

HPC Written Assignment 1B Nathan Thomson P141062

by Nathan James Thomson

FILE	11983_NATHAN_JAMES_THOMSON_HPC_WRITTEN_ASSIGNMENT_1B_NATHAN_THOMSON_P141062_107698_572386298.DOCX (1.82M)		
TIME SUBMITTED	08-MAY-2019 10:04PM (UTC+0100)	WORD COUNT	11216
SUBMISSION ID	106698295	CHARACTER COUNT	55922

The Parallelization of K-means algorithm using CUDA

By Nathan James Thomson – P141062

Contents

Abstract.....	6
Introduction	7
What is K-Means algorithm?.....	8
What data will I provide my K-means algorithm?.....	8
What is Euclidian Distance checking?	8
Euclidian Distance in 2 dimensional space.....	9
Euclidian Distance in 3 dimensional space.....	10
Why is this algorithm suitable to be parallelized by cuda?.....	11
Implementation	12
CPU Implementation.....	12
GPU Implementation	17
Program running in 2D.....	23
Program running in 3D.....	24
Performance Profile.....	25
PC Specifications	25
2D Results	26
512 Points, 2 clusters	26
512 Points, 5 clusters	26
1024 Points, 2 clusters	27
1024 points, 5 clusters	27
25600 Points CPU 2 clusters	28
25600 Points CPU 5 clusters	28
3D Results	29
512 Points 2 clusters	29
512 Points 5 clusters	29
1024 Points 2 clusters	30

1024 Points 5 clusters	30
25600 Points 2 clusters	31
25600 Points 5 clusters	31
Results Evaluation	32
2D Results Evaluation.....	32
3D Results Evaluation.....	33
Conclusion & Future work	34
References.....	35
Appendix - CPU implementation code	36
_Cluster.h	36
_KMeans.h	37
_Math.h	38
_Point.h	39
RNG.h	40
STDAFX.h	41
_Cluster.cpp	42
_KMeans.cpp	44
_Math.cpp	53
_Point.cpp	55
RNG.cpp	58
main.cpp	61
Appendix - GPU implementation code	62
_Cluster.h	62
_Math.h	63
_Point.h	64
RNG.h	65
_Cluster.cpp	66
_Math.cpp	68
_Point.cpp	70
RNG.cpp	73
Kernel.cu	77

Table of Figures

Figure 1 - Picture of structure used for points Implementation	12
Figure 2 - Declaration of array that was used to hold the points structures implementation	12
Figure 3- Picture of structure used for clusters Implementation	13
Figure 4 - Declaration of array that was used to hold the clusters structures implementation	13
Figure 5 - Dynamically creating the size of the array of points, to however many the user wishes to have	13
Figure 6 - RNG X co-ordinate Implementation	14
Figure 7- RNG Y Co-Ordinate Implementation	14
Figure 8-RNG Z Co-Ordinate Implemen	14
Figure 1 - Picture of structure used for points Implementation	12
Figure 2 - Declaration of array that was used to hold the points structures implementation	12
Figure 3 - Dynamically creating the size of the array of points, to however many the user wishes to have	12
Figure 4- Picture of structure used for clusters Implementation	13
Figure 5 - Declaration of array that was used to hold the clusters structures implementation	13
Figure 6 - Dynamically creating the size of the array of clusters, to however many the user wishes to have	13
Figure 7 - RNG X co-ordinate Implementation	14
Figure 8- RNG Y Co-Ordinate Implementation	14
Figure 9-RNG Z Co-Ordinate Implementation	15
Figure 10 - 2D Euclidian Distance Equation Implementation	15
Figure 11- 3D Euclidian Distance Equation Implementation	16
Figure 12 - 2D K-means CPU implementation	16
Figure 13 - 3D K-means CPU implementation	17
Figure 14- 2D Euclidian Distance Device code	18
Figure 15 - 3D Euclidian Distance Device Code	18
Figure 16 - Setting up block and number of threads per block	18
Figure 17 - setting i to the correct point	18
Figure 18 - setting up buffers	19
Figure 19 - allocation of buffers	19
Figure 20 - copying data into the buffers	20
Figure 21 - calling the kernel	20
Figure 22 - freeing the buffers	20
Figure 23 - kernel function prototype	21
Figure 24 - copying data back from kernel to host	21
Figure 25 - 2D k-means kernel implementation	21
Figure 26 - 3D k-means kernel implementation	22
tation	15
Figure 9 - 2D Euclidian Distance Equation Implementation	15
Figure 10- 3D Euclidian Distance Equation Implementation	16
Figure 11- 2D Euclidian Distance Device code	18
Figure 12 - 3D Euclidian Distance Device Code	18



Abstract

The project identifies what K-Means algorithm is, implements k-means and tests the effectiveness of CUDA on the part of the algorithm that was parallelized and ultimately compares the speed the programs.

The project successfully implemented the CPU and GPU versions of the K-Means algorithm, and it was documented in this report. The project also compared the time it took to run both the CPU and GPU versions and evaluated the results found.

Introduction

The aim of the project is to implement the K-Means algorithm in a developed environment, and to speed up the efficiency of the K-means algorithm by using CUDA parallelization.

Firstly, research must be done into an algorithms that the K-means algorithm can utilizes and decide upon a suitable algorithm for the implementation and a suitable part of the algorithm for parrallization.

The next step in the project was to develop the K-Means algorithm using the CPU, and thus following this the next step in the project was to develop the project using the GPU using CUDA parallelization.

Following the implementation the project final step was to using NVIDEA's profiler, identify the speed of running the program on just the CPU or just the GPU and compare the results.

What is K-Means algorithm?

"K-means clustering is a type of unsupervised learning, which is used when you have unlabeled data (i.e., data without defined categories or groups). The goal of this algorithm is to find groups in the data, with the number of groups represented by the variable K . The algorithm works iteratively to assign each data point to one of K groups based on the features that are provided. Data points are clustered based on feature similarity." [1] What Andrea Trevino states in this quote, is that the K-means algorithm is useful for grouping similar data together, it is a machine learning algorithm in which it only receives data, and trawls this data and groups patches of the data provided using the feature that was required by the user.

What data will I provide my K-means algorithm?

Games have three dimensional space capabilities in modern times alongside having capability to function in two dimensional space, as a result, it will be important to be able to find the distance between two points in either two dimensional or three dimensional space. It may be beneficial for the game to be able to group an area of points into a "cluster", the label for this cluster could be something such as a safe zone, or save point. For this distance checking will be required, as a result contextualising my project into games, the implementation will be aiming to group a series of points in two or three dimensional space into clusters.

What is Euclidian Distance checking?

"The Euclidean distance or Euclidean metric is the "ordinary" straight-line distance between two points in Euclidean space. With this distance, Euclidean space becomes a metric space." [2] What this refers to, is that the Euclidian distance is the direct distance, from point a, to point b. The Euclidian distance is as the crow flies, meaning it does not take into account any data other than the point's co-ordinates. Euclidian distance equations have been detailed below, there is a different equation for both two Dimensional space and three dimensional space.

Euclidian Distance in 2 dimensional space

$$d(\mathbf{p}, \mathbf{q}) = \sqrt{(q_1 - p_1)^2 + (q_2 - p_2)^2}.$$

Equation 1- Two Dimensional Space equation [2]

What the equation above is documenting, is how to calculate the Euclidian distance in two dimensional space. Some arbitrary values have been assigned to point P and point Q in order to demonstrate how the two dimensional formula works.

Point 'P' has the x co-ordinate of 122.4, and the y co-ordinate of 64.7

Point 'Q' has the x co-ordinate of 223.7, and the y co-ordinate of 172.2

In order to find the Euclidian distance in two dimensional space, the values will need to be substituted into the equation, the resulting equation would look like the following. 'p1' & 'q1' will refer to the x values of either point and 'p2' & 'q2' will refer to the y values of each point.

$$d(p, q) = \sqrt{(223.7 - 122.4)^2 + (172.2 - 64.7)^2}$$

=

$$d(p, q) = \sqrt{(101.3)^2 + (107.5)^2}$$

=

$$d(p, q) = \sqrt{10261.69 + 11556.25}$$

$$d(p, q) = \sqrt{21817.94}$$

=

$$d(p, q) = 147.70897$$

The resulting distance between point p and point q in two dimensional space with the arbitrary values provided is 147.70897

Euclidian Distance in 3 dimensional space

$$d(p, q) = \sqrt{(p_1 - q_1)^2 + (p_2 - q_2)^2 + (p_3 - q_3)^2}.$$

Equation 2 Three Dimensional Space equation [2]

What the equation above is documenting, is how to calculate the Euclidian distance in three dimensional space. Some arbitrary values have been assigned to point P and point Q in order to demonstrate how the three dimensional formula works.

Point 'P' has the x co-ordinate of 122.4, the y co-ordinate of 64.7 and the z co-ordinate of 35.2

Point 'Q' has the x co-ordinate of 223.7, the y co-ordinate of 172.2 and the z co-ordinate of 9.2

In order to find the Euclidian distance in two dimensional space, the values will need to be substituted into the equation, the resulting equation would look like the following. 'p1' and 'q1' will refer to the x values of either point, 'p2' and 'q2' will refer to the y values of each point and 'p3' and 'q3' will refer to the z values of each of the points.

$$d(p, q) = \sqrt{(122.4 - 223.7)^2 + (64.7 - 172.2)^2 + (35.2 - 9.2)^2}$$

$$=$$

$$d(p, q) = \sqrt{10261.69 + 11556.25 + 676}$$

$$=$$

$$d(p, q) = \sqrt{22493.94}$$

$$=$$

$$d(p, q) = 149.97979$$

The resulting distance between point p and point q in three dimensional space with the arbitrary values provided is 149.97979

Why is this algorithm suitable to be parallelized by cuda?

Considering all the above, this algorithm is suitable for parallelization using CUDA, the reasoning behind this is that the distance check equations have a lot of individual functions that need to be carried out, for example, each x, y, z value from points when calculating the distance need to be subtracted from the other points x,y,z values. After this the resulting numbers need to be squared. Following this the resulting numbers left in the equation need to be added together to get the squared distance, before finally square rooting the number to get the absolute value of distance. Performing all of these for every point, could make the program run slower on the CPU rather than multi-threaded on the GPU on many threads and as a result being more efficient as the CPU would only be able to do one point at a time, thus being time consuming.

Another reasoning behind this algorithm being suitable for parallelization by CUDA is that as it is an unsupervised algorithm, there is not a lot of shared memory.

Implementation

CPU Implementation

In the CPU implementation, the project has an array of structures for the points, and a separate array of structures for the clusters. Each structure has an X, Y, Z coordinate representing the points and clusters place in 2D or 3D space. The number of points is inputted via the user on runtime, and using this the program groups the points into clusters based on their point in 3D space.

```
// structure for each point
struct Point
{
    int pointId;
    float x;
    float y;
    float z;
    float cGroup; // which group do they belong to?
};
```

Figure 1 - Picture of structure used for points Implementation

```
Point* pointArray; // declare an array that will be used to hold the structs
```

Figure 2 - Declaration of array that was used to hold the points structures implementation

```
void _Point::generatePoints(int i)
{
    // dynamically create array size for array declared in header using argument passed in
    pointArray = new Point[i]; // ARRAY OF STRUCTURES
}
```

Figure 3 - Dynamically creating the size of the array of points, to however many the user wishes to have

```
// structure for each cluster
struct Cluster
{
    int clusterID;
    float x;
    float y;
    float z;
};
```

Figure 4- Picture of structure used for clusters Implementation

```
Cluster * Clusters; // declare an array that will be used to hold the structs
```

Figure 5 - Declaration of array that was used to hold the clusters structures implementation

```
void _Cluster::generateClusters(int i)
{
    // dynamically create array size for array declared in header using argument passed in
    Clusters = new Cluster[i]; // ARRAY OF STRUCTURES
}
```

Figure 6 - Dynamically creating the size of the array of clusters, to however many the user wishes to have

