασίες των 8 CPUs, μιας και η δεύτερη παρουσιάζει καλύτερη απόδοση. Λόγω των μεγάλων χρόνων εκτέλεσης του

Nearest Neighbors, δημιουργήσαμε ξεχωριστά διαγράμματα χωρίς τα αποτελέσματα του αλγορίθμου αυτού έτσι ώστε να είναι πιο εύκολη η ανάγνωση των υπολοίπων.

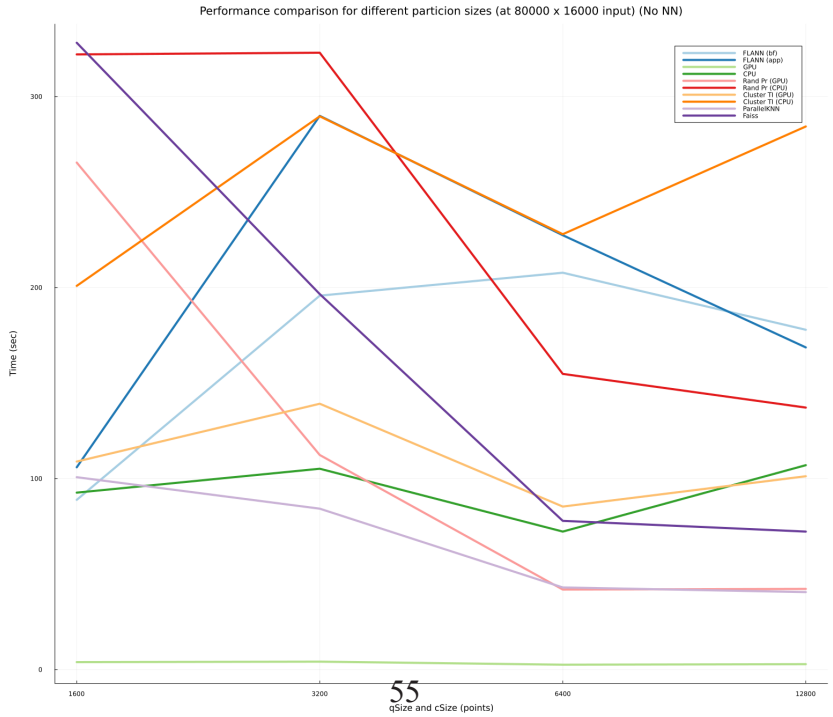
Βλέποντας τα αποτελέσματα του 4.17α□ όσο και του 4.19α□ (και κυρίως των 4.17β□ όσο και του 4.19β□)μπορούμε να παρατηρήσουμε ότι σε γενικές γραμμές η επιλογή υπερβολικά μικρών τιμών για τα qSize και cSize είναι ιδιαίτερα κακή για την περίπτωση της τυχαίας προβολής (τόσο σε CPU όσο και σε GPU). Αντιθέτως η μέθοδος που βασίζεται στο clustering δεν έδωσε ξεκάθαρα αποτελέσματα ώς προς την επιρροή που έχουν τα qSize και cSize μιας και ο σχηματισμός των ίδιων των clusters έχει την μεγαλύτερη επίδραση στην απόδοση. Ωστόσο στην περίπτωση της GPU έκδοσης του clustering, στο 4.17β□ υπάρχει μια τάση να βελτιώνεται η απόδοση για μεσαίες τιμες qSize και cSize (ανάμεσα σε 3200 - 6400) ενώ τόσο για μικρές όσο και για μεγάλες τιμές η απόδοση μειώνεται. Η συμπεριφορά αυτή δεν παρατηρείται στο 4.19β□ καθώς η χρήση περισσότερων διεργασιών επιταχύνει κάπως τα γραμμικά κομμάτια του κώδικα για το σχηματισμό των clusters, μιας και έχουμε συνολικά μεγαλύτερο παραλληλισμό.

Κάτι το οποίο είναι άμεσα παρατηρήσιμο από τα 4.18α□ και 4.20α□ είναι ότι η ακρίβεια της μεθόδου που βασίζεται στην τυχαία προβολή μειώνεται σημαντικά για μεγαλύτερα μεγέθη qSize και cSize καθώς ο αριθμός των σημείων που προβάλλουμε σε κάθε επανάληψη γίνεται μεγαλύτερος. Για να καταφέρουμε μεγαλύτερη ακρίβεια σε αυτές τις περιπτώσεις θα χρειαστεί να αυξήσουμε τον αριθμό των επαναλήψεων P, ωστόσο βλέποντας ότι η ακρίβεια παραμένει καλή έως και τα 3200 σημεία για τα Size και cSize, μπορούμε να περιοριστούμε σε αυτό το εύρος τιμών (1600 - 3200) για να αποφύγουμε περαιτέρω αύξηση του χρόνου εκτέλεσης που θα προκύπτει από ένα μεγαλύτερο P.

Στην περίπτωση των αμιγώς CPU μεθόδων η υλοποίηση του brute force αλγορίθμου μας παρουσιάζει μια παρόμοια συμπεριφορά με αυτή της GPU υλοποίησης της μεθόδου

