K-Means Algorithm

Implementation of sequential and parallel version using CUDA

Marco Trambusti - 7135350

A.A. 2024/2025

Introduction

Purpose of the Assignement

- Implement the K-Means algorithm in C++ in both sequential and parallelized versions using CUDA.
- Dataset: 300147 2D points related to age and total spending of supermarket customers.
- Performance comparison between sequential and parallel version using speedup and efficiency .

K-Means Algorithm

Def.: The K-Means algorithm is a widely used clustering method for partitioning a data set into K distinct clusters

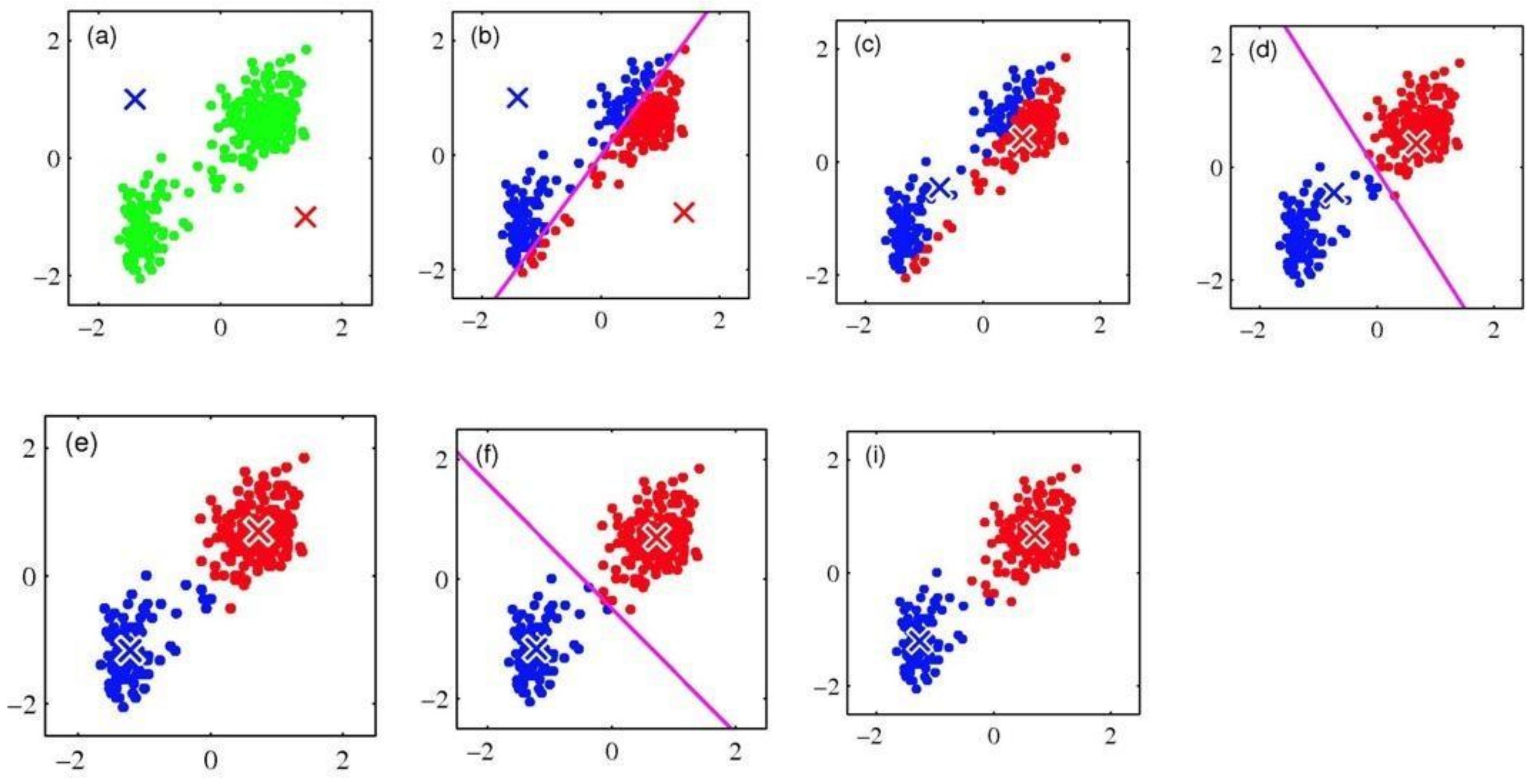
Steps:

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

Stopping criterion used: maximum number of iterations.

Algorithm Parameters

- **Number of clusters (K):**
  - Specified a priori
  - Defined in an indicative manner (alternatively could have been used methods like Elbow, Silhouette or Gap Static)
- **Initialization of centroids:**
  - Randomly among the data points
- **Choice of distance metric:**
  - squared Euclidean distance
- **Maximum number of iterations**



Example of K-Means

Sequential Implementation

- The project is in C++
- Chosen CUDA for the parallel implementation.
- 2D points points are readed from a csv file and normalized
- Clusters are initialized randomly;

The points in the dataset are represented using a struct **Point**, composed by:

- x, y coordinates (double)
- cluster (int);
- minDist (double)
- distance(Point p) (method)

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.