Each point gets randomly assigned a number, using a random number generator that I have developed.

```

1 // Load RNG source (Fisher's)
2 s = Rg
3
4 // makes more sense to my head if its 0-500 than minus 250
5 // 0 is the first time running
6 if (c = 0) do { prev1 = 0; prev2 = 0; prev3 = 0; prev4 = 0; prev5 = 0; }
7 {
8     x = randi(5, (500, 500) - prev1, prev2);
9
10    else
11
12    //generate random values between -250 and 250
13    x = randi(5, (500, 500) - prev1, prev2);
14
15    // if it is the value of the previous 3's's, regenerate until it isn't
16    while (c == prev1 || c == prev2 || c == prev3 || c == prev4 || c == prev5)
17    {
18        x = randi(5, (500, 500) - prev1, prev2);
19        c = prev1 || c == prev2 || c == prev3 || c == prev4 || c == prev5
20        x = prev1 || c == prev2 || c == prev3 || c == prev4 || c == prev5
21        x = randi(5, (500, 500) - prev1, prev2);
22    }
23
24    // assign the values to the next X point.
25    prev5 = prev4;
26    prev4 = prev3;
27    prev3 = prev2;
28    prev2 = prev1;
29    prev1 = x;
30
31    return x;
32 }

```

Figure 7 - RNG X co-ordinate Implementation

[illegible]

Figure 8- RNG Y Co-Ordinate Implementation

[illegible]

Figure 9-RNG Z Co-Ordinate Implementation

Using the random numbers generated, the program, tests each points distance to each cluster using the Euclidian distance equation(s) that were illustrated earlier in the report.

```
// 2D Math euclidean Distance formula
float Math::euclideanDist(float Px, float Qx, float Py, float Qy)
{
    // formula = sqrt(((P1 - Q1)^2) + ((P2 - Q2)^2))

    double dist;

    double x = (Px - Qx);
    double y = (Py - Qy);

    //cout << "x = " << x << endl;
    //cout << "y = " << y << endl;

    x *= x;
    y *= y;

    //cout << "x = " << x << endl;
    //cout << "y = " << y << endl;

    dist = x + y;
    //cout << dist << endl;

    dist = sqrt(dist);
    //cout << dist << endl;

    return dist;
}
```

Figure 10 - 2D Euclidian Distance Equation Implementation

Figure 11- 3D Euclidian Distance Equation Implementation

[illegible]

Figure 12 - 2D K-means CPU implementation

Implementation of K-means algorithm in 3D

[illegible]

Figure 13 - 3D K-means CPU implementation

GPU Implementation

The GPU implementation had all the functionality of the CPU implementation, however was refactored and functions were added in order to make it able to be used by CUDA.

In the kernel.cu file, I had as a device, the Euclidian Distance functions. This is due to the kernel being unable to access the functions outside of scope. The kernel had in it, the k-means implementation as this is what was being serialized, it was set up in such a way to allow each point to run on its own thread, thus running all points simultaneously. The screenshots below show the important code that was added to allow this to work. The K-means class was removed, and the functions were implemented into the kernel.cu file, as kernel will not be able to access functions outside of its scope, as a result, functions will as mentioned previously need to be set up as devices.

```

// Function wraps up a device function in order to access the function from the kernel
// All variables of euclidean distance
__device__ float euclideanDist(float Px, float Py, float Pz, float Qx, float Qy, float Qz) // FIND THE DISTANCE IN 3 DIMENSIONS
{
    // Formula = sqrt((Px - Qx)^2 + (Py - Qy)^2 + (Pz - Qz)^2)

    float dist;

    double x = (Px - Qx);
    double y = (Py - Qy);
    double z = (Pz - Qz);

    //double x1 = x * x + y * y + z * z;
    //double x1 = x * x + y * y + z * z;

    x *= x;
    y *= y;
    z *= z;

    //double x1 = x + y + z;
    //double x1 = x + y + z;

    dist = x + y;
    //double x1 = dist + z;

    dist = sqrt(dist);
    //double x1 = dist + z;

    return dist;
}

```

Figure 14- 2D Euclidian Distance Device code

```

// Function wraps up a device function in order to access the function from the kernel
// All variables of euclidean distance
__device__ float euclideanDist(float Px, float Py, float Pz, float Qx, float Qy, float Qz, float Qz) // FIND THE DISTANCE IN 3 DIMENSIONS
{
    // Formula = sqrt((Px - Qx)^2 + (Py - Qy)^2 + (Pz - Qz)^2)

    float dist;

    double x = (Px - Qx);
    double y = (Py - Qy);
    double z = (Pz - Qz);

    x *= x;
    y *= y;
    z *= z;

    dist = x + y;
    //double x1 = dist + z;

    dist = sqrt(dist);
    //double x1 = dist + z;

    return dist;
}

```

Figure 15 - 3D Euclidian Distance Device Code

The Kernel also sets the number of blocks equal to the amount needed, this is done by having one block of 512 threads for every 512 points.

```

// number of threads per block
int threadsPerBlock = 512;
// blocks
dim3 block;

```

Figure 16 - Setting up block and number of threads per block

The code below allows 'i' to be set to the thread it should be correctly, this allows the thread to be to the point it should be, every point is its own thread.

```

//set i to the thread number, the thread number + block id allows the thread to be correctly assigned to correct point
int i = blockIdx.x * blockDim.x + threadIdx.x;

```

Figure 17 - setting i to the correct point

Below shows the buffers being set up, for the variables and arrays that will need to be passed through into the kernel.

```
// Helper function for using CUDA to add vectors in parallel.
cudaError_t addWithCuda(int *c, const int *a, const int *b, unsigned int size, int num0)
{
    int *dev_a = 0;
    int *dev_b = 0;
    int *dev_c = 0;

    //devices to be used in allowing me to pass the data I require to be passed to the kernel
    int *dev_Updated = 0;
    bool *dev_Convergence = 0;
    Point *dev_PointArray = 0;
    Cluster *dev_Clusters = 0;

    cudaError_t cudaStatus;
```

Figure 18 - setting up buffers

Below is showing the GPU buffers being allocated for the variables that will be passed through to the kernel.

```
// Allocate GPU buffers for point array.
cudaStatus = cudaMalloc((void**) &dev_PointArray, num0 * sizeof(Point));
if (cudaStatus != cudaSuccess) {
    fprintf(stderr, "cudaMalloc failed!");
    goto Error;
}

// Allocate GPU buffers for cluster array
cudaStatus = cudaMalloc((void**) &dev_Clusters, numc * sizeof(Cluster));
if (cudaStatus != cudaSuccess) {
    fprintf(stderr, "cudaMalloc failed!");
    goto Error;
}

// Allocate GPU buffers for updated variable stored in array
cudaStatus = cudaMalloc((void**) &dev_Updated, sizeof(int));
if (cudaStatus != cudaSuccess) {
    fprintf(stderr, "cudaMalloc failed!");
    goto Error;
}

// Allocate GPU buffers for convergence variable stored in array
cudaStatus = cudaMalloc((void**) &dev_Convergence, sizeof(bool));
if (cudaStatus != cudaSuccess) {
    fprintf(stderr, "cudaMalloc failed!");
    goto Error;
}
```

Figure 19 - allocation of buffers

Below shows the data being copied into the GPU buffers that were set up

```
// Copy Point array from host memory to GPU buffers.
cudaStatus = cudaMemcpy(dev_PointArray, _point.pointArray, num * sizeof(Point), cudaMemcpyHostToDevice);
if (cudaStatus != cudaSuccess) {
    printf(stderr, "cudaMemcpy failed!");
    goto Error;
}

// Copy cluster array from host memory to GPU buffers.
cudaStatus = cudaMemcpy(dev_Clusters, _cluster.Clusters, numC * sizeof(Cluster), cudaMemcpyHostToDevice);
if (cudaStatus != cudaSuccess) {
    printf(stderr, "cudaMemcpy failed!");
    goto Error;
}

// Copy updated array from host memory to GPU buffers.
cudaStatus = cudaMemcpy(dev_Updated, updated, sizeof(int), cudaMemcpyHostToDevice);
if (cudaStatus != cudaSuccess) {
    printf(stderr, "cudaMemcpy failed!");
    goto Error;
}

// Copy convergence array from host memory to GPU buffers.
cudaStatus = cudaMemcpy(dev_Convergence, convergence, sizeof(bool), cudaMemcpyHostToDevice);
if (cudaStatus != cudaSuccess) {
    printf(stderr, "cudaMemcpy failed!");
    goto Error;
}
```

Figure 20 - copying data into the buffers

Below shows the kernel being launched, with all the buffers passed through to the kernel

```
// Launch a kernel on the GPU with one thread for each element. pass in point array, cluster array and variables needed
// kernel uses the block.x variable that was set earlier in the file for number of blocks, and the threads per block to state how many threads per block
distanceChecksKernel <<< block.x, threadsPerBlock >>>(dev_PointArray, dev_Clusters, desiredDimension, convergence[0], updated[0], numC);
// Check for any errors launching the kernel
```

Figure 21 - calling the kernel

Below shows the code to free up memory that were allocated to the buffers

```
Error:
    cudaFree(dev_c);
    cudaFree(dev_a);
    cudaFree(dev_b);

    //free the memory
    cudaFree(dev_PointArray);
    cudaFree(dev_Clusters);
    cudaFree(dev_Convergence);
    cudaFree(dev_Updated);

    return cudaStatus;
}
```

Figure 22 - freeing the buffers

Below shows the function prototype for the kernel which contains the k-means algorithm implementation.

```
__global__ void distanceChecksKernel(Point* pointArray, Cluster* Clusters, int desiredDimension, bool convergence, int updated, int numC)
{
```

Figure 23 - kernel function prototype

Below shows the updated information being copied back from the buffers back into their original source.

```
// Copy updated point array back into point array
cudaStatus = cudaMemcpy_Point_pointArray, dev_pointArray, num * sizeof(Point), cudaMemcpyDeviceToHost);
if (cudaStatus != cudaMemcpySuccess) {
    fprintf(stderr, "cudaMemcpy failed!");
    goto Error;
}

// Copy updated cluster array back into cluster array
cudaStatus = cudaMemcpy_Kcluster.Clusters, dev_clusters, num * C * sizeof(Cluster), cudaMemcpyDeviceToHost);
if (cudaStatus != cudaMemcpySuccess) {
    fprintf(stderr, "cudaMemcpy failed!");
    goto Error;
}

// Copy updated value back into updated variable
cudaStatus = cudaMemcpy(updated, dev_updated, sizeof(int), cudaMemcpyDeviceToHost);
if (cudaStatus != cudaMemcpySuccess) {
    fprintf(stderr, "cudaMemcpy failed!");
    goto Error;
}
```

Figure 24 - copying data back from kernel to host

Below shows the refactored k-means cpu code to work with the kernel for 2D.

[illegible]

Figure 25 - 2D k-means kernel implementation

Below shows the refactored k-means cpu code to work with the kernel for 3D.

```
case B//D
// If the distance to cluster 1 centroid is greater than the distance
for (clusterIndex = 0; clusterIndex < numClusters; clusterIndex++)
{
    // Get the centroid
    Px = centroidVec[1][x]; Py = centroidVec[1][y]; Pz = centroidVec[1][z];

    // Get the centroid
    currentCluster = centroidVec[1].id;
    Qx = Centroid[currentCluster][x]; Qy = Centroid[currentCluster][y]; Qz = Centroid[currentCluster][z];

    // Get the centroid
    Mx = Centroid[currentCluster][x]; My = Centroid[currentCluster][y]; Mz = Centroid[currentCluster][z];

    // If the distance from point to current cluster centroid is larger than the distance from point to potential new cluster centroid
    // (Centroid[currentCluster][x], Qy, Qy, Qz) > Centroid[currentCluster][x], My, My, Mz)
    {
        // Set the new cluster to be the potential new cluster id
        centroidVec[1].id = currentCluster;
        // Print to screen that it has been changed from cluster 1 to currentCluster or "a cluster" as currentCluster is null.
        // A point has been added to current variable
    }
}
// End of loop
// Assigning a new
// Print to "Screen" if null, the the program is null.
break;
}
default: // This should never happen
// Print to "Screen" if null, it would be impossible for the dimension since there is no 3D to 3D, only from 3D to 3D.
break;
}
```

Figure 26 - 3D k-means kernel implementation

Program running in 2D



```
C:\Users\JH82\Desktop>ls -l *.exe
ls.exe             -rwxr-xr-x    10    01/01/2006
C:\Users\JH82\Desktop>ls -l *.exe
ls.exe             -rwxr-xr-x    10    01/01/2006
C:\Users\JH82\Desktop>ls -l *.exe
ls.exe             -rwxr-xr-x    10    01/01/2006
```

[illegible]