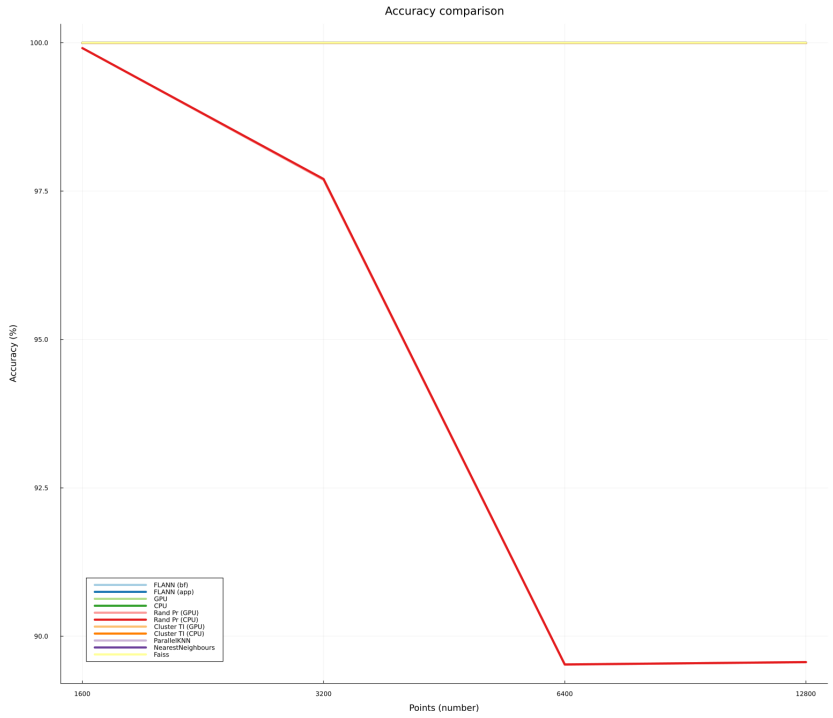


(α□) Χρόνος εκτέλεσης για διαφορετικά qSize και cSize

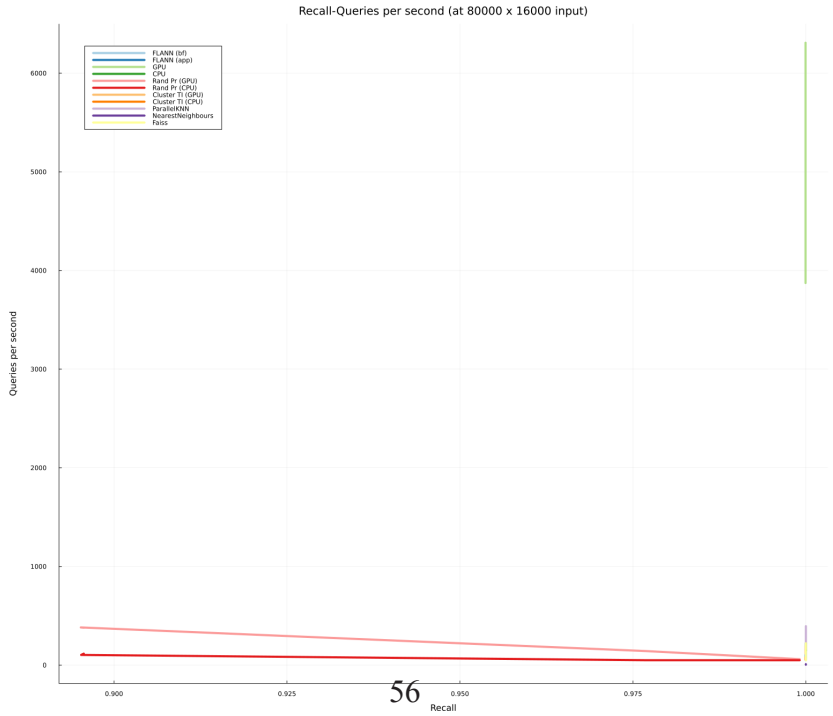


(β□) Χρόνος εκτέλεσης για διαφορετικά qSize και cSize (χωρίς το Nearest Neighbors)

Σχήμα 4.17: Σύγκριση της επιρροής των qSize και cSize για 4 διεργασίες των 16 CPUs

