K-Means Parallelization in CUDA Assignment

Marco Trambusti
Matricola - 7135350
marco.trambusti1@edu.unifi.it

Abstract

The K-means algorithm, utilized for clustering points, is implemented in C++ first in a sequential version and then parallelize it through the use of CUDA, in order to compare and analyze the two versions. The parallel version uses a struct of arrays to represent points, ensuring high scalability and performance . Analysis of the runtime results shows a significant improvement in the parallel version compared to the serial version in term of speedup but introduces challenges in terms of efficiency.

1. Introduction

The purpose of the mid-term paper is to implement the K-Means algorithm, utilized for clustering points, in C++ first in a sequential version and then parallelize it through the use of CUDA, in order to compare and analyze the two versions . The dataset used is formed by 300147 values related to age and total spent of supermarket customers. The dataset is staggered into subsets of sizes that gradually increase to full size in order to compare performance with different numbers of points. At the end, analysis and performance comparisons of the two versions were performed. To evaluate the performance, the speedup obtained by switching from the sequential to the parallel version was measured considering also the increase in the number of threads allocated from 2 to 16.

1.1. The K-Means algorithm

The K-Means algorithm is a widely used clustering method for partitioning a data set into K distinct clusters.Formally it can be defined as the task of finding a partition $S = \S_1, S_2, ..., S_k\}$ where S satisfies:

$$\arg\min \sum_{i=1}^{k} \sum_{x \in S_i} \|x - \mu_i\|^2 \tag{1}$$

Although it is relatively simple to implement, its computational cost can become a limiting factor, especially with large volumes of data. Parallelization of the algorithm is an effective strategy to improve its performance.
The K-Means algorithm consists of a few basic steps:

1. **Initialization**: Random selection of K initial centers for clusters.
2. **Cluster assignment**: Each data point is assigned to the nearest centroid.
3. **Updating centroids**: Centroids are recalculated as the average of the points assigned to each cluster.
4. **Repetition**: The assignment and update steps are repeated until the centroids no longer change significantly or a maximum number of iterations is reached.

1.2. Parameters

The algorithm depends on certain parameters that need to be specified. First, the number K of clusters must be known and chosen a priori; In the project code, values of K have been defined in an indicative manner based on the point ranges used; alternatively, more complex solutions can be adopted to choose the best K given the points as Elbow Method, Silhouette Method or Gap Statistic. However, it is not among the objectives of the paper to use the optimal number of clusters so approximate values were used. After that, the way the centroids are defined and how they are initialized must be chosen : in the case of the paper, where the points lie in a two-dimensional Euclidean space, the centroids are initialized randomly among the points in the analyzed set , after which the centroid is in turn recalculated as a point in the space defined by the average of the points belonging to that cluster. Another important detail is the choice of distance: several solutions are possible; a square Euclidean distance is used in this project. The last parameter is the choice of the maximum number of iterations, which could vary from the nature and number of points and clusters; therefore, the number of iterations varies according to the saglions of points that are used. Another stopping criterion that can be defined is convergence; however, this was not implemented since it is not in the scope of the paper to obtain the most accurate clusters.

1.3. Considerations

Is an algorithm that converges very quickly, although it does not guarantee finding the global optimum. The quality of the final result depends mainly on the number and initial position of the clusters. Also, since we have initialized the clusters randomly, it is not certain that repeating the algorithm 2 times on the same points will give the same result.

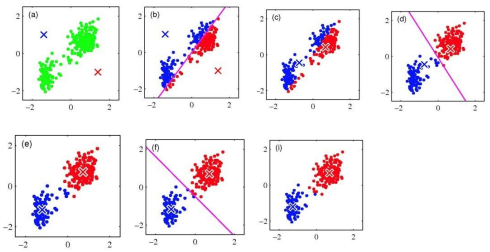

Figure 1. Example of K-Means

2. Implementation

2.1. Representation of Points

The points in the dataset are represented using a struct **Point**

```
struct Point {
   double x, y;
   int cluster;
   double minDist;

   Point() : x(0.0), y(0.0), cluster(-1),
    minDist(std::numeric_limits<double>::max()) {}

   Point(double x, double y) : x(x), y(y), cluster(-1),
    minDist(std::numeric_limits<double>::max()) {}

   [[nodiscard]] double distance(Point p) const {
    return (p.x - x) * (p.x - x) + (p.y - y) * (p.y - y);
   }
};
```