Program running in 3D

```
C:\Users\14102\Desktop\Hand-In\VC\GPU\Lab\Debug\VCGPU.exe
Warning: Attention would you like the program to be performed in
0: No
1: Yes
Please select how many points you would like to generate
112, 1813 or 11000
112
```

```
C:\Users\14102\Desktop\Hand-In\VC\GPU\Lab\Debug\VCGPU.exe
Point [400] : Cluster group = 1 , x = 100 , y = -15 , z = -60 ,
Point [401] : Cluster group = 1 , x = 116 , y = -20 , z = -89 ,
Point [402] : Cluster group = 1 , x = 104 , y = 197 , z = -201 ,
Point [403] : Cluster group = 1 , x = 170 , y = -50 , z = -66 ,
Point [404] : Cluster group = 1 , x = 72 , y = 168 , z = 76 ,
Point [501] : Cluster group = 1 , x = 110 , y = 230 , z = -189 ,
Point [502] : Cluster group = 1 , x = 137 , y = -244 , z = -150 ,
Point [503] : Cluster group = 1 , x = -157 , y = 39 , z = -152 ,
Point [504] : Cluster group = 1 , x = 100 , y = -54 , z = -92 ,
Point [505] : Cluster group = 1 , x = -205 , y = 213 , z = -211 ,
Point [506] : Cluster group = 1 , x = -155 , y = -230 , z = -227 ,
Point [507] : Cluster group = 1 , x = -83 , y = -200 , z = -216 ,
Point [508] : Cluster group = 1 , x = 106 , y = -61 , z = -123 ,
Point [509] : Cluster group = 1 , x = -92 , y = -39 , z = -189 ,
Point [510] : Cluster group = 1 , x = -105 , y = 6 , z = -82 ,
Point [511] : Cluster group = 1 , x = 172 , y = 8 , z = 184 ,
Please select how many clusters you would like to generate
1 or 5
1
Cluster [0] : x = 37 , y = -155 , z = -181 ,
Cluster [1] : x = 130 , y = 184 , z = 5 ,
[1,2,3,4,5] : [10,20,30,40,50] : [0,0,0,0,0]
Initialization complete
Iteration 0 complete
Should be exiting the program
The points that are in cluster 0 are:
0, 5, 7, 8, 9, 12, 14, 16, 19, 21, 28, 35, 38, 46, 51, 55, 57, 59, 62, 73, 80, 90, 96, 98, 100, 103, 105, 108, 109, 111,
112, 116, 121, 123, 128, 129, 130, 131, 139, 140, 144, 149, 147, 146, 150, 151, 152, 153, 160, 161, 164, 165,
166, 167, 168, 169, 170, 177, 179, 180, 181, 186, 188, 189, 194, 195, 200, 201, 202, 205, 206, 209, 210, 212, 217, 219,
220, 222, 223, 225, 227, 228, 229, 230, 232, 234, 238, 240, 243, 244, 245, 247, 248, 250, 252, 252, 256, 259, 260,
267, 268, 271, 276, 279, 281, 283, 286, 291, 294, 298, 299, 300, 306, 307, 309, 310, 311, 314, 317, 319, 321, 323, 326,
328, 330, 332, 336, 337, 341, 342, 343, 345, 347, 355, 356, 357, 360, 362, 363, 364, 366, 370, 372, 376, 378, 384,
385, 386, 387, 389, 394, 396, 398, 399, 402, 408, 409, 410, 411, 415, 416, 417, 420, 423, 426, 433, 434, 435,
445, 446, 450, 451, 452, 457, 462, 463, 465, 466, 472, 473, 478, 479, 480, 483, 487, 488, 490, 492, 493, 500, 511,
The points that are in cluster 1 are:
1, 2, 3, 4, 6, 10, 11, 13, 15, 17, 18, 20, 22, 23, 24, 25, 26, 27, 29, 30, 31, 32, 33, 34, 36, 37, 39, 40, 41, 42, 43, 44,
45, 47, 48, 49, 50, 52, 53, 54, 55, 59, 60, 61, 63, 64, 65, 66, 67, 68, 69, 70, 71, 72, 74, 75, 76, 77, 78, 79, 81, 82,
83, 84, 86, 88, 89, 91, 92, 93, 94, 95, 97, 99, 101, 102, 104, 106, 107, 108, 113, 115, 116, 117, 118, 119, 120,
122, 125, 126, 127, 132, 133, 134, 135, 136, 139, 141, 142, 143, 146, 149, 153, 154, 157, 158, 159, 163, 169, 171,
172, 175, 176, 178, 179, 182, 183, 184, 185, 187, 190, 193, 192, 193, 196, 197, 198, 199, 204, 206, 207, 208,
211, 213, 215, 216, 218, 219, 221, 224, 226, 230, 231, 235, 236, 237, 239, 241, 242, 246, 249, 251, 254, 255, 257, 259, 261,
262, 263, 264, 265, 269, 270, 273, 274, 275, 277, 278, 280, 282, 283, 284, 286, 287, 288, 293, 295, 296, 297,
299, 301, 302, 303, 304, 305, 306, 312, 313, 315, 316, 318, 320, 322, 324, 325, 327, 329, 331, 332, 333, 334, 338, 340,
342, 344, 346, 348, 349, 350, 351, 352, 353, 354, 355, 356, 357, 358, 359, 361, 365, 368, 371, 372, 374, 376, 377, 379, 381,
382, 383, 388, 389, 391, 392, 393, 394, 395, 396, 397, 398, 400, 401, 404, 406, 407, 412, 413, 414, 418, 419, 421, 421, 424, 425, 426,
427, 429, 430, 431, 432, 434, 435, 436, 437, 438, 439, 441, 442, 443, 444, 445, 446, 447, 448, 449, 453, 453, 454, 455, 456, 457, 458,
459, 461, 464, 466, 467, 469, 470, 471, 474, 475, 476, 477, 481, 482, 484, 485, 486, 488, 491, 494, 495, 496, 497, 498, 499,
500, 501, 502, 503, 504, 505, 506, 507, 509, 510,
```

Performance Profile

All tests were done using the following computer specifications. The profile tests were done using CUDA's performance profilers, performance wizard, using the instrumentation option.

PC Specifications

MY RIG

 GeForce GTX 1080 Ti

 Driver version 388.13

 Intel(R) Core(TM) i7-7700 CPU @ 3.60GHz

 31.96 GB RAM

 2560 x 1440, 59Hz

2D Results

512 Points, 2 clusters

	CPU	GPU
Test 1	2.4 seconds	3.9 seconds
Test 2	2.6 seconds	4.0 seconds
Test 3	2.5 seconds	3.8 seconds
Average	2.5 seconds	3.9 seconds

512 Points, 5 clusters

	CPU	GPU
Test 1	2.2 seconds	4.1 seconds
Test 2	2.1 seconds	3.6 seconds
Test 3	2.2 seconds	3.5 seconds
Average	2.2 seconds	3.7 seconds

1024 Points, 2 clusters

	CPU	GPU
Test 1	2.2 seconds	4.5 seconds
Test 2	3.2 seconds	4.4 seconds
Test 3	2.8 seconds	3.8 seconds
Average	2.7 seconds	4.2 seconds

1024 points, 5 clusters

	CPU	GPU
Test 1	2.9 seconds	3.7 Seconds
Test 2	3.2 seconds	4.3 Seconds
Test 3	3.3 seconds	4.0 Seconds
Average	3.1 seconds	4 seconds

25600 Points CPU 2 clusters

	CPU	GPU
Test 1	21.3 seconds	23.5 Seconds
Test 2	20.4 seconds	21.9 Seconds
Test 3	21.1 seconds	22.7 Seconds
Average	20.9 seconds	22.7 seconds

25600 Points CPU 5 clusters

	CPU	GPU
Test 1	22.9 seconds	22.8 Seconds
Test 2	21.8 seconds	21.9 Seconds
Test 3	22.8 seconds	22.7 Seconds
Average	22.5 seconds	22.5 seconds

3D Results

512 Points 2 clusters

	CPU	GPU
Test 1	2.6 seconds	2.7 seconds
Test 2	2.4 seconds	2.7 seconds
Test 3	1.9 seconds	3.2 seconds
Average	2.3 seconds	2.9 seconds

512 Points 5 clusters

	CPU	GPU
Test 1	2.2 seconds	2.8 seconds
Test 2	2.0 seconds	3.6 seconds
Test 3	1.9 seconds	2.9 seconds
Average	2.0 seconds	3.1 seconds

1024 Points 2 clusters

	CPU	GPU
Test 1	2.8 seconds	4.0 seconds
Test 2	3.2 seconds	4.3 seconds
Test 3	2.6 seconds	3.9 seconds
Average	2.9 seconds	4.1 seconds

1024 Points 5 clusters

	CPU	GPU
Test 1	2.7 seconds	3.8 seconds
Test 2	2.5 seconds	4.4 seconds
Test 3	2.7 seconds	3.5 seconds
Average	2.6 seconds	3.9 seconds

25600 Points 2 clusters

	CPU	GPU
Test 1	21.0 seconds	22.3 seconds
Test 2	21.2 seconds	22.1 seconds
Test 3	20.6 seconds	22.6 seconds
Average	20.9 seconds	22.3 seconds

25600 Points 5 clusters

	CPU	GPU
Test 1	21.0 seconds	22.7 seconds
Test 2	21.8 seconds	21.4 seconds
Test 3	22.0 seconds	21.6 seconds
Average	21.6 seconds	21.9 seconds

Results Evaluation

2D Results Evaluation

	CPU	GPU
512 points, 2 clusters	2.5 seconds	3.9 seconds
512 points, 5 clusters	2.2 seconds	3.7 seconds
1024 points, 2 clusters	2.7 seconds	4.2 seconds
1024 points, 5 clusters	3.1 seconds	4 seconds
25600 points, 2 clusters	20.9 seconds	22.7 seconds
25600 points, 5 clusters	22.5 seconds	22.5 seconds

To conclude the 2D CPU and GPU results, surprisingly, whether the project uses enough points to run on one block of 512 threads of fifty blocks of 512 threads, the CPU implementation and the CPU has results which indicate that the implementation that has been done using CUDA does not speed up the program, it has the opposite effect. The GPU was slower.

3D Results Evaluation

	CPU	GPU
512 points, 2 clusters	2.3 seconds	2.9 seconds
512 points, 5 clusters	2.0 seconds	3.1 seconds
1024 points, 2 clusters	2.9 seconds	4.1 seconds
1024 points, 5 clusters	2.6 seconds	3.9 seconds
25600 points, 2 clusters	20.9 seconds	22.3 seconds
25600 points, 5 clusters	21.6 seconds	21.9 seconds

To conclude the 3D CPU and GPU results, as above in the 2D results, whether the project uses enough points to run on one block of 512 threads of fifty blocks of 512 threads, the CPU implementation and the CPU has results which indicate that the implementation that has been done using CUDA does not speed up the program, it has the opposite effect. The GPU was slower.

Conclusion & Future work

The result of the project was not what I expected at the start of the project, I had expected at the start of the project that due to the set up required in setting up CUDA to work, that CUDA parallelisation would be slower on the smaller amount of point numbers, but I had expected that using a large amount of points, for example in the project I used 25600, enough for fifty blocks of 512 threads, that the GPU would have been quicker. It may be due to the way that the parallelization was implemented, or it may be due to not enough points being used. Further testing should be done.

Future work in the project should look to parallelizing more areas of the project, in order to test if serializing more areas of the project would improve efficiency, for example, generating all the points on the CPU in both the GPU and CPU version takes a lot of time, Further future work would be to add visual representation of the points, using OpenGL and using a CUDA kernel draw each point with the same amount of blocks and threads that was needed to perform the distance checks.

References

- [1] A. Trevino, "Introduction to K-means Clustering," 6 December 2016 2019. [Online]. Available: <https://www.datascience.com/blog/introduction-to-k-means-clustering-algorithm-learn-data-science-tutorials>. [Accessed 1 January 2019].
- [2] "Euclidian Distance," [Online]. Available: https://en.wikipedia.org/wiki/Euclidean_distance. [Accessed 1 January 2019].
- [3] "Step-By-Step K-Means Example," [Online]. Available: <http://mnemstudio.org/clustering-k-means-example-1.htm>. [Accessed 06 May 19].
- [4] P. Baret, "Euclidean Distance," [Online]. Available: <http://www.pbarrett.net/techpapers/euclid.pdf>.