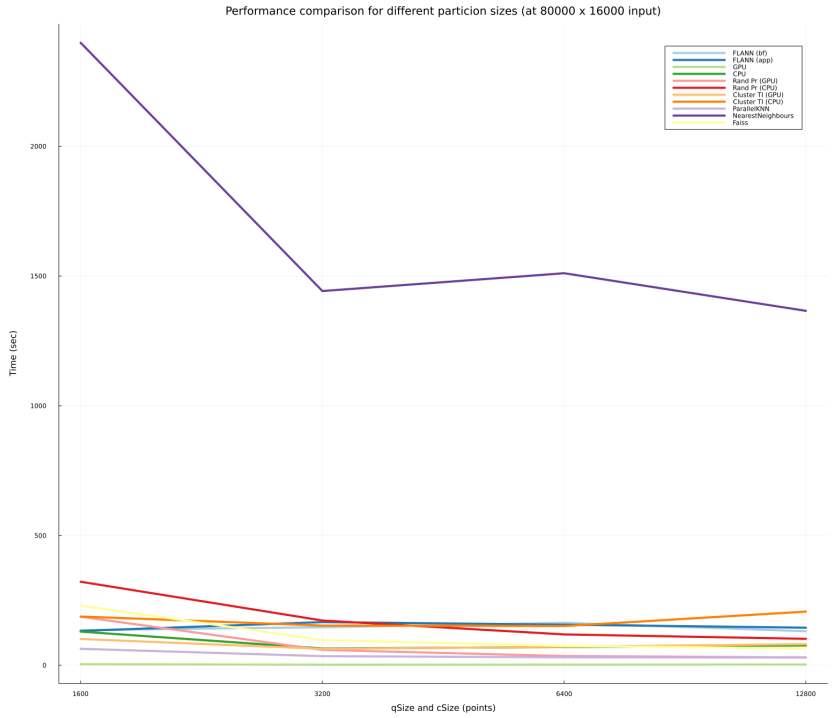


(α□) Ακρίβεια εκτέλεσης για διαφορετικά qSize και cSize

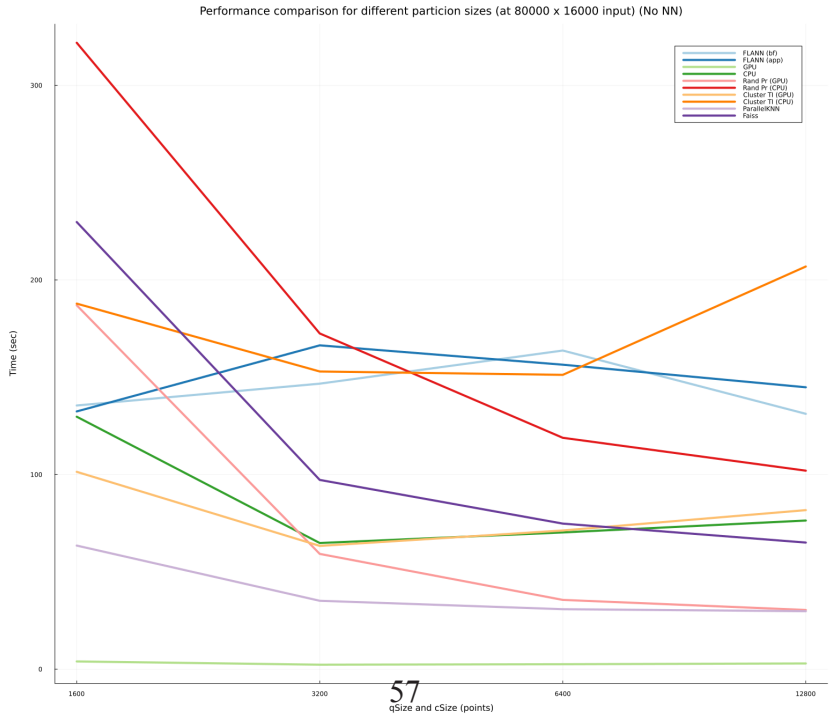


(β□) Queries per second σε σχέση με το recall για διαφορετικά qSize και cSize

Σχήμα 4.18: Σύγκριση ακρίβειας και QPSRecall για διαφορετικά qSize και cSize για 4 διεργασίες των 16 CPUs

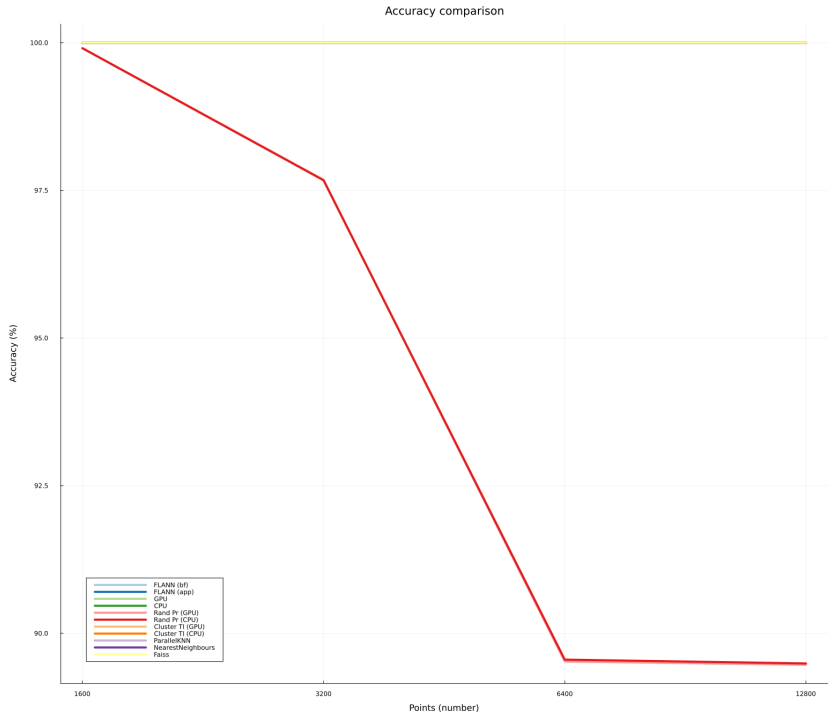


(α□) Χρόνος εκτέλεσης για διαφορετικά qSize και cSize

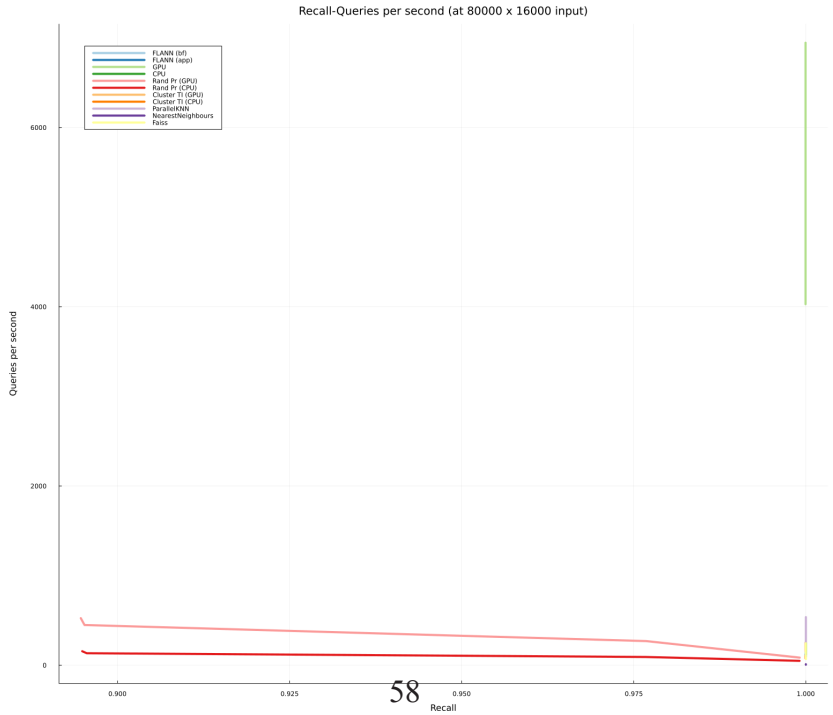


(β□) Χρόνος εκτέλεσης για διαφορετικά qSize και cSize (χωρίς το Nearest Neighbors)

Σχήμα 4.19: Σύγκριση της επιρροής των qSize και cSize για 8 διεργασίες των 8 CPUs



(α□) Ακρίβεια εκτέλεσης για διαφορετικά qSize και cSize



(β□) Queries per second σε σχέση με το recall για διαφορετικά qSize και cSize

Σχήμα 4.20: Σύγκριση ακρίβειας και QPSRecall για διαφορετικά qSize και cSize για 8 διεργασίες των 8 CPUs

clustering που περιγράψαμε παραπάνω. Αυτό δεν αποτελεί τόσο μεγάλη έκπληξη μιας και το στάδιο του ίδιου του clustering γίνεται στη CPU, οπότε και η ίδια η συμπεριφορά παρατηρείται και στην περίπτωση του brute force CPU αλγορίθμου μας, αν και εδώ έχει να κάνει με την ίδια την απόδοση των υπολογισμών απόστασης. Γενικότερα η αύξηση του αριθμού των διεργασιών φαίνεται ότι αυξάνει την απόδοση, αλλά ο διαμοιρασμός των ίδιων CPUs σε περισσότερες διεργασίες έχει ως αποτέλεσμα να μειώνεται η απόδοση όσο λιγότερες CPUs έχουμε ανα διεργασία. Αυτό σημαίνει ότι γενικότερα η απόδοση τείνει να γίνει βέλτιστη για μεσαίου μεγέθους qSize και cSize (ανάμεσα σε 3200 - 6400), ενώ για μικρές τιμές ο μεγάλος αριθμός των κατατμήσεων μειώνει την απόδοση. Από την άλλη, μεγάλες τιμές qSize και cSize οδηγούν σε χειρότερη αξιοποίηση της cache του επεξεργαστή μας, που έχει ως αποτέλεσμα την μείωση της απόδοσης.