In the first lines we define the coordinates of a point, as well as the cluster to which it belongs and the distance from that cluster. Initially, the point does not belong to any cluster, and was arbitrarily set to -1. As a result, minDist was set to the maximum possible value.

```
double minDist = std::numeric_limits<double>::max();
```

We also define a method, which calculates the Euclidean distance (squared) between this point and the other point

```
[[nodiscard]] double distance(Point p) const {
        return (p.x - x) * (p.x - x) + (p.y - y) * (p.y - y);
}
```

Regardless of the versions the initial steps read the points from the csv file and normalize them via the function `readAndNormalizeCSV`. After that, you will initialize the clusters randomly among the set of points via the function `init_centroids`

2.2. K-means Sequential

The algorithm is divided into two main parts, **assignment to cluster** and **updating centroids** these steps will be repeated for the number of iterations specified a priori.
For the cluster assignment part, the code is constituted as follows:

```
void kmeans(std::vector<Point>& points, std::vector<Point> centroids,
    int k, int epochs) {
    for (int epoch = 0; epoch < epochs; ++epoch) {
        int nPoints[k]= {0};
        double sumX[k] = {0.0};
        double sumY[k] = {0.0};
        for (auto& point : points) {
            for (int i = 0; i < k; ++i) {
                double dist = point.distance(centroids[i]);
                if (dist < point.minDist) {
                    point.minDist = dist;
                    point.cluster = i;
                }
            }
            //append data to centroids
            int clusterId = point.cluster;
            nPoints[clusterId]++;
            sumX[clusterId] += point.x;
            sumY[clusterId] += point.y;
            //reset distance
            point.minDist = std::numeric_limits<double>::max();
        }
        ...
```

As can be guessed from the code for each point in the dataset the squared Euclidean distance is calculated and the minimum distance values and cluster index are updated, the clusters are initialized into an array of k points passed as an argument to the function. This part of the code encapsulates a critical point regarding the complexity of the algorithm, considering the execution of the algorithm for datasets with a tens or hundreds of thousands of points and with tens of clusters.
After deriving the cluster to which the point belongs, the values related to the sum of the number and coordinates of the points for the cluster identified as the closest (`nPoints`, `sumX`, `sumY`) are updated and the distance of the point is reset to the maximum possible value, in view of the next epoch.

As for the part about updating centroids and carried out in the following portion of the code:

```
    for (int i = 0; i < k; ++i) {
        if (nPoints[i] > 0) {
            double newX = sumX[i] / nPoints[i];
            double newY = sumY[i] / nPoints[i];
            centroids[i].x = newX;
            centroids[i].y = newY;
        }
    }
```

Where the new centoids are calculated as the average of the coordinates of the points belonging to the same cluster.

Using GnuPlot, the result of the K-Means algorithm for the maximum number of points (301487) is:

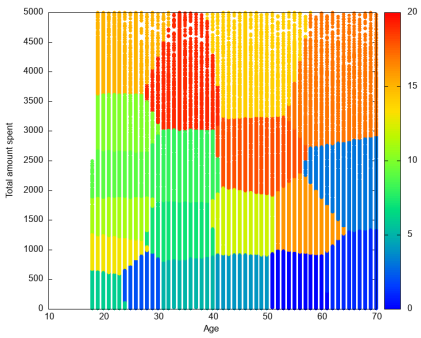

Figure 2. Result of K-Means with 301487 points

3. Parallelization

3.1. Parallelization Points

- **From Array of Structs to Struct of Array:** in order to make parallelization with CUDA more efficient by using the GPU, it is optimal to use a struct of arrays to represent points, effectively exploiting the computational power of the GPU. therefore we define the struct `Points` defined as follows :

```
struct Points {
    std::vector<float> flat_coord;
    std::vector<int> clusters;
    Points(const int numPoints,
        const std::vector<Point> &points) :
        flat_coord(numPoints * 2), clusters(numPoints, -1) {
            for (int i = 0; i < points.size(); i++) {
                flat_coord[i * 2] = points[i].x;
                flat_coord[i * 2 + 1] = points[i].y;
            }
        }
};
```

constructor transforms the array of struct `Point` into a flat array of point co-ordinates.