Sequential Implementation

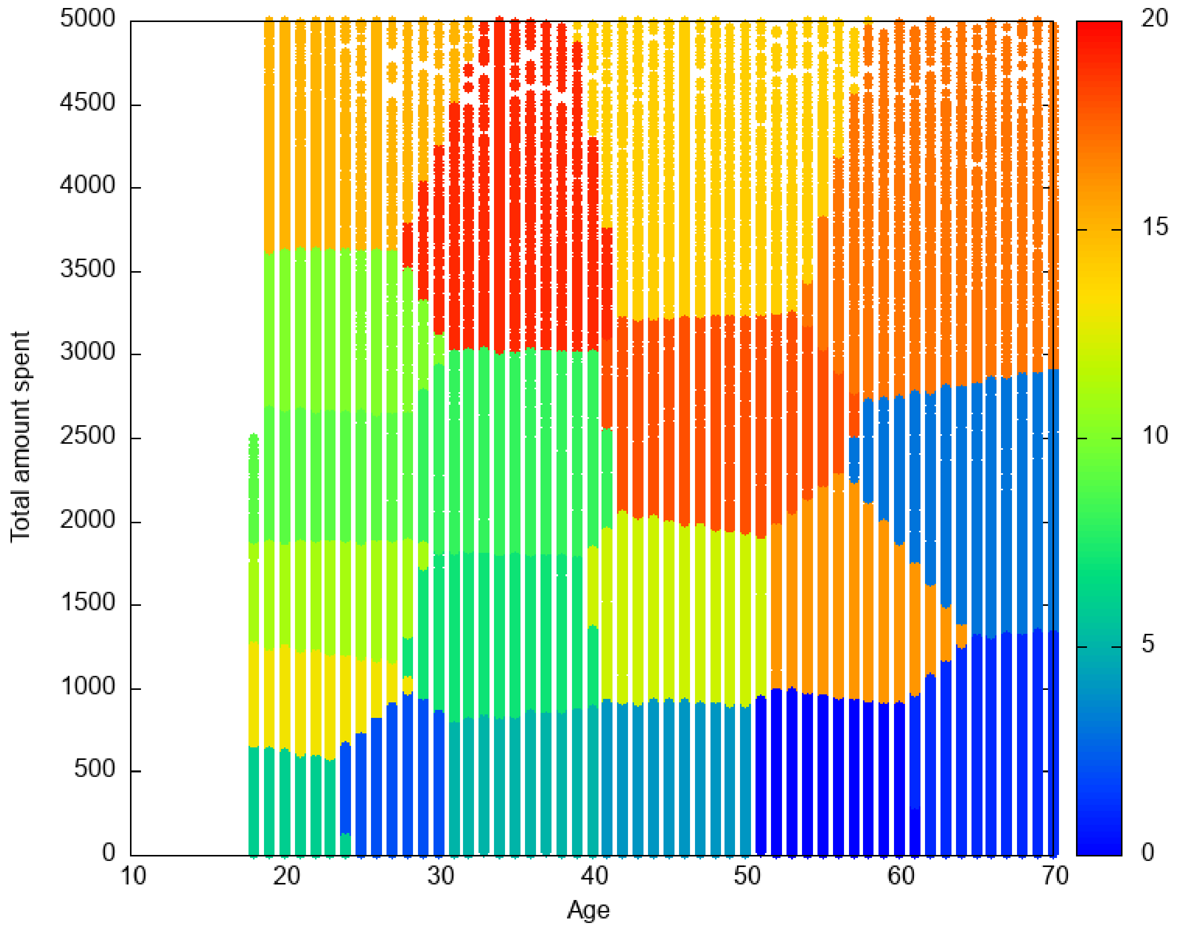
Kmeans method: Cluster assigment part

```
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();
        }
        ...
```

Sequential Implementation

Kmeans method: Centroid Update 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;
    }
}
```



Example of results plotted using GnuPlotted

Parallel Implementation

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

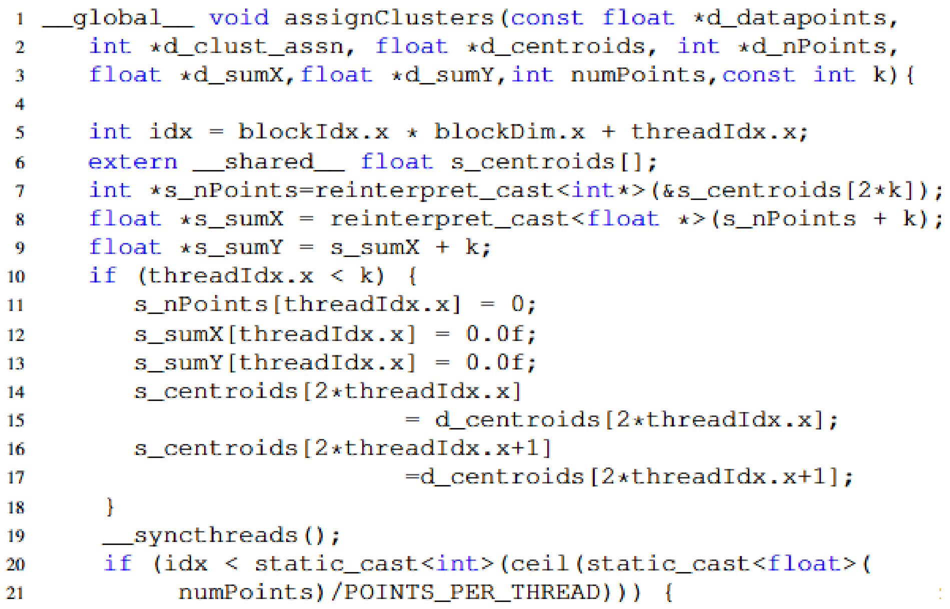
```
struct Points {
    std::vector<float> flat_coord;
    std::vector<int> clusters;
    Points(const int numPoints,
        const std::vector<Point> &points) :
        flat_coord(numPoints * 2), clusters(numPoints, -1)
            for (int i = 0; i < points.size(); i++) {
                flat_coord[i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

Cluster Assignment

- **Parallelism Exploited:**
  - The cluster assignment function implements the K-means algorithm on GPUs using CUDA to maximize parallelism.