Κάτι που έχει ιδιαίτερο ενδιαφέρον είναι η περίπτωση του FLANN, τόσο στην brute force έκδοσή του (linear search) όσο και στην approximated εκδοχή του (kd trees) η συμπεριφορά του είναι αντιδιαμετρικά αντίθετη με αυτή της brute force μεθόδου μας (για τη CPU), παρουσιάζοντας καλύτερη απόδοση για μικρά και μεγάλα μεγέθη qSize και cSize, ενώ για μεσαία μεγέθη η απόδοση μειώνεται. Μάλιστα αυτό ισχύει τόσο στην περίπτωση του 4.17β□ όσο και του 4.19β□, σε αντίθεση με την περίπτωση της brute force CPU μεθόδου μας, που μόνο για το 4.19β□ παρουσιάζει διαφοροποίηση στην συμπεριφορά της βάσει των qSize και cSize.

Οι brute force GPU μέθοδοι (τόσο ο δικός μας αλγόριθμος όσο και το ParallelNeighbours) φαίνονται να μην επηρεάζονται σε τόσο μεγάλο βαθμό από την επιλογή των qSize και cSize, αν και στην περίπτωση που χρησιμοποιήσουμε πολύ μικρές τιμές τότε πρέπει να χωρίσουμε το πρόβλημά μας σε πάρα πολλά κομμάτια. Κάτι τέτοιο έχει ως άμεσο αποτέλεσμα να χρειάζεται να μεταφέρουμε δεδομένα μεταξύ της κεντρικής RAM και της VRAM

των GPUs μας, και οι επιπλέον αυτές καθυστερήσεις μειώνουν την απόδοση όταν συμβαίνουν πάρα πολύ συχνά. Στην γενικότερη περίπτωση μεγαλύτερα μεγέθη qSize και cSize φαίνεται να λειτουργούν καλύτερα στους αλγόριθμους αυτούς, δεδομένου ότι ο μαζικός παραλληλισμός που προσφέρουν οι GPUs εξασφαλίζει καλύτερο χρόνο εκτέλεσης στα περισσότερα προβλήματα. Μια διαφορετική συμπεριφορά παρατηρούμε στο Faiss το οποίο παρουσιάζει καλύτερη απόδοση όσο μεγαλώνει το μέγεθος των qSize και cSize, μιας και φαίνεται ότι η μέθοδός μας με τον κατακερματισμό του προβλήματος σε μικρότερα κομμάτια δεν είναι και η πιο κατάλληλη για το Faiss. Αυτό γίνεται ιδιαίτερα εμφανές για qSize και cSize 1600, όπου με εξαίρεση τον αλγόριθμο Nearest Neighbors, γίνεται ο πιο αργός αλγόριθμος στο 4.17β□ και ο δεύτερος πιο αργός στο 4.19β□. Ωστόσο πρέπει να δοθεί ιδιαίτερη προσοχή στο γεγονός στο ότι η VRAM έχει μικρότερο γενικά μέγεθος σε σύγκριση με την κύρια RAM ενός συστήματος (ή τη συνολική RAM των κόμβων μιας συστοιχίας), οπότε και θα πρέπει να περιοριζόμαστε σε τιμές qSize και cSize που δεν αυξάνουν την χρήση της VRAM σε βαθμό που να μην είναι δυνατή η εκτέλεση των προβλημάτων στο υπάρχον υλικό. Έτσι αν και είναι γενικά γρηγορότερη η εκτέλεση με μικρότερο κατακερματισμό του σετ δεδομένων μας, κάτι τέτοιο μπορεί εύκολα να οδήγησει σε εξάντληση των πόρων του συστήματος μας.

5

Συμπεράσματα

Από τη μελέτη των πειραμάτων μας, και έχοντας ως γνώμονα τον αρχικό στόχο της διπλωματικής καταφέραμε να παρουσιάσουμε μια σειρά από διαφορετικές μεθόδους που έχουν ως σκοπό το να αντιμετωπίσουν όσο το δυνατόν πιο αποτελεσματικά μεγάλα προβλήματα kNN με χρήση κατανεμημένου προγραμματισμού και υπολογισμούς σε GPUs. Έτσι μπορούμε να πούμε με μία σχετική σιγουριά ότι η χρήση των GPUs μπορεί να βοηθήσει στην ταχύτερη επίλυση των kNN προβλημάτων στο σύνολο των περιπτώσεων, αν και σε δεδομένα μικρού αριθμού διαστάσεων τα οφέλη δεν είναι τόσο θεαματικά. Αντιθέτως τα πλεονεκτήματα της χρήσης GPUs για τους υπολογισμούς αποστάσεων γίνονται πολύ πιο αισθητά όταν ο αριθμός των διαστάσεων των δεδομένων μας είναι αρκετά μεγάλος. Στα ίδια πλαίσια κινούνται και οι μέθοδοι που προσπαθούν να μειώσουν τον αριθμό των υπολογισμών, όπως το φιλτράρισμα με χρήση clustering, καθώς και η τυχαία προβολή. Παρά το γεγονός ότι η τυχαία προβολή είναι μία προσεγγιστική μέθοδος, είναι δυνατόν να επιτύχει υψηλά ποσοστά ακρίβειας για κατάλληλες τιμές r και P. Βέβαια κάτι τέτοιο δίνει

ανταγωνιστικές επιδόσεις όταν το υλικό μας φτάνει στα όριά του, μιας και οι απαιτούμενες επαναλήψεις των προβολών προσθέτουν μια σχετικά μεγάλη χρονική επιβάρυνση, η οποία μπορεί να αποσβεστεί μόνο στην περίπτωση πολύ μεγάλου αριθμού διαστάσεων στα δεδομένα. Το ίδιο δυστυχώς δεν μπορεί να ισχύσει και για την περίπτωση του clustering μιας και δεν είναι εύκολο να αντισταθμίσουμε την επιβάρυνση που προσθέτει το clustering στην περίπτωση της προσέγγισής μας που βασίζεται στην διάσπαση του προβληματος μας σε κομμάτια.