Appendix - CPU implementation code

_Cluster.h

```
#pragma once
#include "STDAFX.h"
#include "RNG.h"

class _Cluster
{
private:
    // structure for each cluster
    struct Cluster
    {
        int clusterID;
        float x;
        float y;
        float z;
    };

    void generateClusters(int i); // how many clusters as argument
    void RNGCoOrds(int i); // take size of array (amount of clusters) as argument generate random point co-ordinates

public:
    _Cluster();
    ~_Cluster();

    Cluster * Clusters; // declare an array that will be used to hold the structs
    int uiCluster;
    void printClusters();
    void selectMode(); //select an amount of clusters to be done. (2||3||5||10) this function will call other functions depending on input

};
```

_KMeans.h

```
#pragma once
// includeS
#include "STDAFX.H"
// include class names
#include "_Cluster.h"
#include "_Point.h"
#include "_Math.h"

class _KMeans
{
public:
    _KMeans();
    ~_KMeans();
    void kmeans(); // does the algorithm

private:
    int desiredDimension; // the desired dimension variable
    int pointIndex; // the point index used in distance checking
    int clusterIndex; // cluster index, used in distance checking
    int maxIteration; // maximum amount of iterations
    int iterationIndex = 0; // current iteration number

    int updated; // if points have swapped
    bool convergence; // has the algorithm found convergence?

    void initialization(); // performs generation of points & clusters
    void iteration(); // iteration of the algorithm
    void distanceChecks(); // performs distance checks from points to clusters
    void recalculatClusterMeans(); // recalculate cluster means
    void endProgram(); // end the program

    void recalculatX(); // recalculate the X value of cluster centroids
    void recalculatY(); // recalculate the Y value of cluster centroids
    void recalculatZ(); // recalculate the Z value of cluster centroids

    void printClusterGroups(); // prints a list of points in each cluster
};
```

Math.h

```
#pragma once
#include <math.h>
#include "STDAFX.h"
class _Math
{
public:
    _Math();
    ~_Math();

    float euclidianDist(float Px, float Qx, float Py, float Qy); // FIND THE DISTANCE IN 2 DIMENSIONS
    float euclidianDist(float Px, float Qx, float Py, float Qy, float Pz, float Qz); // FIND THE DISTANCE IN 3 DIMENSIONS
};
```

_Point.h

#pragma once

class _Point
{

private :

// structure for each point

struct Point

{

int pointId;

float x;

float y;

float z;

float cGroup; // which group do they belong to?

};

void generatePoints(int i); // how many points as argyment

void genRandomPoints(int i); // take size of array as argument generate random point co-ordinates

public:

_Point();

~_Point();

Point* pointArray; // declare an array that will be used to hold the structs

int uiPoint; // user inputted points

void printPoints();

void selectMode(); // pre set amount of points to generate

};

RNG.h

```
#pragma once
#include "STDAFX.h"
#include <ctime>
#include <chrono>
#include <thread>

class RNG
{
private:
    // mg vars to be used to prevent the same number being used repeatedly
    // X
    float prevX;
    float prevX1;
    float prevX2;
    float prevX3;
    // Y
    float prevY;
    float prevY1;
    float prevY2;
    float prevY3;
    // Z
    float prevZ;
    float prevZ1;
    float prevZ2;
    float prevZ3;
public:
    RNG();
    ~RNG();

    float genX(float); // generates a random x value and returns it
    float genY(float); // generates a random y value and returns it
    float genZ(float); // generates a random z value and returns it
};
```

STDAFX.h

```
#pragma once
// ALL HEADERS AND INCLUDES IN HERE
#include <stdio.h>
#include <iostream>
```

```
using namespace std;
```

Cluster.cpp

```
#include "_Cluster.h"

RNG numGenCluster;

_Cluster::_Cluster()
{
    uiCluster = 0;
}

_Cluster::~_Cluster()
{
    uiCluster = 0;
}

void _Cluster::selectMode()
{
    //select how many clusters to be generated
    cout << "please select how many clusters you would like to generate " << endl;
    cout << "2 or 5" << endl;
    cin >> uiCluster;

    switch (uiCluster)
    {
    case 2 :
        generateClusters(uiCluster);
        RNGCoOrds(uiCluster);
        break;
    case 5:
        generateClusters(uiCluster);
        RNGCoOrds(uiCluster);
        break;
    default :
        cout << "please enter a valid amount of clusters " << endl;
        selectMode();
    }
    cout << endl;
    printClusters();
}
```

```

}

void _Cluster::generateClusters(int i)
{
    // dynamically create array size for array declared in header using argument passed in
    Clusters = new Cluster[i]; // ARRAY OF STRUCTURES

}

void _Cluster::RNGCoOrds(int num)
{
    for (int i = 0; i < num; i++)
    {
        float tmp = 0;

        // Generate value for the current element properties
        // initialize the point variable cluster group to 1
        this->Clusters[i].clusterID = i;

        // generate x,y,z values
        this->Clusters[i].x = numGenCluster.genX(tmp);
        this->Clusters[i].y = numGenCluster.genY(tmp);
        this->Clusters[i].z = numGenCluster.genZ(tmp);;
    }
}

void _Cluster::printClusters()
{
    // print out all of the array
    for (int i = 0; i < uiCluster; i++)
    {
        cout << "Cluster [" << Clusters[i].clusterID << "], ";
        cout << "x = " << Clusters[i].x << " ";
        cout << "y = " << Clusters[i].y << " ";
        cout << "z = " << Clusters[i].z << " " << endl;
    }
}

```

KMeans.cpp

```
#include "_KMeans.h"
// declare an object to use for each class
_Cluster _kcluster;
_Point _kpoint;
_Math _kmath;

_KMeans::_KMeans()
{
    pointIndex = 0;
    clusterIndex = 0;
    updated = 0;
}

_KMeans::~_KMeans()
{
    pointIndex = 0;
    clusterIndex = 0;
    updated = 0;
}

void _KMeans::kmeans()
{
    // initialize the points and clusters
    initialization();

    cout << "initialization complete " << endl;

    /*cout << "max # of iterations of the program if convergence is unable to be met? " << endl;*/
    /*cin >> maxIteration;*/
    maxIteration = 100;

    // whilst convergence has not been met AND max iterations have not been met continue loop
    while (iterationIndex < maxIteration)
    {
        // set updated variable to 0
        updated = 0;

        // do iteration
        cout << "iteration " << iterationIndex << " complete" << endl;
        iteration();
    }
}
```

```

        // if convergence has been met ( no points changing groups), set convergence to true
        if (updated == 0 )
        {
            cout << "should be exiting the program" << endl;
            break; // exit loop once convergence found
            /*iterationIndex = maxIteration;*/
        }
        iterationIndex++;
    };

    // end the program
    endProgram();
}

void _KMeans::initialization()
{
    // what dimension will the algorithm run in
    cout << "what dimension would you like the program to be performed in" << endl; cout << "1. 2D " << endl; cout << "2. 3D" << endl;
    cin >> desiredDimension;

    // check if not an option, dont allow crap data
    if (desiredDimension != 1 && desiredDimension != 2)
    {
        system("cls");
        cout << "please choose a correct option" << endl;
        initialization(); // recursively call the function to start again
    }

    //generate points and clusters
    //////////////////////////////////// POINT GEN ////////////////////////////////////
    //n of points is == 10||20||50||100||200
    _kpoint.selectMode();

    //////////////////////////////////// CLUSTER GEN ////////////////////////////////////
    // n of clusters == 2||4||6||8||10
    _kcluster.selectMode();
}

```

```

void _KMeans::iteration()
{
    distanceChecks(); // PERFORM DISTANCE CHECKS
    recalClusterMeans(); // RECALCULATE CLUSTER MEANS
}

void _KMeans::distanceChecks()
{
    int iterationNumDistanceCheck;
    /// DISTANCE CHECK VARIABLES
    float Px = 0, Py = 0, Pz = 0; // Point
    float Qx = 0, Qy = 0, Qz = 0; int currentClustId = 0; // Current Cluster
    float NQx = 0, NQy = 0, NQz = 0; int nCurrentClustId = 0; // Potential New Cluster

    int numUpdated;

    clusterIndex = 0;
    pointIndex = 0;
    numUpdated = 0;
    // check what dimension program is running in

    switch (desiredDimension)
    {
        cout << iterationNumDistanceCheck << endl;
    case 1: //2D
        // if the distance to cluster 1 centroid is shorter than the distance
        for (clusterIndex = 0; clusterIndex < _kcluster.uiCluster; clusterIndex++)
        {
            //from the current point index, starting at point 0 each cluster index, loop through point array and check the euclidian distances
            for (pointIndex = 0; pointIndex < _kpoint.uiPoint; pointIndex++)
            {
                /// POINT VARIABLES
                Px = _kpoint.pointArray[pointIndex].z; Py = _kpoint.pointArray[pointIndex].y;

                /// CURRENT CLUSTER VARIABLES
                currentClustId = _kpoint.pointArray[pointIndex].cGroup;
                Qx = _kcluster.Clusters[currentClustId].z; Qy = _kcluster.Clusters[currentClustId].y;

                /// POTENTIAL NEW CLUSTER VARIABLES
                nCurrentClustId = clusterIndex;
                NQx = _kcluster.Clusters[nCurrentClustId].z; NQy = _kcluster.Clusters[nCurrentClustId].y;

                // P = POINT, C = CLUSTER , // NQ = Potential new cluster
            }
        }
    }
}

```

```

        // If the euclidian distance from point to current cluster centroid is larger than euclidian distance from point to potential new cluster centroid
        if (_kmath.euclidianDist(Px, Qx, Py, Qy) > _kmath.euclidianDist(Px, NQx, Py, NQy))
        {
            // set the points cluster id to the potential new cluster id's id
            _kpoint.pointArray[pointIndex].cGroup = nCurrentClustId;

            //cout << "point " << pointIndex << "has changed from cluster" << currentClustId << "-> cluster " << nCurrentClustId << endl;
            updated++;
        }
    }
}
if (updated == 0) // if nothing has been updated
{
    convergence = true;
    cout << "CONVERGENCE IS MET, END THE PROGRAM" << endl;
}
break;

```

case 2://3D

```

// if the distance to cluster 1 centroid is shorter than the distance
for (clusterIndex = 0; clusterIndex < _kcluster.uiCluster; clusterIndex++)
{
    //from the current point index, starting at point 0 each cluster index, loop through point array and check the euclidian distances
    for (pointIndex = 0; pointIndex < _kpoint.uiPoint; pointIndex++)
    {
        /// POINT VARIABLES
        Px = _kpoint.pointArray[pointIndex].x; Py = _kpoint.pointArray[pointIndex].y; Pz = _kpoint.pointArray[pointIndex].z;

        /// CURRENT CLUSTER VARIABLES
        currentClustId = _kpoint.pointArray[pointIndex].cGroup;
        Qx = _kcluster.Clusters[currentClustId].x; Qy = _kcluster.Clusters[currentClustId].y; Qz = _kcluster.Clusters[currentClustId].z;

        /// POTENTIAL NEW CLUSTER VARIABLES
        nCurrentClustId = clusterIndex;
        NQx = _kcluster.Clusters[nCurrentClustId].x; NQy = _kcluster.Clusters[nCurrentClustId].y; NQz = _kcluster.Clusters[nCurrentClustId].z;

        // P = POINT, C = CLUSTER, // NQ = Potential new cluster
        // If the euclidian distance from point to current cluster centroid is larger than euclidian distance from point to potential new cluster centroid
        if (_kmath.euclidianDist(Px, Qx, Py, Qz) > _kmath.euclidianDist(Px, NQx, Py, NQy))
        {
            // set the points cluster id to the potential new cluster id's id
            _kpoint.pointArray[pointIndex].cGroup = nCurrentClustId;
        }
    }
}

```

```

        cout << "point " << pointIndex << "has changed from cluster" << currentClustId << "-> cluster " << nCurrentClustId << endl;
        // a point has been updated so increment variable
        updated++;
    }
}

if (updated == 0) // if nothing has been updated
{
    convergence = true;
    cout << "CONVERGENCE IS MET, END THE PROGRAM" << endl;
}
break;

default: // this should never happen
    cout << "ERROR, IT SHOULD BE IMPOSSIBLE FOR ANY DIMENSION OTHER THAN 2D OR 3D TO BE PICKED, CHECK ERROR CHECKING" << endl;
    break;
}

}

void _KMeans::recalcClusterMeans()
{
    // call the recalculate cluster means functions based on dimension
    switch (desiredDimension)
    {
        case 1: // 2D
            recalcX();
            recalcY();
            break;
        case 2: // 3D
            recalcX();
            recalcY();
            recalcZ();
            break;

        default: // should never get here
            cout << "ERROR IN RECALC MEANS FUNCTION, WAS NOT SET AS 2D OR 3D" << endl;
            break;
    }
}

void _KMeans::endProgram()
{

