CUDA-ACCELERATED

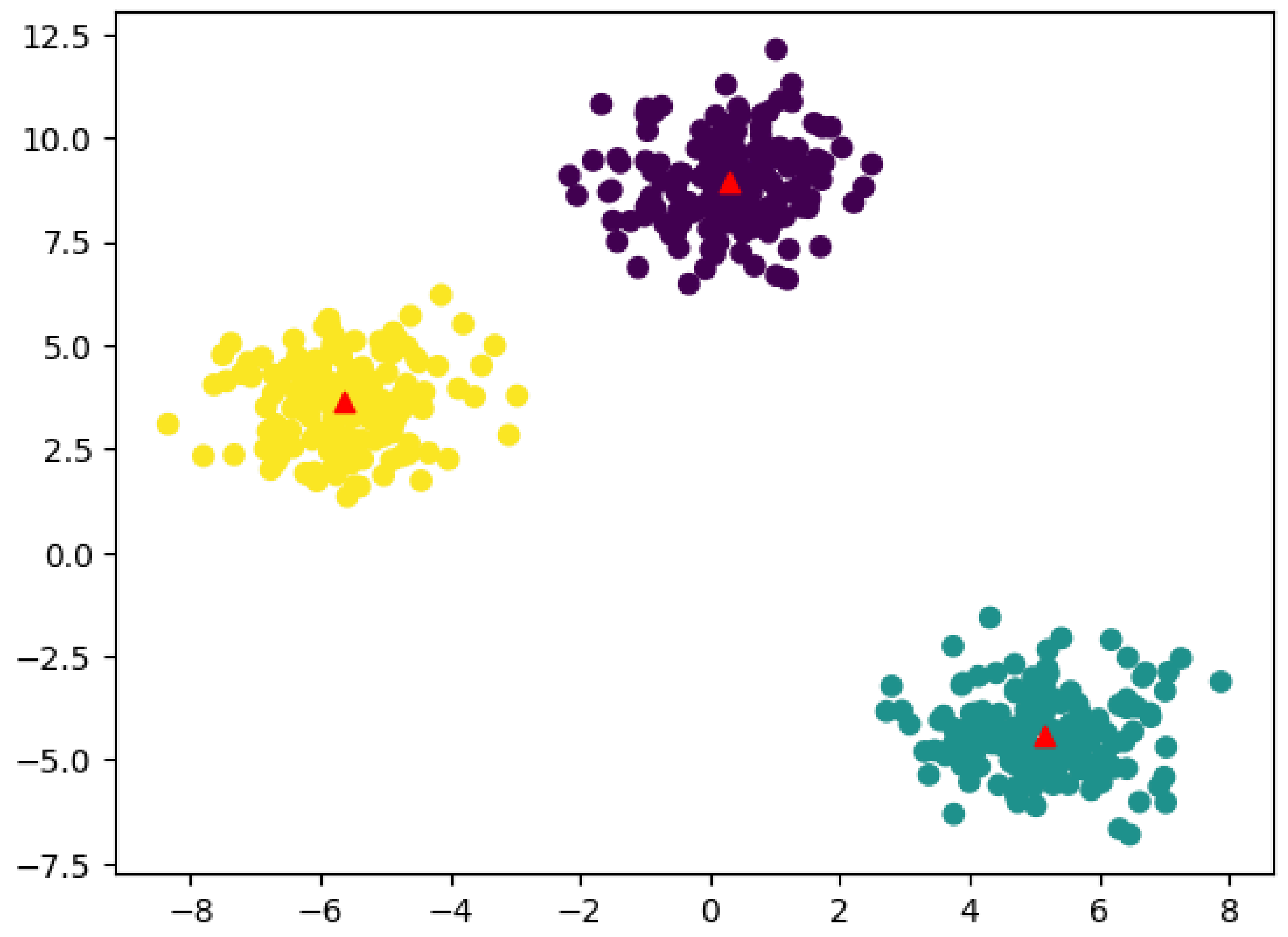
# K-MEANS

## PARALLEL COMPUTING WITH CUDA PROJECT

by Possawat Joodkong 6510405695

## K-MEANS CLUSTERING

- Clustering Algorithm (Unsupervised Machine Learning)
- Partitions a dataset into a pre-determined number $(k)$ of clusters based on feature similarity
- Used in Market segmentation, Image compression/Segmentation, Anomaly detection, feature learning, etc.



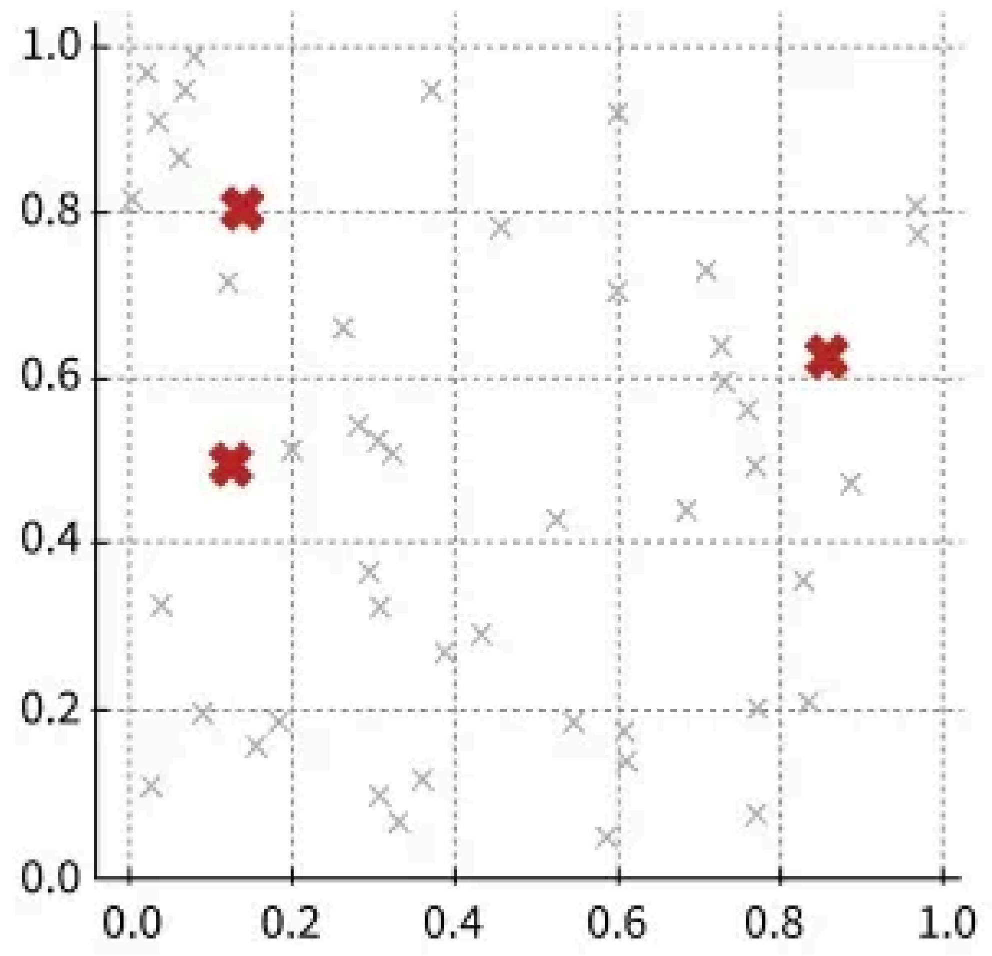
source : https://www.geeksforgeeks.org/machine-learning/k-means-clustering-introduction/

## THE ALGORITHM

### Choose Initial Centroids

Centroids are randomly chosen from the data points. These represent the initial cluster centers.

✖ Centroids × Data Points



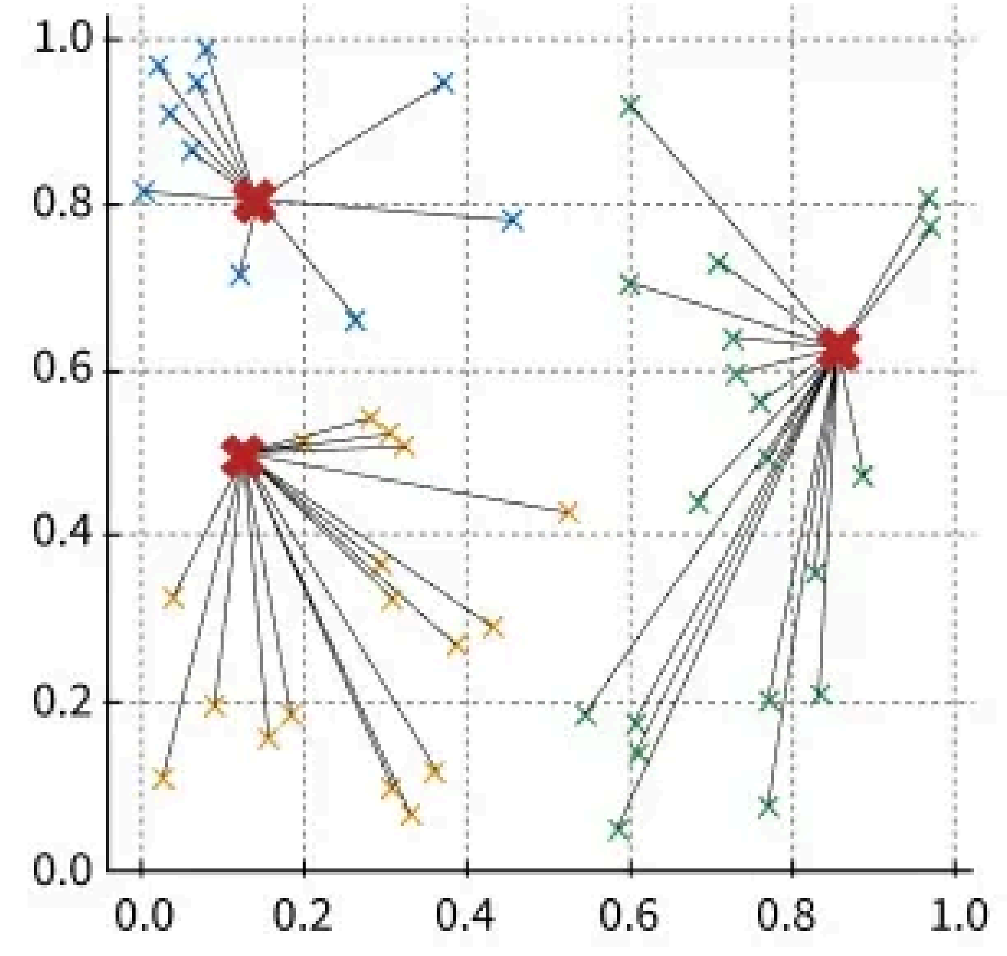
source : https://www.geeksforgeeks.org/machine-learning/k-means-clustering-introduction/

# THE ALGORITHM

## Assign Points to Nearest Centroid

Each point is assigned to the nearest centroid, forming clusters

| | |
|---|---|
| ✖ Centroids | × Cluster 1 |
| × Cluster 2 | × Cluster 3 |



source : https://www.geeksforgeeks.org/machine-learning/k-means-clustering-introduction/

