ΠΑΡΑΛΛΗΛΑ ΚΑΙ ΔΙΑΝΕΜΕΙΜΕΝΑ ΣΥΣΤΗΜΑΤΑ

25/01/2019

Αναφορά 3ης Εργασίας

Συγγραφέας: Αντωνιάδης Δημήτριος

Email: akdimitri@auth.gr

ΑΕΜ: 8462

ΠΕΡΙΕΧΟΜΕΝΑ

Παράλληλα και Διανεμημένα Συστήματα

ΑΝΑΦΟΡΑ 3ΗΣ ΕΡΓΑΣΙΑΣ

1. ΕΙΣΑΓΩΓΗ.

Το παρόν έγγραφο αποτελεί την αναφορά της 3ης εργασίας που υλοποιήθηκε στο πλαίσιο του μαθήματος Παράλληλα και Διανεμημένα Συστήματα για το ακαδημαϊκό έτος 2018-2019. Σκοπός της εργασίας αυτής είναι η παραλληλοποίηση του αλγορίθμου εύρεσης κοντινότερου γείτονα σε ομοιόμορφα διαιρεμένο χώρο τριών διαστάσεων.

1.1 Περιεχόμενα φακέλου εργασίας.

Η παρούσα εργασία περιλαμβάνει τρία αρχεία πηγαίου κώδικα:

- knn-grid-cuda-parallel-file-float.cu
- knn-grid-cuda-serial-file-float.cu
- knn-grid-cuda-parallel-file-float-shared.cu

Ένα ***Makefile*** με το οποίο μπορεί να πραγματοποιηθεί **compile** για τα παραπάνω αρχεία με τις εξής εντολές:

- knn-grid-cuda-parallel-file-float.cu: ***>>make parallel***
- knn-grid-cuda-serial-file-float.cu: ***>>make serial***
- knn-grid-cuda-parallel-file-float-shared.cu: ***>>make shared***

Επιπλέον περιλαμβάνεται ένα *Matlab* script:

- matlabScript.m

1.2 Εκτέλεση της εργασίας.

Για την εκτέλεση της εργασίας πρώτα πρέπει να εκτελεσθεί το *matlabScript.m* εντός του φακέλου που βρίσκονται τα εκτελέσιμα αρχεία. Για τον προσδιορισμό του μεγέθους των δεδομένων **N**, μπορεί κανείς να αλλάξει την μεταβλητή **N_power** του αρχείου *matlabScript.m*. Η εκτέλεση του *matlabScript.m* δημιουργεί τέσσερα(4) αρχεία txt μέσα στον φάκελο *test*:

- **./test/MatlabQ21.txt** , το αρχείο αυτό περιλαμβάνει τις συντεταγμένες των σημείων του Q συνόλου.
- **./test/MatlabC21.txt** , το αρχείο αυτό περιλαμβάνει τις συντεταγμένες των σημείων του Q συνόλου.
- **./test/MatlabDistances21.txt** , το αρχείο αυτό περιλαμβάνει την απόσταση του κοντινότερου γείτονα για κάθε σημείο του συνόλου Q.

- **./test/MatlabIndeces21.txt** , το αρχείο αυτό περιέχει τη θέση του κοντινότερου γείτονα για κάθε σημείο του Q στο σύνολο στοιχείων C.

Σημείωση: Σε περίπτωση που αλλαχθούν τα ονόματα των παραγόμενων αρχείων από το *matlabScript.m* πρέπει να αλλαχθούν αντίστοιχα και τα ονόματα εντός του πηγαίου κώδικα των αρχείων.

Η εκτέλεση του παράλληλου προγράμματος μπορεί να γίνει με την εντολή:

- ***>>cudaParallel d_power***

Η εκτέλεση του παράλληλου προγράμματος το οποίο χρησιμοποιεί shared Memory μπορεί να γίνει με την εντολή:

- ***>>cudaParallelShared d_power***

Η εκτέλεση του σειριακού προγράμματος μπορεί να γίνει με την εντολή:

- ***>>cudaSerial d_power***

όπου d_power η δύναμη στην οποία θα υψωθεί το 2 για τον προσδιορισμό των διαστάσεων των μπλοκ κατακερματισμού του χώρου $d = 2^{d\_power}$.

Ο πηγαίος κώδικας βρίσκεται ολόκληρος καθώς και όλα τα παραπάνω αρχεία στη διεύθυνση:

https://github.com/akdimitri/Nearest-Neighbor-Cuda-v.1.git

1.3 Δομή του παρόντος εγγράφου.

Αρχικά, στο έγγραφο παρουσιάζεται η μέθοδος παραγωγής τυχαίων αριθμών, οι οποίοι ακολουθούν ομοιόμορφη κατανομή στο διάστημα [0,1). Στην τρίτη($3^η$) ενότητα περιγράφεται ο σειριακός αλγόριθμος και στην επόμενη ενότητα($4^η$) παρουσιάζεται ο παράλληλος αλγόριθμος, ο οποίος είναι υλοποιημένος σε γλώσσα προγραμματισμού ***C*** σε συνδυασμό με τη χρήση της Διεπαφής Προγραμματισμού Εφαρμογών ***CUDA***. Στην πέμπτη($5^η$) ενότητα παρουσιάζονται τα αποτελέσματα του αλγορίθμου και στην τελευταία ενότητα παρουσιάζονται τα συμπεράσματα της εργασίας.

