# Εργασία 4
# Παράλληλα και Διανεμημένα Συστήματα

Καρελής Παναγιώτης (9099)
6/9/2020

## *Εισαγωγή*

Στην παρούσα εργασία υλοποιείται η κατασκευή των δομών Vantage-Point Tree και KD-Tree με χρήση της GPU, σε CUDA C/C++. Επίσης υλοποιούνται οι αλγόριθμοι k-Nearest Neighbor search με χρήση των παραπάνω δομών. Έπειτα γίνεται σύγκριση των χρόνων κατασκευής των δύο δένδρων και του μέσου αριθμού κόμβων που χρειάζεται να επισκεφθεί ο κάθε αλγόριθμος kNN search για τα ίδια datasets. Λόγω της ομοιότητας των δύο δομών, παρακάτω θα γίνει περιγραφή του κώδικα μόνο για το VP-Tree. Ο κώδικας του project βρίσκεται στο Github.

Για την εργασία, έγιναν δοκιμές σε πραγματικά datasets του UCI. Οι μετρήσεις και τα αποτελέσματα στα γραφήματα έγιναν με τυχαία δεδομένα.

## *Κατασκευή VP-Tree με GPU*

Η πρώτη προσπάθεια για την κατασκευή του δένδρου παραλληλοποιούσε τον αναδρομικό αλγόριθμο που παρουσιάστηκε στην πρώτη εργασία του μαθήματος[1] με τη βοήθεια του CUDA Dynamic Parallelism (Adinets, 2014). Στη προσπάθεια αυτή, έγινε χρήση ενός kernel με 1 thread και 1 block, που καλούσε αναδρομικά τον εαυτό του ώστε να κατασκευάζει το αριστερό και δεξί υπό-δένδρο παράλληλα. Εσωτερικά του παραπάνω kernel, γινόταν κλήση σε ένα άλλο kernel (με αρκετά blocks & threads) που υπολόγιζε τις αποστάσεις παράλληλα. Έπειτα τα σημεία χωρίζονταν σε inner και outer με μια QuickSelect(). Η υλοποίηση αυτή παρήγαγε σωστά αποτελέσματα, όμως ήταν πολύ αργή, περίπου 30 με 100 φορές πιο αργή από τη sequential έκδοση. Τα αποτελέσματα αυτά προέκυψαν από δοκιμαστικά πειράματα που έγιναν στην GPU της συστοιχίας του ΑΠΘ (Nvidia Tesla P100).

Από τα παραπάνω πειράματα γίνεται φανερό ότι ο συγκεκριμένος αναδρομικός αλγόριθμος δεν είναι αποδοτικός για GPU. Έτσι, στην εργασία αυτή υλοποιείται ένας επαναληπτικός αλγόριθμος για την κατασκευή του VP-Tree, ο οποίος χτίζει το δένδρο επίπεδο-προς-επίπεδο.

### Περιγραφή Αλγορίθμου

Ο αλγόριθμος χρειάζεται $\lfloor log_2(n) \rfloor$ iterations, όσο δηλαδή και το ύψος του δένδρου. Ορίζουμε τους πίνακες $idArr$ και $distArr$, μεγέθους n. Το στοιχείο $distArr[i]$ ορίζεται ως η απόσταση του σημείου με $id = idArr[i]$ προς κάποιο vantage-point. Σε κάθε κόμβο του δένδρου αντιστοιχεί ένα διάστημα $[start, end]$. Ο πίνακας $idArr$ από τη θέση $start$ έως $end - 1$ περιέχει όλα τα $id$ των απογόνων του κόμβου. Το vantage-point κάθε κόμβου επιλέγεται να είναι το σημείο με $id = idArr[end]$. Για τα παιδιά του κάθε κόμβου έχουμε

- Αριστερό παιδί: $[start_{left}, end_{left}] = [start, \frac{start+(end-1)}{2}]$
- Δεξί παιδί: $[start_{right}, end_{right}] = [\frac{start+(end-1)}{2} + 1,\ end - 1]$

Τέλος, ορίζουμε τους βοηθητικούς πίνακες $start$ και $end$, μεγέθους n, ώστε στις θέσεις $start[i]$ και $end[i]$ να αποθηκεύουμε το διάστημα $[start, end]$ στο οποίο ανήκουν τα $distArr[i]$ και $idArr[i]$ σε κάθε iteration. Οι πίνακες αυτοί ανανεώνονται σε κάθε iteration με το $update\_arrays()$ kernel.