## THE ALGORITHM

### Update Centroids

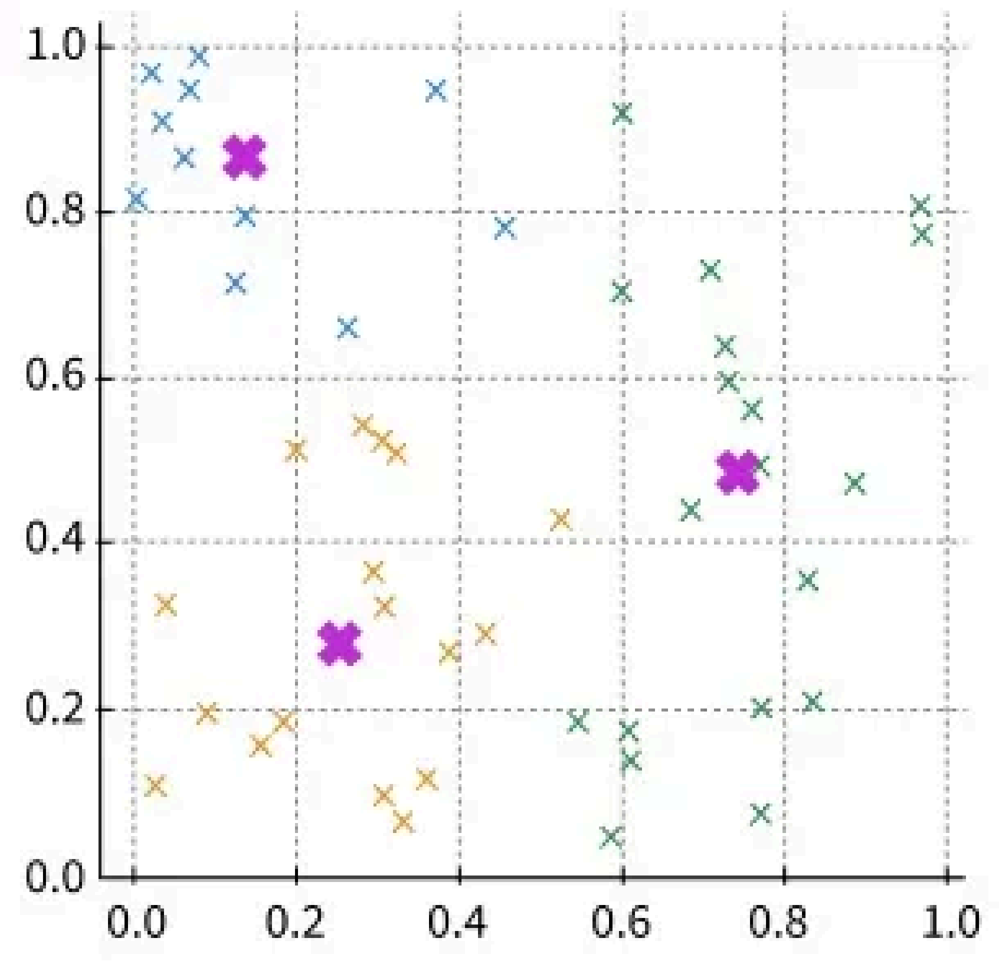
Centroids are recalculated as the mean of the points in each cluster

✖ New Centroids

× Cluster 1

× Cluster 2

× Cluster 3



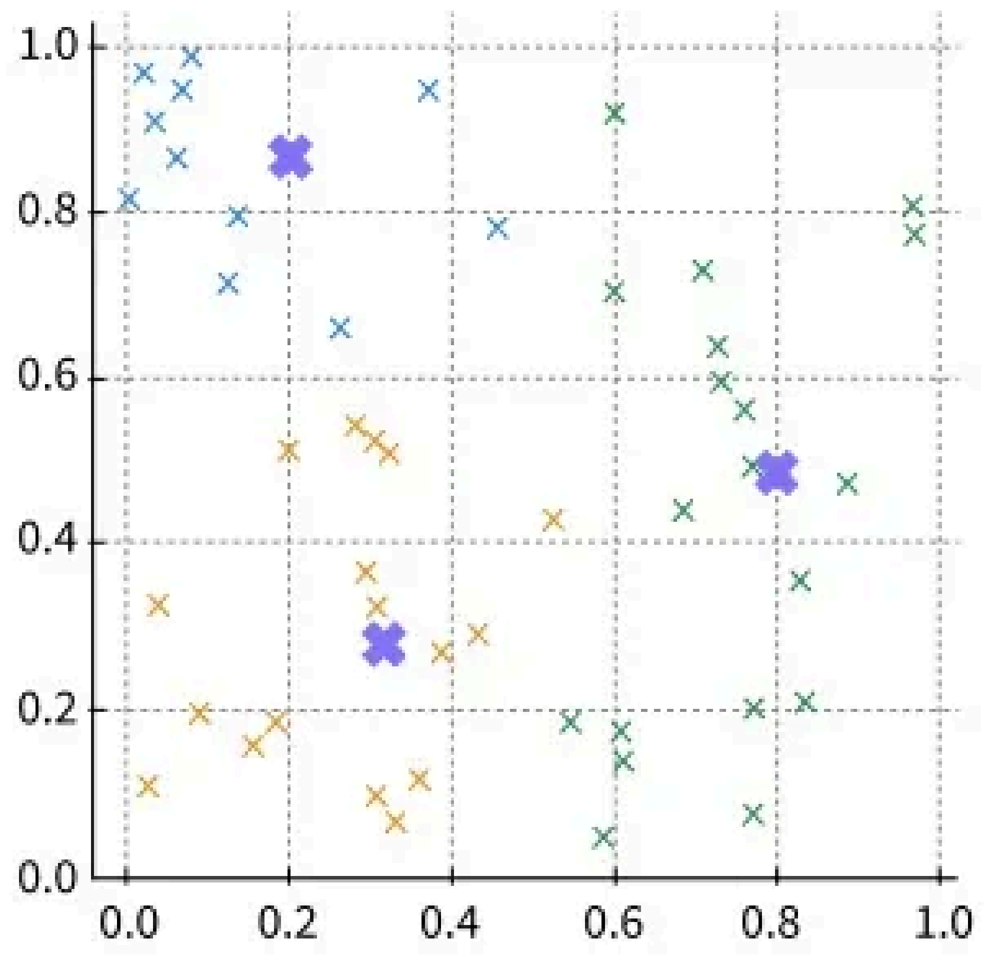
source : https://www.geeksforgeeks.org/machine-learning/k-means-clustering-introduction/

# THE ALGORITHM

## Repeat Until Convergence

This process repeats until the centroids stabilize and do not move further.

- ✖ Final Centroids
- × Cluster 1
- × Cluster 2
- × Cluster 3



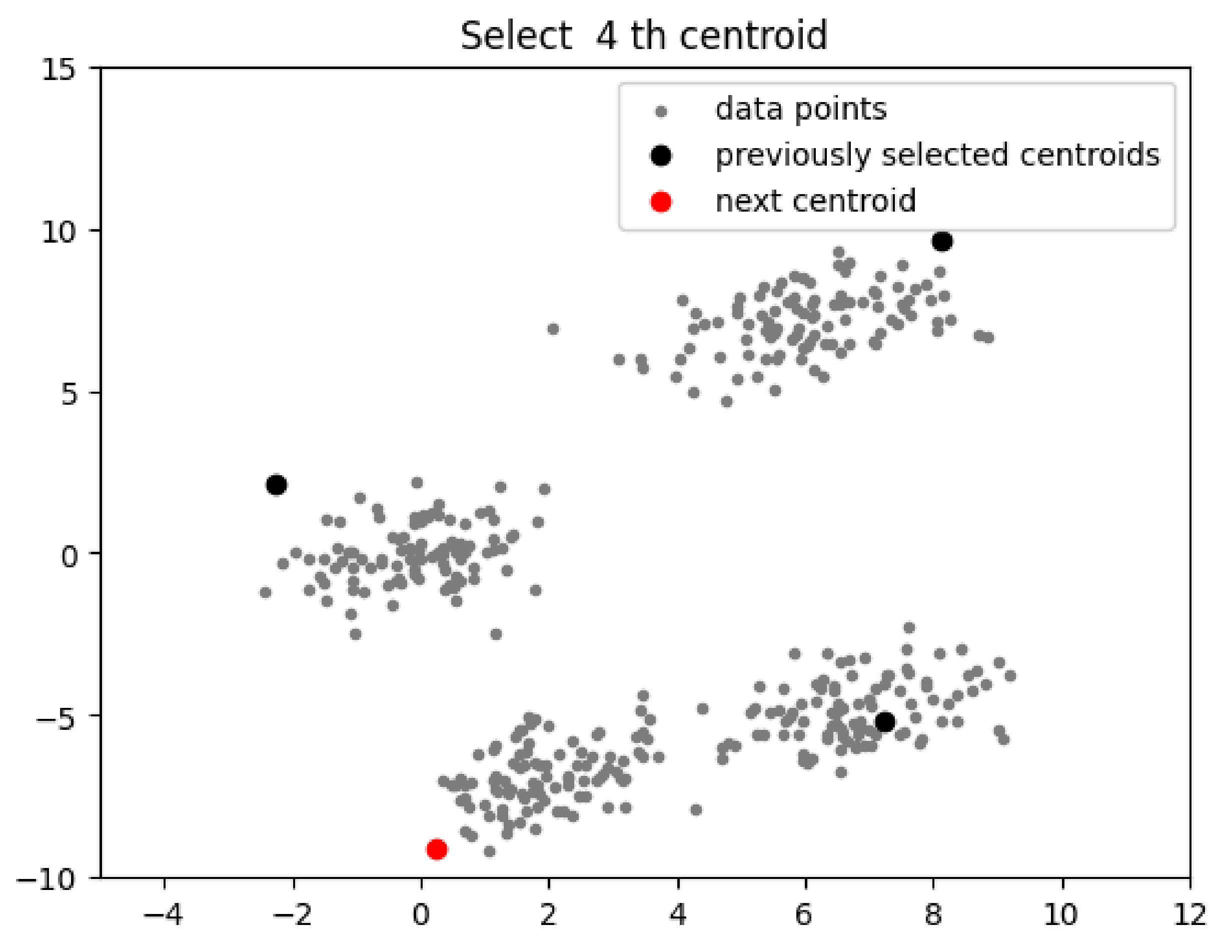
source : https://www.geeksforgeeks.org/machine-learning/k-means-clustering-introduction/

## K-MEANS CLUSTERING

- Complexity → $O(n \cdot k \cdot d \cdot i)$ where:
  - n is the number of d-dimensional vectors (to be clustered)
  - k the number of clusters
  - i the number of iterations needed until convergence.
- i can be superpolynomial in worst case (rarely happen in practice)
- In implementation like scikit-learn bound i by max_iter