2. ΠΑΡΑΓΩΓΗ ΤΥΧΑΙΩΝ ΑΡΙΘΜΩΝ.

Για την παράγωγή τυχαίων αριθμών στο διάστημα [0,1) χρησιμοποιήθηκε το πρόγραμμα *Matlab*. Οι εντολές που χρησιμοποιήθηκαν φαίνονται παρακάτω. Με την εντολή ***rand()***[1], δημιουργούνται δύο πίνακες διαστάσεων *N*3*, οι πίνακες Q και C. Στη συνέχεια εκτελείται η εντολή ***knnsearch(C, Q)***[2] η οποία επιστρέφει για κάθε στοιχείο του πίνακα Q την κοντινότερη απόσταση από τα στοιχεία του πίνακα C και τη θέση της γραμμής του C. Τέλος, οι αρχικοί πίνακες και τα αποτελέσματα αποθηκεύονται σε αρχεία.

matlabScript.m

```
%Matlab Script Q,C Sets Generation%NN calculation
N_power = 21;
N = 2^N_power
C = rand(N,3);
Q = rand(N,3);
%Specify Indeces and shortest Distances
[Idx, D] = knnsearch( C, Q);

%Write Distances to files
fileID = fopen('./test/MatlabDistances21.txt','w');
fprintf( fileID,'%f\n', D);
fclose(fileID);

%Write Indeces to File
fileID = fopen('./test/MatlabIndeces21.txt','w');
fprintf( fileID,'%d\n', Idx);
fclose(fileID);

%Write Q to File
fileID = fopen('./test/MatlabQ21.txt','w');
for i = 1:N
fprintf( fileID,'%f %f %f\n', Q(i,1), Q(i,2), Q(i,3));
end
fclose(fileID);

%Write C to File
fileID = fopen('./test/MatlabC21.txt','w');
for i = 1:N
fprintf( fileID,'%f %f %f\n', C(i,1), C(i,2), C(i,3));
end
fclose(fileID);
```

3. ΣΕΙΡΙΑΚΟΣ ΑΛΓΟΡΙΘΜΟΣ.

Ο σειριακός αλγόριθμος είναι ιδιαίτερα απλός. Αρχικά, λαμβάνονται οι τιμές των στοιχείων από τα αρχεία txt που παράχθηκαν από το *matlabScript*. Συνεπώς, τα δεδομένα είναι δύο πίνακες Q, C μεγέθους N*3 ο καθένας. Η πρώτη στήλη του πίνακα απευθύνεται στον άξονα x, η δεύτερη στον άξονα y και η τρίτη στον άξονα z.

2.1 Αρχικοποίηση και προεπεξεργασία Δεδομένων.

Με βάση τον αριθμό των διαστάσεων d, ο χώρος κατακερματίζεται σε ***d*d*d*** κουτιά. Τα κουτιά μοντελοποιούνται με την εξής λογική: x + d*y +d*d*z προκειμένου οι πράξεις να γίνονται σε μονοδιάστατους πίνακες. Αρχικά, ο κώδικας εντοπίζει για κάθε στοιχείο το κουτί στο οποίο θα πρέπει να συμπεριληφθεί και αποθηκεύει την πληροφορία. Στη συνέχεια τα κουτιά αναμετατίθονται και ομαδοποιούνται με βάση το κουτί στο οποίο ανήκουν. Επιπλέον δημιουργείται ένας πίνακας-ευρετήριο της θέσης του πρώτου στοιχείου κάθε κουτιού στον πίνακα. Ακόμη δημιουργείται ένας μεταβατικός πίνακας ο οποίος περιλαμβάνει την παλιά θέση του κάθε στοιχείου. Ο πίνακας αυτός είναι σημαντικός στο τέλος του προγράμματος για την επαλήθευση. Η παραπάνω διαδικασία συμβαίνει και για τους δύο πίνακες Q και C.

Υλοποίηση αλγορίθμου:

```
***main function***
….
readElementsFromFile( hostQ, N,  "./test/MatlabQ21.txt"); //Αρχικοποίηση πίνακα Q
readElementsFromFile( hostC, N,  "./test/MatlabC21.txt"); //Αρχικοποίηση πίνακα C
….
binning( hostQ, hostBoxIndexQ, N, d, blocksSizeQ, oldIndexQ); //Τοποθέτηση στοιχείων σε κουτιά
binning( hostC, hostBoxIndexC, N, d, blocksSizeC, oldIndexC); //Τοποθέτηση στοιχείων σε κουτιά
….
initIndexArray( indexArrayC, blocksSizeC, d); //Δημιουργία πίνακα ευρετηρίου
initIndexArray( indexArrayQ, blocksSizeQ, d); //Δημιουργία πίνακα ευρετηρίου
```

Όπου, hostQ, hostC οι πίνακες με τα στοιχεία και blockSizeC, blockSizeQ, πίνακες καταγραφής του μεγέθους κάθε κουτιού. Οι πίνακες oldIndexQ, oldIndexC αποτελούν τους μεταβατικούς πίνακες και οι πίνακες indexArrayC,indexArrayQ είναι οι πίνακες-ευρετήρια.

