KMeans Clustering Algorithm Parallelization with CUDA

Task II

Presentation of Solution with required Scenarios:

Clustering algorithms belong to the data mining field. It divides a dataset into smaller chunks known as clusters. A cluster includes dataset records that are similar and is obtained through running an iterative set of algorithms over a dataset for a targeted range of instances. Giving the facts that a clustering algorithm is an iterative method, the quantity of time that is required for splitting a processed dataset in clusters is growing with the dimensions of the processed dataset.

For accelerating the speed of a clustering process, there are approaches: **you either run a parallel clustering on multiple machines or the use of an optimized clustering algorithm.** The resulting clusters can be post-processed to achieve further information or records. Primarily for this project, I will attempt to use a partition-based clustering algorithm based on the k-means clustering algorithm on multiple machines. This proposed algorithm is an improved model of the k-means algorithm that reduces the number of iterations (cycles) needed for acquiring the very final clusters. **By lowering the number of iterations that are needed for walking the clustering algorithm, the whole processing time required for obtaining the final clusters will also be reduced, that means that the volume of data that may be processed by using the equal computing device in the equal number of time can be increased.**

The Classical KMeans Algorithm

The k-means algorithm is an unsupervised learning algorithm that divides a dataset in clusters and it is one of the most used algorithms for data mining. The k-means algorithm is able to process only numerical values, so if a dataset that contains non-numerical value needs to be processed, a convert operation needs to be performed to bring all dataset vectors (records) to the form of a d– dimensional set:

$$\{ [x_i = [x_{i1} X_{i2} \dots x_i^d]^T \in \mathcal{R}^d \mid i = 1 \dots n\}, \quad (1)$$

Where the superscript T stands for matrix transposition and n is the number of database records. The objective of k-means algorithm is represented by minimizing the intra-cluster variance by solving the optimization problem:

$$c = \underset{C \in \mathcal{R}^{dk}}{argmin} V(c),\ V(c) = \sum_{\substack{j=1 \\ x_i \in c_j}}^{k} \sum_{i=1}^{n} \| x_i \dots c_j \| \quad (2)$$

Where:

k - the number of clusters C_j, j = 1 ... k,

$c_j = [c_{j1}\ c_{j2}\ \dots\ c_{jd}]^T \in \mathcal{R}^d$ - the centroid of the cluster

$c = [\ c^T_1\ \ c^T_2\ \dots\ c^T_k] \in \mathcal{R}^d$ - the centroid of the vector

$\{ c_1\ \ c_2\ \dots\ c_k\} \in \mathcal{R}^d$ - the set of the centroids

$c^* = [(c^*_1)^T\ \ (c^*_2)^T\ \dots\ (c^*_p)^T]^T \in \mathcal{R}^d$ - the vector of the optimal centroids and also the solution to the optimization problem (2). The k-means algorithm uses a mathematical formula for calculating the distance $d_{xi\ cj}$ from every record x_i and a centroid c_j . If the Euclidian distance $d_{xi\ cj}$ is considered:

$$d_{xi\ cj} = \| x_i \dots c_j \| = \sqrt{\sum_{i=1}^{d} (x_{i1} - c_{j1})^2} \quad (3)$$

by replacing the distance formula in (2) with the formula for calculating the Euclidian distance, the optimization problem becomes:

$$c = argmin V(c),\ V(c) = \sum_{j=1}^{k} \sum_{\substack{i=1 \\ x_i \in c_j}}^{n} \sqrt{\sum_{i=1}^{d} (x_{i1} - c_{j1})^2} \qquad (4)$$

The formula included in (4) that is used for calculating the distance between every record x_i and all centroids c_j determines the shape of the final clusters.

The k-means algorithm consists of five major steps, **0 to 4,** presented as follows. The step **0** is executed once, and the other ones are executed in an **iterative (cyclic)** manner until the final clusters are generated.