```

```

// displaying result depending on what happened

// if convergence was found
if (convergence == true)
{
    cout << "convergence was met " << endl << endl;
}
else if (iterationIndex == maxIteration) // if iteration index reached max iterations
{
    cout << "convergence was not met" << endl << endl;
}

// print cluster groups
cout << "final cluster groups are " << endl;
printClusterGroups();
cout << endl << endl;
}

void _KMeans::recalcX()
{
    float newX = 0; float oldX = 0; int numX = 0;

    clusterIndex = 0;

    // recalculate each clusters x co-ordinate average
    for (int i = 0; i < 1; i++)
    {
        for (clusterIndex = 0; clusterIndex < _kcluster.uiCluster; clusterIndex++)
        {
            // reset new x to 0
            newX = _kcluster.Clusters[clusterIndex].x;
            oldX = _kcluster.Clusters[clusterIndex].x;
            numX = 0;

            // loop through the points, adding every point which is part of the clusters x value to the new x value ready to be averaged at the end
            for (pointIndex = 0; pointIndex < _kpoint.uiPoint; pointIndex++)
            {
                if (_kpoint.pointArray[pointIndex].cGroup == _kcluster.Clusters[clusterIndex].clusterID)
                {
                    // Add the points x value to the newX
                    newX += _kpoint.pointArray[pointIndex].x;
                    // Increment the amount of points in the cluster

```

```

        numX++;
    }
}
for (int l = 0; l < 1; l++)
{
    // DIVIDE THE X VALUES OF THE CLUSTER BY THE AMOUNT OF POINTS WITHIN THE CLUSTER TO GET THE MEAN
    newX /= numX;

    // ASSIGN THE NEW MEAN TO THE CLUSTER CENTROID
    _kcluster.Clusters[clusterIndex].x = newX;
}
}
}

void _KMeans::recalcY()
{
    float newY = 0; float oldY = 0; int numY = 0;

    clusterIndex = 0;

    // recalculate each clusters y co-ordinate average
    for (int i = 0; i < 1; i++)
    {
        for (clusterIndex = 0; clusterIndex < _kcluster.uiCluster; clusterIndex++)
        {
            // reset new x to 0
            newY = _kcluster.Clusters[clusterIndex].y;
            oldY = _kcluster.Clusters[clusterIndex].y;
            numY = 0;

            // loop through the points, adding every point which is part of the clusters x value to the new x value ready to be averaged at the end
            for (pointIndex = 0; pointIndex < _kpoint.uiPoint; pointIndex++)
            {
                if (_kpoint.pointArray[pointIndex].cGroup == _kcluster.Clusters[clusterIndex].clusterID)
                {
                    // Add the points y value to the newX
                    newY += _kpoint.pointArray[pointIndex].y;
                    // Increment the amount of points in the cluster
                    numY++;
                }
            }
            for (int l = 0; l < 1; l++)
            {
                // DIVIDE THE Y VALUES OF THE CLUSTER BY THE AMOUNT OF POINTS WITHIN THE CLUSTER TO GET THE MEAN

```

```

        newY /= numY;

        // ASSIGN THE NEW MEAN TO THE CLUSTER CENTROID
        _kcluster.Clusters[clusterIndex].y = newY;
    }
}

void _KMeans::recalcZ()
{
    float newZ = 0; float oldZ = 0; int numZ = 0;

    clusterIndex = 0;

    // recalculate each clusters x co-ordinate average
    for (int i = 0; i < 1; i++)
    {
        for (clusterIndex = 0; clusterIndex < _kcluster.uiCluster; clusterIndex++)
        {
            // reset new x to 0
            newZ = _kcluster.Clusters[clusterIndex].z;
            oldZ = _kcluster.Clusters[clusterIndex].z;
            numZ = 0;
            // loop through the points, adding every point which is part of the clusters x value to the new x value ready to be averaged at the end
            for (pointIndex = 0; pointIndex < _kpoint.uiPoint; pointIndex++)
            {
                if (_kpoint.pointArray[pointIndex].cGroup == _kcluster.Clusters[clusterIndex].clusterID)
                {
                    // Add the points x value to the newX
                    newZ += _kpoint.pointArray[pointIndex].z;
                    // Increment the amount of points in the cluster
                    numZ++;
                }
            }
            for (int l = 0; l < 1; l++)
            {
                // DIVIDE THE X VALUES OF THE CLUSTER BY THE AMOUNT OF POINTS WITHIN THE CLUSTER TO GET THE MEAN
                newZ /= numZ;
                // ASSIGN THE NEW MEAN TO THE CLUSTER CENTROID
                _kcluster.Clusters[clusterIndex].z = newZ;
            }
        }
    }
}

```

```

}

void _KMeans::printClusterGroups()
{
    int clusterID; // whats the cluster id
    int pointClusterGroup; // what group does the point belong to

    int numC; // number of clusters
    int numP; // number of points

    numC = _kcluster.uiCluster; // assign the number of clusters to this variable
    numP = _kpoint.uiPoint; // assign the number of points to this variable

    // print the cluster groups out
    for (int k = 0; k < numC; k++)
    {
        clusterID = k;

        cout << "the points that are in cluster " << k << " are," << endl;

        for (int j = 0; j < numP; j++)
        {
            pointClusterGroup = _kpoint.pointArray[j].cGroup;
            if (clusterID == pointClusterGroup)
            {
                cout << _kpoint.pointArray[j].pointId << " ";
            }
        }
        cout << endl << endl;
    }
}

```

Math.cpp

```
#include "_Math.h"

_Math::_Math()
{
}

_Math::~~_Math()
{
}

// 3D Math euclidian Distance formula
float _Math::euclidianDist(float Px, float Qx, float Py, float Qy, float Pz, float Qz)
{
    // formula = sqrt(((P1 - Q1)*2) + ((P2 - Q2)*2) + ((P3 - Q3)*2)))

    double dist;

    double x = (Px - Qx);
    double y = (Py - Qy);
    double z = (Pz - Qz);

    x *= x;
    y *= y;

    z *= z;
    dist = x + y + z;

    dist = sqrt(dist);

    return dist;
}

// 2D Math euclidian Distance formula
float _Math::euclidianDist(float Px, float Qx, float Py, float Qy)
{
    // formula = sqrt(((P1 - Q1)*2) + ((P2 - Q2)*2))

    double dist;
```

```
double x = (Px - Qx);
double y = (Py - Qy);

//cout << " x = " << x << endl;
//cout << " y = " << y << endl;

x *= x;
y *= y;

//cout << " x = " << x << endl;
//cout << " y = " << y << endl;

dist = x + y;
/*cout << dist << endl;*/

dist = sqrt(dist);
/*cout << dist << endl;*/

return dist;
```

```
}
```

Point.cpp

```
#include "_Point.h"
#include "STDAFX.h"
#include "RNG.h"

RNG numGenPoint;

_Point::_Point()
{
    uiPoint = 0;
}

_Point::~_Point()
{
    uiPoint = 0;
}

void _Point::selectMode()
{
    cout << "please select how many points you would like to generate " << endl;
    cout << "512, 1024 or 25600" << endl;
    cin >> uiPoint;

    switch (uiPoint)
    {
    case 512:
        generatePoints(uiPoint);
        genRandomPoints(uiPoint);
        break;
    case 1024:
        generatePoints(uiPoint);
        genRandomPoints(uiPoint);
        break;
    case 25600:
        generatePoints(uiPoint);
        genRandomPoints(uiPoint);
        break;
    default:
```

```

        cout << "please enter a valid amount of points " << endl;
        selectMode();
    }
    cout << endl;
    printPoints();
}

void _Point::generatePoints(int i)
{
    // dynamically create array size for array declared in header using argument passed in
    pointArray = new Point[i]; // ARRAY OF STRUCTURES
}

void _Point::genRandomPoints(int num)
{
    int pointID = 0;

    for (int i = 0; i < num; i++)
    {
        float tmp = 0;

        // Generate value for the current element properties
        // initialize the point variable cluster group to 1
        this->pointArray[i].cGroup = 1;

        // generate x,y,z values
        this->pointArray[i].x = numGenPoint.genX(tmp);
        this->pointArray[i].y = numGenPoint.genY(tmp);
        this->pointArray[i].z = numGenPoint.genZ(tmp);

        // Assign Point ID
        this->pointArray[i].pointId = pointID;
        pointID += 1;
    }
}

//
void _Point::printPoints()
{
    uiPoint;

```

```
// print out all of the array
for (int i = 0; i < uiPoint; i++)
{
    cout << "Point [" << pointArray[i].pointId << "], ";
    cout << "Cluster group = " << pointArray[i].cGroup << " ";
    cout << "x = " << pointArray[i].x << " ";
    cout << "y = " << pointArray[i].y << " ";
    cout << "z = " << pointArray[i].z << " " << endl;
}
```

```
}
```

RNG.cpp

```
#include "RNG.h"
```

```
RNG::RNG()
```

```
{  
    // initialize variables to 0  
    // X  
    prevX = 0;  
    prevX1 = 0;  
    prevX2 = 0;  
    prevX3 = 0;  
    // Y  
    prevY = 0;  
    prevY1 = 0;  
    prevY2 = 0;  
    prevY3 = 0;  
    // Z  
    prevZ = 0;  
    prevZ1 = 0;  
    prevZ2 = 0;  
    prevZ3 = 0;  
    // time seed to be used in RNG generators  
    srand(time(0));  
}
```

```
RNG::~RNG()
```

```
{  
}
```

```
float RNG::genX(float x)
```

```
{  
    x = 0;  
  
    // makes more sense in my head if its 0-500 then minus 250  
    // if it is the first time running  
    if (x != 0 && prevX != 0 && prevX1 && prevX2 != 0 && prevX3 != 0)  
    {  
        x = rand() % (500, 500) - 250;  
    }  
    else
```

```

{
    //generate random value between -250 and 250
    x = rand() % (500, 500) - 250;
}

// if x is the value of the previous 3 x's, regenerate until it isn't
while (x == prevX || x == prevX1 || x == prevX2 || x == prevX3 ||
       x == prevY || x == prevY1 || x == prevY2 || x == prevY3 ||
       x == prevZ || x == prevZ1 || x == prevZ2 || x == prevZ3)
{
    x = rand() % (500, 500) - 250;
}
// assign the values to the next x point.
prevX3 = prevX2;
prevX2 = prevX1;
prevX1 = prevX;
prevX = x;

return x;
}

float RNG::genY(float y)
{
    y = 0;

    // makes more sense in my head if its 0-500 then minus 250
    // if it is the first time running
    if (y != 0 && prevY != 0 && prevY1 && prevY2 != 0 && prevY3 != 0)
    {
        y = rand() % (500, 500) - 250;
    }
    else
    {
        // generate random value between -250 and 250
        y = rand() % (500, 500) - 250;
    }

    // if y is the value of the previous 3 y's, x's or z's, regenerate until it isn't
    while (y == prevX || y == prevX1 || y == prevX2 || y == prevX3 ||
           y == prevY || y == prevY1 || y == prevY2 || y == prevY3 ||
           y == prevZ || y == prevZ1 || y == prevZ2 || y == prevZ3)
    {

```

```

        y = rand() % (500, 500) - 250;
    }
    // assign the values to the next Y point.
    prevY3 = prevY2;
    prevY2 = prevY1;
    prevY1 = prevY3;
    prevY = y;

    return y;
}

float RNG::genZ(float z)
{
    z = 0;

    // makes more sense in my head if its 0-500 then minus 250
    // if it is the first time running
    if (z != 0 && prevZ != 0 && prevZ1 && prevZ2 != 0 && prevZ3 != 0)
    {
        z = rand() % (500, 500) - 250;

    }
    else
    {
        // generate random value between -250 and 250
        z = rand() % (500, 500) - 250;

    }

    // if Z is the value of the previous 3 Z's, regenerate until it isn't
    while (z == prevX || z == prevX1 || z == prevX2 || z == prevX3 ||
           z == prevY || z == prevY1 || z == prevY2 || z == prevY3 ||
           z == prevZ || z == prevZ1 || z == prevZ2 || z == prevZ3)
    {
        z = rand() % (500, 500) - 250;

    }
    // assign the values to the next Z point.
    prevZ3 = prevZ2;
    prevZ2 = prevZ1;
    prevZ1 = prevZ3;
    prevZ = z;

    return z;
}