Επομένως, οι πίνακες Q και C έχουν τμηματοποιηθεί και είναι έτοιμοι για τη χρήση του προβλήματος εύρεσης του κοντινότερου γείτονα.

2.2 Εύρεση κοντινότερου γείτονα.

Έχοντας τμηματοποιηθεί οι πίνακες βάση της μοντελοποίησης που έχει γίνει στα κουτιά εκκινείτε η διαδικασία εύρεσης του κοντινότερου γείτονα. Αρχικά για κάθε στοιχείο ενός κουτιού του πίνακα Q αναζητείται το κοντινότερο στοιχείο του πίνακα C που ανήκει στο αντίστοιχο κουτί. Η διαδικασία αυτή επαναλαμβάνεται για όλα τα στοιχεία του πίνακα Q. Συνεπώς, το πρώτο μέρος του αλγορίθμου ολοκληρώνεται με την εύρεση του κοντινότερου γείτονα στο αντίστοιχο κουτί.

Στη συνέχεια ελέγχεται η απόσταση του στοιχείο Q από τα όρια του κουτιού. Αν η απόσταση του κουτιού από τα όρια είναι μικρότερη της αποστάσεως του κοντινότερου γείτονα, τότε ενδεχομένως να υπάρχει κοντινότερος γείτονας στα γειτονικά κουτιά. Συνεπώς, ελέγχεται η απόσταση από όλα τα γειτονικά κουτιά και όπου το επιβάλλουν οι συνθήκες αναζητείται κοντινότερος γείτονας.

```
***main function***
/* κλήση συνάρτησης εύρεσης κοντινότερου γείτονα */
nearestNeighbor( NNindeces, NNdistances, hostC, hostQ, indexArrayC, indexArrayQ, N, d);
```

```
/*Search for Primary Candidates */
        for( i = 0; i < d*d*d; i++){        //Επανάλαβε για κάθε κουτί
                if( i != d*d*d-1)//Όρισε τη θέση του τελευταίου στοιχείου στον πίνακα για κάθε κουτί
                        {       limitC = indexArrayC[i+1]; limitQ = indexArrayQ[i+1]; }
                else
                        {       limitC = N; limitQ = N; }
                for( j = indexArrayQ[i]; j < limitQ; j++){ // Για κάθε στοιχείο του κουτιού
                        minDistance = 2;
                        tempIndex = -1;
                        for( k = indexArrayC[i]; k < limitC; k++){ //Βρες την απόσταση με τα στοιχεία
                                distance = calculateDistance( &hostQ[j], &hostC[k]); // του αντίστοιχου
                                if( distance < minDistance){                //κουτιού στον πίνακα C
                                        minDistance = distance;
                                        tempIndex = k;
                                }
                        }
                        NNdistances[j] = minDistance;
                        NNindeces[j]     = tempIndex;
                }
```

```
/* Search for Secondary Candidates */
        for(i = 0; i < d*d*d; i++){
                /* 1st Define Blocks id [x,y,z] */
                …
                /* For Every Element in box i search near boxes */
                for( j = indexArrayQ[i]; j < limitQ; j++){  //Για κάθε στοιχείο του κουτιού i
                        /* Define near Box to search */
                        /* Define z */
                        Για z από -1 έως + 1 του κουτιού i
                        /* Define y */
                        Για y από -1 έως + 1 του κουτιού i
                        /* Define x*/
                        Για x από – 1 έως +1    του κουτιού i
                        /* Block to search has been defined */
                        /* Calculate Distance from Block */
                        blockDistance = sqrt( pow(distanceX, 2) + pow(distanceY, 2) + pow(distanceZ,
2));
                        /* Set id of box to be searched */
                        searchBoxId = x + d*y +d*d*z;
                        if( blockDistance < NNdistances[j]){
                        /* Search All his Elements */
                        for( k = indexArrayC[searchBoxId]; k < limitC; k++){
                                distance = calculateDistance( &hostQ[j], &hostC[k]);
                                if( distance < NNdistances[j]){
                                        NNdistances[j] = distance;
                                        NNindeces[j] = k;
                                }
                        }
                }
        }
}
```

Με την ολοκλήρωση του αλγορίθμου ο πίνακας NNindeces περιλαμβάνει τη γραμμή(θέση) του κοντινότερου γείτονα στον πίνακα C και ο πίνακας NNdistancces περιλαμβάνει την κοντινότερη απόσταση γείτονα για κάθε στοιχείο του Q.

3. ΠΑΡΑΛΛΗΛΟΣ ΑΛΓΟΡΙΘΜΟΣ.