Step 0: this is an initialization step, where the number of clusters p, the initial centroids, the maximum number of iterations and the accepted error are chosen. The general notation μ will be used as follows for the current iteration, $\mu = 1 \ldots \mu_{max,}$ where μ_{max} is the maximum number of iterations, which needs to be initialized. The initial value $\mu = 1$ will be used.

There are several ways to initialize the **centroid vector c.** For this project I will use an initial centroid vector that contains the points where we believe that the final centroids will be. **This initial centroid vector is:**

$$c^{(1)} = [\, c^{T(1)}_1 \ \ c^{T(1)}_2 \ \ldots \ c^{T(1)}_p \,] \in \mathcal{R}^{d \quad p} \qquad (5)$$

and it will initialize the set of initial centroids:

$$\{ c^{(1)}_1 \ \ c^{(1)}_2 \ \ldots \ c^{(1)}_p \} \subset \mathcal{R}^d \qquad (6)$$

The accepted error $\square > 0$ needs to be initialized and it will be used as a stop condition in the algorithm.

Step 1. The distance $d_{xi\ cj}$ between every vector x_{i1} and a centroid c_j of cluster C_j needs to be calculated using (3) for i = 1 ... n and j = 1 ... k.

Step 2. Based on the distances calculated in step 1, a new set of clusters can be generated. Each new cluster $C^{(\mu)}_j$ will contain the vectors x_m that are closer to its center, the centroid $C_j^{(\mu)}$. The cluster generation process is based on the distances that were computed in step 1. Each point x_m has a number of k distances $d_{xm}\, c^{(\mu)}$ i = 1 ... k that were computed, the vector x_m will be included into a cluster $C_j^{(\mu)}$ if the distance $d_{xm}\, c_j^{(\mu)}$ is minimum, which means: Step 3. The new centroid

$$C_j^{(\mu)} = \{x_m \mid d_{xm}\, c^{(\mu)}_j \ \forall i = 1 \ldots k\} \tag{7}$$

where every vector is assigned only to a single cluster $C_j^{(\mu)}$

Step 3. The new centroid $\mathbf{c}_j^{(\mu+1)}$ needs to be computed for each cluster $C_j^{(\mu)}$ based on the vectors x_i that are assigned to the cluster . The new centroid is computed on the basis of the next formula that calculates the average of the coordinates for all vectors $x_j : \in C_j^{(\mu)}$:

$$\mathbf{c}_j^{(\mu+1)} = \frac{1}{|\mathbf{c}_j^{(\mu+1)}|} \sum_{x_i \in c_j^{(\mu)}} x_i \ ,j = 1 \ldots p, \tag{8}$$

Where $|C_j^{(\mu)}|$ is the number of vectors assigned to cluster $C_j^{(\mu)}$. Equation (8) generates the next centroid vector: $= c^{(1)} = [\, c^{T(1)}_1 \ \ c^{T(1)}_2 \ \ldots \ c^{T(1)}_p] \ \in \mathcal{R}^{dp}$

$$\mathbf{c}^{(\mu+1)} = [\, c_1^{T(\mu+1))} \ \ c_2^{T(\mu+1))} \ \ldots \ c^{T(\mu+1))}_p]^T \ \in \mathcal{R}^{dp} \tag{9}$$

The scalar version of (8) is: $| \mathbf{c}_j^{(\mu)}|$

$$\mathbf{c}_{ji}^{(\mu+1)} = \frac{1}{| \mathbf{c}_j^{(\mu)}|} \sum_{x_i \in c_j^{(\mu)}} x_{il}, \; l - 1 \ldots d, \;\; j = 1 \ldots p, \qquad (10)$$

Step 4. The stop condition of the algorithm is checked:

$$| \mathbf{c}^{(\mu+1)} - \mathbf{c}^{(\mu)}| < \square \qquad (11)$$

If the condition (11) is fulfilled, **this means that the k-means algorithm had run in a cyclic manner until the centroid vector c did not change between two consecutive cycles and the solution to the optimization** problem (2) is the last obtained vector c:

$$\mathbf{c}^* = \mathbf{c}^{(\mu+1)} \qquad (12)$$

which corresponds to the final centroids:

$$\{ c^*_1, \; c^*_{1 \,,\ldots.\,,} c^*_k \} = \{ c_1^{(\mu+1)}, \; c_2^{(\mu+1)} \,_{,\ldots.,} \; c_k^{(\mu+1)} \} \subset \mathcal{R}^d \qquad (13)$$

If the condition (11) is not fulfilled, then an increment of the current iteration replacing μ with $\mu + 1$ and the algorithm will continue with step 1.