```

main.cpp

```
// includes
#include "STDAFX.H"
#include "_KMeans.h"

// declare an object to use for each class
_KMeans kmeans;

void main()
{
    // start Kmeans
    kmeans.kmeans();
    system("pause");
}
```

Appendix - GPU implementation code

_Cluster.h

```
#pragma once
#include "STDAFX.h"
#include "RNG.h"
// structure for each cluster
struct Cluster
{
    int clusterID;
    float x;
    float y;
    float z;
};

class _Cluster
{
private:

    void generateClusters(int i); // how many clusters as argument
    void RNGCoOrds(int i); // take size of array (amount of clusters) as argument generate random point co-ordinates

public:
    _Cluster();
    ~_Cluster();

    Cluster * Clusters; // declare an array that will be used to hold the structs
    int uiCluster;
    void printClusters();
    void selectMode(); //select an amount of clusters to be done. (2||3||5||10) this function will call other functions depending on input
};
```

_Math.h

```
#pragma once
#include <math.h>
#include "STDAFX.h"
class _Math
{
public:
    _Math();
    ~_Math();

    float euclidianDist(float Px, float Qx, float Py, float Qy); // FIND THE DISTANCE IN 2 DIMENSIONS
    float euclidianDist(float Px, float Qx, float Py, float Qy, float Pz, float Qz); // FIND THE DISTANCE IN 3 DIMENSIONS
};
```

_Point.h

```
#pragma once
// structure for each point
struct Point
{
    int pointId;
    float x;
    float y;
    float z;
    float cGroup; // which group do they belong to?
};

class _Point
{
private :

    void generatePoints(int i); // how many points as argyment
    void genRandomPoints(int i); // take size of array as argument generate random point co-ordinates

public:

    _Point();
    ~_Point();

    Point* pointArray; // declare an array that will be used to hold the structs
    int uiPoint;
    void printPoints();
    void selectMode(); // pre set amount of points to generate

};
```

RNG.h

```
#pragma once
#include "STDAFX.h"
#include <ctime>
#include <chrono>
#include <thread>

class RNG
{
private:
    // rng vars to be used to prevent the same number being used repeatedly
    // X
    float prevX;
    float prevX1;
    float prevX2;
    float prevX3;
    // Y
    float prevY;
    float prevY1;
    float prevY2;
    float prevY3;
    // Z
    float prevZ;
    float prevZ1;
    float prevZ2;
    float prevZ3;
public:
    RNG();
    ~RNG();

    float genX(float); // generates a random x value and returns it
    float genY(float); // generates a random y value and returns it
    float genZ(float); // generates a random z value and returns it
};
```

Cluster.cpp

```
#include "_Cluster.h"

RNG numGenCluster;

_Cluster::_Cluster()
{
    uiCluster = 0;
}

_Cluster::~_Cluster()
{
    uiCluster = 0;
}

void _Cluster::selectMode()
{
    cout << "please select how many clusters you would like to generate " << endl;
    cout << "2 or 5" << endl;
    cin >> uiCluster;

    switch (uiCluster)
    {
    case 2 :
        generateClusters(uiCluster);
        RNGCoOrds(uiCluster);
        break;
    case 5:
        generateClusters(uiCluster);
        RNGCoOrds(uiCluster);
        break;
    default :
        cout << "please enter a valid amount of clusters " << endl;
        selectMode();
    }
    cout << endl;
    printClusters();
}
```

```

}

void _Cluster::generateClusters(int i)
{
    // dynamically create array size for array declared in header using argument passed in
    Clusters = new Cluster[i]; // ARRAY OF STRUCTURES

}

void _Cluster::RNGCoOrds(int num)
{
    for (int i = 0; i < num; i++)
    {
        float tmp = 0;

        // Generate value for the current element properties
        // initialize the point variable cluster group to 1
        this->Clusters[i].clusterID = i;

        // generate x,y,z values
        this->Clusters[i].x = numGenCluster.genX(tmp);
        this->Clusters[i].y = numGenCluster.genY(tmp);
        this->Clusters[i].z = numGenCluster.genZ(tmp);;
    }
}

void _Cluster::printClusters()
{
    // print out all of the array
    for (int i = 0; i < uiCluster; i++)
    {
        cout << "Cluster [" << Clusters[i].clusterID << "], ";
        cout << "x = " << Clusters[i].x << " ";
        cout << "y = " << Clusters[i].y << " ";
        cout << "z = " << Clusters[i].z << " " << endl;
    }
}

```

Math.cpp

```
#include "_Math.h"
```

```
_Math::_Math()  
{  
}
```

```
_Math::~~_Math()  
{  
}
```

```
// 3D Math euclidian Distance formula
```

```
float _Math::euclidianDist(float Px, float Qx, float Py, float Qy, float Pz, float Qz)
```

```
{  
    // formula = sqrt(((P1 - Q1)*2) + ((P2 - Q2)*2) + ((P3 - Q3)*2)))
```

```
    double dist;
```

```
    double x = (Px - Qx);
```

```
    double y = (Py - Qy);
```

```
    double z = (Pz - Qz);
```

```
    x *= x;
```

```
    y *= y;
```

```
    z *= z;
```

```
    dist = x + y + z;
```

```
    dist = sqrt(dist);
```

```
    return dist;
```

```
}
```

```
// 2D Math euclidian Distance formula
```

```
float _Math::euclidianDist(float Px, float Qx, float Py, float Qy)
```

```
{
```

```
    // formula = sqrt(((P1 - Q1)*2) + ((P2 - Q2)*2))
```

```
double dist;
```

```
double x = (Px - Qx);
```

```
double y = (Py - Qy);
```

```
//cout << " x = " << x << endl;
```

```
//cout << " y = " << y << endl;
```

```
x *= x;
```

```
y *= y;
```

```
//cout << " x = " << x << endl;
```

```
//cout << " y = " << y << endl;
```

```
dist = x + y;
```

```
/*cout << dist << endl;*/
```

```
dist = sqrt(dist);
```

```
/*cout << dist << endl;*/
```

```
return dist;
```

```
}
```

Point.cpp

```
#include "_Point.h"
#include "STDAFX.h"
#include "RNG.h"

RNG numGenPoint;

_Point::_Point()
{
    uiPoint = 0;
}

_Point::~_Point()
{
    uiPoint = 0;
}

void _Point::selectMode()
{
    cout << "please select how many points you would like to generate " << endl;
    cout << "512, 1024 or 25600" << endl;
    cin >> uiPoint;

    switch (uiPoint)
    {
    case 512:
        generatePoints(uiPoint);
        genRandomPoints(uiPoint);
        break;
    case 1024:
        generatePoints(uiPoint);
        genRandomPoints(uiPoint);
        break;
    case 25600:
        generatePoints(uiPoint);
        genRandomPoints(uiPoint);
        break;
    default:
```

```

        cout << "please enter a valid amount of points " << endl;
        selectMode();
    }
    cout << endl;
    printPoints();
}

void _Point::generatePoints(int i)
{
    // dynamically create array size for array declared in header using argument passed in
    pointArray = new Point[i]; // ARRAY OF STRUCTURES
}

void _Point::genRandomPoints(int num)
{
    int pointID = 0;

    for (int i = 0; i < num; i++)
    {
        float tmp = 0;

        // Generate value for the current element properties
        // initialize the point variable cluster group to 1
        this->pointArray[i].cGroup = 1;

        // generate x,y,z values
        this->pointArray[i].x = numGenPoint.genX(tmp);
        this->pointArray[i].y = numGenPoint.genY(tmp);
        this->pointArray[i].z = numGenPoint.genZ(tmp);

        // Assign Point ID
        this->pointArray[i].pointId = pointID;
        pointID += 1;
    }
}

//
void _Point::printPoints()
{
    uiPoint;

```

```
// print out all of the array
for (int i = 0; i < uiPoint; i++)
{
    cout << "Point [" << pointArray[i].pointId << "], ";
    cout << "Cluster group = " << pointArray[i].cGroup << " ";
    cout << "x = " << pointArray[i].x << " ";
    cout << "y = " << pointArray[i].y << " ";
    cout << "z = " << pointArray[i].z << " " << endl;
}
```

```
}
```

RNG.cpp

```
#include "RNG.h"
```

```
RNG::RNG()
```

```
{  
    // initialize variables to 0  
    // X  
    prevX = 0;  
    prevX1 = 0;  
    prevX2 = 0;  
    prevX3 = 0;  
    // Y  
    prevY = 0;  
    prevY1 = 0;  
    prevY2 = 0;  
    prevY3 = 0;  
    // Z  
    prevZ = 0;  
    prevZ1 = 0;  
    prevZ2 = 0;  
    prevZ3 = 0;  
    // time seed to be used in RNG generators  
    srand(time(0));  
}
```

```
RNG::~RNG()
```

```
{  
}
```

```
float RNG::genX(float x)
```

```
{  
    x = 0;  
  
    // makes more sense in my head if its 0-500 then minus 250  
    // if it is the first time running  
    if (x != 0 && prevX != 0 && prevX1 && prevX2 != 0 && prevX3 != 0)  
    {  
        x = rand() % (500, 500) - 250;  
    }  
}
```

```

else
{
    //generate random value between -250 and 250
    x = rand() % (500, 500) - 250;
}

// if x is the value of the previous 3 x's, regenerate until it isn't
while (x == prevX || x == prevX1 || x == prevX2 || x == prevX3 ||
      x == prevY || x == prevY1 || x == prevY2 || x == prevY3 ||
      x == prevZ || x == prevZ1 || x == prevZ2 || x == prevZ3)
{
    x = rand() % (500, 500) - 250;
}
// assign the values to the next X point.
prevX3 = prevX2;
prevX2 = prevX1;
prevX1 = prevX;
prevX = x;

return x;
}

float RNG::genY(float y)
{
    y = 0;

    // makes more sense in my head if its 0-500 then minus 250
    // if it is the first time running
    if (y != 0 && prevY != 0 && prevY1 && prevY2 != 0 && prevY3 != 0)
    {
        y = rand() % (500, 500) - 250;
    }
    else
    {
        // generate random value between -250 and 250
        y = rand() % (500, 500) - 250;
    }

    // if y is the value of the previous 3 y's, x's or z's, regenerate until it isn't
    while (y == prevX || y == prevX1 || y == prevX2 || y == prevX3 ||
          y == prevY || y == prevY1 || y == prevY2 || y == prevY3 ||
          y == prevZ || y == prevZ1 || y == prevZ2 || y == prevZ3)

```

```

    {
        y = rand() % (500, 500) - 250;
    }
    // assign the values to the next Y point.
    prevY3 = prevY2;
    prevY2 = prevY1;
    prevY1 = prevY3;
    prevY = y;

    return y;
}

float RNG::genZ(float z)
{
    z = 0;

    // makes more sense in my head if its 0-500 then minus 250
    // if it is the first time running
    if (z != 0 && prevZ != 0 && prevZ1 && prevZ2 != 0 && prevZ3 != 0)
    {
        z = rand() % (500, 500) - 250;

    }
    else
    {
        // generate random value between -250 and 250
        z = rand() % (500, 500) - 250;
    }

    // if Z is the value of the previous 3 Z's, regenerate until it isn't
    while (z == prevX || z == prevX1 || z == prevX2 || z == prevX3 ||
           z == prevY || z == prevY1 || z == prevY2 || z == prevY3 ||
           z == prevZ || z == prevZ1 || z == prevZ2 || z == prevZ3)
    {
        z = rand() % (500, 500) - 250;
    }
    // assign the values to the next Z point.
    prevZ3 = prevZ2;
    prevZ2 = prevZ1;
    prevZ1 = prevZ3;
    prevZ = z;

    return z;
}