Η υλοποίηση του παράλληλου αλγορίθμου βασίστηκε στον σειριακό αλγόριθμο. Οι αρχικοποιήσεις των πινάκων και η προεπεξεργασία των δεδομένων παρέμεινε ίδια. Επιπλέον, προστέθηκαν οι αρχικοποιήσεις των αντίστοιχων πινάκων στη μνήμη της GPU και οι μεταφορές των δεδομένων μεταξύ κάρτας γραφικών και επεξεργαστή.

Με βάση τα δεδομένα που εισάγονται στον αλγόριθμο:

- Μέγεθος πλέγματος.
- Μέγεθος Δεδομένων

Υπολογίζεται ο αριθμός των στοιχείων μέσα σε κάθε μπλοκ. Ο αριθμός αυτός ισούται με

$$l = 2^{l_{power}} = \frac{2^{N_{power}}}{2^{3*d_{ppower}}}$$

Η εκτέλεση του παράλληλου αλγορίθμου περιλαμβάνει ένα grid ΄τριών διαστάσεων [d*d*d] το οποίο αντιστοιχεί στα blocks που χρησιμοποιήθηκαν για τον κατακερματισμό των δεδομένων. Εάν το μέγεθος του αριθμού των στοιχείων για ένα μπλοκ είναι μικρότερος από τον μέγιστο αριθμό threads που επιτρέπεται σε ένα μπλοκ, τότε κάθε thread μέσα στο μπλοκ αντιστοιχεί σε έναν αριθμό του μπλοκ. Διαφορετικά ο αριθμός των threads προσαρμόζεται στο μέγιστο δυνατό.

```
/* Initialize Dimensions */
        int dimX, dimY, dimZ;
        if( (1 << l_power) > prop.maxThreadsPerBlock){        //Case: number in each Block are more than the maximum number of threads per block.
                l_power = log(prop.maxThreadsPerBlock)/log(2);
        }
        dimX = 1<<(l_power/3 + l_power%3); dimY = 1 << l_power/3; dimZ = 1 << l_power/3;
        if( l_power/3 == 0){    //Case: l_power < 3, then [x*1*1] threads per block.
                dimY = 1;       dimZ = 1;
        }
        dim3 numberOfBlocks( d, d, d);
        dim3 threadsPerBlock( dimX, dimY, dimZ); printf("Threads per Block: [%d %d %d]\n", dimX, dimY, dimZ);
```

Εφόσον έχουν οριστεί τα μπλοκς και τα threads που θα χρησιμοποιηθούν καλείται ο kernel.

```
nearestNeighbor<<< numberOfBlocks, threadsPerBlock>>>( devQ, devC, devIndexArrayQ, devIndexArrayC, devNNindeces, devNNdistances, N, d);
```

3.1 Kernel Nearest Neighbor.

Για την καλύτερη κατανόηση του κώδικα δίνεται πρώτα σε μορφή ψευδοκώδικα.

1. Προσδιόρισε το Id του Thread μέσα στο μπλοκ.
2. Προσδιόρισε το Id του μπλόκ μέσα στο γκριντ.

Επανέλαβε έως ότου να έχουν ελεγχθεί όλα τα στοιχεία του μπλοκ του πίνακα C για το μπλοκ blockIdInGrid

Για i = threadIdInBlock+ Αρχή του μπλοκ BlockIdInGrid ΕΩΣ το τελευταίο στοιχείο του μπλοκ ή έως να διαβάσεις στοιχεία λιγότερα του (Μεγεθος shared memory)/12(μέγεθος στοιχείου)
Επανέλαβε:

Αντέγραψε το στοιχείο i στη shared Memory

Για j = αρχή του στοιχείου του πίνακα Q έως το τέλος του πίνακα

Για k = 0 έως τέλος shared Memory

Υπολόγισε απόσταση, Αν είναι μικρότερη αποθήκευσε την.

Επανέλαβε Για [z-1, z+1]

Επανέλαβε για [y-1, y+1]

Επανέλαβε [x-1,x+1]

Την παραπάνω διαδικασία με τη διαφορά ότι το υπό εξέταση μπλοκ είναι το x + d*y +d*d*z

Ουσιαστικά κάθε thread αναλαμβάνει να υπολογίσει την απόσταση του στοιχείου Q[i] με όλα τα στοιχεία του αντίστοιχου μπλόκ στον πίνακα C. Με την προϋπόθεση ότι τα στοιχεία του πίνακα C έχουν περασθεί στην Shared Memory του κάθε μπλοκ. Σε περίπτωση που τα μεγέθη δεν είναι επιτρεπτά, η διαδικασία προσαρμόζεται μέσω επαναλήψεων χρησιμοποιώντας έξυπνα strides.

Στη συνέχεια η ίδια διαδικασία επαναλαμβάνεται και για όλα τα γειτονικά μπλοκς με τον επιπλέον έλεγχο της απόσταση του στοιχείου από τα όρια του μπλοκ. Αν η απόσταση από τα όρια του μπλοκ είναι μεγαλύτερη προσπερνάει τη διαδικασία.