RELATED WORK

The k-means algorithm can be applied to small datasets, medium datasets and even on big datasets. The total processing time that k-means needs to find an acceptable solution is proportional with the size of the processed dataset. If the hardware components that are used for running the k-means algorithm are not considered, it can be concluded that the total processing time that the k-means algorithm needs to process a dataset depends on two major things: the position of the initial centroids and the number of the iterations that are performed until an acceptable solution is found. **Because of the importance of initial centroid selection, different methods were proposed in the literature for improving the k-means by picking wisely the initial centroids.** The k-means++ algorithm was proposed in [13] in order to speed up the performance of the k-means algorithm. The k-means++ algorithm picks the first centroid in a random manner as the k-means algorithm does, but the other centroids are picked by the probability proportional to the shortest distance from all existing centroids. **A disadvantage of the k-means++ algorithm is the fact that it is inherently sequential**. Another version of k-means algorithm was suggested in [14], where the initial set of clusters is generated by computing pair-wise distances and all the closer points create a cluster. There is a threshold value for the minimum number of the points that can form a cluster. The initial centroids will result as the mean value of these centroids. Another method for selecting the initial centroids presented in [15] deals with sorting all processed records in a table, splitting the table in multiple sections (number of sections equals the number of desired centroids), and the mean value of each section will

represent an initial centroid. Another way of improving the performance of k-means algorithms is to reduce the number of the computation steps that are performed until an acceptable solution is found. The solution given in [16] to speed up the k-means algorithm consists in storing information about the data that is clustered. By keeping information about closer centroid for every processed record, many of the distance calculations can be avoided.

A disadvantage of this algorithm is the fact that it does not perform well when high dimension vectors are processed. Others versions of the k-means algorithm were modified to use the triangle inequality [17], [18] to avoid the unnecessary point-center distance calculation. **This algorithm caches distance bounds for reducing the number of distance calculations. Every time when a centroid moves, the cached distances are updated based on the triangle inequality. Another way to increase the speed of the k-means algorithm was proposed in [19] by observing the fact that in some cases, some of the clusters were not changing anymore between two consecutive iterations**. For all points that are assigned to these static clusters, the distance calculation can be skipped. O (nk) Another direction that evolved into the last years is represented by the online clustering that uses fuzzy logic [22]–[38] for classifying a dataset in a non-iterative manner. This kind of application is suitable for online data that cannot be loaded or stored into the memory for applying iterative clustering algorithms.

NEW K-MEANS ALGORITHM WITH CENTROID APPROACH

The proposed version of k-means algorithm has as a main target maintaining the objective specified in (2) but reducing the number of cycles that are performed for splitting all records in clusters. For achieving this objective a new step will be added between steps 3 and 4 of the classical algorithm and it has as a main objective the estimation of the way of how the centroids will evolve based on their previous values.

A. Monitoring the Centroids Evolution

The k-means algorithm stops when the centroids $\{ c_1^{(\mu)}, c_2^{(\mu)}, \ldots, c_k^{(\mu)} \}$ are not changing anymore between two consecutive iterations. If a centroid $\mathbf{c}_j$ is considered along with its value at the iterations μ and $\mu - 1$ and , then based on $c_j^{(\mu)}$ and $c_j^{(\mu-1)}$ the increment of centroid at the iteration μ is defined a the vector : $\Delta c_j^{(\mu)}$:

$$\Delta c_j^{(\mu)} = c_j^{(\mu)} - c_j^{(\mu-1)} = [\Delta c_{j1}^{(\mu)}, \Delta c_{j2}^{(\mu)} \ldots \Delta c_{jd}^{(\mu)}]^T \in \mathcal{R}^d \qquad (14)$$

$$\Delta c_{jl}^{(\mu)} = c_{jl}^{(\mu)} - c_{jl}^{(\mu-1)} , l = 1 \ldots d, j = 1 \ldots k$$