- **Thread-Based Processing:**
  - Each thread is identified and processes a group of data points independently.
  - Defined constant POINTS_PER_THREAD=10 optimizes parallelization and maintains speedup.

Cluster Assignment

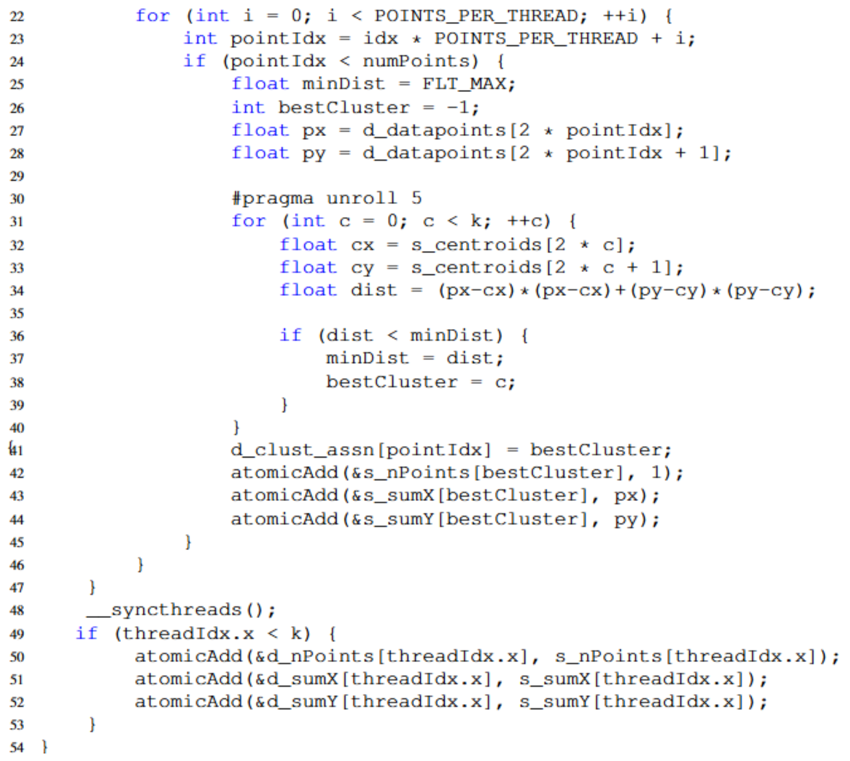


```
__global__ void assignClusters(const float *d_datapoints,
    int *d_clust_assn, float *d_centroids, int *d_nPoints,
    float *d_sumX,float *d_sumY,int numPoints,const int k){

    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))) {
```

- centroids are loaded into the shared memory at the beginning and variables nPoints, sumX, sumY are defined in the shared memory to speed up the memory access operations for these variables.
- After the initialization, the threads are synchronized

Cluster Assignment



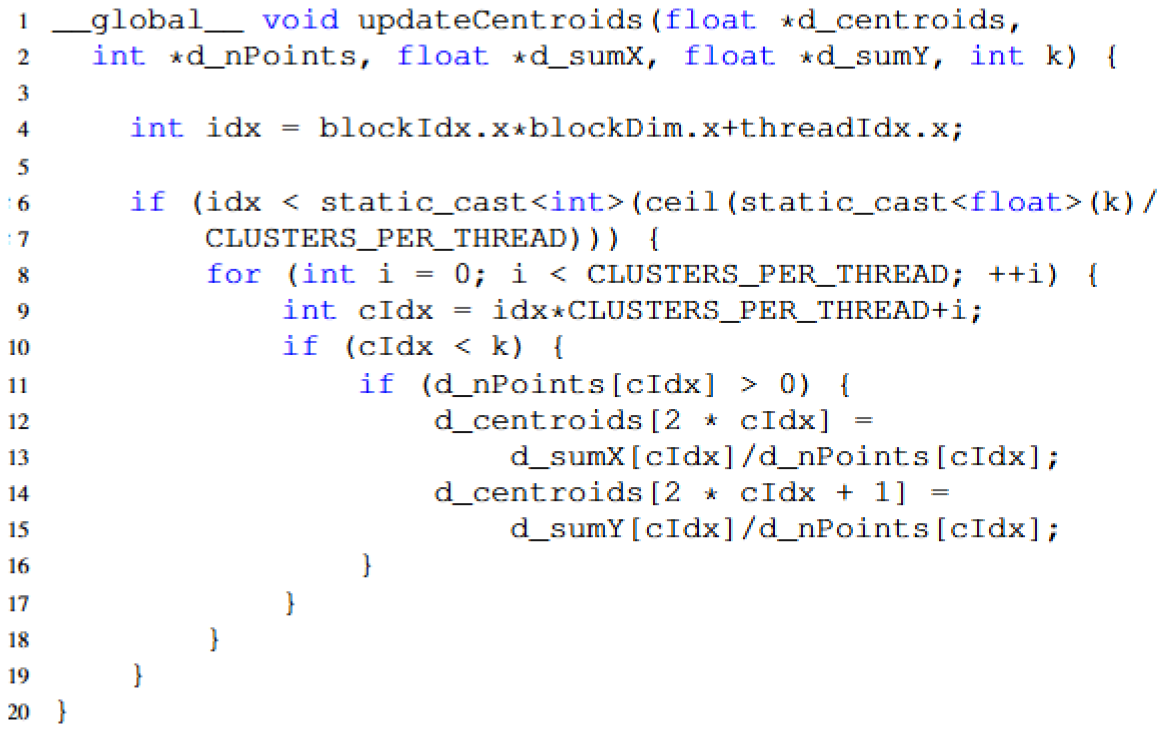
```
        for (int i = 0; i < POINTS_PER_THREAD; ++i) {
            int pointIdx = idx * POINTS_PER_THREAD + i;
            if (pointIdx < numPoints) {
                float minDist = FLT_MAX;
                int bestCluster = -1;
                float px = d_datapoints[2 * pointIdx];
                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]);
    }
}
```