[1] Ο κώδικας του αναδρομικού αλγορίθμου βρίσκεται στο αρχείο *src/vptree_sequential* και η περιγραφή του έχει γίνει σε προηγούμενη εργασία (Karelis & Michalainas, 2019)

Επομένως, για να χτιστεί το δένδρο, σε κάθε iteration θα πρέπει να υπολογίζεται ο πίνακας $distArr$ και έπειτα τα σημεία που ανήκουν σε κάθε διάστημα $[start, end]$ να χωρίζονται σε inner και outer. Τον πίνακα $distArr$ τον γεμίζουμε με τιμές, παράλληλα, με το $distCalc()$ kernel. Για το δεύτερο ζητούμενο, χρησιμοποιούμε ταξινόμηση. Η κλήση μιας συνάρτησης ταξινόμησης για κάθε διάστημα δεν είναι αποδοτική, μπορούμε όμως να πετύχουμε το ίδιο αποτέλεσμα κάνοντας δύο κλήσεις της $stable\_sort\_by\_key()$ για ολόκληρο τον πίνακα, χρησιμοποιώντας και έναν ακόμη βοηθητικό πίνακα $segments$, όπως περιγράφεται από τον Nathan Bell (Bell, 2009). Για να σχηματίσουμε τον πίνακα $segments$ όπως στην παραπάνω πηγή, αρχικά θέτουμε στον πίνακα τις τιμές

$$segments[i] = \begin{cases} 1, & i = start[i]\ OR\ i = end[i] \\ 0, & elsewhere \end{cases}$$ με το $make\_segments()$ kernel.

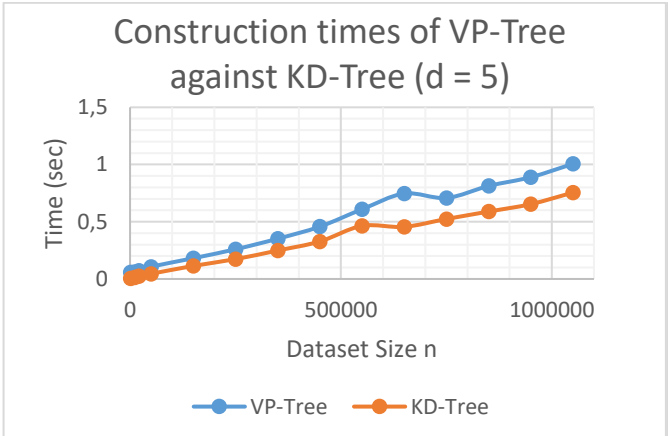
Έπειτα με μια διαδικασία Prefix-Sum/Scan, η οποία είναι efficient με GPU, στον πίνακα $segments$ έχουμε το επιθυμητό αποτέλεσμα.

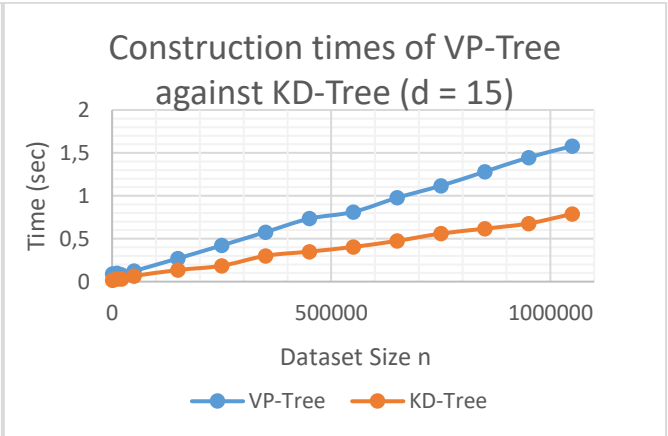
Η κατασκευή του δένδρου στη μνήμη της GPU, έχει ως αποτέλεσμα οι pointers $inner$ και $outer$ να δείχνουν σε λάθος διευθύνσεις όταν μεταφερθεί η δομή στη RAM. Επομένως, είναι αναγκαία η διάσχιση του δένδρου στον host, ώστε να διορθωθούν οι pointers. Παρατηρούμε όμως, ότι είναι εφικτό να κατασκευάσουμε το δένδρο στη CPU έχοντας μόνο τους πίνακες $idArr$ και $distArr$, εφόσον έχουν επεξεργαστεί από τον παραπάνω αλγόριθμο στη GPU. Έτσι γλυτώνουμε το κόστος μεταφοράς λανθασμένων δεδομένων.