- **Cluster Assignment**: The cluster assignment function, designed to implement the K-means algorithm on GPUs via CUDA, exploits parallelism to optimize the allocation of data points to clusters. Each thread, identified by an index calculated from the current block and thread, independently processes a group of points (defined in the constant `POINTS_PER_THREAD=10`) in order to maximize the parallelization without decreasing the speedup, thus reducing the overall computation time. The kernel calculates the quadratic distance between the point and each cluster centroid, identifying the closest cluster through a parallel iteration over all centroids. This architecture makes full use of the GPU's computing power, ensuring high scalability and performance even for large data sets, making the K-means algorithm considerably more efficient than sequential CPU implementations. The previous code will change as follows:

```
__global__ void assignClusters(const float *d_datapoints,
    int *d_clust_assn, float *d_centroids, int *d_nPoints,
    float *d_sumX,float *d_sumY,int numPoints,const int k){

    int idx = blockIdx.x * blockDim.x + threadIdx.x;
    extern __shared__ float s_centroids[];
    int *s_nPoints=reinterpret_cast<int*>(&s_centroids[2*k]);
    float *s_sumX = reinterpret_cast<float *>(s_nPoints + k);
    float *s_sumY = s_sumX + k;
    if (threadIdx.x < k) {
       s_nPoints[threadIdx.x] = 0;
       s_sumX[threadIdx.x] = 0.0f;
       s_sumY[threadIdx.x] = 0.0f;
       s_centroids[2*threadIdx.x]
                          = d_centroids[2*threadIdx.x];
       s_centroids[2*threadIdx.x+1]
                          =d_centroids[2*threadIdx.x+1];
    }
    __syncthreads();
    if (idx < static_cast<int>(ceil(static_cast<float>(
         numPoints)/POINTS_PER_THREAD))) {
        for (int i = 0; i < POINTS_PER_THREAD; ++i) {
            int pointIdx = idx * POINTS_PER_THREAD + i;
            if (pointIdx < numPoints) {
                float minDist = FLT_MAX;
                int bestCluster = -1;
                float px = d_datapoints[2 * pointIdx];
                float py = d_datapoints[2 * pointIdx + 1];

                #pragma unroll 5
                for (int c = 0; c < k; ++c) {
                    float cx = s_centroids[2 * c];
                    float cy = s_centroids[2 * c + 1];
                    float dist = (px-cx)*(px-cx)+(py-cy)*(py-cy);

                    if (dist < minDist) {
                        minDist = dist;
                        bestCluster = c;
                    }
                }
                d_clust_assn[pointIdx] = bestCluster;
                atomicAdd(&s_nPoints[bestCluster], 1);
                atomicAdd(&s_sumX[bestCluster], px);
                atomicAdd(&s_sumY[bestCluster], py);
            }
        }
    }
    __syncthreads();
    if (threadIdx.x < k) {
        atomicAdd(&d_nPoints[threadIdx.x], s_nPoints[threadIdx.x]);
        atomicAdd(&d_sumX[threadIdx.x], s_sumX[threadIdx.x]);
        atomicAdd(&d_sumY[threadIdx.x], s_sumY[threadIdx.x]);
    }
}
```

In order to enhance the efficiency of the process, the centroids are loaded into the shared memory at the beginning and variables are defined in the shared memory concerning `nPoints, sumX, sumY` in order to speed up the memory access operations for these variables. After the initialization, the threads are synchronized and the centroid assignment part begins, using the unroll of the inner loop to reduce the GPU overhead for the for iterating on the centroids (which can be up to 20), the outer loop in this case does not need to unroll since it iterates over 10 points, which is a negligible value so the unroll could introduce more overhead. Finally, the threads synchronize and update the variables on the device related to `nPoints, sumX, sumY`.