## K-MEANS++

- Centroid initialization technique
- Make the algorithm converge faster
- Implementation
    - First center: Choose the first cluster center uniformly at random from the data points
    - Subsequent centers: For each remaining center:
        - Calculate the distance from each data point to its nearest existing center.
        - Choose the next center with probability proportional to the square of this distance
        - Points farther from existing centers have a higher chance of being selected.
    - Standard K-means: Once all k centers are initialized, proceed with the standard K-means algorithm
- O(k · n · d)



source : https://www.geeksforgeeks.org/machine-learning/ml-k-means-algorithm/

## DATASET

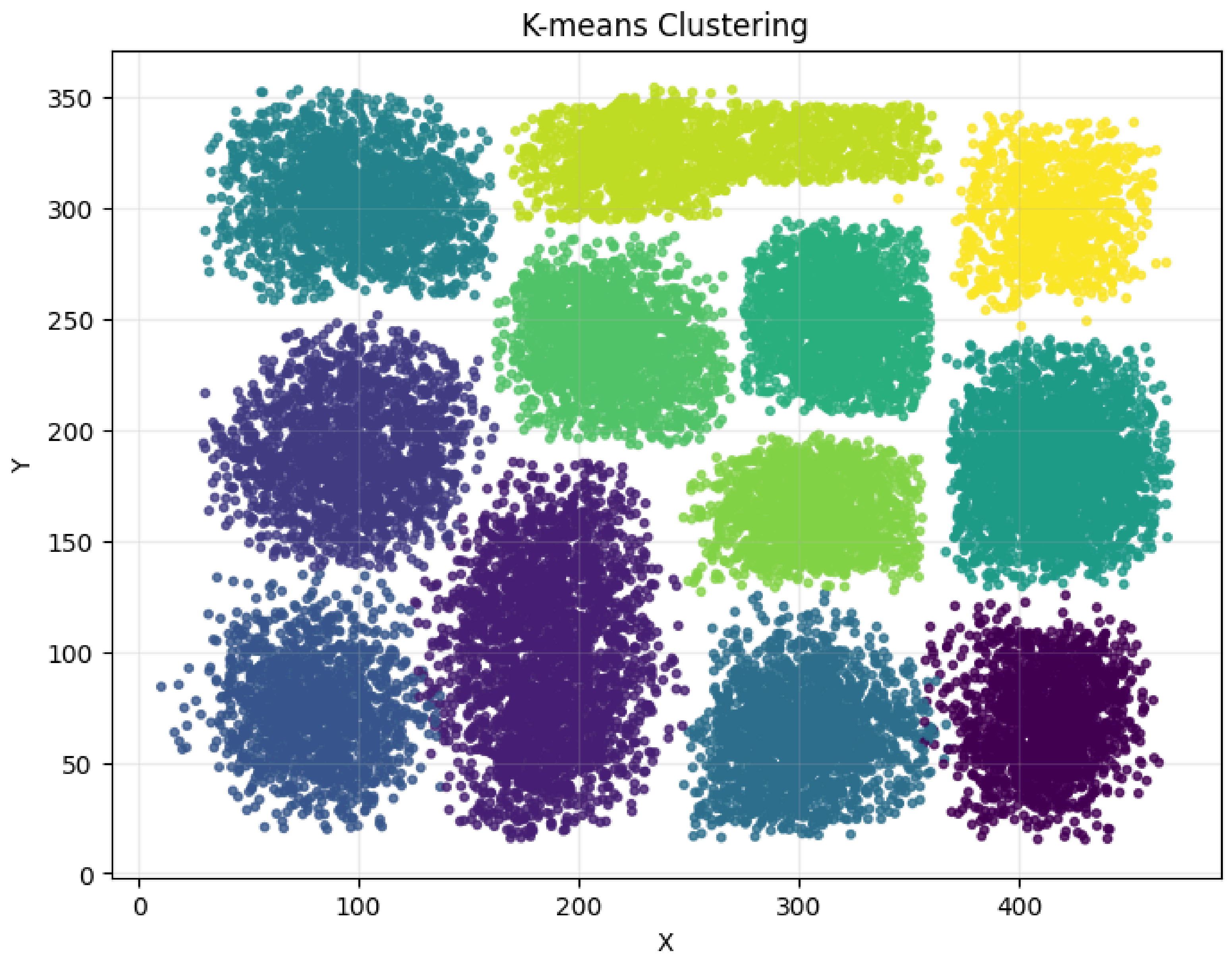
boxes3.csv from Clustering Exercises Dataset