Εκτός από τα πλεονεκτήματα στον χρόνο των υπολογισμών, η προτεινόμενη κατανεμημένη προσέγγιση παρέχει και διευκολύνσεις στην διαχείριση των πόρων των κόμβων μας, μιας και με σωστή επιλογή των cSize και qSize μπορούμε να κρατήσουμε την χρήση της RAM αλλά και της VRAM της GPU σε όρια που κρίνονται ικανοποιητικά για την εκτέλεση των απαιτούμενων υπολογισμών. Χωρίς αυτή την προσέγγιση κατακερματισμού του προβλήματος, για μεγάλα σετ δεδομένων τόσο οι πίνακες αποστάσεων, όσο και οι ενδιάμεσοι πίνακες που χρησιμοποιούνται στην διάρκεια των υπολογισμών μας, μπορούν να ξεπεράσουν τις δυνατότητες του υλικού μας.

Ένα από τα μεγαλύτερα προβλήματα που αντιμετωπίσαμε είναι η σχετικά απογοητευτική απόδοση της μεθόδου φιλτραρίσματος μέσω clustering. Αν και η προσέγγιση με τις τυχαίες προβολές παρουσιάζει και αυτή χαμηλή απόδοση σε αρκετές περιπτώσεις, βάσει των πειραμάτων μας μπορέσαμε να εξακριβώσουμε ότι για δεδομένα πολύ μεγάλων διαστάσεων είναι δυνατόν να δημιουργηθούν σενάρια όπου η μέθοδος αυτή παρουσιάζει πλεονεκτήματα. Αντιθέτως το ίδιο δεν ισχύει και για το φιλτράρισμα μέσω clustering και τριγωνική ανισότητα, που σε όλες τις περιπτώσεις παρουσίασε χαμηλή απόδοση.

Εδώ θα μπορούσαμε να αναφέρουμε κάποια σημεία που θα ήταν ιδιαίτερα χρήσιμα σαν αντικείμενα μελλοντικής έρευνας που θα βελτίωναν τόσο την προσέγγισή μας, όσο

και παρεμφερείς εργασίες πάνω στον υπολογισμό kNN προβλημάτων. Αν και η μέθοδος μέσω clustering και εφαρμογή της τριγωνικής ανισότητας παρουσιάζει θεωρητικά κάποια πλεονεκτήματα, στην πράξη παρατηρήσαμε ότι ο διαχωρισμός των σετ δεδομένων μας σε clusters είναι μία μη αποδοτική μέθοδος για δεδομένα ιδιαίτερα μεγάλων διαστάσεων. Επομένως η εύρεση μιας αποδοτικότερης μεθόδου clustering θα μπορούσε να επηρεάσει σημαντικά τον επιπρόσθετο φόρτο εργασίας για το φιλτράρισμα που κάνουμε πριν τους υπολογισμούς αποστάσεων.

Το σημαντικότερο όμως με διαφορά σημείο βελτίωσης που επηρεάζει βασικά όλες τις μεθόδους που παρουσιάσαμε είναι η ανάγκη για ταξινόμηση των αποτελεσμάτων μέσω της CPU. Αυτό αποτελεί ένα σοβαρό περιορισμό, μιας και μας αναγκάζει να επιλέξουμε αρκετά ισχυρότερες CPUs (ή τουλάχιστον να χρησιμοποιήσουμε περισσότερους πυρήνες) σε συνδυασμό με τις GPUs μας. Αν μάλιστα δεν καταφέρουμε να πετύχουμε την σωστή ισορροπία μεταξύ των δύο αυτών υπολογιστικών μονάδων των κόμβων μας, τότε αναγκάζουμε την μία από αυτές να περιμένει την άλλη, μειώνοντας έτσι την συνολική απόδοση των υπολογισμών μας. Για καλύτερα αποτελέσματα θα ήταν ιδιαίτερα χρήσιμο να έχουμε μία αποδοτική υλοποίηση της partialsortperm (partial sort permutation) συνάρτησης της Julia σε κώδικα CUDA έτσι ώστε να μπορούμε όχι μόνο να εκτελέσουμε τους υπολογισμού αποστάσεων στην GPU αλλά και να ταξινομήσουμε τα αποτελέσματα. Αν και υπάρχει παρούσα υλοποίηση της sortpem (sort permutation), χωρίς την δυνατότητα να σταματήσουμε την ταξινόμηση μετά τα k αποτελέσματα για το κάθε σημείο του σετ δεδομένων Q, η απόδοση είναι σχετικά χαμηλότερη σε σχέση με την partialsortperm ιδίως όταν υπάρχει μεγάλη διαφορά μεταξύ του αριθμού των ζητούμενων γειτόνων k και του κάθε τμήματος (μεγέθους cSize) στο οποίο σπάσαμε τον συνολικό πρόβλημα.

Παράρτημα Α

Πρώτο παράρτημα

Οδηγίες χρήσης

Η βασική συνάρτηση του πακέτου kNN-JuliaGPU για τον υπολογισμό ενός προβλήματος kNN είναι η distributedKNN που μπορεί να κληθεί με δύο βασικούς συνδυασμούς παραμέτρων που καθορίζουν το αν τα αποτελέσματα θα επιστραφούν ως πίνακες ή ως δυαδικά αρχεία (δύο αρχεία, για τα indexes και distances). Στην δεύτερη περίπτωση ο χρήστης ορίζει τα ονόματα των αρχείων αποτελεσμάτων, και μετά το τέλος της εκτέλεσης της distributedKNN τα αποτελέσματα αποθηκεύονται τα αρχεία αυτά.