Ο συνολικός κώδικας βρίσκεται στο παράρτημα.

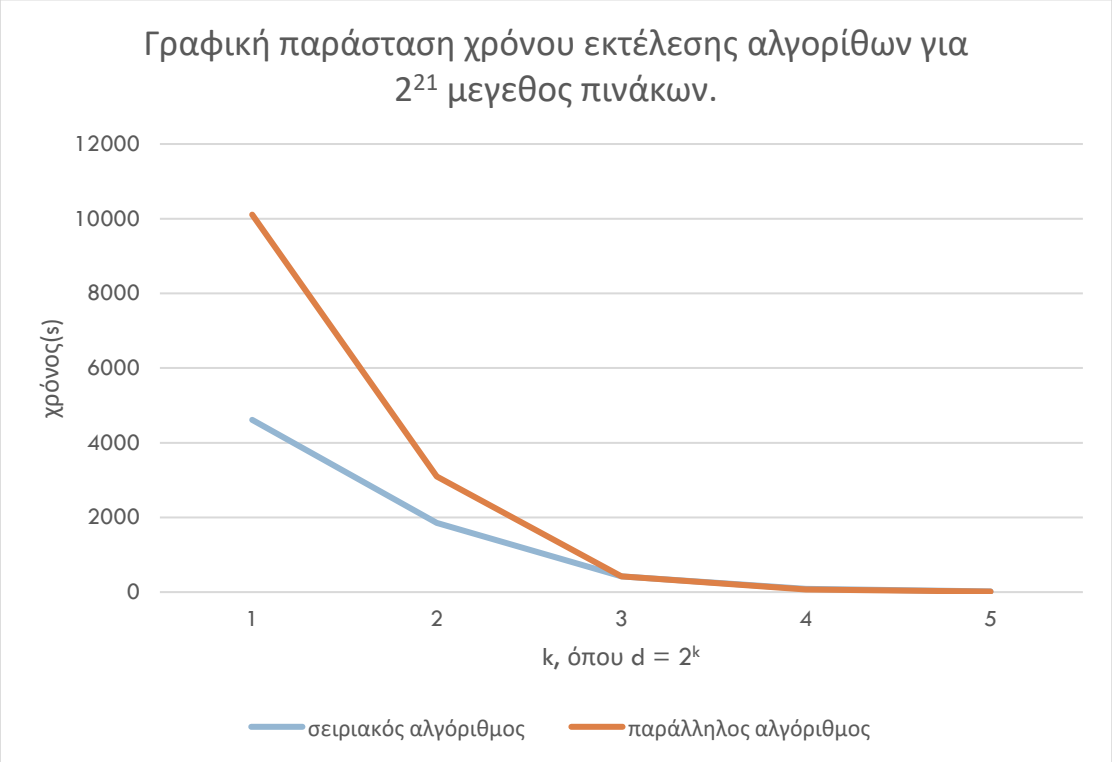
Συμπερασματικά υπολογίζεται παράλληλα ο κοντινότερος γείτονας σε πακέτα των multiprocessors*wraps στοιχείων.

4. ΑΠΟΤΕΛΕΣΜΑΤΑ ΑΛΓΟΡΙΘΜΩΝ.

Οι μετρήσεις για τον σειριακό κώδικα έγιναν σε έναν desktop υπολογιστή με επεξεργαστή 3.9 GHz, ενώ οι μετρήσεις για τον παράλληλο κώδικα σε μία κάρτα γραφικών NVIDIA GeForce 650 Ti με computational capability 3, 4 multiprocessors, μνήμη 1Gb και base clock 928 MHz.

Δυστυχώς λόγω της παλαιότητας της κάρτας γραφικών και του μικρού μεγέθους μνήμης δεν ήταν δυνατή η μέτρηση για πολύ μεγάλες διαστάσεις διότι απαιτούνταν δέσμευση μνήμης: 12*2*N + 4*4*N bytes.

Δίνονται οι μετρήσεις για N = 2^21 της έκδοσης που δεν περιλαμβάνει shared memory.



Συμπεραίνεται ότι η παράλληλη έκδοση είναι ταχύτερη της σειριακής ειδικά για μικρό κατακερματισμό και για όσο το δυνατόν μεγαλύτερο μέγεθος δεδομένων. Αυτό οφείλεται στο ότι όσο μειώνεται ο κατακερματισμός τόσο μειώνεται και η πιθανότητα να πραγματοποιηθεί αποκλίνουσα συμπεριφορά μεταξύ των threads κατά την εκτέλεση. Δηλαδή, αν είναι μικρός ο κατακερματισμός, είναι πολύ πιθανό αρκετά threads να χρειαστεί να ψάξουν γειτονικά μπλοκς με αποτέλεσμα πολλά threads να περιμένουν τα υπόλοιπα για να ολοκληρωθούν. Αντίθετα, όταν υπάρχει μεγάλος κατακερματισμός, ελάχιστα threads θα χρειαστεί να ψάξουν τα γειτονικά.

