My parallelization approach is largely the same as it was for the PThreads and OpenMP implementations from assignment 1. I divide process distances from each point to each medoid in parallel, and then within each cluster parallelize the computation of total distance to all other points in that cluster. I then use a min reduce on the distances in the second step in order to find the new medoid. The main changes are that I do some pre-processing to get all points in one cluster identified and don't have a Cluster struct to group points together, since Cluster Structs did not have a set memory size due to the variable number of points they did not work well with CUDA.

I have 4 main kernels, 3 of which are actually parallelizing work, and the 4th just reduces the amount of host and device mem copies needed. I also use a handful of Thrust and CUB functions as well. My clusterPoints kernel has each thread calculates distance to each medoid for a given point, and writes the ID of the medoid with the lowest distance to an array that tracks cluster assignment of each point. I considered writing all distances to an array, and then performing reductions on each num_groups chunk of the array, but this ends up having at least half a million floats, and cub::DeviceReduce cannot be called in kernels, so these must be done in sequence.

My next kernel is very simple and each thread just sets a flag in a flag array by looking at the cluster association array, and seeing if the value matches the current cluster. This array is used both to identify the indices of all points in some cluster, and the total number of points in the cluster using Thrust copy: if and a sum reduction with thrust

My last parallelization kernel generates total distances for each point in a cluster to use in medoid determination. Each thread calculates total distance to all other points for some point in the cluster. It writes these values to an array that can have an argmin reduction run on it to find the index of the new medoid.This kernel benefits least from increased threads in part due to the relatively low number of points in each kernel. It is very fast, but has to be run num_groups number of times. I thought about changing this kernel to be run in parallel, but that requires all pre-processing to identify points in each cluster (and number points in each cluster) before calling the kernel, and has memory issues.

My final kernel just updates the medoid stored in device memory to avoid having to copy to host, update and copy back to the device

I arranged threads in 1D blocks because I had issues with dynamic parallelism (calling kernels from another kernel, and there were limits to the benefits of doing different operations in parallel, especially due to the variable size of clusters. Additionally, based on most of the investigation and research on Cuda I did, 2D and 3D blocks are largely syntax abstractions, and everything is flattened to 1D when mapping onto warps anyway. 2D blocks could have been useful if dynamic parallelism had worked for me, but there still would have been some atomic operations that would need to be introduced. Memory costs would also become prohibitive if distance calculation components were written to arrays, due to points having 500 dimensions.

Below is the data On various block/thread configurations, grouped by total number of threads for each number of groups. Results are grouped by total number of threads, and averages of 3 runs. There was some resource contention occurring on the server while these runs were performed, (occasional 10+x slowdowns with no code change) so runs that encountered slowdowns due to resource competition were ignored.

Dynamic block size results are also included. Changes in groupings for some number of threads were stopped once performance decreased, as this usually indicated thread synchronization overhead was beginning to overtake benefits, and would not improve by adding more threads to each block. Increasing threads higher than the values below much higher usually saw benefits gained from the clustering kernel wiped out by additional overhead in the medoid assignment kernel, and net performance decreases.

CPU Serial and multithreaded times. Both using OMP solution from assignment 1 compiled without any optimization flags* with either 1 thread or 16 threads as command line arg.

Optimization flags were not used because they don’t do anything in cuda. Loop unrolling branch prediction and other CPU optimizations lead to much faster times that do not correspond to code implementation that cannot be replicated with nvcc compiler

<u>256 groups</u>

- 1 thread: 500.8432 seconds
- 16 threads 31.1096 seconds

<u>512 groups</u>

- 1 thread: 444.5396 seconds
- 16 threads 28.7445 seconds

<u>1024 groups</u>

- 1 thread: 459.6349 seconds
- 16 threads 29.3484 seconds

CUDA times

<u>256 groups</u>

512 threads:

32 blocks, 16 threads: 39.8462 seconds
16 blocks, 32 threads: 41.3377 seconds

2048 threads:

128 blocks, 16 threads: 31.2708 seconds
64 blocks, 32 threads: 27.3142 seconds
32 blocks, 64 threads: 27.4488 seconds
16 blocks, 128 threads: 34.3482 seconds

4096 Threads:

256 blocks, 16 threads: 30.6600 seconds
128 blocks, 32 threads: 28.7619 seconds
64 blocks, 64 threads: 27.7571 seconds
32 blocks, 128 threads: 28.6116 seconds

9192 threads:

512 blocks, 16 threads: 31.3757 seconds
256 blocks, 32 threads: 29.3497 seconds
128 blocks, 64 threads: 33.9578 seconds

Dynamic block size based on threads per block (should use about as many threads as numPoints)*
*times were a little slower for these runs, I think due to resource contention on the server

16 TPB: 31.6626 seconds
32 TPB: 29.9856 seconds
64 TPB: 31.4460

512 groups

512 threads:

32 blocks, 16 threads: 41.1214 seconds
16 blocks, 32 threads: 42.7452 seconds

2048 threads:

128 blocks, 16 threads: 30.8984 seconds
64 blocks, 32 threads: 28.7632 seconds
32 blocks, 64 threads: 29.2219 seconds
16 blocks, 128 threads: 36.121 seconds

4096 Threads:

256 blocks, 16 threads: 29.7134 seconds
128 blocks, 32 threads: 28.4108 seconds
64 blocks, 64 threads: 28.0760 seconds
32 blocks, 128 threads: 33.5407 seconds

9192 threads:

512 blocks, 16 threads: 40.2064 seconds
256 blocks, 32 threads: 28.6351 seconds
128 blocks, 64 threads: 35.8821 seconds

Dynamic block size based on threads per block (should use about as many threads as numPoints)

16 TPB: 28.0552 seconds
32 TPB: 28.5375 seconds
64 TPB: 32.9207 seconds

<u>1024 groups</u>

512 threads:

32 blocks, 16 threads: 56.6419 seconds
16 blocks, 32 threads: 59.2321 seconds

2048 threads:

128 blocks, 16 threads: 41.1325 seconds
64 blocks, 32 threads: 38.3198 seconds
32 blocks, 64 threads: 38.9222 seconds
16 blocks, 128 threads: 44.8281 seconds

4096 Threads:

256 blocks, 16 threads: 46.6629 seconds
128 blocks, 32 threads: 37.2828 seconds
64 blocks, 64 threads: 36.5080 seconds
32 blocks, 128 threads: 41.7060 seconds

9192 threads:

512 blocks, 16 threads: 29.9537 seconds

256 blocks, 32 threads: 37.5490 seconds
128 blocks, 64 threads: 47.3064 seconds

Dynamic block size based on threads per block (should use about as many threads as numPoints)

16 TPB: 36.9408 seconds
32 TPB: 35.6115 seconds
64 TPB: 39.6343 seconds