Προσοχή: αν τα αρχεία Indxs_out και dists_out υπάρχουν, τότε διαγράφονται κατά την εκκίνηση της distributedKNN.

Αξίζει να σημειωθεί ότι για καλύτερα αποτελέσματα σε χρόνο εκτέλεσης χρειαζόμαστε ένα συνδυασμό διεργασιών και νημάτων. Η Julia διαθέτει τα οριίσματα -p και - t για τον ορισμό των διεργασιών και νημάτων αντίστοιχα. Ωστόσο η ταυτόχρονη χρήση τους μπορεί

να οδηγήσει σε παρανοήσεις ως προς την λειτουργία τους.

Τα ορίσματα αυτά αφορούν την κεντρική διεργασία της Julia στην οποία μπορούμε να προσθέσουμε περισσότερες διεργασίες ή νήματα όπως θεωρούμε απαραίτητο για τις ανάγκες των προγραμμάτων μας. Όμως αυτό δεν σημαίνει ότι και οι διεργασίες που προσθέτουμε αποκτούν πρόσβαση σε -t νήματα η κάθε μία.

Για να εξασφαλίσουμε κάτι τέτοιο καλό είναι να ορίσουμε τον αριθμό των νημάτων ως μεταβλητή περιβάλλοντος και να εκτελέσουμε την Julia μόνο με το -p όρισμα. Για παράδειγμα αν θέλουμε να έχουμε 4 διεργασίες από 4 νήματα η κάθε μία τότε πρέπει να ορίσουμε:

```
export JULIA_NUM_THREADS=4
```

και να εκτελέσουμε έπειτα την Julia ως

```
julia -p 4 Our_program.jl
```

Δεν υπάρχει λόγος η μεταβλητή περιβάλλοντος να τεθεί μόνιμα σε περίπτωση που δεν το επιθυμούμε. Αρκεί να τρέξουμε το κατάλληλο export στη γραμμή εντολών πριν την εκκίνηση της Julia.

A□.0.1 Ορίσματα

Αρχικά ας δούμε τα ορίσματα της distributedKNN για την περίπτωση όπου επιστρέφονται πίνακες.

distributedKNN(filenameC, filenameQ, k, d, nC, nQ, cSize, qSize, algorithm, in_memory, r, P)

Τα ορίσματα της συνάρτησης αυτής έχουν ως εξής:

◇ filenameC (String) : Το όνομα το αρχείου που περιέχει το σετ δεδομένων Corpus.

◇ filenameQ (String) : Το όνομα το αρχείου που περιέχει το σετ δεδομένων Query.

- k (Int) : Ο αριθμός των κοντινότερων γειτόνων που αναζητούμε για τα σημεία του Query.
- d (Int): Ο αριθμός των διαστάσεων των σετ Corpus και Query.
- nC (Int): Ο αριθμός των σημείων του σετ δεδομένων Corpus.
- nQ (Int): Ο αριθμός των σημείων του σετ δεδομένων Query.
- cSize (Int): Το μέγιστο μέγεθος των κομματιών που θα χωρίσουμε το σετ Corpus.
- qSize (Int): Το μέγιστο μέγεθος των κομματιών που θα χωρίσουμε το σετ Query.
- algorithm (Int): Ο αλγόριθμος που επιθυμούμε να χρησιμοποιήσουμε για των υπολογισμό των αποτελεσμάτων. Οι διαθέσιμες επιλογές είναι οι εξής:
  - Algorithm 0: brute force FLANN (algorithm 0)
  - Algorithm 1: approximated FLANN (kd trees)
  - Algorithm 2: brute force GPU
  - Algorithm 3: brute force CPU
  - Algorithm 4: ParallelNeighbors (GPU)
  - Algorithm 5: random projection CPU
  - Algorithm 6: random projection GPU
  - Algorithm 7: TI filtering CPU
  - Algorithm 8: TI filtering GPU
  - Algorithm 9: NearestNeighbors (kd trees)

▷ Algorithm 10: Faiss (GPU)

◇ in_memory (Bool): Επιλογή κράτησης των ενδιάμεσων αποτελεσμάτων στην RAM (true) ή αποθήκευσή τους σε δυαδικά αρχεία (false) για μικρότερη χρήση μνήμης κατά την εκτέλεση.

◇ r (Int): (προαιρετικό) Ο αριθμός των μειωμένων διαστάσεων που επιθυμούμε για τις μεθόδους random projection.

◇ P (Int): (προαιρετικό) Ο αριθμός των επαναλήψεων για τις μεθόδους random projection.

Η μέθοδος random projection (Algorithm 5 & 6) είναι μια προσεγγιστική μέθοδος που βασίζεται στη μείωση των διαστάσεων του αρχικού προβλήματος από d σε r. Μία ικανοποιητική τιμή για το r είναι η στρογγυλοποιημένη προς τα πάνω τετραγωνική ρίζα του d. Αντίστοιχα, για καλύτερη ακρίβεια στα αποτελέσματα χρειάζεται να εκτελεστεί παραπάνω από μία φορές. Επιλέγοντας ένα κατάλληλο P μπορούμε να επιτύχουμε καλύτερα αποτελέσματα, αν και όσες περισσότερες επαναλήψεις, τόσο πιο αργή γίνεται η εκτέλεση της μεθόδου αυτής. Τα μεγαλύτερα πλεονεκτήματα της random projection τα παίρνουμε για σετ δεδομένων πολύ υψηλού αριθμού διαστάσεων.

Η έκδοση της distributedKNN για επιστρεφόμενα αποτελέσματα σε δυαδικά αρχεία έχει τα εξής ορίσματα:

distributedKNN(filenameC, filenameQ, file_indxs_out, file_dists_out, k, d, nC, nQ, cSize, qSize, algorithm, in_memory, r, P)

Τα περισσότερα από τα ορίσματα είναι κοινά με την πρώτη έκδοση της distributedKNN. Τα δύο νέα ορίσματα είναι τα file_indxs_out και file_dists_out, που έιναι και τα δύο μορφής String και στα οποία ο χρήστης δίνει τα ονόματα των αρχείων με τα indexes των k

κοντινότερων γειτόνων, και τις αποστάσεις τους από τα σημεία του Q αντίστοιχα.