- The inner loop, responsible for iterating over the centroids (up to 20), is unrolled to reduce GPU overhead.
- The outer loop does not require unrolling as the value is negligible
- Post centroid assignment, threads synchronize again.
- The relevant variables on the device, including nPoints, sumX, and sumY, are updated to reflect the latest computation results

Centroids Update



```
__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];
                }
            }
        }
    }
}
```

Each thread 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)

Results and Performance Evaluation

Setup:

- Tests conducted on an Intel Core i7-6700K processor (4 cores and 8 threads), and a NVIDIA GeForce GTX 970 GPU, 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.
- Data sets of various sizes with a commensurate number of clusters and epochs.
- The results are averaged across five runs.

Results and Performance Evaluation

Duration:

Average execution time of each configuration.

- operations on larger data sets take significantly longer.

Small Data Set (500 points):

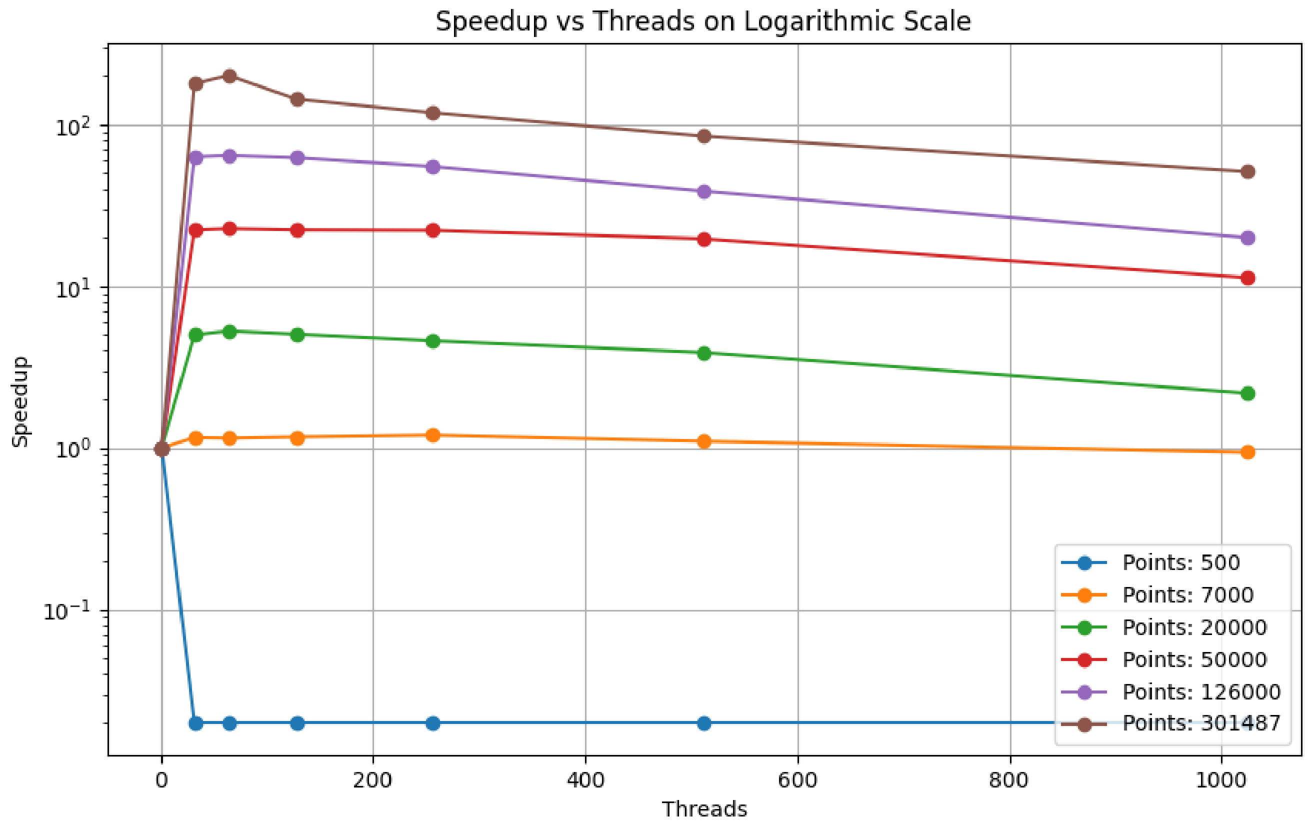
- 0.0011 seconds (1 thread)
- 0.0637 seconds (32 threads): Duration does not decrease with additional threads indicates management overhead.

Large Data Set (301,487 points):

- 22.8738 seconds (1 thread)
- 0.1279 seconds (32 threads) - shows effectiveness of parallelization.