Η βέλτιστη επιτάχυνση είναι περίπου *5. Ωστόσο η μέτρηση αυτή είναι **σχετική** καθώς σε μεγάλο βαθμό οφείλεται στο γεγονός ότι ο επεξεργαστής είναι τελευταίας γενιάς και η κάρτα αντίστοιχα πολύ παλιά. Εάν η μέτρηση γινόταν σε μία κάρτα γραφικών ανάλογου επίπεδου του επεξεργαστή του υπολογιστή (όπως για παράδειγμα σε μία GeForce 1080) με *2 base clock και 5 φορές περισσότερα CUDA CORES, ενδεχομένως η επιτάχυνση να ήταν ***15-20** φορές πιο καλή σε σχέση με αυτή του επεξεργαστή.

Η μη χρησιμοποίηση της shared memory είναι εμφανής. Επιπλέον, ο αλγόριθμος επιδέχεται σημαντικές επιδιορθώσεις βελτιστοποίησης. Χαρακτηριστικά αναφέρεται ότι στη δημοσίευση [3] ο παράλληλος

αλγόριθμος KNN που σχεδιάστηκε έτσι ώστε να χρησιμοποιεί shared Memory, επέφερε βελτίωση της απόδοσης έως και *40 φορές.

4. ΜΕΘΟΔΟΙ ΕΛΕΓΧΟΥ.

Για τον έλεγχο των αποτελεσμάτων μπορεί να χρησιμοποιηθεί η συνάρτηση ***validateResultsFromFile()***. Η συγκεκριμένη συνάρτηση μπορεί να χρησιμοποιηθεί με δύο τρόπου:

- Χρήση των **ελαχίστων αποστάσεων** που υπολογίσθηκαν από το **Matlab** και σύγκριση με αυτές του αλγορίθμου.
- Χρήση των **δεικτών της θέσης του κοντινότερου γείτονα** όπως αυτός υπολογίσθηκε για κάθε στοιχείο στο **Matlab** και σύγκριση με αυτούς που υπολογίσθηκαν στον αλγόριθμο.

Η σωστή χρήση αυτής της συνάρτησης απαιτούσε τη δημιουργία μεταβατικού πίνακα μεταξύ της θέσης πριν και μετά της προεπεξεργασίας κάθε στοιχείου.

Τα αποτελέσματα των μετρήσεων όταν συγκρίνονται οι αποστάσεις μεταξύ τους είναι **απόλυτα σωστά.**

Στην περίπτωση που χρησιμοποιούνται οι δείκτες παρουσιάζεται ένα σφάλμα της τάξης του 270/2^21 = 0.0001 % το οποίο οφείλεται στις ισοβαθμίες των αποστάσεων και την επιλογή του δείκτη σε περίπτωση ισοβαθμίας.

Ο συνολικός κώδικας περιλαμβάνεται στο παράρτημα.

Επιπλέον, υπάρχει και η συνάρτηση *validateBinning()* που ελέγχει την επιτυχή προεπεξεργασία των δεδομένων.

5. ΣΥΜΠΕΡΑΣΜΑΤΑ.

Από τις παραπάνω μετρήσεις και τη συνολική προσπάθεια που έγινε για την υλοποίηση της εργασίας παρατηρήθηκε ότι η οι κάρτες GPU αποτελούν μία φθηνή επιλογή η οποία παρέχει υπερβολικά μεγάλη υπολογιστική ισχύ. Η λογική του προγραμματισμού αυτών των υπολογιστικών μέσων διαφέρει από την κλασσική λογική παραλληλοποίηση στον CPU καθώς στις GPU η επιτάχυνση επιτυγχάνεται κατά ένα μεγάλο μέρος μέσω της αποτελεσματικής μαζικής χρήσης κοινού χώρου μνήμης. Η σωστή χρησιμοποίηση του υλικού της κάρτας μπορεί να επιφέρει σημαντική βελτίωση της απόδοσης σε έντονα υπολογιστικά προβλήματα όπως είναι το πρόβλημα της εύρεσης του κοντινότερου γείτονα.

6. ΑΝΑΦΟΡΕΣ.

[1] https://www.mathworks.com/help/matlab/ref/rand.html

[2] https://www.mathworks.com/help/stats/knnsearch.html

[3] S. Liang, Y. Liu, C. Wang and L. Jian, "Design and evaluation of a parallel k-nearest neighbor algorithm on CUDA-enabled GPU," 2010 IEEE 2nd Symposium on Web Society, Beijing, 2010, pp. 53-60.
doi:10.1109/SWS.2010.5607480
URL: http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=5607480&isnumber=5607344

7. ΠΑΡΑΡΤΗΜΑ.