A□.0.2 Μορφή δεδομένων εισόδου και εξόδου

Όλοι οι πίνακες των αποτελεσμάτων είναι μεγέθους k x nQ, με τις στήλες να αποθηκεύουν τα k αποτελέσματα (ως γραμμές του πίνακα) για το κάθε σημείο του Q. Αυτό γίνεται γιατί η Julia είναι μια column major order γλώσσα σε αντίθεση με άλλες, όπως για παράδειγμα η C που είναι row major order.

Τα δυαδικά αρχεία που χρειάζεται να δημιουργήσουμε για να περάσουμε τα δεδομένα μας στην distributedKNN μπορούμε να τα δημιουργήσουμε με τη βοήθεια των συναρτήσεων που βρίσκονται στο read_file.jl

Για την αποθήκευση ενός πίνακα data χρησιμοποιούμε την store_file(filename, data, append) με τα εξής ορίσματα:

- ◇ filename (String): Το όνομα του αρχείου δεδομένων που θα αποθηκεύσουμε τα περιεχόμενα του πίνακα data. Η επέκταση θα πρέπει να είναι της μορφής .dat για να αναγνωριστεί από τις συναρτήσεις ανάγνωσης που με τη σειρά τους φορτώνουν το αρχείο στη μνήμη.

- ◇ data (AbstractArray): Ο πίνακας δεδομένων μεγέθους d x N που θέλουμε να μετατρέψουμε σε δυαδικό αρχείο.

- ◇ append (Bool): Αν θα ανοίξουμε το αρχείο filename (σε περίπτωση που υπάρχει ήδη) για προσάρτηση στο τέλος του (true), ή θα το δημιουργήσουμε από την αρχή (false).

Για την ανάγνωση ενός δυαδικού αρχείου filename χρησιμοποιούμε την load_file(filename, d, n, dtype) με τα εξής ορίσματα:

◇ filename (String): Το όνομα του αρχείο (με το απόλυτο ή σχετικό path του) που θέλουμε να διαβάσουμε. Η επέκταση πρέπει να είναι .dat

◇ d (Int): Ο αριθμός των διαστάσεων των δεδομένων

◇ n (Int): Ο αριθμός των εγγραφών που θέλουμε να διαβάσουμε από το filename.

◇ dtype (DataType): (προαιρετικό) Ο τύπος των δεδομένων που θα διαβάσουμε. Η προεπιλογή είναι Float32 και θα πρέπει να παραμείνει η ίδια, μιας και οι μέθοδοί της distributedKNN χρησιμοποιούν δεδομένα εισόδου Float32.

Ο επιστρεφόμενος πινακας από την load_file έχει μέγεθος d x n.

A□.0.3 Παράδειγμα χρήσης

Ανάγνωση και εγγραφή σε δυαδικά αρχεία.

```
function buildata(
         nc,
         nq,
         d
         )

         C = randn(d,nc)
         C = convert(Array{Float32},C)

         Q = randn(d,nq)
         Q = convert(Array{Float32},Q)

         return C, Q
end
#set parameters for generating random data
nc = 5000
nq = 1000
d = 10

#create two random arrays C and Q
C, Q = buildata(nc, nq, d)

```

```
    #store the arrays in two binary files
    filename_c = "./data_c."dat
    filename_q = "./data_q."dat

    store_file(filename_c, C, false)
    store_file(filename_q, Q, false)

    #load them again in new arrays
    data_C = load_file(filename_c, d, nc)
    data_Q = load_file(filename_q, d, nq)
```

Παράδειγμα εκτέλεσης υπολογισμού προβλήματος kNN στη μνήμη και επιστροφή αποτελεσμάτων σε πίνακες.

```
    nQ = 20000
    nC = 100000
    d = 28
    k = 150
    filename_input_C = "/path/to/datafile_c."dat
    filename_input_Q = "/path/to/datafile_q."dat
    C_size = 3200    #The size of the fragments 'well break C during
   calculations
    Q_size = 3200    #The size of the fragments 'well break C during
   calculations
    algorithm = 2         #brute force GPU
    in_memory = true
    indxs_g = zeros(Int32,k,nQ)
    dists_g = zeros(Float32,k,nQ)

    indxs_g, dists_g = distributedKNN(filename_input_C,
   filename_input_Q , k, d, nC, nQ, C_size, Q_size, algorithm,
   in_memory)
```

Αντίστοιχα ακολουθεί παράδειγμα για περίπτωση που θέλουμε τα αποτελέσματα σε δυαδικά αρχεία.

```
    nQ = 20000
    nC = 100000
    d = 28
    k = 150
    filename_input_C = "/path/to/datafile_c."dat
    filename_input_Q = "/path/to/datafile_q."dat
    file_indxs = "/path/to/output/indxs."dat
    file_dists = "/path/to/output/dists ."dat
```

```
9     C_size = 3200    #The size of the fragments 'well break C during
      calculations
10    Q_size = 3200    #The size of the fragments 'well break C during
      calculations
11
12    algorithm = 2         #brute force GPU
13    in_memory = true
14
15    distributedKNN(filename_input_C, filename_input_Q, file_indxs,
      file_dists, k, d, nC, nQ, C_size, Q_size, algorithm, in_memory)
```

Παράρτημα Β

Δεύτερο παράρτημα

Εκτέλεση σε περιβάλλον υπολογιστικής συστοιχίας χρειαζόμαστε δύο βασικά προαπαιτούμενα. Το πρώτο είναι το script για τον Workload Manager (που ούτως ή άλλως είναι απαραίτητο για την εκτέλεση μιας εργασίας) και το δεύτερο είναι να καλέσουμε το πακέτο ClusterManagers.jl για να αποκτήσει πρόσβαση το πρόγραμμά μας στους πόρους που ζητήσαμε στο script του Workload Manager.

