# Report

## Introduction

The aim of this project was to implement k-means clustering on CPU and parallelize it for GPU and GPU, using tiling. The implementations operate 4 dimensional data, clustering it in 4 clusters. The number of dimensions was chosen to fully demonstrate the advantages of GPU over CPU.

## Theory and implementation

Figure . Reachability graph

### Sequential algorithm

Implemented sequential algorithm is very standard and basic version of k-means clustering. It is:

1. Initialize centroids (4 random coordinates were seeded)
2. Find closest centroid (change membership of a point)
3. Recalculate the means
4. Go to step 2 for 400 times

For testing purposes the clustered data was generated. As can be seen from Figures 1,2 the algorithm is finite and deadlock free. It has a few bottlenecks. They are calculation of new closest centroid and calculation of the new means. This particular implementation features a structure created for storing point’s data. It consists of dimi’s variables, where i - # of dimension and true cluster, current cluster variables.

### Parallel algorithm

Implemented algorithm employs the power of GPU to calculate the closest centroid.

1. Initialize centroids (4 random coordinates were seeded)
2. Copy data and centroids array to the GPU global memory
3. Find closest centroid (change membership of a point) on the GPU
4. Copy data back
5. Recalculate the means
6. Go to step 2 for 400 times

There were several options to implement on the GPU; however, the closest\_centroid function was chosen for two reason. First, comparing with recalculation the means, closest\_centroid occupies longer time, as it requires sqrt and pow math functions. Second, recalculate the means function is very simple, its GPU implementation won’t significantly increase performance.

Figure . Petri net

### Tiled parallel algorithm

Implemented algorithm employs the power of GPU’s shared memory to calculate the closest centroid.

1. Initialize centroids (4 random coordinates were seeded)
2. Copy data and centroids array to the GPU global memory
3. Find closest centroid (change membership of a point) on the GPU
   * 1. Copy point to shared memory (each thread copies 1 point to the specific shared memory cell)
     2. If first thread in the block, copy centroids’ coordinates
     3. Synchronize
     4. Compare distances and write the closest
     5. Synchronize
     6. Copy back to global memory
4. Copy data back
5. Recalculate the means
6. Go to step 2 for 400 times

## Discussion

Figure .Benchmarking (in seconds)

To demonstrate the difference in the performance between different implementations 100,000 4-dimensianal points were generated. 400 and 800 iterations were timed for three algorithms. 16 and 32 tiles were used (sequential algorithm does not use tiles). 32 tile size was predicted to be the fastest because 32 is the size of warp on the testing machine, consequently it utilizes the whole warp for calculations. Next, 100,000 is divisible by 32 without reminder, so no half warps were running. Both above reasons allow to gain benefit from spatial locality, allowing fast data transfer between global memory and registers (or shared memory).

## Conclusion

The aim of this project was achieved: three algorithms were implemented. As can be seen on the Figure 3. 32 tile size performed the best in 800 iterations in both parallel and tiled parallel algorithms and in 400 iterations in tiled parallel algorithm. According to Figure 3. the difference between parallel and tiled parallel implementations increases with the number of iterations, the time losses on data transfer between global and shared memory become insignificant.