**Εργαλεία/Τεχνικές**

Για την υλοποίηση του αλγορίθμου χρησιμοποιήθηκε η βιβλιοθήκη CUDA Thrust. Η βιβλιοθήκη αυτή περιέχει βελτιστοποιημένες παράλληλες υλοποιήσεις των *inclusive_scan()* και *stable_sort_by_key()*, όπως επίσης και πιο τετριμμένες συναρτήσεις όπως η *sequence()* και η *fill()*. Επιπλέον, η βιβλιοθήκη περιλαμβάνει τη κλάση $device\_vector$ που είναι χρήσιμη για την αυτόματη διαχείριση της μνήμης GPU. Επειδή χρειάζεται να ταξινομούνται ταυτόχρονα τα στοιχεία δύο πινάκων με βάση τα κλειδιά (keys), χρησιμοποιούμε $zip\_iterator$. Τέλος, ο κώδικας των kernel χρησιμοποιεί Grid-Stride Loops ώστε να αναθέτει περισσότερη δουλειά σε κάθε thread (Harris, 2013).

Παρακάτω φαίνονται οι χρόνοι κατασκευής των VP-Tree και KD-Tree συναρτήσει του μεγέθους του dataset (με χρήση μιας Nvidia GT 640).




Οι χρόνοι του VP-Tree είναι χειρότεροι, κι αυτό οφείλεται στον υπολογισμό του πίνακα $distArr$.

## *kNN Search με VP-Tree*

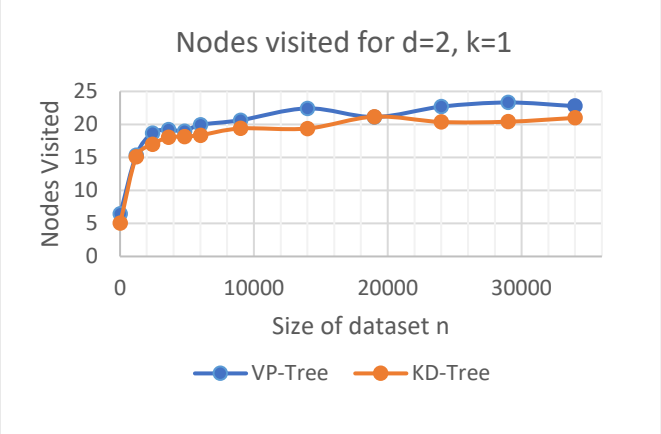
Η αναζήτηση των k κοντινότερων γειτόνων ενός query point περιγράφεται στη πηγή (VP Tree, n.d.). Ο αλγόριθμος συνοψίζεται με 3 διαφορετικές περιπτώσεις (βλέπε στην πηγή παρ. "Searching"):

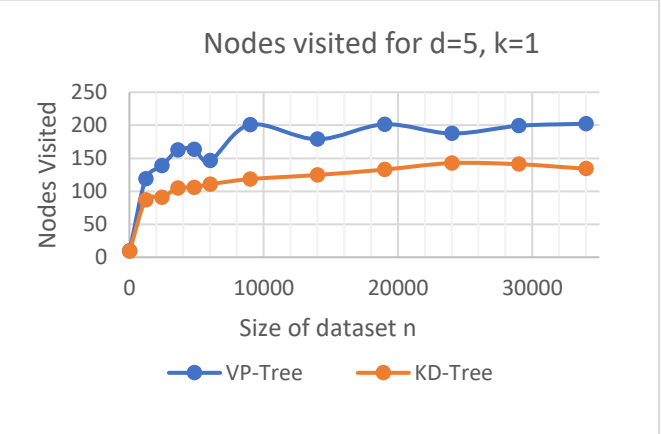
- (Case 1) Το μοναδικό υπό-δένδρο που πρέπει να ελέγξουμε είναι το outer. Μπορούμε με ασφάλεια να αγνοήσουμε το inner.
- (Case 2) Είναι η αντίθετη από τη πρώτη περίπτωση. Χρειάζεται μόνο να ψάξουμε στο inner subtree.
- (Case 3) Έχουμε επικάλυψη και δεν μπορούμε να κλαδέψουμε κάποιο subtree.

Η σειρά με την οποία θα ψάξουμε στα inner και outer (για την Case 3) έχει μεγάλη επίδραση στον τελικό αριθμό nodes που θα επισκεφθεί ο αναδρομικός αλγόριθμος! Η ευρετική μέθοδος που χρησιμοποιήθηκε είναι να ξεκινούμε από το outer αν η απόσταση query-vp είναι μεγαλύτερη από $md$ κι έπειτα στο inner. Αλλιώς πρώτα inner, έπειτα outer.