Τα παρακάτω παραδείγματα αφορούν το SLURM Workload Manager, αλλά αντίστοιχα μπορούν να τροποποιηθούν και για διαφορετικά περιβάλλοντα εκτέλεσης σε άλλους Workload Managers. Στη συνγεκτιμένη περίπτωση ζητούμε 4 διεργασίες από 16 CPUs η κάθε μία (σύνολο 64 CPUs) και 2 GPUs.

```
#!/bin/bash
#SBATCH --job-name=Julia-DistKNN
#SBATCH --ntasks-per-node=4
#SBATCH --nodes=1
#SBATCH --time=06:00:00
#SBATCH --partition=ampere
#SBATCH --gres=gpu:2
```

```
#SBATCH --cpus-per-task=16

module load gcc/10.2.0 julia cuda/11.1.0 cudnn/8.0.4.30-11.0-linux-x64

export JULIA_NUM_THREADS=$SLURM_CPUS_PER_TASK

julia Test_DistKNN.jl
```

Στην περίπτωση που χρησιμοποιούμε το Faiss, πρέπει να ορίσουμε το περιβάλλον της Python στο συστημά μας, οπότε το script εκτέλεσης διαμορφώνεται ως εξής:

```
#!/bin/bash
#SBATCH --job-name=Julia-DistKNN
#SBATCH --ntasks-per-node=4
#SBATCH --nodes=1
#SBATCH --time=06:00:00
#SBATCH --partition=ampere
#SBATCH --gres=gpu:4
#SBATCH --cpus-per-task=16

module load gcc/10.2.0 julia cuda/11.1.0 cudnn/8.0.4.30-11.0-linux-x64
    miniconda3

source $CONDA_PROFILE/conda.sh
conda activate myEnv
export PATH=$CONDA_PREFIX/bin:$PATH

export JULIA_NUM_THREADS=$SLURM_CPUS_PER_TASK
export JULIA_PYTHONCALL_EXE="$CONDA_PREFIX/bin/python"

export LD_LIBRARY_PATH=""

julia Test_DistKNN.jl

conda deactivate
```

Η παράμετρος LD_LIBRARY_PATH είναι προαιρετική και μπορεί να χρησιμοποιηθεί και στο προηγούμενο παράδειγμα αν μας παρουσιάζεται πρόβλημα με την χρήση κάποιων πακέτων όπως το Plots που μπορεί να έχει εγκατασταθεί με προηγούμενη έκδοση της glib, κάτι που οδηγεί σε αποτυχία το precompile του.

Στο πρόγραμμά μας, θα πρέπει στην αρχή του να δηλώσουμε τα εξής:

```
    # load packages
    IN_SLURM && using ClusterManagers

    # Here we create our parallel julia processes
    if IN_SLURM
        pids = addprocs_slurm(parse(Int, ENV["SLURM_NTASKS"]))
        print("\n")
    else
        pids = addprocs()
    end

    # See ids of our workers. Should be same length as SLURM_NTASKS
    # The output of this `println` command will appear in the
    # SLURM output file julia_in_parallel.output
    println(workers())
```

Βιβλιογραφία

[1] 2nd gen amd epyctm 7742.

[2] Διαθέσιμοι υπολογιστικοί πόροι - aristotelis docs 2023, 2023.

[3] Dimitris Achlioptas. Database-friendly random projections. In *Proceedings of the twentieth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*. ACM, May 2001.

[4] Tim Besard, Valentin Churavy, Alan Edelman, and Bjorn De Sutter. Rapid software prototyping for heterogeneous and distributed platforms. *Advances in Engineering Software*, 132:29–46, 2019.

[5] Tim Besard, Christophe Foket, and Bjorn De Sutter. Effective extensible programming: Unleashing Julia on GPUs. *IEEE Transactions on Parallel and Distributed Systems*, 2018.

[6] Guoyang Chen, Yufei Ding, and Xipeng Shen. Sweet knn: An efficient knn on gpu through reconciliation between redundancy removal and regularity. *2017 IEEE 33rd International Conference on Data Engineering (ICDE)*, pages 621–632, 2017.

[7] T. Cover and P. Hart. Nearest neighbor pattern classification. *IEEE Transactions on Information Theory*, 13(1):21–27, January 1967.

[8] Sampath Deegalla, Keerthi Walgama, Panagiotis Papapetrou, and Henrik Boström. Random subspace and random projection nearest neighbor ensembles for high dimensional data. *Expert Systems with Applications*, 191:116078, April 2022.

[9] Jeff Johnson, Matthijs Douze, and Hervé Jégou. Billion-scale similarity search with GPUs. *IEEE Transactions on Big Data*, 7(3):535–547, 2019.

[10] JuliaStats. A julia package for data clustering.

[11] Carlsson Kristoffer. Julia wrapper around the faiss library for similarity search with pythoncall.jl.

[12] KristofferC. High performance nearest neighbor data structures and algorithms for julia.

[13] David Muhr and Michael Affenzeller. Hybrid (CPU/GPU) exact nearest neighbors search in high-dimensional spaces. In *IFIP Advances in Information and Communication Technology*, pages 112–123. Springer International Publishing, 2022.

[14] Marius Muja and David G. Lowe. Fast approximate nearest neighbors with automatic algorithm configuration. In *International Conference on Computer Vision Theory and Applications*, 2009.

[15] Nvidia. NVIDIA A100 GPUs Power the Modern Data Center — nvidia.com. [Accessed 07-08-2023].

[16] Nvidia. CUDA C++ Programming Guide — docs.nvidia.com, 2023. [Accessed 13-Jul-2023].

[17] Jarrett Revels. GitHub - JuliaCI/BenchmarkTools.jl: A benchmarking framework for the Julia language — github.com. [Accessed 08-08-2023].

[18] C. J. Stone. Consistent nonparametric regression. *Annals of Statistics*, 5:595–620, 1977.

[19] Naveen Venkat. The curse of dimensionality: Inside out. 2018.

[20] Daniel Whiteson. Higgs. UCI Machine Learning Repository, 2014.

[21] Wildart. A julia wrapper for fast library for approximate nearest neighbors flann.

[22] Xindong Wu, Vipin Kumar, J. Ross Quinlan, Joydeep Ghosh, Qiang Yang, Hiroshi Motoda, Geoffrey J. McLachlan, Angus Ng, Bing Liu, Philip S. Yu, Zhi-Hua Zhou, Michael Steinbach, David J. Hand, and Dan Steinberg. Top 10 algorithms in data mining. *Knowledge and Information Systems*, 14(1):1–37, December 2007.

[23] Bo Xiao and George Biros. Parallel algorithms for nearest neighbor search problems in high dimensions. *SIAM Journal on Scientific Computing*, 38(5):S667–S699, January 2016.