7.1 Παράλληλος Αλγόριθμος.

```
__global__ void nearestNeighbor( struct Element *devQ, struct Element *devC, int *devIndexArrayQ, int *devIndexArrayC,
int *devNNindeces, float *devNNdistances,  int N,  int d){
  int i, k, j, l;
  int limitQ, limitC, startQ, startC, index, nearestNeighborIndex, blocksSizeC;
  float minDistance, distance;
  /* Calculate 1D index of a thread relative to its block */
          int threadId_block = threadIdx.x + (threadIdx.y)*blockDim.x + (threadIdx.z)*(blockDim.x)*(blockDim.y);
          /* Calculate 1D index of a block relative to the grid */
          int blockId_grid = blockIdx.x + (blockIdx.y)*(gridDim.x) + (blockIdx.z)*(gridDim.x)*(gridDim.y);
          /* Calculate 1D index o a thread relative to the grid */
          //int threadId_grid = blockId_grid*(blockDim.x*blockDim.y*blockDim.z) + threadId_block;
  /* Calculate Block Stride */
  int blockStride = (blockDim.x)*(blockDim.y)*(blockDim.z); //In case Elements in Query Array are more than Threads
          //printf("blockStride %d\n", blockStride);
  /* Each Block corresponds to the block of the previous pre-processing Clustering */
  /* Start point of a block is known in both arrays Q and C */
  /* Thread Id corresponding to the block represents an element of threadId_block position + start_position of the block in
Array C and Q */
          /* Define Search Limits of Block*/


  if( blockId_grid == d*d*d - 1){//Check for Out of Bounds limit
                  limitQ = N;
                  limitC = N;
          }
          else{     //Stop when you reach 1st Element of next block
                  limitQ = devIndexArrayQ[blockId_grid + 1];
                  limitC = devIndexArrayC[blockId_grid + 1];
          }

  /* Define Start Index of Block*/
  startQ = devIndexArrayQ[blockId_grid];
```

```
 startC = devIndexArrayC[blockId_grid];

 /* Define corresponding index in the Array Q */
 index = startQ + threadId_block;

        blocksSizeC = limitC - startC;
        struct Element Query;
        /* move C to share memory */
        extern __shared__ int sharedMemory[];
        int count = 0;
        struct Element *s = ( struct Element*)(&(sharedMemory[0]));
        j = 0;
        while( startC + j*count < limitC){
                for( l = 0; l < 5120 || l < limitC; l++){ //Do not exceed Shared Memory
                        s[l] = devC[startC + l + 5120*count];
                }
                /* Synchronize Threads to read simultaneously from shared memory */
                __syncthreads();


         /* Primary Nearest Neighbor Set */
         for( i = index; i < limitQ; i = i + blockStride){
                        minDistance = 2;
                        Query = devQ[i];
                        for( k = 0 + 5120*count; k < 5120 || k < blocksSizeC; k++){
                                distance = sqrt( pow(Query.x - s[k].x, 2) + pow(Query.y - s[k].y, 2) + pow(Query.z -
s[k].z, 2));
                                if( distance < minDistance){
                                        minDistance = distance;
            nearestNeighborIndex = k;
          }
         }
         /* Store the results */
         devNNindeces[i] = nearestNeighborIndex;
         devNNdistances[i] = minDistance;
                }
                count++;
 }

 /* Secondary Nearest Neighbor Set */
 int x, y, z, searchBoxId;
        float distanceZ, distanceX, distanceY, interval, blockDistance;

 /* Define Interval between boxes */
        interval = 1/((float)d);
        /* Reinitialize index */
 index = startQ + threadId_block;
        /* Choose Which Box to Search */
        int startZ, startX, startY, endZ, endY, endX;
        startZ = blockIdx.z - 1;
        endZ = blockIdx.z + 1;
        startX = blockIdx.x - 1;
        endX = blockIdx.x + 1;
        startY = blockIdx.y - 1;
        endY = blockIdx.y + 1;

        /* Define z */
 for( z = startZ; z <= endZ; z++ ){
                if( z >= 0 && z < d){
                        /* Define y */
                        for( y = startY; y <= endY; y++ ){
                          if( y >= 0 && y < d){
                            /* Define x*/
                            for( x = startX; x <= endX; x++){
```

```
                        if( x >= 0 && x < d){
                        /* Block to be searched has been defined */
                              searchBoxId = x + d*y +d*d*z;
                              /* Choose to search or not */
                              if(searchBoxId != blockId_grid){
                                                    /* Set Limit of the Box */
                                                    if( searchBoxId != d*d*d-1)
                                                          limitC                                    =
devIndexArrayC[searchBoxId+1];
                                                    else
                                                          limitC = N;
                                                    /* Set start Index of Box to be searched */
                                                    startC = devIndexArrayC[searchBoxId];
                                                    /* Size of searchBoxId */
                                                    blocksSizeC = limitC - startC;
                                                    /* move C to share memory */
                                                    count = 0;
                                                    j = 0;
                                                    while( startC + j*count < limitC){
                                                          for( l = 0; l < 5120 || l < limitC; l++){
//Do not exceed Shared Memory
                                                                s[l]    =    devC[startC    +    l    +
5120*count];
                                                          }
                                                          /*    Synchronize    Threads    to    read
simultaneously from shared memory */
                                                          __syncthreads();
                                    for( i = index; i < limitQ; i += blockStride){
                                      minDistance = devNNdistances[i];
                                      nearestNeighborIndex = devNNindeces[i];
                                                                Query = devQ[i];
                                      /* Calculate shortest distance Of Query from Box on z axis */
                                      if( z == startZ)
                                        distanceZ = abs( Query.z - (z + 1)*interval);
                                      else if( z == blockIdx.z)
                                        distanceZ = 0;
                                      else if( z == endZ)
                                        distanceZ = abs( z*interval - Query.z);
                                      else
                                        printf("Z error: z = %d\n", z);
                                      /* Calculate shortest distance Of Query from Box on y axis */
                                      if( y == startY)
                                        distanceY = abs( Query.y - (y + 1)*interval);
                                      else if( y == blockIdx.y)
                                        distanceY = 0;
                                      else if( y == endY)
                                        distanceY = abs( y*interval - Query.y);
                                      else
                                        printf("Y error: y = %d\n", y);
                                      /* Calculate shortest distance Of Query from Box on y axis */
                                      if( x == startX)
                                        distanceX = abs( Query.x - (x + 1)*interval);
                                      else if( x == blockIdx.x)
                                        distanceX = 0;
                                      else if( x == endX)
                                        distanceX = abs( x*interval - Query.x);
                                      else
                                        printf("X error: x = %d\n", x);
                                      /* Calculate Distance from Block */
                                      blockDistance = sqrt( pow(distanceX, 2) + pow(distanceY, 2) + pow(distanceZ,
2));

                                      if( blockDistance < minDistance){
```

```
                                        /* Search All his Elements */
                                                                                for( k = 0 +
5120*count; k < 5120 || k < blocksSizeC; k++){
                                                                                    distance =
sqrt( pow(Query.x - s[k].x, 2) + pow(Query.y - s[k].y, 2) + pow(Query.z - s[k].z, 2));
                                                                                    if( distance <
minDistance){

        minDistance = distance;
                                                                                        nearestNeighborIndex = k;
                                                                                    }
                                                }
                                                devNNdistances[i] = minDistance;
                                                devNNindeces[i] = nearestNeighborIndex;
                                            }
                                        }
                                                                                count++;
                                                                            }
                                    }
                                }
                            }
                        }
                    }
 }
}
```