## DATASOURCE

https://www.kaggle.com/datasets/joonasyoon/clustering-exercises

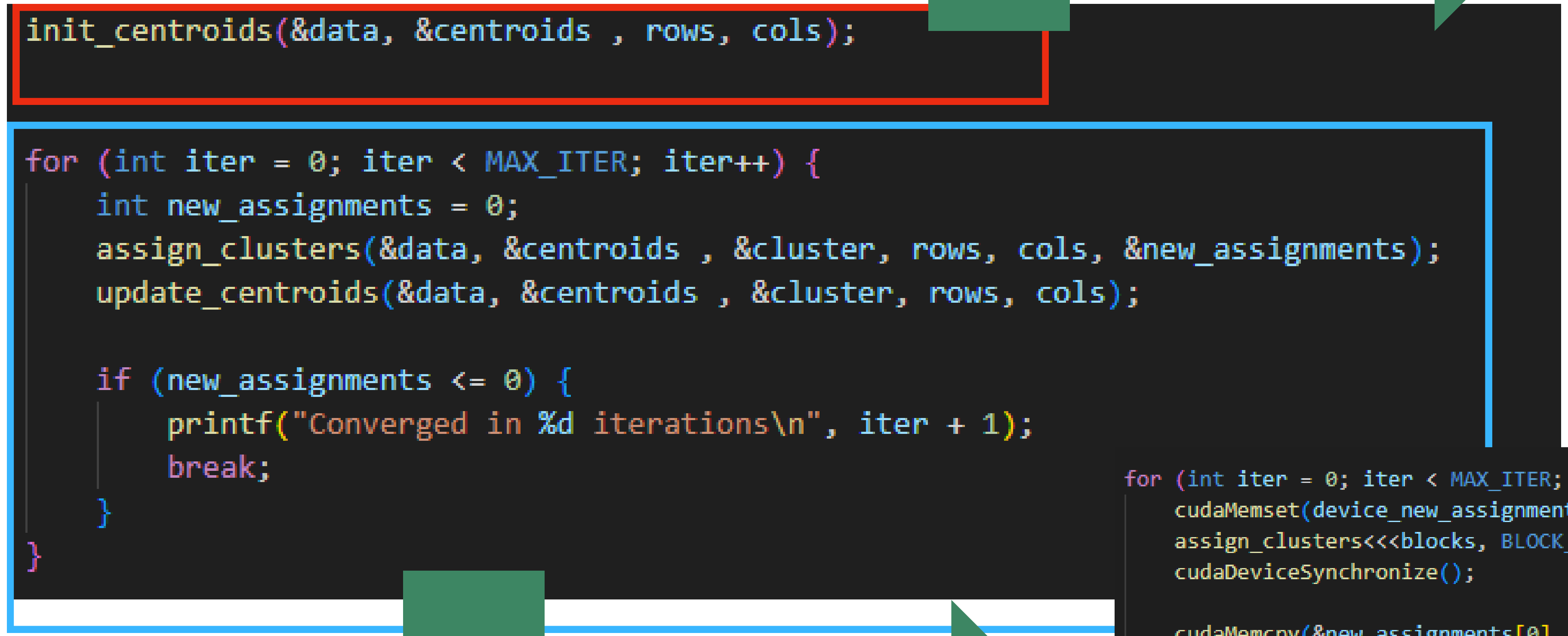
## SHAPE

21600 rows × 2 columns



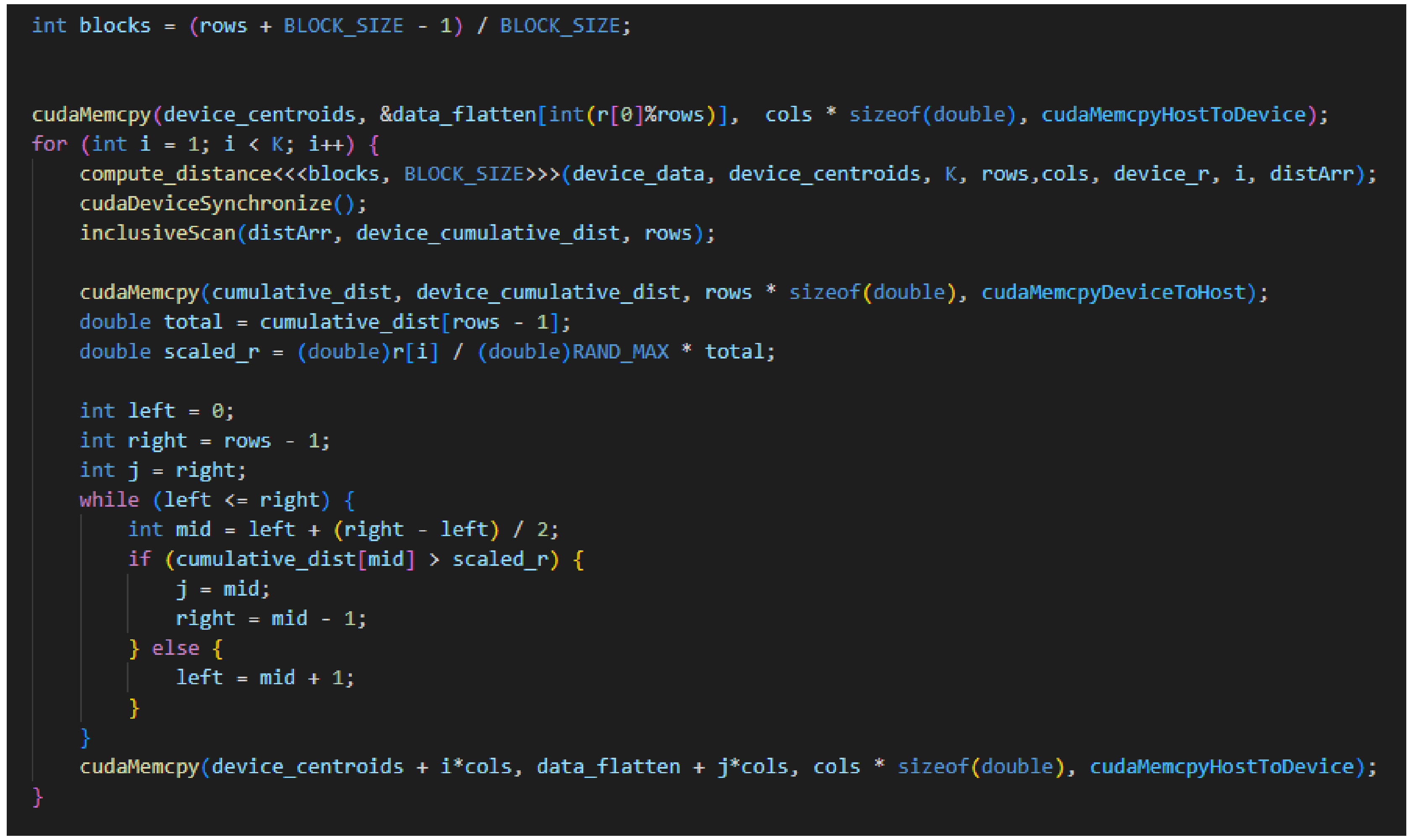
# IMPLEMENTATION

## SEQUENTIAL

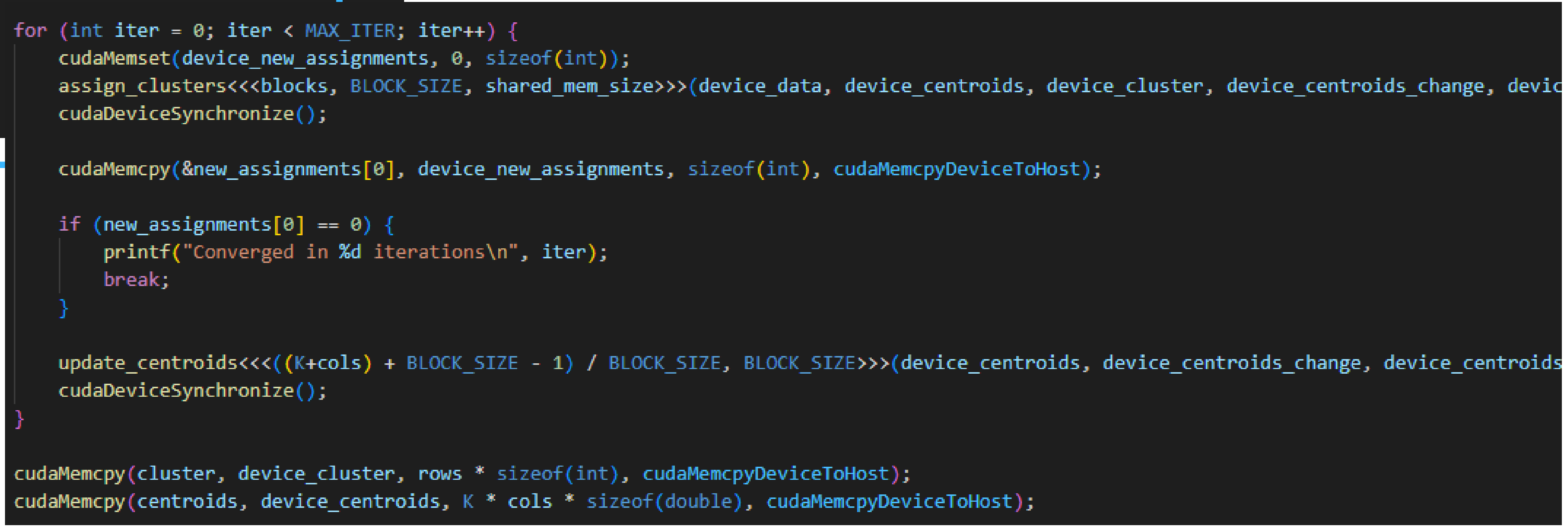


```c
init_centroids(&data, &centroids , rows, cols);

for (int iter = 0; iter < MAX_ITER; iter++) {
    int new_assignments = 0;
    assign_clusters(&data, &centroids , &cluster, rows, cols, &new_assignments);
    update_centroids(&data, &centroids , &cluster, rows, cols);

    if (new_assignments <= 0) {
        printf("Converged in %d iterations\n", iter + 1);
        break;
    }
}
```

## PARALLEL



```c
int blocks = (rows + BLOCK_SIZE - 1) / BLOCK_SIZE;


cudaMemcpy(device_centroids, &data_flatten[int(r[0]%rows)],  cols * sizeof(double), cudaMemcpyHostToDevice);
for (int i = 1; i < K; i++) {
    compute_distance<<<blocks, BLOCK_SIZE>>>(device_data, device_centroids, K, rows,cols, device_r, i, distArr);
    cudaDeviceSynchronize();
    inclusiveScan(distArr, device_cumulative_dist, rows);

    cudaMemcpy(cumulative_dist, device_cumulative_dist, rows * sizeof(double), cudaMemcpyDeviceToHost);
    double total = cumulative_dist[rows - 1];
    double scaled_r = (double)r[i] / (double)RAND_MAX * total;

    int left = 0;
    int right = rows - 1;
    int j = right;
    while (left <= right) {
        int mid = left + (right - left) / 2;
        if (cumulative_dist[mid] > scaled_r) {
            j = mid;
            right = mid - 1;
        } else {
            left = mid + 1;
        }
    }
    cudaMemcpy(device_centroids + i*cols, data_flatten + j*cols, cols * sizeof(double), cudaMemcpyHostToDevice);
}
```

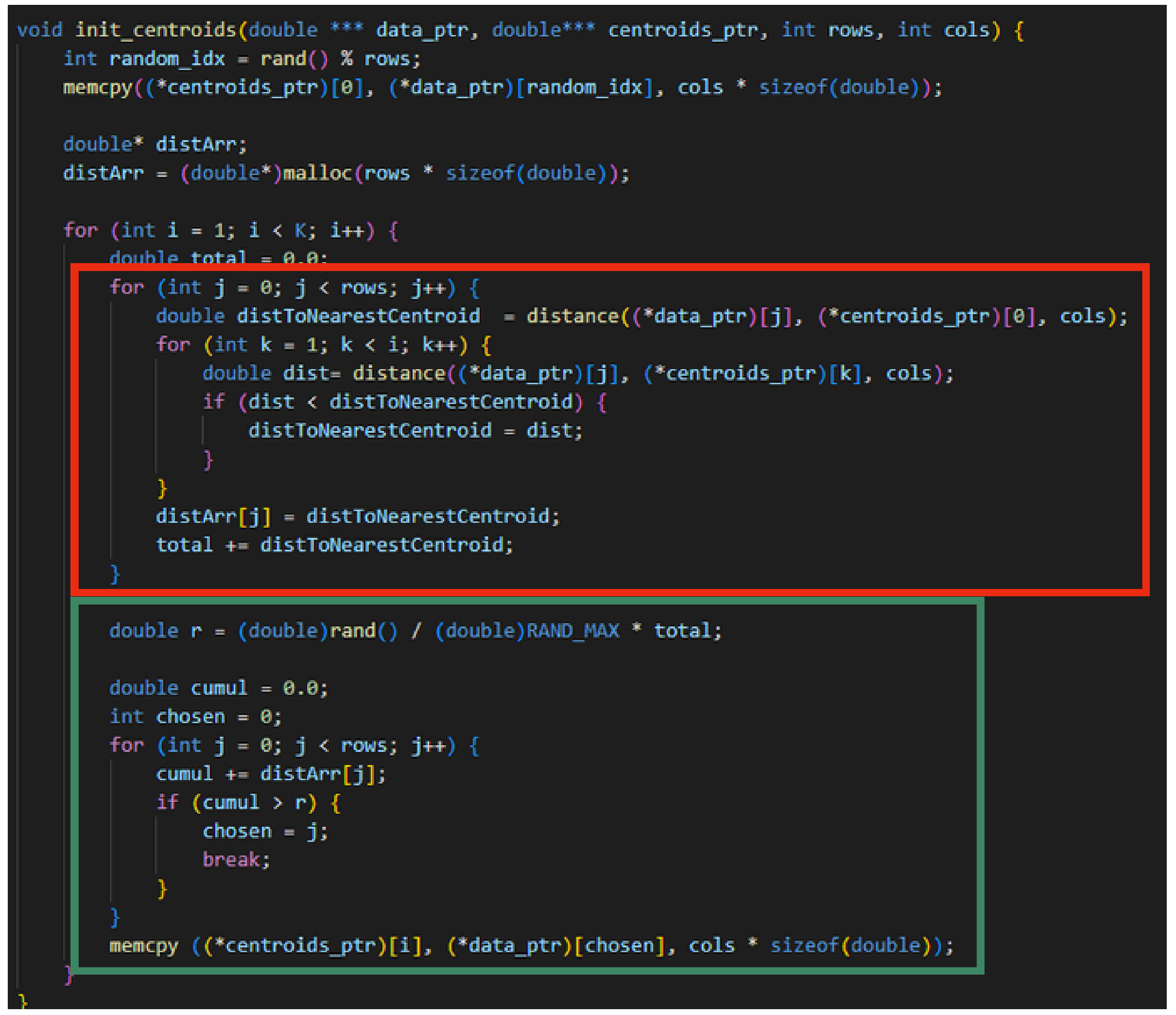


```c
for (int iter = 0; iter < MAX_ITER; iter++) {
    cudaMemset(device_new_assignments, 0, sizeof(int));
    assign_clusters<<<blocks, BLOCK_SIZE, shared_mem_size>>>(device_data, device_centroids, device_cluster, device_centroids_change, devic
    cudaDeviceSynchronize();

    cudaMemcpy(&new_assignments[0], device_new_assignments, sizeof(int), cudaMemcpyDeviceToHost);

    if (new_assignments[0] == 0) {
        printf("Converged in %d iterations\n", iter);
        break;
    }

    update_centroids<<<((K+cols) + BLOCK_SIZE - 1) / BLOCK_SIZE, BLOCK_SIZE>>>(device_centroids, device_centroids_change, device_centroids
    cudaDeviceSynchronize();
}

cudaMemcpy(cluster, device_cluster, rows * sizeof(int), cudaMemcpyDeviceToHost);
cudaMemcpy(centroids, device_centroids, K * cols * sizeof(double), cudaMemcpyDeviceToHost);
```