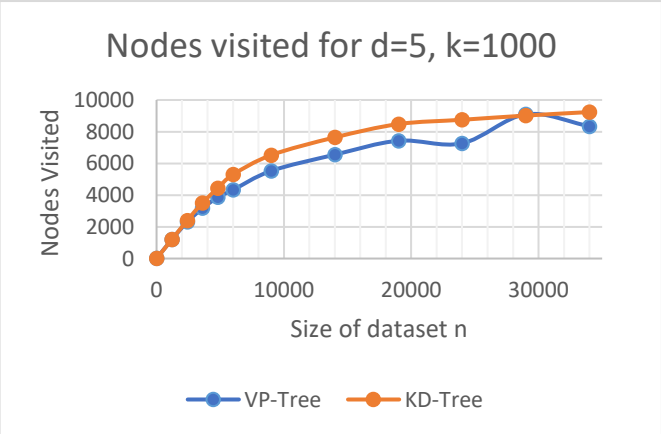
Στον κώδικα χρησιμοποιήθηκε μια $std::priority\_queue$ μεγέθους $k$ για να αποθηκεύονται οι kNN. Ο πιο μακρινός γείτονας βρίσκεται στην κορυφή της ουράς και αφαιρούμε αυτόν, αν βρεθεί κοντινότερος. Η αναζήτηση των all-kNN γίνεται παράλληλα, εκτελώντας τον αλγόριθμο kNN search για κάθε query point, με χρήση $cilk\_for$. Επίσης τροποποιήθηκε ο κώδικας της 2ης εργασίας για all-kNN search σε σύστημα διανεμημένης μνήμης με τοπολογία δακτυλίου ώστε να χρησιμοποιεί το VP-Tree.

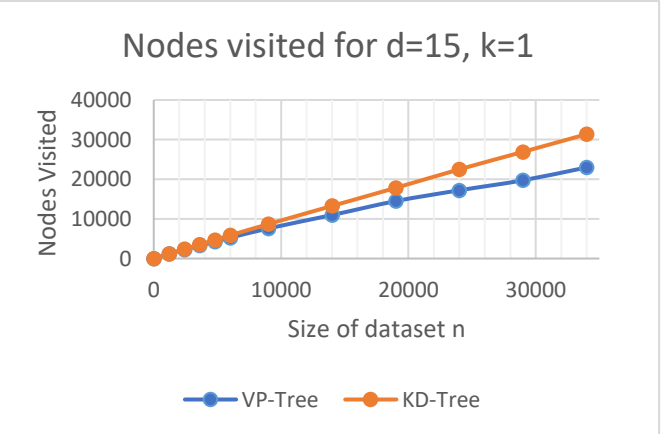
Πιο κάτω παρουσιάζεται ο μέσος αριθμός κόμβων που ελέγχονται χρησιμοποιώντας VP-tree και KD-Tree. Ο μέσος όρος προκύπτει από 100 τυχαία query points ($m = 100$).




Για την εύρεση του κοντινότερου γείτονα ($k = 1$) οι δύο δομές έχουν την ίδια επίδοση για μικρό αριθμό διαστάσεων. Η επίδοση του VP-Tree είναι ελάχιστα χειρότερη.




Με την αύξηση του αριθμού των γειτόνων $k$ ή του αριθμού διαστάσεων $d$ υπάρχει ξεκάθαρο πλεονέκτημα για το VP-Tree.

# Αναφορές


Adinets, A. (2014, May 20). *CUDA Dynamic Parallelism API and Principles*. Ανάκτηση από NVIDIA Developer Blog: https://developer.nvidia.com/blog/cuda-dynamic-parallelism-api-principles/

Bell, N. (2009, December). *vectorized/batch sorting?* Ανάκτηση από Google Groups: https://groups.google.com/g/thrust-users/c/BoLsxO6b4FY?pli=1

Harris, M. (2013, April 22). *CUDA Pro Tip: Write Flexible Kernels with Grid-Stride Loops*. Ανάκτηση από NVIDIA Developer Blog: https://developer.nvidia.com/blog/cuda-pro-tip-write-flexible-kernels-grid-stride-loops/

Karelis, P., & Michalainas, E. (2019). *Εργασία 1: Παράλληλα και Διανεμημένα Συστήματα.* Aristotle University of Thessaloniki, School of Electrical and Computer Engineering. Ανάκτηση από https://github.com/karelispanagiotis/VP_Tree_Construction/blob/master/report.pdf

*VP Tree*. (χ.χ.). Ανάκτηση από fribbels.github.io: https://fribbels.github.io/vptree/writeup