```



Kernel.cu

```
#include "cuda_runtime.h"
#include "device_launch_parameters.h"

#include <stdio.h>
#include "STDAFX.H"
#include "_Point.h"
#include "_Cluster.h"
#include "_Math.h"

// declare an object to use for each class
_Cluster _kcluster;
_Point _kpoint;
_Math _kmath;

// Variables to be used
int desiredDimension = 0; // the desired dimension variable
int pointIndex = 0; // the point index used in distance checking
int clusterIndex = 0; // cluster index, used in distance checking
int maxIteration = 0; // maximum amount of iterations
int iterationIndex = 0; // current iteration number
int numC; // num of clusters
int numP; // num of points

// Did not work if I used these variables as int or bool,
//so had to pass them to kernel as a array of 1, in order to get it back from the kernel
int updated[1]; // if points have swapped
bool convergence[1]; //has the algorithm found convergence?

//Functions
void initialization(); // performs generation of points & clusters
void iteration(); // iteration of the algorithm

void recalculClusterMeans(); // recalculate cluster means
void endProgram(); // end the program

void recalculX(); //recalculate cluster X centroids
void recalculY(); // recalculate cluster Y centroids
void recalculZ(); // recalculate cluster Z centroids

// the 4 variables below were provided with cuda file upon creation
const int arraySize = 5;
```

```

const int a[arraySize] = { 1, 2, 3, 4, 5 };
const int b[arraySize] = { 10, 20, 30, 40, 50 };
int c[arraySize] = { 0 };

void printClusterGroups(); // prints a list of points in each cluster

// number of threads per block
int threadsPerBlock = 512;
// blocks
dim3 block;

cudaError_t addWithCuda(int *c, const int *a, const int *b, unsigned int size, int pass);

// function needed as a device function in order to access the function from the kernel
// 2D version of euclidian distance
__device__ float euclidianDist2D(float Px, float Qx, float Py, float Qy) // FIND THE DISTANCE IN 2 DIMENSIONS
{
    // formula = sqrt(((P1 - Q1)*2) + ((P2 - Q2)*2))

    double dist;

    double x = (Px - Qx);
    double y = (Py - Qy);

    //cout << " x = " << x << endl;
    //cout << " y = " << y << endl;

    x *= x;
    y *= y;

    //cout << " x = " << x << endl;
    //cout << " y = " << y << endl;

    dist = x + y;
    /*cout << dist << endl;*/

    dist = sqrt(dist);
    /*cout << dist << endl;*/

    return dist;
}
// function needed as a device function in order to access the function from the kernel

```

```

//3D version of euclidian distance
__device__ float euclidianDist3D(float Px, float Qx, float Py, float Qy, float Pz, float Qz) // FIND THE DISTANCE IN 3 DIMENSIONS
{
    // formula = sqrt(((P1 - Q1)*2) + ((P2 - Q2)*2) + ((P3 - Q3)*2)))

    double dist;

    double x = (Px - Qx);
    double y = (Py - Qy);
    double z = (Pz - Qz);

    x *= x;
    y *= y;

    z *= z;
    dist = x + y + z;

    dist = sqrt(dist);

    return dist;
}

__global__ void addKernel(int *c, const int *a, const int *b)
{
    int i = threadIdx.x;
    c[i] = a[i] + b[i];
}

__global__ void distanceChecksKernel(Point* pointArray, Cluster* Clusters, int desiredDimension, bool convergence, int updated, int numC)
{
    //set i to the thread number, the thread number + block id allows the thread to be correctly assigned to correct point
    int i = blockIdx.x * blockDim.x + threadIdx.x;

    /// DISTANCE CHECK VARIABLES
    float Px = 0, Py = 0, Pz = 0; // Point
    float Qx = 0, Qy = 0, Qz = 0; // Current Cluster
    float NQx = 0, NQy = 0, NQz = 0; // Potential New Cluster
    int currentClustId = 0; // Current Cluster
    int nCurrentClustId = 0; // Potential New Cluster

    int numUpdated;

    int clusterIndex = 0;
    numUpdated = 0;
    // check what dimension program is running in

```

```

switch (desiredDimension)
{

case 1: //2D
    // if the distance to cluster 1 centroid is shorter than the distance
    for (clusterIndex = 0; clusterIndex < numC; clusterIndex++)
    {

        /// POINT VARIABLES
        Px = pointArray[i].z;    Py = pointArray[i].y;

        /// CURRENT CLUSTER VARIABLES
        currentClustId = pointArray[i].cGroup;
        Qx = Clusters[currentClustId].z;    Qy = Clusters[currentClustId].y;

        /// POTENTIAL NEW CLUSTER VARIABLES
        nCurrentClustId = clusterIndex;
        NQx = Clusters[nCurrentClustId].z;    NQy = Clusters[nCurrentClustId].y;

        // P = POINT, C = CLUSTER , // NQ = Potential new cluster
        // If the euclidian distance from point to current cluster centroid is larger than euclidian distance from point to potential new cluster centroid
        if (euclidianDist2D(Px, Qx, Py, Qy) > euclidianDist2D(Px, NQx, Py, NQy))
        {
            // set the points cluster id to the potential new cluster id's id
            pointArray[i].cGroup = nCurrentClustId;

            /*cout << "point " << i << "has changed from cluster" << currentClustId << "-> cluster " << nCurrentClustId << endl;*/
            updated++;
        }
    }
    if (updated == 0)
    {
        convergence = true;
        /*cout << "CONVERGENCE IS MET, END THE PROGRAM" << endl;*/
    }
    break;

case 2: //3D
    // if the distance to cluster 1 centroid is shorter than the distance
    for (clusterIndex = 0; clusterIndex < numC; clusterIndex++)
    {

```

```

        /// POINT VARIABLES
        Px = pointArray[i].z;    Py = pointArray[i].y; Pz = pointArray[i].z;

        /// CURRENT CLUSTER VARIABLES
        currentClustId = pointArray[i].cGroup;
        Qx = Clusters[currentClustId].z;    Qy = Clusters[currentClustId].y; Qz = Clusters[currentClustId].z;

        /// POTENTIAL NEW CLUSTER VARIABLES
        nCurrentClustId = clusterIndex;
        NQx = Clusters[nCurrentClustId].z;    NQy = Clusters[nCurrentClustId].y; NQz = Clusters[nCurrentClustId].z;

        // P = POINT, C = CLUSTER , // NQ = Potential new cluster
        // If the euclidian distance from point to current cluster centroid is larger than euclidian distance from point to potential new cluster centroid
        if (euclidianDist3D(Px, Qx, Py, Qy, Pz, Qz) > euclidianDist3D(Px, NQx, Py, NQy, Pz, NQz))
        {
            // set the points cluster id to the potential new cluster id's id
            pointArray[i].cGroup = nCurrentClustId;
            /*cout << "point " << i << "has changed from cluster" << currentClustId << "-> cluster " << nCurrentClustId << endl;*/
            // a point has been updated so increment variable

        }

    }
    if (updated == 0)
    {
        convergence = true;
        /*cout << "CONVERGENCE IS MET, END THE PROGRAM" << endl;*/
    }
    break;

default: // this should never happen
    /*cout << "ERROR, IT SHOULD BE IMPOSSIBLE FOR ANY DIMENSION OTHER THAN 2D OR 3D TO BE PICKED, CHECK ERROR CHECKING" << endl;*/
    break;
}

}

void kmeans()
{
    // initialize the points and clusters
    //initialization();

    cout << "initialization complete " << endl;
}

```

```

updated[0] = 0;
convergence[0] = false;

/*cout << "max # of iterations of the program if convergence is unable to be met? " << endl;*/
/*cin >> maxIteration;*/
maxIteration = 100;

// whilst convergence has not been met AND max iterations have not been met continue loop
while (iterationIndex < maxIteration)
{
    updated[0] = 0;

    // do iteration
    cout << "iteration " << iterationIndex << " complete" << endl;
    iteration();

    // if convergence has been met ( no points changing groups), set convergence to true
    if (updated[0] == 0)
    {
        cout << "should be exiting the program" << endl;
        break; // exit loop once convergence found
        /*iterationIndex = maxIteration;*/
    }
    iterationIndex++;
};

// end the program
endProgram();
}

void initialization()
{
    // set the block x size to Number of points divided by thenumber of threads per block
    block.x = (numP / threadsPerBlock) + 1;

    // what dimension will the algorithm run in
    cout << "what dimension would you like the program to be performed in" << endl; cout << "1. 2D " << endl; cout << "2. 3D" << endl;
    cin >> desiredDimension;

    // check if not an option, dont allow crap data
    if (desiredDimension != 1 && desiredDimension != 2)

```

```

{
    system("cls");
    cout << "please choose a correct option" << endl;
    initialization(); // recursively call the function to start again
}

//generate points and clusters
///*////////// POINT GEN //////////*/
//n of points is == 10||20||50||100||200
_kpoint.selectMode();

///*////////// CLUSTER GEN //////////*/
// n of clusters == 2||4||6||8||10
_kcluster.selectMode();

}

void iteration()
{
    //distanceChecks(); // PERFORM DISTANCE CHECKS

    // added functionality here to pass in the number of points in the program
    cudaError_t cudaStatus = addWithCuda(c, a, b, arraySize, numP);
    if (cudaStatus != cudaSuccess) {
        fprintf(stderr, "addWithCuda failed!");
        return;
    }

    // cudaDeviceReset must be called before exiting in order for profiling and
    // tracing tools such as Nsight and Visual Profiler to show complete traces.
    cudaStatus = cudaDeviceReset();
    if (cudaStatus != cudaSuccess) {
        fprintf(stderr, "cudaDeviceReset failed!");
        return;
    }

    recalcClusterMeans(); // RECALCULATE CLUSTER MEANS
}

void recalcClusterMeans()

```

```

{
    // call the recalculate cluster means functions based on dimension
    switch (desiredDimension)
    {
    case 1: // 2D
        recalcX();
        recalcY();
        break;
    case 2: // 3D
        recalcX();
        recalcY();
        recalcZ();
        break;

    default: // should never get here
        cout << "ERROR IN RECALC MEANS FUNCTION, WAS NOT SET AS 2D OR 3D" << endl;
        break;
    }
}

void endProgram()
{
    // displaying result depending on what happened

    // if convergence was found
    if (convergence[0] == true)
    {
        cout << "convergence was met " << endl << endl;
    }
    else if (iterationIndex == maxIteration) // if iteration index reached max iterations
    {
        cout << "convergence was not met" << endl << endl;
    }

    // print cluster groups
    cout << "final cluster groups are " << endl;
    printClusterGroups();
    cout << endl << endl;
}

void recalcX()
{

```

```

float newX = 0; float oldX = 0; int numX = 0;

clusterIndex = 0;

// recalculate each clusters x co-ordinate average
for (int i = 0; i < 1; i++)
{
    for (clusterIndex = 0; clusterIndex < _kcluster.uiCluster; clusterIndex++)
    {
        // reset new x to 0
        newX = _kcluster.Clusters[clusterIndex].x;
        oldX = _kcluster.Clusters[clusterIndex].x;
        numX = 0;

        // loop through the points, adding every point which is part of the clusters x value to the new x value ready to be averaged at the end
        for (pointIndex = 0; pointIndex < _kpoint.uiPoint; pointIndex++)
        {
            if (_kpoint.pointArray[pointIndex].cGroup == _kcluster.Clusters[clusterIndex].clusterID)
            {
                // Add the points x value to the newX
                newX += _kpoint.pointArray[pointIndex].x;
                // Increment the amount of points in the cluster
                numX++;
            }
        }
        for (int l = 0; l < 1; l++)
        {
            // DIVIDE THE X VALUES OF THE CLUSTER BY THE AMOUNT OF POINTS WITHIN THE CLUSTER TO GET THE MEAN
            newX /= numX;

            // ASSIGN THE NEW MEAN TO THE CLUSTER CENTROID
            _kcluster.Clusters[clusterIndex].x = newX;
        }
    }
}

void recalcY()
{
    float newY = 0; float oldY = 0; int numY = 0;

    clusterIndex = 0;

    // recalculate each clusters y co-ordinate average

```

```

for (int i = 0; i < 1; i++)
{
    for (clusterIndex = 0; clusterIndex < _kcluster.uiCluster; clusterIndex++)
    {
        // reset new x to 0
        newY = _kcluster.Clusters[clusterIndex].y;
        oldY = _kcluster.Clusters[clusterIndex].y;
        numY = 0;
        // loop through the points, adding every point which is part of the clusters x value to the new x value ready to be averaged at the end
        for (pointIndex = 0; pointIndex < _kpoint.uiPoint; pointIndex++)
        {
            if (_kpoint.pointArray[pointIndex].cGroup == _kcluster.Clusters[clusterIndex].clusterID)
            {
                // Add the points y value to the newX
                newY += _kpoint.pointArray[pointIndex].y;
                // Increment the amount of points in the cluster
                numY++;
            }
        }
        for (int l = 0; l < 1; l++)
        {
            // DIVIDE THE Y VALUES OF THE CLUSTER BY THE AMOUNT OF POINTS WITHIN THE CLUSTER TO GET THE MEAN
            newY /= numY;

            // ASSIGN THE NEW MEAN TO THE CLUSTER CENTROID
            _kcluster.Clusters[clusterIndex].y = newY;
        }
    }
}

void recalcZ()
{
    float newZ = 0; float oldZ = 0; int numZ = 0;

    clusterIndex = 0;

    // recalculate each clusters x co-ordinate average
    for (int i = 0; i < 1; i++)
    {
        for (clusterIndex = 0; clusterIndex < _kcluster.uiCluster; clusterIndex++)
        {
            // reset new x to 0
            newZ = _kcluster.Clusters[clusterIndex].z;