## PARALLEL

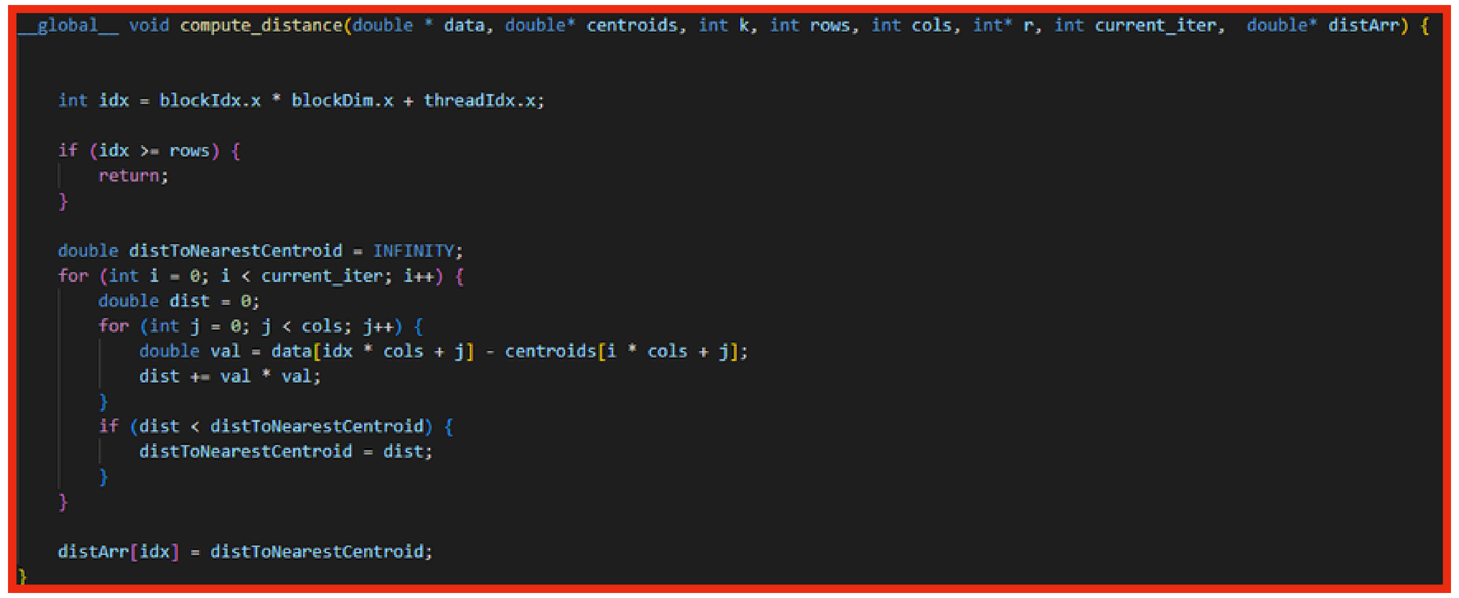
# IMPLEMENTATION

## SEQUENTIAL



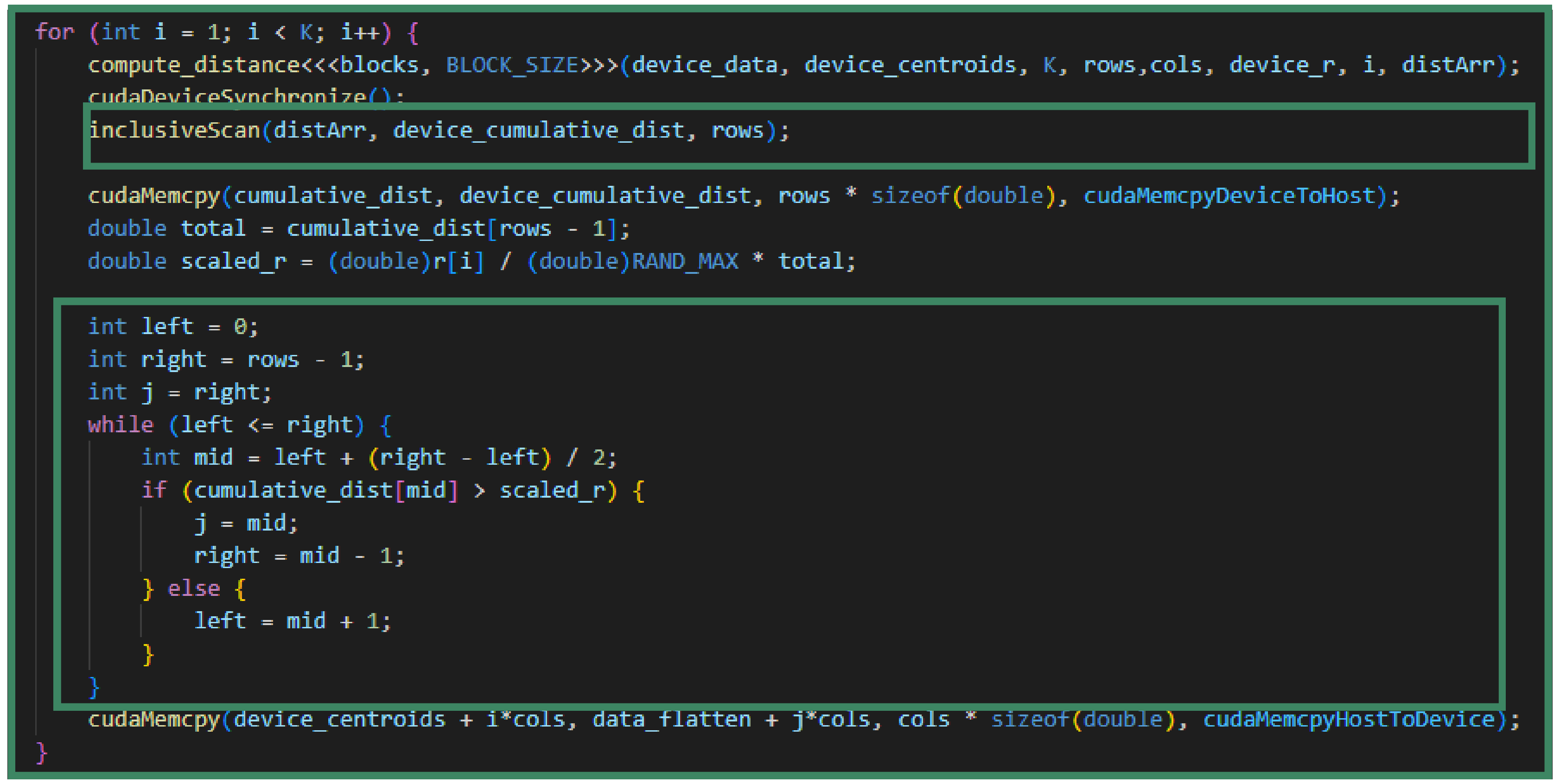
```c
void init_centroids(double *** data_ptr, double*** centroids_ptr, int rows, int cols) {
    int random_idx = rand() % rows;
    memcpy((*centroids_ptr)[0], (*data_ptr)[random_idx], cols * sizeof(double));

    double* distArr;
    distArr = (double*)malloc(rows * sizeof(double));

    for (int i = 1; i < K; i++) {
        double total = 0.0;
        for (int j = 0; j < rows; j++) {
            double distToNearestCentroid  = distance((*data_ptr)[j], (*centroids_ptr)[0], cols);
            for (int k = 1; k < i; k++) {
                double dist= distance((*data_ptr)[j], (*centroids_ptr)[k], cols);
                if (dist < distToNearestCentroid) {
                    distToNearestCentroid = dist;
                }
            }
            distArr[j] = distToNearestCentroid;
            total += distToNearestCentroid;
        }
        double r = (double)rand() / (double)RAND_MAX * total;

        double cumul = 0.0;
        int chosen = 0;
        for (int j = 0; j < rows; j++) {
            cumul += distArr[j];
            if (cumul > r) {
                chosen = j;
                break;
            }
        }
        memcpy ((*centroids_ptr)[i], (*data_ptr)[chosen], cols * sizeof(double));
    }
}
```

O(k^2 · n · d)



```c
__global__ void compute_distance(double * data, double* centroids, int k, int rows, int cols, int* r, int current_iter,  double* distArr) {


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

    if (idx >= rows) {
        return;
    }

    double distToNearestCentroid = INFINITY;
    for (int i = 0; i < current_iter; i++) {
        double dist = 0;
        for (int j = 0; j < cols; j++) {
            double val = data[idx * cols + j] - centroids[i * cols + j];
            dist += val * val;
        }
        if (dist < distToNearestCentroid) {
            distToNearestCentroid = dist;
        }
    }

    distArr[idx] = distToNearestCentroid;
}
```

span O(k · d)



```c
for (int i = 1; i < K; i++) {
    compute_distance<<<blocks, BLOCK_SIZE>>>(device_data, device_centroids, K, rows,cols, device_r, i, distArr);
    cudaDeviceSynchronize();
    inclusiveScan(distArr, device_cumulative_dist, rows);

    cudaMemcpy(cumulative_dist, device_cumulative_dist, rows * sizeof(double), cudaMemcpyDeviceToHost);
    double total = cumulative_dist[rows - 1];
    double scaled_r = (double)r[i] / (double)RAND_MAX * total;

    int left = 0;
    int right = rows - 1;
    int j = right;
    while (left <= right) {
        int mid = left + (right - left) / 2;
        if (cumulative_dist[mid] > scaled_r) {
            j = mid;
            right = mid - 1;
        } else {
            left = mid + 1;
        }
    }
    cudaMemcpy(device_centroids + i*cols, data_flatten + j*cols, cols * sizeof(double), cudaMemcpyHostToDevice);
}
```

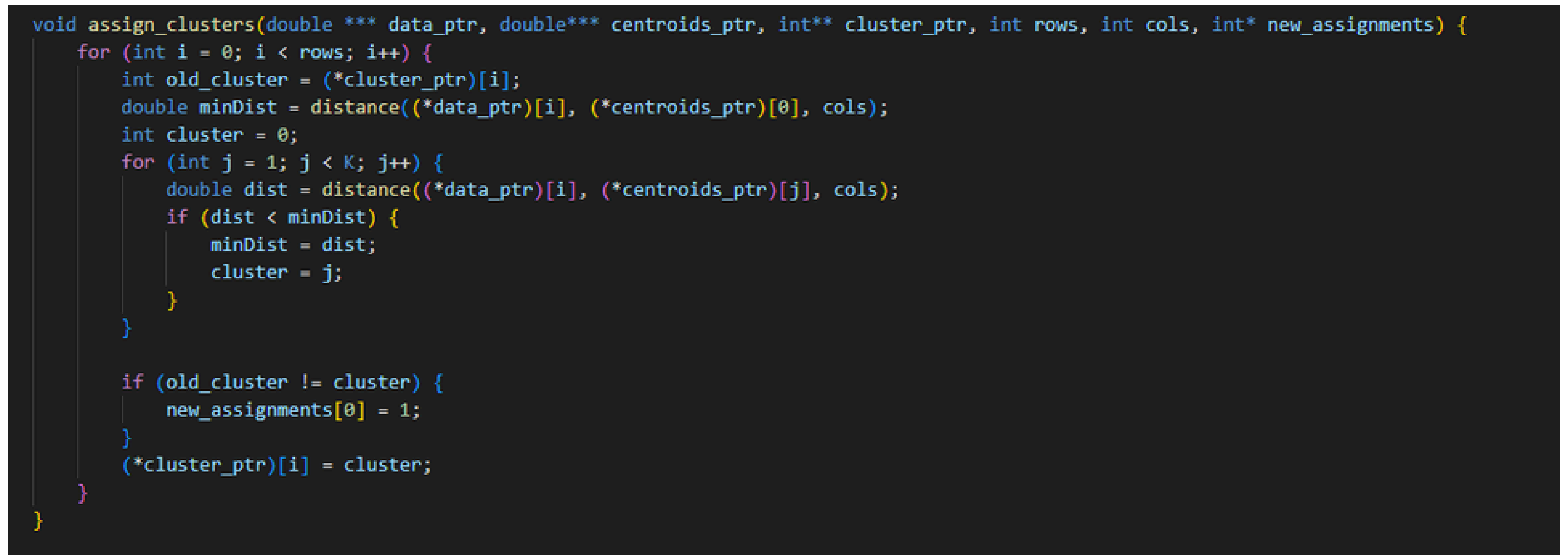
span O(log(n))

span O(log(n))

Total span O(k^2 · d +k · log(n))