- **Updating Centroids**: This step can also be optimized using a parallel loop.

```
__global__ void updateCentroids(float *d_centroids,
  int *d_nPoints, float *d_sumX, float *d_sumY, int k) {

    int idx = blockIdx.x*blockDim.x+threadIdx.x;

    if (idx < static_cast<int>(ceil(static_cast<float>(k)/
        CLUSTERS_PER_THREAD))) {
        for (int i = 0; i < CLUSTERS_PER_THREAD; ++i) {
            int cIdx = idx*CLUSTERS_PER_THREAD+i;
            if (cIdx < k) {
                if (d_nPoints[cIdx] > 0) {
                    d_centroids[2 * cIdx] =
                        d_sumX[cIdx]/d_nPoints[cIdx];
                    d_centroids[2 * cIdx + 1] =
                        d_sumY[cIdx]/d_nPoints[cIdx];
                }
            }
        }
    }
}
```

The update centroids function, implemented as a CUDA kernel, is designed to update cluster centroids in the K-means algorithm using the parallelism offered by GPUs. As in the cluster assignment kernel, each thread is identified by an index calculated from the current block and thread and independently processes a group of centroids (defined in the constant `CLUSTERS_PER_THREAD=5`) calculating the centroid id as `int cIdx = idx*CLUSTERS_PER_THREAD+i;` and updating and normalizing the centroids coordinates of device variable `d_centroids`.

- **`kmeans_cuda` function**: implements the K-means algorithm using CUDA to accelerate the clustering process on the GPU. This function starts by allocating memory on the GPU for data points, cluster allocations, centroids and cluster sizes.

  – **Memory Allocation**

```
int* clust_assn = new int[numPoints];

float *d_datapoints;
int *d_clust_assn;
float *d_centroids;
int *d_nPoints;
float *d_sumX;
float *d_sumY;

cudaMalloc(&d_datapoints, numPoints * 2 * sizeof(float));
cudaMalloc(&d_clust_assn, numPoints * sizeof(int));
cudaMalloc(&d_centroids, k * 2 * sizeof(float));
cudaMalloc(&d_nPoints, k * sizeof(int));
cudaMalloc(&d_sumX, k * sizeof(float));
cudaMalloc(&d_sumY, k * sizeof(float));
```

An array for cluster allocations (`clust_assn`) is allocated on the CPU, while memory is allocated on the GPU for `datapoints`, `d_clust_assn`, `d_centroids`, `d_nPoints`, `d_sumX` and `d_sumY`.

  – **Point transformation in Struct of Array**

```
auto pointsSoA = Points(numPoints, points);
cudaMemcpy(d_datapoints,pointsSoA.flat_coord.data(),
    numPoints*2*sizeof(float),cudaMemcpyHostToDevice);
auto centroidsSoA = Points(k, centroids);
cudaMemcpy(d_centroids, centroidsSoA.flat_coord.data(),
    k * 2 * sizeof(float), cudaMemcpyHostToDevice);
```

The points and centroids are converted into a one-dimensional array for the memory of the GPU.

  – **Setting Block Size and Size of Shared Memory**

```
float totalThreads=
     static_cast<float>(numPoints)/POINTS_PER_THREAD;
int numBlocks=
     static_cast<int>(ceil(totalThreads/numThreads));
float totalThreadsB =
ceil(static_cast<float>(numClusters)/CLUSTERS_PER_THREAD);
int numBlocksB=
    static_cast<int>(ceil(totalThreadsB/numThreads));
size_t sharedMemSize =
    (4 * k * sizeof(float))+(k*sizeof(int));
```

The number of blocks required for the GPU is calculated based on the total number of points, points handled per thread and threads per block. the value is calculated in the function `main()`. The size of the shared memory is computed to handle the centroid array and the arrays for the sums of the coordinates and the number of points assigned to the centroids.