Results and Performance Evaluation

Speedup: Ratio of sequential to parallel execution time.



Speedup:

Small Data Sets (500 & 7000 points):

- Speedup ~0.02; minimal improvement with additional threads due to overhead.

Medium Data Set (20,000 points):

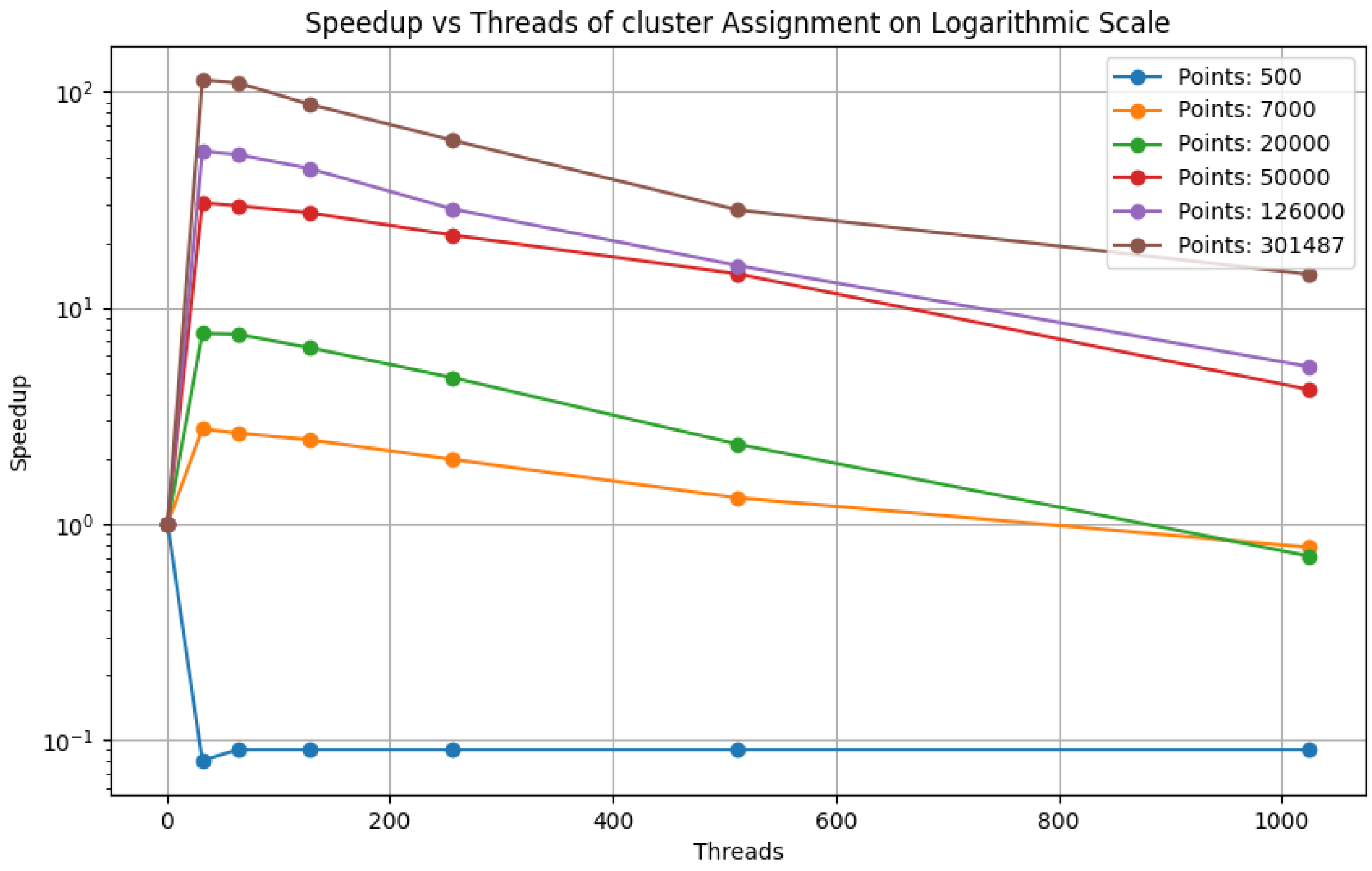
- Speedup: 5.00 (32 threads), 5.28 (64 threads).
- Indicates a sub-linear trend.