# IMPLEMENTATION

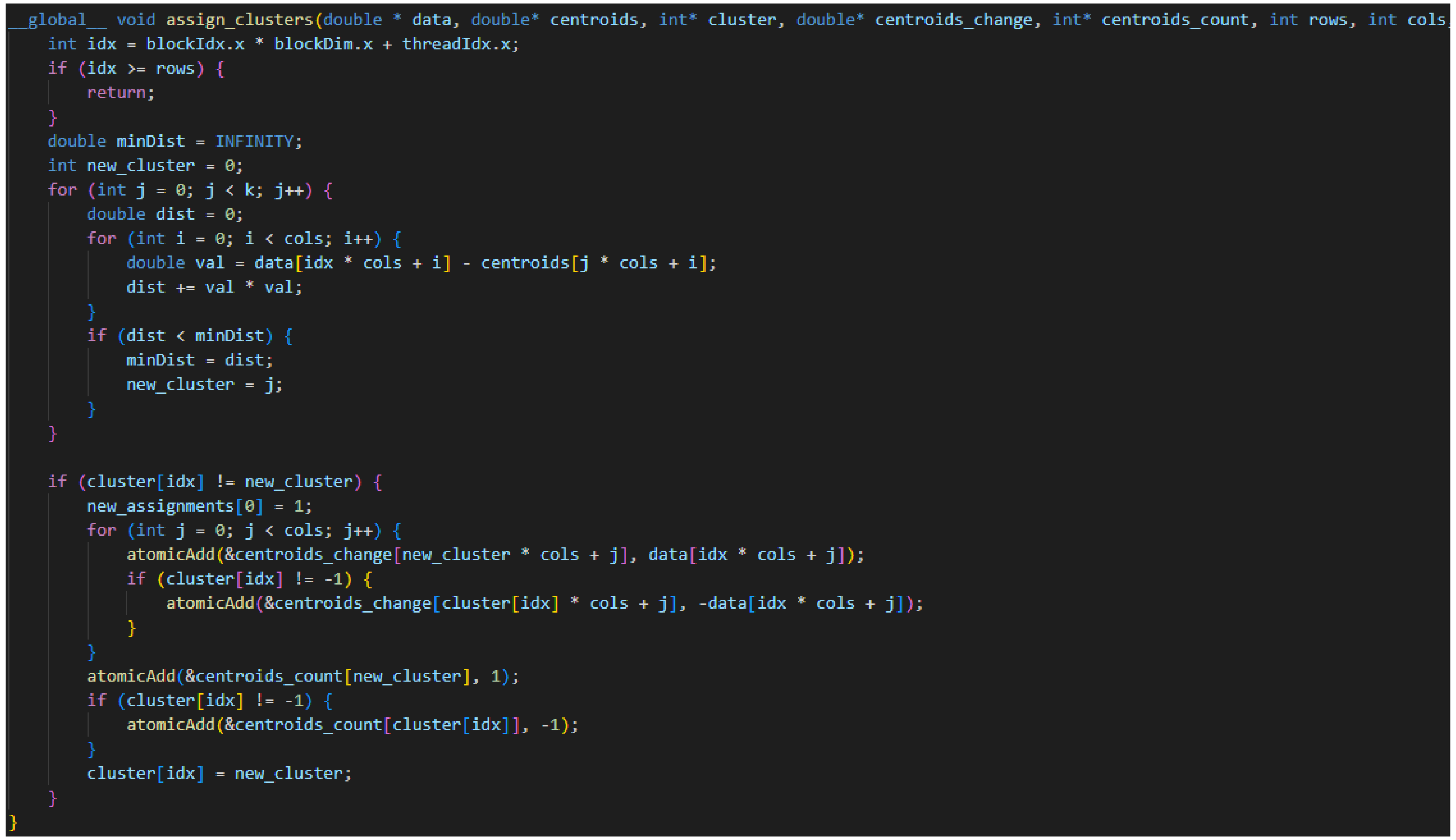
## SEQUENTIAL



```
void assign_clusters(double *** data_ptr, double*** centroids_ptr, int** cluster_ptr, int rows, int cols, int* new_assignments) {
    for (int i = 0; i < rows; i++) {
        int old_cluster = (*cluster_ptr)[i];
        double minDist = distance((*data_ptr)[i], (*centroids_ptr)[0], cols);
        int cluster = 0;
        for (int j = 1; j < K; j++) {
            double dist = distance((*data_ptr)[i], (*centroids_ptr)[j], cols);
            if (dist < minDist) {
                minDist = dist;
                cluster = j;
            }
        }

        if (old_cluster != cluster) {
            new_assignments[0] = 1;
        }
        (*cluster_ptr)[i] = cluster;
    }
}
```

$O(k \cdot n \cdot d)$

## PARALLEL

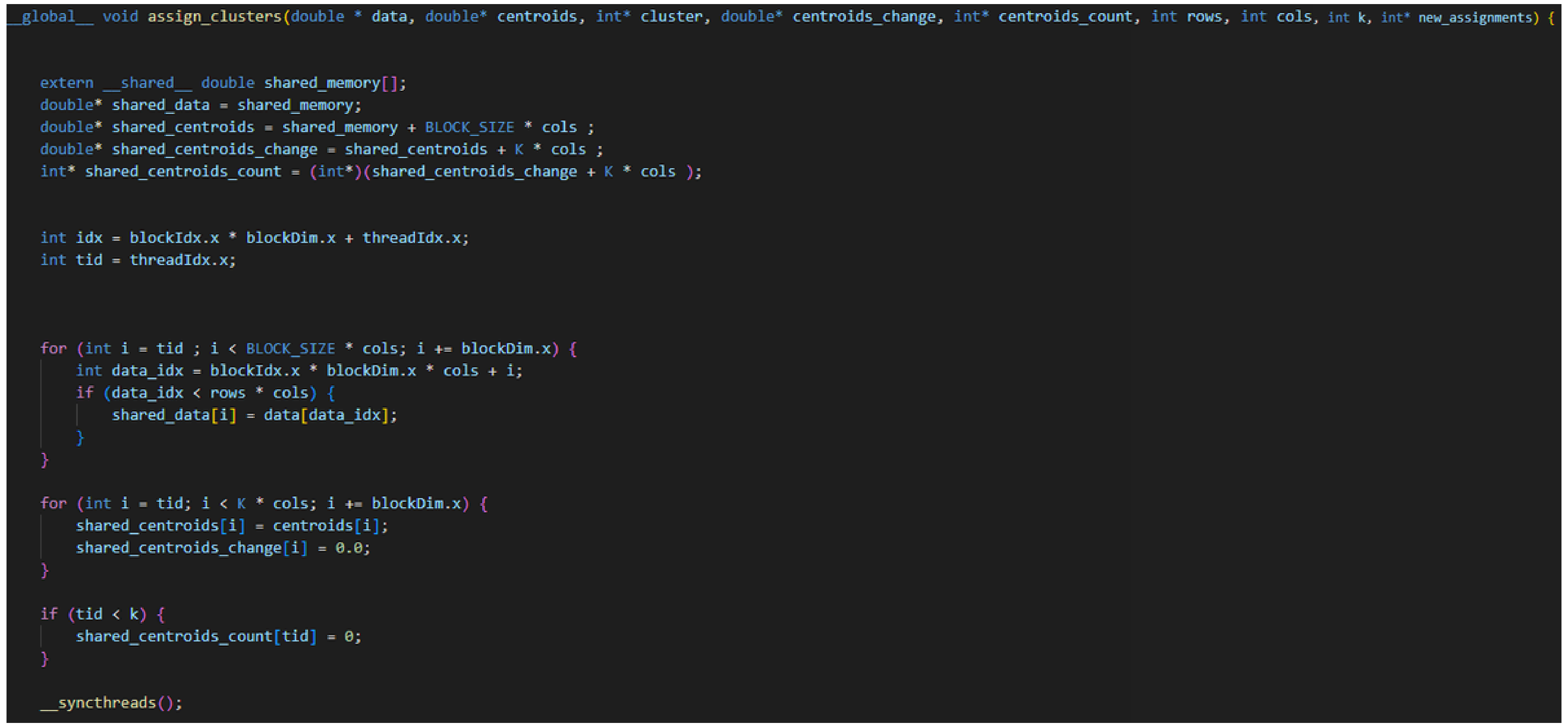


```
__global__ void assign_clusters(double * data, double* centroids, int* cluster, double* centroids_change, int* centroids_count, int rows, int cols
    int idx = blockIdx.x * blockDim.x + threadIdx.x;
    if (idx >= rows) {
        return;
    }
    double minDist = INFINITY;
    int new_cluster = 0;
    for (int j = 0; j < k; j++) {
        double dist = 0;
        for (int i = 0; i < cols; i++) {
            double val = data[idx * cols + i] - centroids[j * cols + i];
            dist += val * val;
        }
        if (dist < minDist) {
            minDist = dist;
            new_cluster = j;
        }
    }

    if (cluster[idx] != new_cluster) {
        new_assignments[0] = 1;
        for (int j = 0; j < cols; j++) {
            atomicAdd(&centroids_change[new_cluster * cols + j], data[idx * cols + j]);
            if (cluster[idx] != -1) {
                atomicAdd(&centroids_change[cluster[idx] * cols + j], -data[idx * cols + j]);
            }
        }
        atomicAdd(&centroids_count[new_cluster], 1);
        if (cluster[idx] != -1) {
            atomicAdd(&centroids_count[cluster[idx]], -1);
        }
        cluster[idx] = new_cluster;
    }
}
```