  – **Cycle of Epochs**

```
for (int epoch = 0; epoch < epochs; ++epoch) {
    cudaMemset(d_nPoints, 0, k * sizeof(int));
    cudaMemset(d_sumX, 0, k * sizeof(float));
    cudaMemset(d_sumY, 0, k * sizeof(float));

    double startClusterAssign = omp_get_wtime();
    assignClusters<<<numBlocksA, numThreads, sharedMemSize>>>(
        d_datapoints, d_clust_assn, d_centroids, d_nPoints,d_sumX,
                    d_sumY, numPoints, k);
            ...
    cudaDeviceSynchronize();
    updateCentroids<<<numBlocksB, numThreads>>>(d_centroids,
        d_nPoints, d_sumX, d_sumY, k);
    cudaDeviceSynchronize();
```

For each epoch, the kernel `assignClusters` is invoked to assign points to clusters. The use of `cudaDeviceSynchronise()` ensures that all threads complete the assignment before proceeding. The kernel `updateCentroids` is invoked, which recalculates the values of the centroids by normalizing them, as described above.

  – **Copying Cluster Assignments and Memory Cleaning**

```
cudaMemcpy(clust_assn, d_clust_assn,
    numPoints * sizeof(int),
    cudaMemcpyDeviceToHost);

for (int i = 0; i < numPoints; ++i) {
    points[i].cluster = clust_assn[i];
}

delete[] clust_assn;
cudaFree(d_datapoints);
cudaFree(d_clust_assn);
cudaFree(d_centroids);
cudaFree(d_clust_sizes);
```

Finally, the cluster allocations are copied to the host array and the allocated memory is freed on the GPU and CPU.

The function `kmeans_cuda` implements the K-means algorithm by harnessing the power of the GPU, using optimized techniques to efficiently manage data and improve performance. The detailed analysis proposed in this paper highlights the importance of each step in the implementation of the clustering algorithm.

4. Results and Performance Evaluation

4.1. Setup

The tests were carried out on a machine with an Intel Core i7-6700K processor, comprising four cores and eight threads, and a NVIDIA GeForce GTX 970 GPU. Due to Hyper-Threading technology, the processor is capable of handling up to eight instruction streams simultaneously. The GTX 970, based on Maxwell architecture, features 1664 CUDA cores, a base clock speed of 1050 MHz, 4 GB of GDDR5 memory with a 256-bit memory interface, and a bandwidth of 224.4 GB/s . The tests were conducted on data sets of varying sizes with differing numbers of clusters and epochs, contingent on the size of the data set. The objective of the tests is to compare the sequential version, corresponding to the version with a single thread, and the parallel version with a greater number of threads (32,64, 128, 256,512, 1024). It was possible to ascertain the ***speedup*** and ***efficiency*** parameters:

$$S_p = t_s/t_p; E = S_p/n_{totThreads}$$

With: $n_{totThreads} = n_{threads} * n_{BlocksClusterAssignment} + n_{threads} * n_{BlockCentroidsUpdate}$ These tests are used to evaluate the performance of parallel version. The results are averaged over five runs.

4.2. Results and Evaluation

The analysis of runtime results showed a significant improvement in the parallel version compared to the serial version, with noticeable performance differences depending on the size of the data sets.

Analysis of results

4.2.1 Duration

The execution duration of the KMEANS algorithm varies significantly depending on the number of points and threads per block. For a small number of points (500), the duration is very short, with an execution time of 0.0011 seconds with a single thread. However, by increasing the number of threads, the duration does not decrease proportionally, indicating a thread management overhead. For a larger number of points (301487), the duration with a single thread is 22.8738 seconds, but it decreases dramatically to 0.1279 seconds with 32 threads. This shows the effectiveness of parallelization, although increasing the number of threads beyond a certain point does not lead to significant improvements

4.2.2 Speedup

1. **Small data set (500 points)**: Performance doesn't increase with additional threads due to the parallelism overhead. The speedup remains around 0.02 regardless of the number of threads (64, 128, 256, 512, or 1024), indicating a very weak sublinear trend. This suggests that for very small datasets, the overhead associated with thread management outweighs the advantages of parallelization.