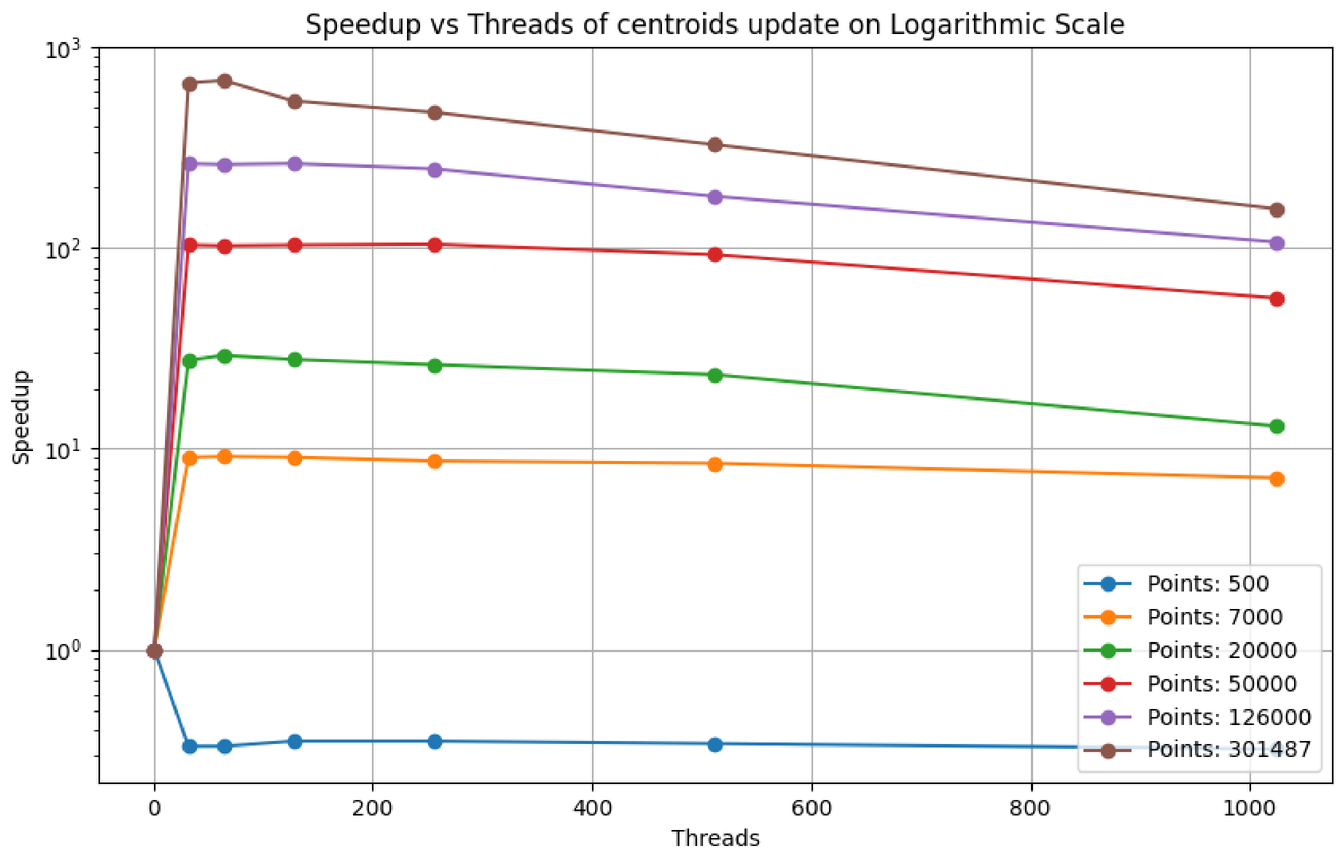
Large Data Sets (50,000, 126,000, 301,487 points):

- 50,000 points: Speedup = 22.27 (32), 22.68 (64).
- 126,000 points: Speedup = 63.05 (32), 64.30 (64).
- 301,487 points: Speedup = 178.82 (32), peaks at 200.73 (64), then drops with 128 threads.

Beyond a certain number of threads, speedup almost stabilizes, suggesting saturation

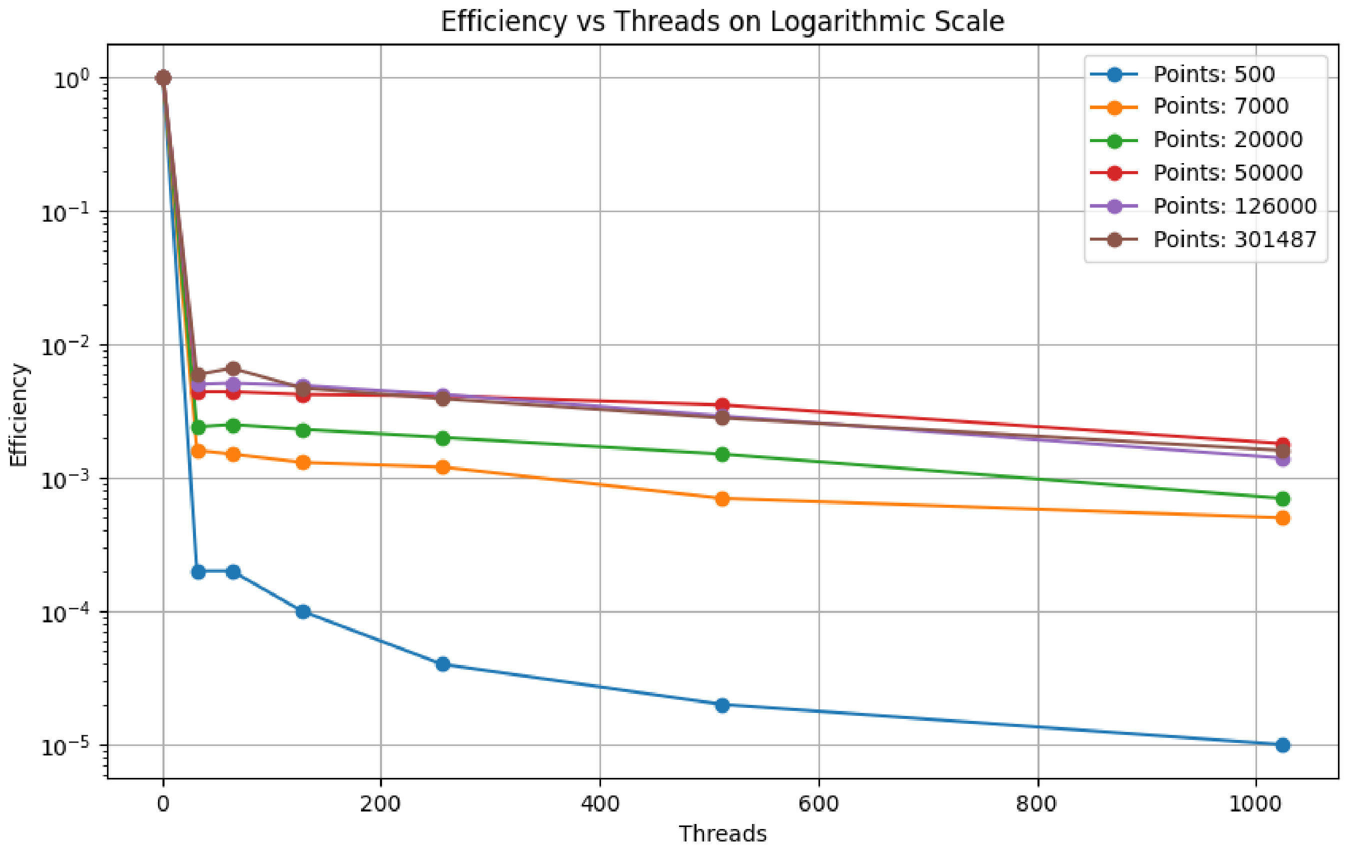
Most tests show optimal performance gains between 32 and 64 threads