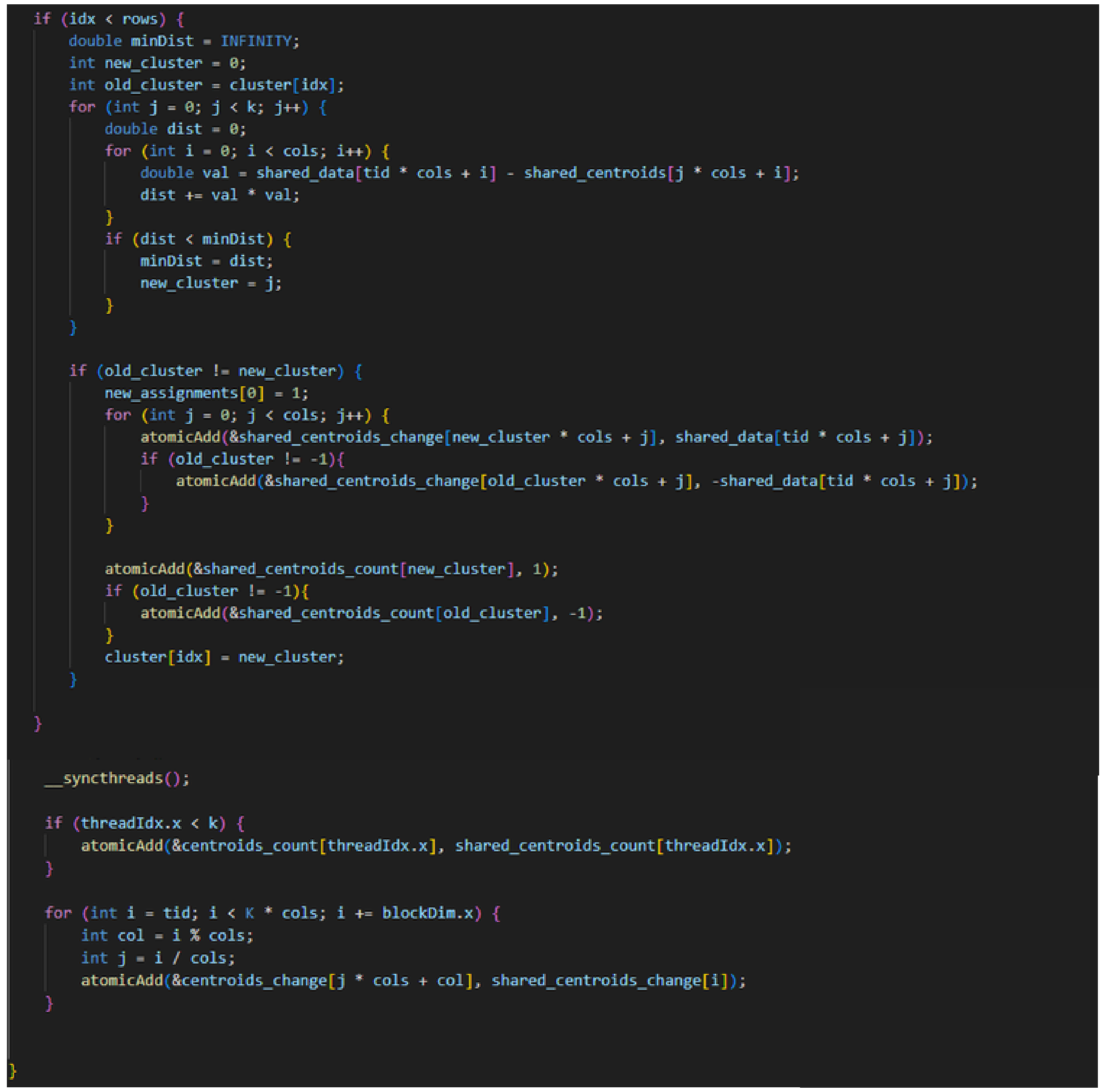
span $O(k \cdot d)$

# IMPLEMENTATION

## PARALLEL-SHARED MEMORY



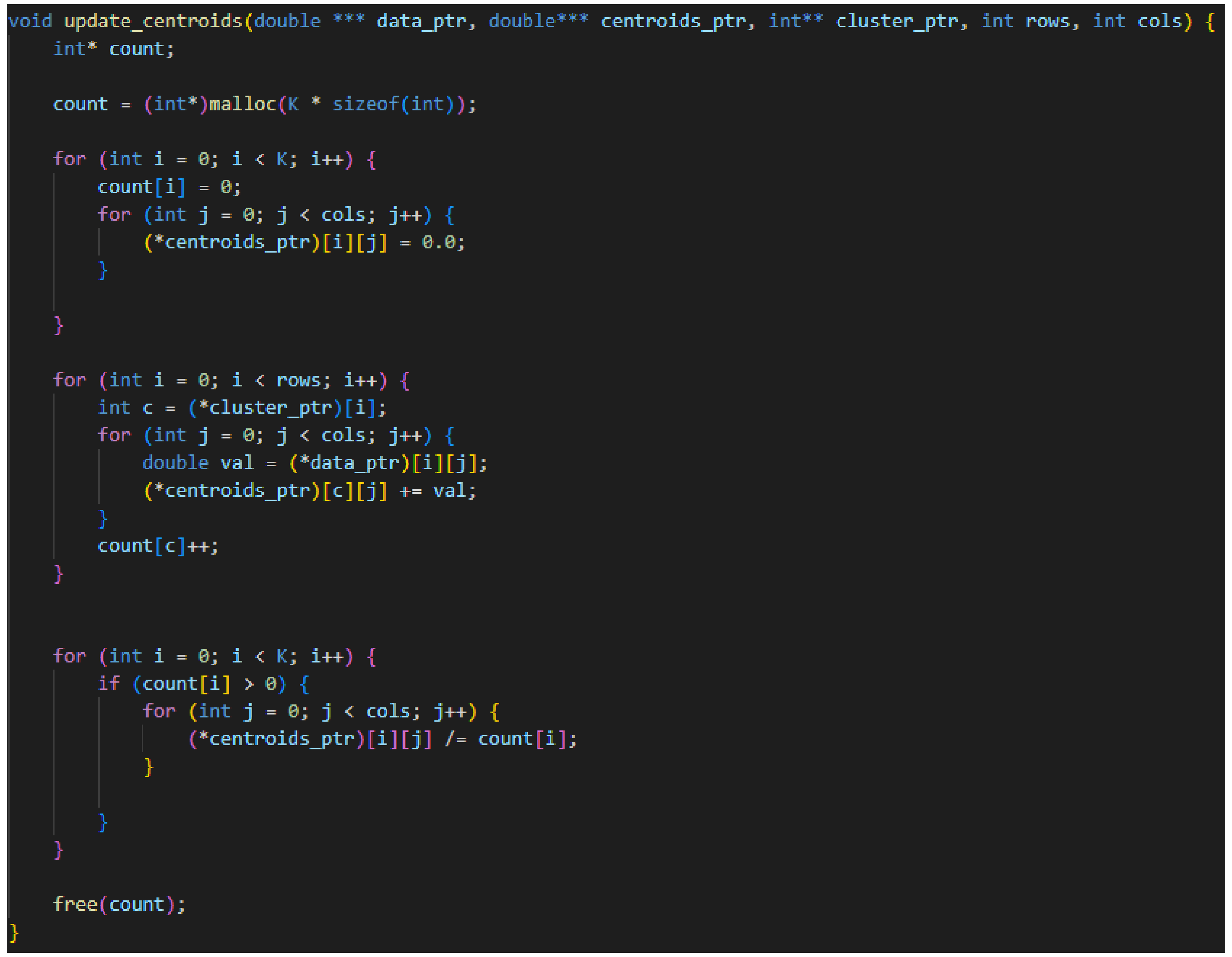
```
__global__ void assign_clusters(double * data, double* centroids, int* cluster, double* centroids_change, int* centroids_count, int rows, int cols, int k, int* new_assignments) {


    extern __shared__ double shared_memory[];
    double* shared_data = shared_memory;
    double* shared_centroids = shared_memory + BLOCK_SIZE * cols ;
    double* shared_centroids_change = shared_centroids + K * cols ;
    int* shared_centroids_count = (int*)(shared_centroids_change + K * cols );


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



    for (int i = tid ; i < BLOCK_SIZE * cols; i += blockDim.x) {
        int data_idx = blockIdx.x * blockDim.x * cols + i;
        if (data_idx < rows * cols) {
            shared_data[i] = data[data_idx];
        }
    }

    for (int i = tid; i < K * cols; i += blockDim.x) {
        shared_centroids[i] = centroids[i];
        shared_centroids_change[i] = 0.0;
    }

    if (tid < k) {
        shared_centroids_count[tid] = 0;
    }

    __syncthreads();
```



```
    if (idx < rows) {
        double minDist = INFINITY;
        int new_cluster = 0;
        int old_cluster = cluster[idx];
        for (int j = 0; j < k; j++) {
            double dist = 0;
            for (int i = 0; i < cols; i++) {
                double val = shared_data[tid * cols + i] - shared_centroids[j * cols + i];
                dist += val * val;
            }
            if (dist < minDist) {
                minDist = dist;
                new_cluster = j;
            }
        }

        if (old_cluster != new_cluster) {
            new_assignments[0] = 1;
            for (int j = 0; j < cols; j++) {
                atomicAdd(&shared_centroids_change[new_cluster * cols + j], shared_data[tid * cols + j]);
                if (old_cluster != -1){
                    atomicAdd(&shared_centroids_change[old_cluster * cols + j], -shared_data[tid * cols + j]);
                }
            }

            atomicAdd(&shared_centroids_count[new_cluster], 1);
            if (old_cluster != -1){
                atomicAdd(&shared_centroids_count[old_cluster], -1);
            }
            cluster[idx] = new_cluster;
        }

    }

    __syncthreads();

    if (threadIdx.x < k) {
        atomicAdd(&centroids_count[threadIdx.x], shared_centroids_count[threadIdx.x]);
    }

    for (int i = tid; i < K * cols; i += blockDim.x) {
        int col = i % cols;
        int j = i / cols;
        atomicAdd(&centroids_change[j * cols + col], shared_centroids_change[i]);
    }


}
```

# IMPLEMENTATION

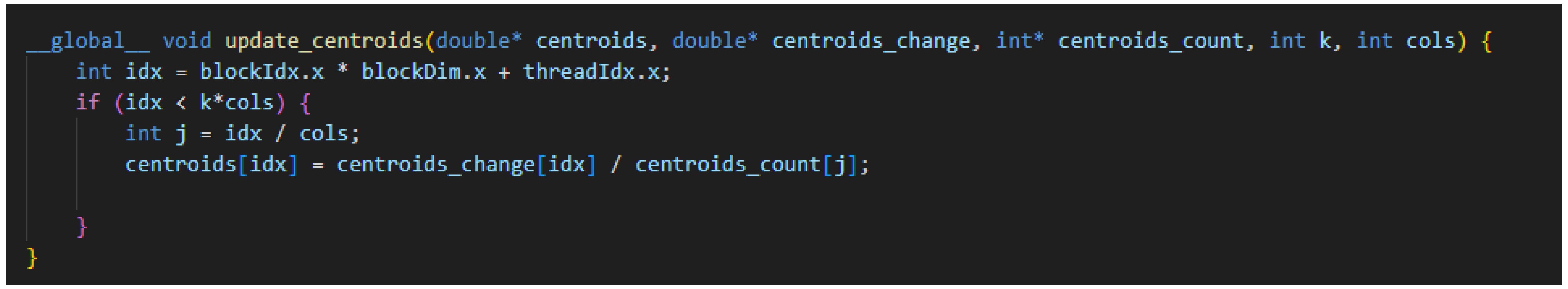
## SEQUENTIAL



```c
void update_centroids(double *** data_ptr, double*** centroids_ptr, int** cluster_ptr, int rows, int cols) {
    int* count;

    count = (int*)malloc(K * sizeof(int));

    for (int i = 0; i < K; i++) {
        count[i] = 0;
        for (int j = 0; j < cols; j++) {
            (*centroids_ptr)[i][j] = 0.0;
        }

    }

    for (int i = 0; i < rows; i++) {
        int c = (*cluster_ptr)[i];
        for (int j = 0; j < cols; j++) {
            double val = (*data_ptr)[i][j];
            (*centroids_ptr)[c][j] += val;
        }
        count[c]++;
    }


    for (int i = 0; i < K; i++) {
        if (count[i] > 0) {
            for (int j = 0; j < cols; j++) {
                (*centroids_ptr)[i][j] /= count[i];
            }

        }
    }

    free(count);
}
```