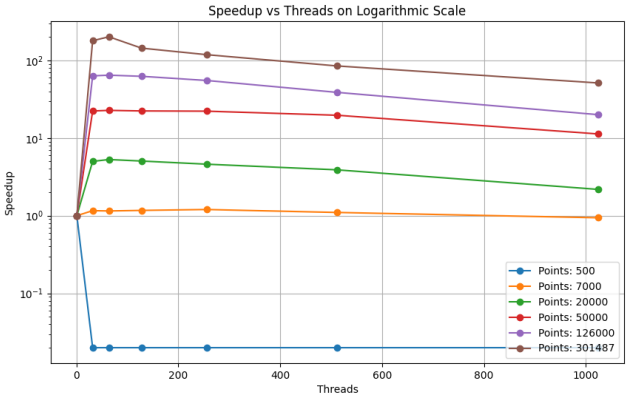

Figure 3. Speedup

2. **Medium data set (7000 points)**: speedup starts to show a slight improvement, but still remains low. With 32 threads, the speedup is 1.16, and with 128 threads it is 1.17. This suggests a sub-linear trend, where an increase in the number of threads does not lead to a proportional increase in speedup.

3. **Large data set size (20000 points)**: speedup shows a more obvious trend. With 32 threads, the speedup is 5.00, and with 64 threads it is 5.28.This indicates a sub-linear trend, where an increase in the number of threads leads to an improvement in speedup, but not proportionally.

4. **Very large data set (50000, 126000 and 301487 points)**: For 50000 points, the speedup becomes much

Points	Threads	Duration (s)	Speedup	Efficiency
500	1	0.0011	1.00	1.0000
500	32	0.0637	0.02	0.0002
500	64	0.0571	0.02	0.0002
500	128	0.0541	0.02	0.0001
500	256	0.0537	0.02	0.00004
500	512	0.0538	0.02	0.00002
500	1024	0.0571	0.02	0.00001
7000	1	0.0680	1.00	1.0000
7000	32	0.0586	1.16	0.0016
7000	64	0.0589	1.15	0.0015
7000	128	0.0583	1.17	0.0013
7000	256	0.0567	1.20	0.0012
7000	512	0.0616	1.10	0.0007
7000	1024	0.0719	0.94	0.0005
20000	1	0.3090	1.00	1.0000
20000	32	0.0619	5.00	0.0024
20000	64	0.0585	5.28	0.0025
20000	128	0.0612	5.05	0.0023
20000	256	0.0672	4.60	0.0020
20000	512	0.0795	3.89	0.0015
20000	1024	0.1415	2.18	0.0007
50000	1	1.4193	1.00	1.0000
50000	32	0.0637	22.27	0.0044
50000	64	0.0626	22.68	0.0044
50000	128	0.0637	22.29	0.0042
50000	256	0.0641	22.14	0.0041
50000	512	0.0723	19.62	0.0035
50000	1024	0.1258	11.28	0.0018
126000	1	5.1135	1.00	1.0000
126000	32	0.0811	63.05	0.0050
126000	64	0.0795	64.30	0.0051
126000	128	0.0822	62.23	0.0049
126000	256	0.0930	54.99	0.0042
126000	512	0.1323	38.65	0.0029
126000	1024	0.2561	19.97	0.0014
301487	1	22.8738	1.00	1.0000
301487	32	0.1279	178.82	0.0059
301487	64	0.1140	200.73	0.0066
301487	128	0.1595	143.41	0.0047
301487	256	0.1937	118.10	0.0039
301487	512	0.2705	84.58	0.0028
301487	1024	0.4475	51.11	0.0016

Table 1. Performance Data

more pronounced. With 32 threads, the speedup is 22.27, and with 64 threads it is 22.68. This suggests an almost linear trend, where increasing the number of threads leads to a significant increase in speedup, but with a slight drop in efficiency beyond a certain point.For 126000 points, speedup continues to show an almost linear trend. With 32 threads, the speedup is 63.05, and with 64 threads it is 64.30. This indicates that the parallelization is very efficient, with a trend approaching linearity, but with a slight decrease in efficiency as the number of threads increases. For 301487 points, the speedup reaches the highest values and shows an initial exponential trend which then stabilizes. With 32 threads, the speedup is 178.82, and with 64 threads it is 200.73. This indicates an initial exponential trend where the increase in the number of threads leads to a very rapid increase in speedup. However, with 128 threads, the speed-up drops to 143.41, suggesting that beyond 64 threads, the thread management overhead begins to negatively affect performance.