Speedup vs Threads of cluster Assignment on Logarithmic Scale
Speedup
Threads
10^{2}
10^{1}
10^{0}
10^{-1}
0
200
400
600
800
1000
Points: 500
Points: 7000
Points: 20000
Points: 50000
Points: 126000
Points: 301487


Speedup vs Threads of centroids update on Logarithmic Scale
Speedup
Threads
10^0
10^1
10^2
10^3
0
200
400
600
800
1000
Points: 500
Points: 7000
Points: 20000
Points: 50000
Points: 126000
Points: 301487

Results and Performance Evaluation

Efficiency: Ratio of Speedup to Number of Threads.



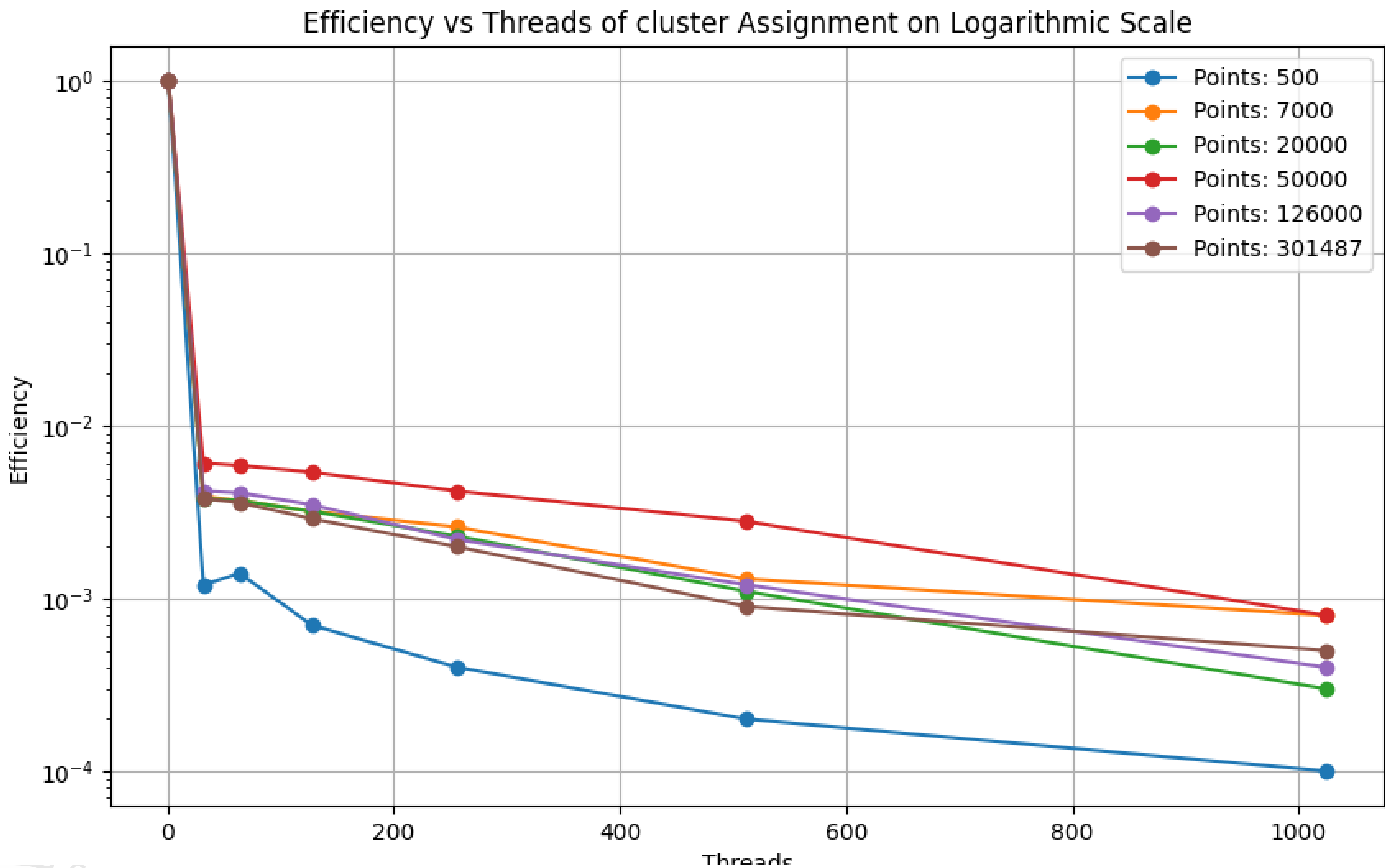
Efficiency:

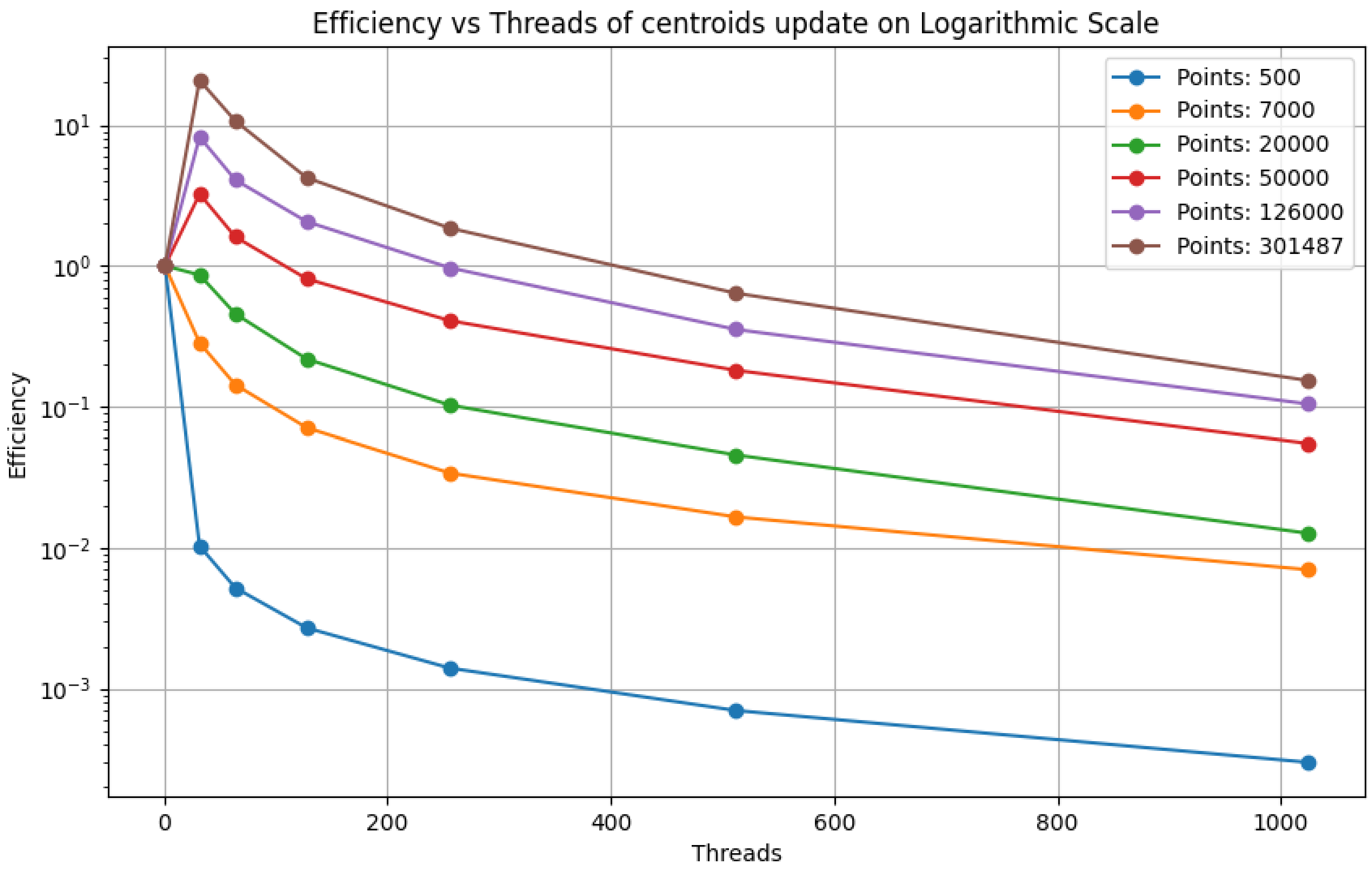
Smaller Datasets:

- Extremely low efficiency (e.g., 0.00001 with 1024 threads).

Larger Datasets:

- Maximum efficiency of 0.0066 with 64 threads.
- Decreasing efficiency as thread count increases due to overheads.


Efficiency vs Threads of cluster Assignment on Logarithmic Scale
Efficiency
Threads
10^0
10^{-1}
10^{-2}
10^{-3}
10^{-4}
0
200
400
600
800
1000
Points: 500
Points: 7000
Points: 20000
Points: 50000
Points: 126000
Points: 301487


Efficiency vs Threads of centroids update on Logarithmic Scale
Points: 500
Points: 7000
Points: 20000
Points: 50000
Points: 126000
Points: 301487
Efficiency
Threads
10^1
10^0
10^{-1}
10^{-2}
10^{-3}
0
200
400
600
800
1000

Conclusions

General Findings:

- Significant Speedup performance gains for large datasets.
- Efficiency is very low and decreases as the number of threads increases

Performance Characteristics:

- Minimal speedup for smaller datasets due to overhead.
- Different KMEANS components respond variably to parallelization.

Key Takeaway:

- CUDA parallelization is effective for large datasets, but efficiency diminishes with excessive threads; a balance is crucial for optimal performance.