$O(n \cdot d + k \cdot d)$

## PARALLEL



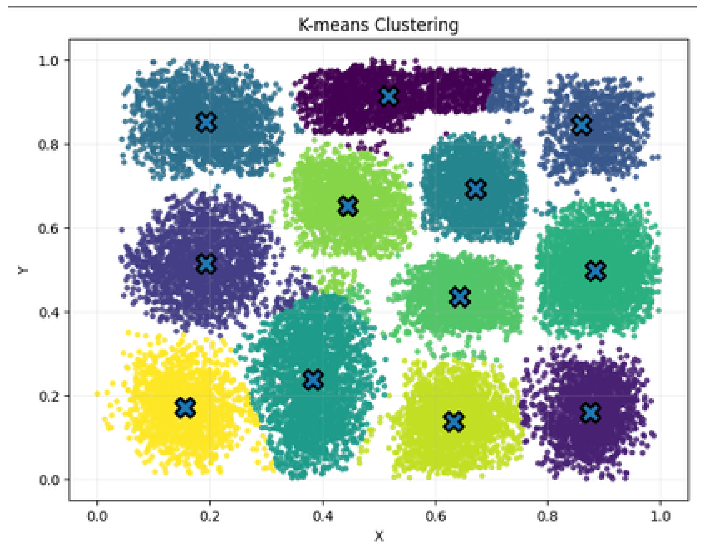
```c
__global__ void update_centroids(double* centroids, double* centroids_change, int* centroids_count, int k, int cols) {
    int idx = blockIdx.x * blockDim.x + threadIdx.x;
    if (idx < k*cols) {
        int j = idx / cols;
        centroids[idx] = centroids_change[idx] / centroids_count[j];

    }
}
```

span $O(1)$

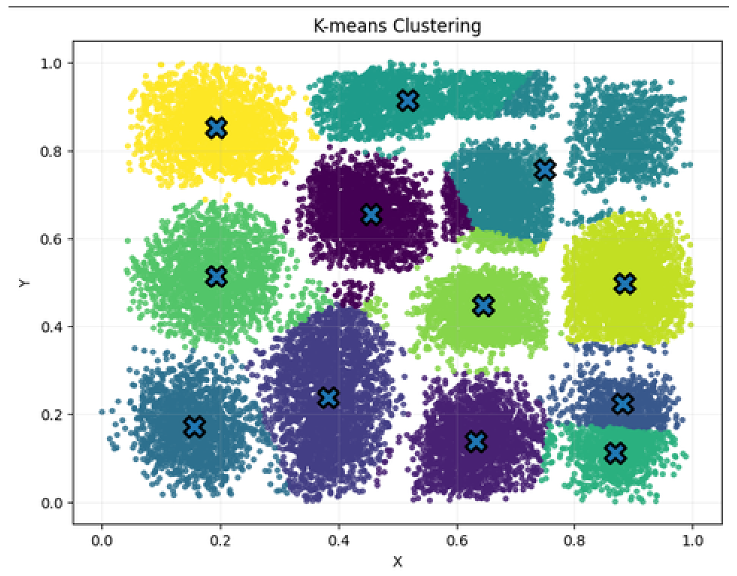
# RESULT

## K = 12

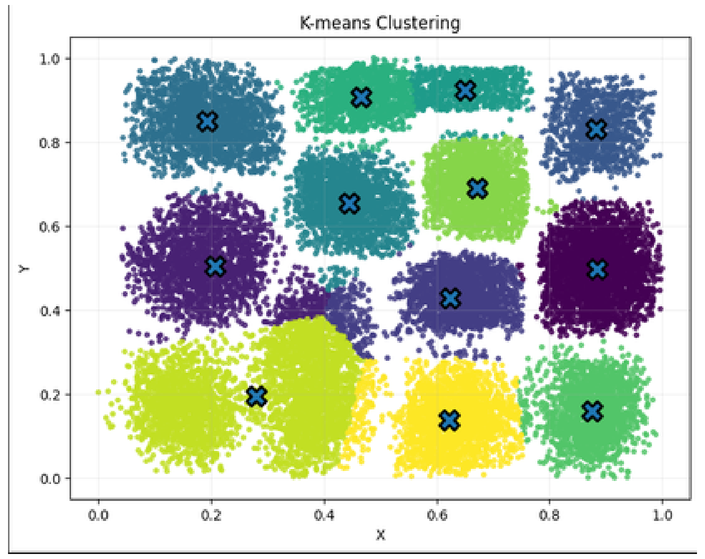
### K-MEANS++ SEQUENTIAL



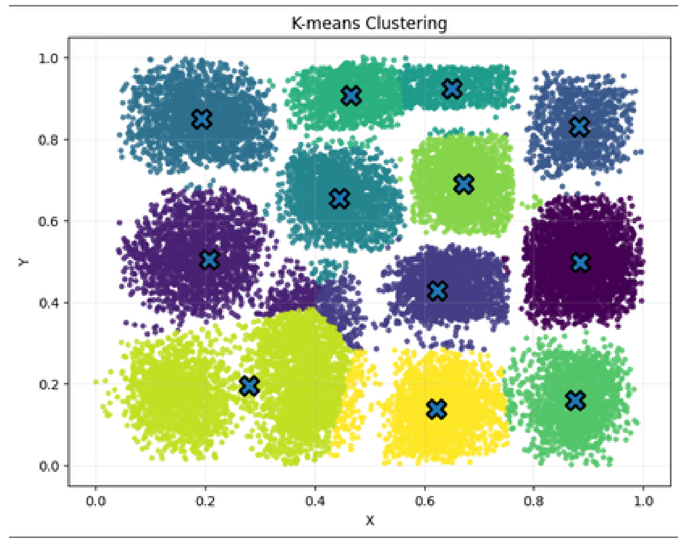
### K-MEANS PARALLEL



### K-MEANS++ PARALLEL



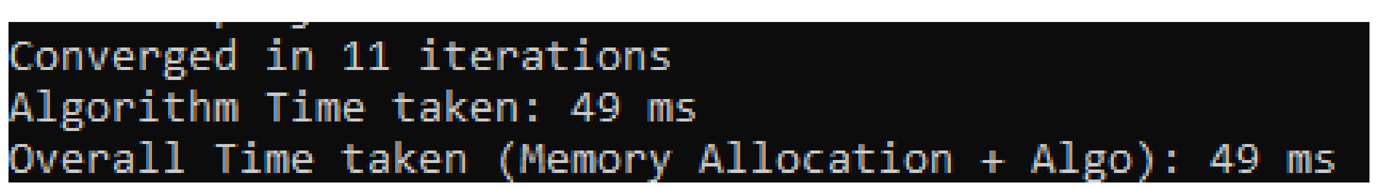
### K-MEANS++ SHARED MEM



## RESULT

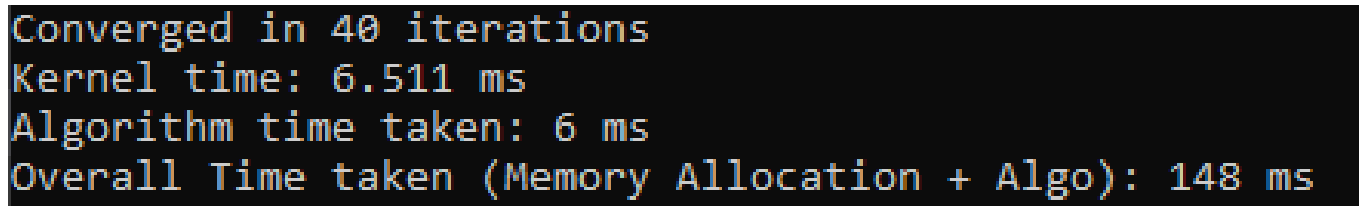
K = 12

### K-MEANS++ SEQUENTIAL



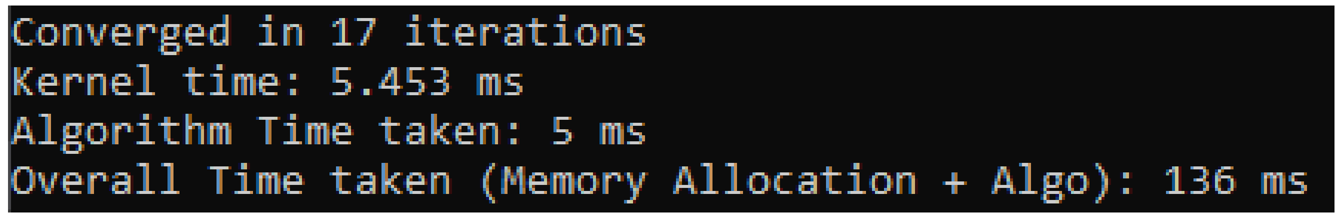
```
Converged in 11 iterations
Algorithm Time taken: 49 ms
Overall Time taken (Memory Allocation + Algo): 49 ms
```

### K-MEANS PARALLEL



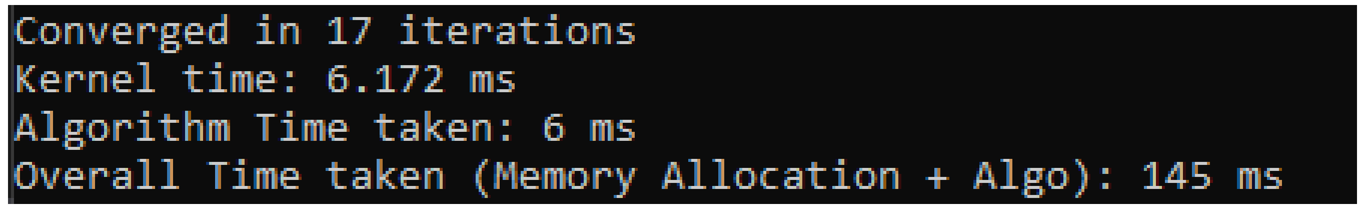
```
Converged in 40 iterations
Kernel time: 6.511 ms
Algorithm time taken: 6 ms
Overall Time taken (Memory Allocation + Algo): 148 ms
```

### K-MEANS++ PARALLEL



```
Converged in 17 iterations
Kernel time: 5.453 ms
Algorithm Time taken: 5 ms
Overall Time taken (Memory Allocation + Algo): 136 ms
```

### K-MEANS++ SHARED MEM



```
Converged in 17 iterations
Kernel time: 6.172 ms
Algorithm Time taken: 6 ms
Overall Time taken (Memory Allocation + Algo): 145 ms
```

## RESULT

### N = 21600, D = 2

| Algo/k | k=12 | k=24 | k=48 | k=96 |
|---|---|---|---|---|
| sequential | 49 | 243 | 786 | 2107 |
| k-means parallel | 148/6.511 | 144/18.059 | 149/12.657 | 167/23.393 |
| k-means++ parallel | 136/5.453 | 136/13.863 | 144/24.849 | 179/53.329 |
| k-means++ shared mem | 145/6.172 | 142/14.489 | 160/25.674 | 174/54.977 |

(OVERALLTIME/KERNEL TIME)

# THANK *You!*

HTTPS://WWW.CANVA.COM/DESIGN/DAG5SQY4NKA/PGD8VZVRISUJ-S9A0ZP_5G/EDIT?
UTM_CONTENT=DAG5SQY4NKA&UTM_CAMPAIGN=DESIGNSHARE&UTM_MEDIUM=LINK2&UTM_SOURCE=SHAREBUTTON