The speedup analysis shows that for small datasets (500 and 7000 points), the trend is sub-linear due to the thread management overhead. For medium-sized datasets (20000 and 50000 points), the trend becomes almost linear, with a significant improvement in speedup. For very large datasets (126000 and 301487 points), the trend is initially exponential, but stabilizes and may even decrease beyond a certain number of threads due to management overhead. This suggests that parallelization is more effective for large datasets, but it is important to find an optimal balance between the number of threads and the workload to maximize performance. Thus, a clear pattern is observed: the greater the number of points, the greater the ability to exploit threads effectively up to a certain point. Beyond a certain number of threads, speedup stabilizes, suggesting saturation in parallelization operations. This is indicative of Amdahl's law, according to which the advantages of parallelism decrease as the number of threads increases due to the limit determined by the non-parallelizable part of the code. This trend may also be due in part to the physical characteristics of the machine on which the experiments were conducted, in addition to the parallelism itself.
By looking specifically at the speedup of the cluster assignment and centroids update part, it is possible to deepen the analysis and criticality of kernel parallelization. Analyzing

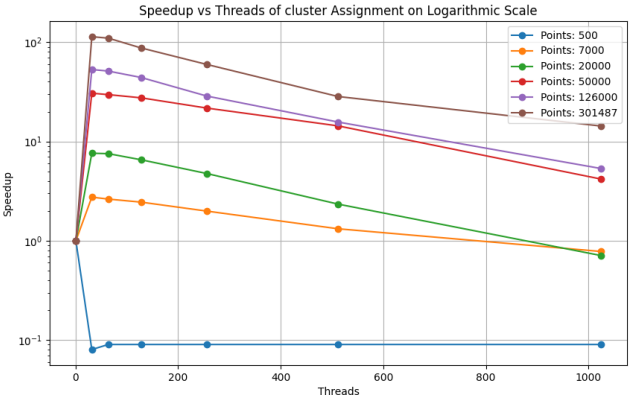

Figure 4. Cluster Assignment Speedup

specifically the speedup of the cluster assignment we note that speedup is present and for fairly large and super-linear sets of points for 32 and 64 threads per block, while for a greater number of threads there is a progressive degradation of speedup until there is about 10^{-1} compared to the initial threads, which suggests the presence of overhead in the management of kernel operations given by the presence of read and write operations in both shared and device memory and the presence of cycles and conditions, as well as generic operations for parallelization. As far as the kernel

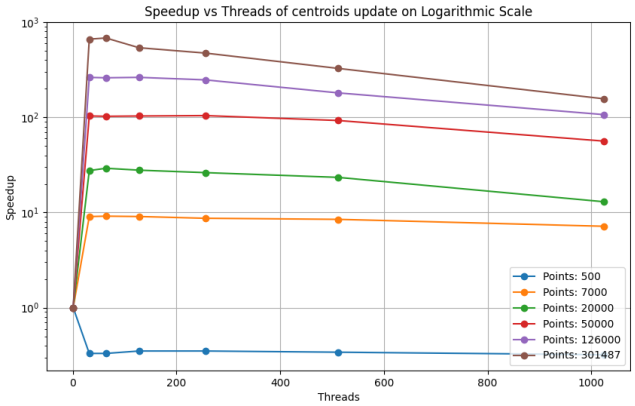

Figure 5. Centroids Update Speedup

of centroid updating is concerned, there is a roughly linear trend for smaller sets of points (up to 20000) and a super-linear trend for larger sets with 32 and 64 threads, for a larger number of threads the speedup tends to stabilize with slight decreases on 1024 threads per block up to 126000 points while for 301487 there is a more marked decrease in speedup.

4.2.3 Efficiency

Efficiency measures how well computing resources are utilized during parallelization.