```

```

oldZ = _kcluster.Clusters[clusterIndex].z;
numZ = 0;
// loop through the points, adding every point which is part of the clusters x value to the new x value ready to be averaged at the end
for (pointIndex = 0; pointIndex < _kpoint.uiPoint; pointIndex++)
{
    if (_kpoint.pointArray[pointIndex].cGroup == _kcluster.Clusters[clusterIndex].clusterID)
    {
        // Add the points x value to the newX
        newZ += _kpoint.pointArray[pointIndex].z;
        // Increment the amount of points in the cluster
        numZ++;
    }
}
for (int l = 0; l < 1; l++)
{
    // DIVIDE THE X VALUES OF THE CLUSTER BY THE AMOUNT OF POINTS WITHIN THE CLUSTER TO GET THE MEAN
    newZ /= numZ;
    // ASSIGN THE NEW MEAN TO THE CLUSTER CENTROID
    _kcluster.Clusters[clusterIndex].z = newZ;
}
}
}
}

void printClusterGroups()
{
    int clusterID; // whats the cluster id
    int pointClusterGroup; // what group does the point belong to

    // print the cluster groups out
    for (int k = 0; k < numC; k++)
    {
        clusterID = k;

        cout << "the points that are in cluster " << k << " are:" << endl;

        for (int j = 0; j < numP; j++)
        {
            pointClusterGroup = _kpoint.pointArray[j].cGroup;
            if (clusterID == pointClusterGroup)
            {
                cout << _kpoint.pointArray[j].pointId << " ";
            }
        }
    }
}

```

```

        }
    }
    cout << endl << endl;
}

}

int main()
{

    initialization();

    numC = _kcluster.uiCluster; // num of clusters, assign the number of clusters to this variable
    numP = _kpoint.uiPoint; // num of points, assign the number of points to this variable

    // was provided when creating the cuda file
    printf("{1,2,3,4,5} + {10,20,30,40,50} = {%d,%d,%d,%d,%d}\n",
        c[0], c[1], c[2], c[3], c[4]);

    // Run k-means algorithm
    kmeans();
    system("pause");
    return 0;
}

// Helper function for using CUDA to add vectors in parallel.
cudaError_t addWithCuda(int *c, const int *a, const int *b, unsigned int size, int numP)
{
    int *dev_a = 0;
    int *dev_b = 0;
    int *dev_c = 0;

    //devices to be used in allowing me to pass the data I require to be passed to the kernel
    int *dev_Updated = 0;
    bool *dev_Convergence = 0;
    Point *dev_PointArray = 0;
    Cluster *dev_Clusters = 0;

    cudaError_t cudaStatus;

    // Choose which GPU to run on, change this on a multi-GPU system.

```

```

cudaStatus = cudaSetDevice(0);
if (cudaStatus != cudaSuccess) {
    fprintf(stderr, "cudaSetDevice failed! Do you have a CUDA-capable GPU installed?");
    goto Error;
}

// Allocate GPU buffers for point array
cudaStatus = cudaMalloc((void**)&dev_PointArray, numP * sizeof(Point));
if (cudaStatus != cudaSuccess) {
    fprintf(stderr, "cudaMalloc failed!");
    goto Error;
}

// Allocate GPU buffers for cluster array
cudaStatus = cudaMalloc((void**)&dev_Clusters, numC * sizeof(Cluster));
if (cudaStatus != cudaSuccess) {
    fprintf(stderr, "cudaMalloc failed!");
    goto Error;
}

// Allocate GPU buffers for updated variable stored in array
cudaStatus = cudaMalloc((void**)&dev_Updated, sizeof(int));
if (cudaStatus != cudaSuccess) {
    fprintf(stderr, "cudaMalloc failed!");
    goto Error;
}

// Allocate GPU buffers for convergence variable stored in array
cudaStatus = cudaMalloc((void**)&dev_Convergence, sizeof(bool));
if (cudaStatus != cudaSuccess) {
    fprintf(stderr, "cudaMalloc failed!");
    goto Error;
}

cudaStatus = cudaMalloc((void**)&dev_c, size * sizeof(int));
if (cudaStatus != cudaSuccess) {
    fprintf(stderr, "cudaMalloc failed!");
    goto Error;
}

cudaStatus = cudaMalloc((void**)&dev_a, size * sizeof(int));
if (cudaStatus != cudaSuccess) {
    fprintf(stderr, "cudaMalloc failed!");
    goto Error;
}

```

```

}

cudaStatus = cudaMalloc((void**)&dev_b, size * sizeof(int));
if (cudaStatus != cudaSuccess) {
    fprintf(stderr, "cudaMalloc failed!");
    goto Error;
}

// Copy Point array from host memory to GPU buffers.
cudaStatus = cudaMemcpy(dev_PointArray, _kpoint.pointArray, numP * sizeof(Point), cudaMemcpyHostToDevice);
if (cudaStatus != cudaSuccess) {
    fprintf(stderr, "cudaMemcpy failed!");
    goto Error;
}

// Copy cluster array from host memory to GPU buffers.
cudaStatus = cudaMemcpy(dev_Clusters, _kcluster.Clusters, numC * sizeof(Cluster), cudaMemcpyHostToDevice);
if (cudaStatus != cudaSuccess) {
    fprintf(stderr, "cudaMemcpy failed!");
    goto Error;
}

// Copy updated array from host memory to GPU buffers.
cudaStatus = cudaMemcpy(dev_Updated, updated, sizeof(int), cudaMemcpyHostToDevice);
if (cudaStatus != cudaSuccess) {
    fprintf(stderr, "cudaMemcpy failed!");
    goto Error;
}

// Copy convergence array from host memory to GPU buffers.
cudaStatus = cudaMemcpy(dev_Convergence, convergence, sizeof(bool), cudaMemcpyHostToDevice);
if (cudaStatus != cudaSuccess) {
    fprintf(stderr, "cudaMemcpy failed!");
    goto Error;
}

cudaStatus = cudaMemcpy(dev_a, a, size * sizeof(int), cudaMemcpyHostToDevice);
if (cudaStatus != cudaSuccess) {
    fprintf(stderr, "cudaMemcpy failed!");
    goto Error;
}

cudaStatus = cudaMemcpy(dev_b, b, size * sizeof(int), cudaMemcpyHostToDevice);

```

```

if (cudaStatus != cudaSuccess) {
    fprintf(stderr, "cudaMemcpy failed!");
    goto Error;
}

// Launch a kernel on the GPU with one thread for each element. pass in point array, cluster array and variables needed
// kernel uses the block.x variable that was set earlier in the file for number of blocks, and the threads per block to state how many threads per block
distanceChecksKernel <<< block.x, threadsPerBlock >>>(dev_PointArray, dev_Clusters, desiredDimension, convergence[0], updated[0], numC);
// Check for any errors launching the kernel
cudaStatus = cudaGetLastError();
if (cudaStatus != cudaSuccess) {
    fprintf(stderr, "addKernel launch failed: %s\n", cudaGetErrorString(cudaStatus));
    goto Error;
}

// cudaDeviceSynchronize waits for the kernel to finish, and returns
// any errors encountered during the launch.
cudaStatus = cudaDeviceSynchronize();
if (cudaStatus != cudaSuccess) {
    fprintf(stderr, "cudaDeviceSynchronize returned error code %d after launching addKernel!\n", cudaStatus);
    goto Error;
}

// Copy updated point array back into point array
cudaStatus = cudaMemcpy( kpoint.pointArray, dev_PointArray, numP * sizeof(Point), cudaMemcpyDeviceToHost);
if (cudaStatus != cudaSuccess) {
    fprintf(stderr, "cudaMemcpy failed!");
    goto Error;
}

// Copy updated cluster array back into cluster array
cudaStatus = cudaMemcpy( kcluster.Clusters, dev_Clusters, numC * sizeof(Cluster), cudaMemcpyDeviceToHost);
if (cudaStatus != cudaSuccess) {
    fprintf(stderr, "cudaMemcpy failed!");
    goto Error;
}

// Copy updated value back into updated variable
cudaStatus = cudaMemcpy(updated, dev_Updated, sizeof(int), cudaMemcpyDeviceToHost);
if (cudaStatus != cudaSuccess) {
    fprintf(stderr, "cudaMemcpy failed!");
    goto Error;
}

```

```
// Copy output vector from GPU buffer to host memory.
cudaStatus = cudaMemcpy(convergence, dev_Convergence, sizeof(bool), cudaMemcpyDeviceToHost);
if (cudaStatus != cudaSuccess) {
    fprintf(stderr, "cudaMemcpy failed!");
    goto Error;
}
```

```
// Copy output vector from GPU buffer to host memory.
cudaStatus = cudaMemcpy(c, dev_c, size * sizeof(int), cudaMemcpyDeviceToHost);
if (cudaStatus != cudaSuccess) {
    fprintf(stderr, "cudaMemcpy failed!");
    goto Error;
}
```

Error:

```
cudaFree(dev_c);
cudaFree(dev_a);
cudaFree(dev_b);
```

```
//free the memory
cudaFree(dev_PointArray);
cudaFree(dev_Clusters);
cudaFree(dev_Convergence);
cudaFree(dev_Updated);
```

```
return cudaStatus;
}
```

GRADEMARK REPORT

FINAL GRADE

GENERAL COMMENTS

51 /100

Instructor

PAGE 1

PAGE 2

PAGE 3

PAGE 4

PAGE 5

PAGE 6

PAGE 7

PAGE 8

PAGE 9

PAGE 10

PAGE 11

PAGE 12

PAGE 13

PAGE 14

PAGE 15

PAGE 16

PAGE 17

PAGE 18

PAGE 19

PAGE 20

PAGE 21

PAGE 22

PAGE 23

PAGE 24

PAGE 25

PAGE 26

PAGE 27

PAGE 28

PAGE 29

PAGE 30

PAGE 31

PAGE 32

PAGE 33

PAGE 34

PAGE 35

PAGE 36

PAGE 37

PAGE 38

PAGE 39

PAGE 40

PAGE 41

PAGE 42

PAGE 43

PAGE 44

PAGE 45

PAGE 46

PAGE 47

PAGE 48

PAGE 49

PAGE 50

PAGE 51

PAGE 52

PAGE 53

PAGE 54

PAGE 55

PAGE 56

PAGE 57

PAGE 58

PAGE 59

PAGE 60

PAGE 61

PAGE 62

PAGE 63

PAGE 64

PAGE 65

PAGE 66

PAGE 67

PAGE 68

PAGE 69

PAGE 70

PAGE 71

PAGE 72

PAGE 73

PAGE 74

PAGE 75

PAGE 76

PAGE 77

PAGE 78

PAGE 79

PAGE 80

PAGE 81

PAGE 82

PAGE 83

PAGE 84

PAGE 85

PAGE 86

PAGE 87

PAGE 88

PAGE 89

PAGE 90

GRADING FORM: SMU GENERIC_GJ

NATHAN JAMES THOMSON**50.50****RESEARCH-INFORMED LITERATURE**

I would expect far more references here. They are all online too. Please avoid relying on too many online references- there are plenty of good quality texts and journal papers that you could have used to help you here.

4**KNOWLEDGE AND UNDERSTANDING**

You've provided a brief overview of the method, reliance on quoting others here. You have identified a suitable component of the algorithm to parallelise - Euclidian distance but why limit yourself to 3D here - a games context? ok but use of a higher dimensional vector is common in Game AI.

7.5**ANALYSIS**

Euclidian distance selected for parallelisation - good but use higher dimensions as mentioned above. This will make your GPU testing more relevant and produce more reasonable load/timing experiments. Low data sizes limit the extent of your timing results.

4.5**PRACTICAL APPLICATION AND DEP.**

You managed to evaluate code that implements 3D and 3D distance checking on both CPU and GPU. It's a shame you didn't manage to fully integrate the K-means into this. Use of larger datasets would have helped produce more interesting practical results.

32**SKILLS FOR PROFESSIONAL PRAC.**



Abstract included. Refs light. Report structured. Some additional diagrams would have helped support your discussions/ lit review.

2.5