7.2 Αλγόριθμος Επαλήθευσης

***Σημείωση: Για τη χρήση του αλγορίθμου με τους δείκτες: χαρακτήρισε ως σχόλιο τη γραμμή σύγκρισης των αποστάσεων και αποχαρακτήρισε την αμέσως επόμενη με τους δείκτες. Κάθε φορά μπορεί να χρησιμοποιηθεί μία από τις δύο επιλογές. .

```
int validateResultsFromFile( struct Element *Q, struct Element *C, float *distances, int *NNindeces,  int N, int *oldIndexQ, int
*oldIndexC)
{

        float *calculatedDistances = (float*)malloc(N*sizeof(float));
        int *calculatedIndeces = (int*)malloc(N*sizeof(int));
        readFloatsFromFile( calculatedDistances, N, "./test/MatlabDistances21.txt");
 readIntegersFromFile( calculatedIndeces, N, "./test/MatlabIndeces21.txt");
        //for(int i = 0; i < N; i++ )
                //printf(" %i %f\n", calculatedIndeces[i], calculatedDistances[i]);
        /*for(int i = 0; i < N; i++){
                printf("%d:Element: [%f %f %f] NN: [%f %f %f] NNdistance:%f, calculatedDistance: %f\n", i, Q[i].x,
Q[i].y, Q[i].z, C[Nindeces[i]].x, C[Nindeces[i]].y, C[Nindeces[i]].z, distances[i], calculatedDistances[oldIndex[i]]);
        }*/
        int error = 0;
 printf("Validation..\n");
        for(int i = 0; i < N; i++){
          if( abs(distances[i] - calculatedDistances[oldIndexQ[i]]) > 0.0001){
   //if( oldIndexC[NNindeces[i]] + 1 != calculatedIndeces[oldIndexQ[i]]){
                        //printf(" Error: C:%f Matlab:%f \n", distances[i], calculatedDistances[oldIndexQ[i]]);
                        //printf("C:%d Matlab:%d\n", oldIndexC[Nindeces[i]]+1, calculatedIndeces[oldIndexQ[i]]);
                        //printf("Failed [New,Old] Q position:[%d,%d] [%f %f %f] NNdistance:%f | NN: C [New,Old]
Position:[%d,%d] [%f %f %f],  calculatedDistance: %f, calculated Matlab Q Index = %d(-1 in my Table)\n",i,
oldIndexQ[i], Q[i].x, Q[i].y, Q[i].z, distances[i], NNindeces[i], oldIndexC[NNindeces[i]], C[NNindeces[i]].x, C[NNindeces[i]].y,
C[NNindeces[i]].z,  calculatedDistances[oldIndexQ[i]], calculatedIndeces[oldIndexQ[i]]);
                        error++;
                }
        }
        printf("ERRORS: %d\n", error);
        free( calculatedDistances);
```

```
  free(calculatedIndeces);
        if(error > 0)
              return 0;

        return 1;
}
```