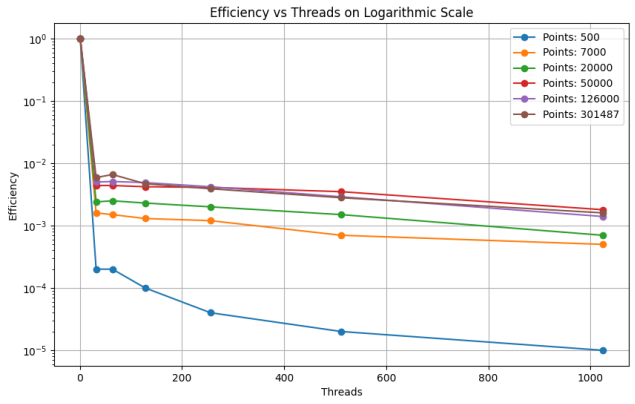

Figure 6. Efficiency

For small datasets (500 points), the efficiency is extremely low, with values falling as low as 0.00001 with 1024 threads. This indicates that the overhead of thread management outweighs the benefits of parallelization. For larger datasets (301487 points), the efficiency is better, but still not optimal, with a maximum value of 0.0066 with 64 threads.This suggests that although parallelization improves performance, efficiency decreases as the number of threads increases, probably due to the overhead of synchronization and communication between threads.

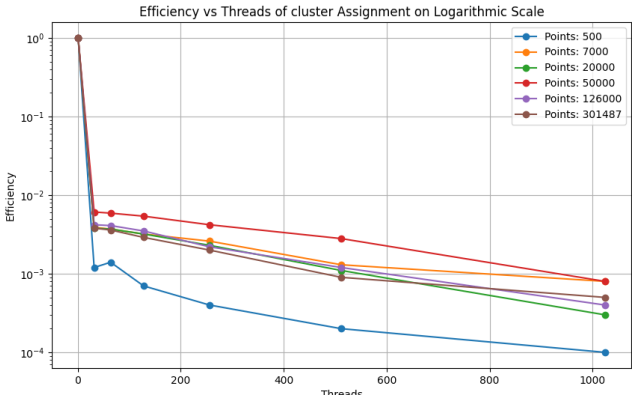

Figure 7. Cluster Assignment Efficiency

The efficiency for the cluster assignment part remains very low with the lowest values for small datasets and as the number of threads increases, emphasising the fact that the overhead introduced by the threads outweighs the benefits in terms of their parallelism.

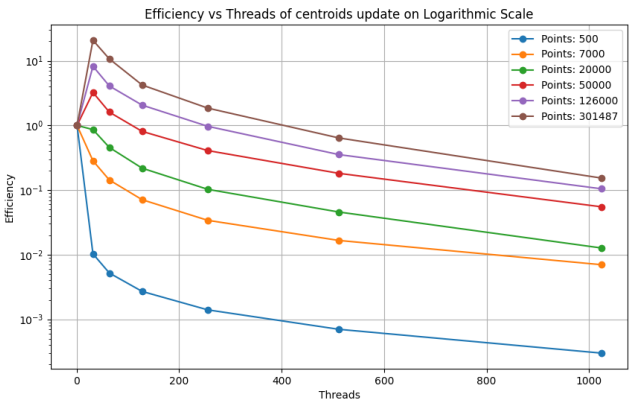

Figure 8. Centroids Update Efficiency

Similar to the cluster allocation kernel, the efficiency in the centroid update kernel tends to decrease as the number of threads increases, but in this case a more promising initial behaviour in terms of efficiency and speedup is observed, especially for larger datasets. Analysis of the centroid update kernel shows a significant improvement with increasing dataset size, allowing efficient use of available

resources. The increased efficiency and improved performance with 32-thread configurations indicate that for larger datasets, a well-designed thread configuration can lead to optimal results. However, it is important to remember that for smaller datasets, the reduced number of threads may be more efficient, minimizing overhead and maximizing efficiency.

5. Conclusions

Parallelization significantly enhances performance for large datasets, with speedups reaching 200.73 for 301,487 points using 64 threads. Performance gains are not linear with thread count. Smaller datasets showed minimal speedup due to overhead from thread management, resulting in low efficiency scores. Different parts of the K-Means algorithm respond uniquely to parallelization. The cluster assignment kernel experienced higher overhead with more threads, whereas the centroid updating kernel showed better efficiency at 32 and 64 threads for larger datasets.
In summary, CUDA parallelization of KMEANS shows significant benefits for large datasets, with noticeable improvements in durability and speedup. However, efficiency is very low and decreases as the number of threads increases, indicating the need for an optimal balance between the number of threads and workload to maximize performance