Each element of the increment vector $\Delta c_j^{(\mu)}$ provides information about the way of how the centroid c_j evolved between two consecutive iterations of the algorithm:

If the l^{th} element of the increment vector $\Delta c_j^{(\mu)}$ equals zero $\Delta c_{jl}^{(\mu)} = 0$ jlc between two consecutive iterations of the algorithm the th l^{th} element of

centroid $c_j^{(\mu)}$ did not change from the previous iteration $c_j^{(\mu-1)}$ namely $c_{jl}^{(\mu)} = c_{jl}^{(\mu-1)}$

If the l^{th} element of the increment vector $\Delta c_j^{(\mu)}$ equals zero $\Delta c_{jl}^{(\mu)} \neq o$ jlc between two consecutive iterations of the algorithm the th l^{th} element of centroid $c_j^{(\mu)}$ has change and did not match the one from the previous iteration $c_j^{(\mu-1)}$, i.e. $c_{jl}^{(\mu)} \neq c_{jl}^{(\mu-1)}$.

If the evolution of the increment of a centroid is analyzed, it is concluded that every element of the increment vector has a bigger value at the beginning. In the end, when an acceptable solution $\{c^*_1, c^*_{1,\ldots,}c^*_k\}$ is found, the values of the elements of the increment vector are zero.

Using the Evolution of the Centroids to Force the Convergence of the K-Means Algorithm Based on Centroid Update Approach

The evolution of the centroids can be used for forcing the convergence of the k-means algorithm to a solution. The most convenient case when a centroid update approach can be applied is the case when all elements of the increment vector $\Delta c_j^{(\mu)}$ of a centroid taken in absolute values are decreasing from the beginning of the algorithm until the algorithm ends, and every element of the increment vector keeps the same sign between consecutives cycles of the algorithm until the algorithm completes.

The worst situation for applying a centroid update approach is the case when all elements of the increment vector of a centroid taken in absolute

value are making an increasing and decreasing between consecutive steps of the algorithm from the beginning of the algorithm until the algorithm ends, and every element of the increment vector changes its sign between consecutives cycles of the algorithm. The centroid update approach can be applied if some of the following conditions are met:

a. The last four centroids $c_j^{(\mu)}$, $c_j^{(\mu-1)}$, $c_j^{(\mu-2)}$ and $c_j^{(\mu-3)}$ are known for $j = 1 \ldots k$.

b. The centroid update approach is applied if every element that has the same position index in the vectors $\Delta c_j^{(\mu)}$, $\Delta c_j^{(\mu-)}$ and $\Delta c_j^{(\mu-2)}$ has the same sign.

c. The centroid update approach is applied as long as a stop condition is not met.

d. The centroid update approach can be applied to one or multiple centroids in the same cycle of the algorithm.

e. The formula used to update the value of a centroid $c_j^{(\mu)}$ is:

$$c_j^{(\mu)} = [c_{j1}^{(\mu)}, c_{j2}^{(\mu)} \ldots c_{jd}^{(\mu)}]^T \quad (15)$$
$$+M[\Delta c_{j1}^{(\mu)}, \Delta c_{j2}^{(\mu)} \ldots \Delta c_{jd}^{(\mu)}]^T, j = 1 .. k,$$

$$\left| \frac{\Delta c_{ji}^{(\mu)}}{c_{ji}^{(\mu)}} \right| 100 > \square, \quad i = 1 \ldots d, \quad \text{(Condition II)}$$

If (there are centroids $c_j^{(\mu)}$ that respect the conditions a and b)

Then

Apply update approach to $c_j^{(\mu)}$ using (15)

Else

Go to step 3.

3. Centroid update algorithm not applicable at this iteration.

Validation

For validating the proposed algorithm, multiple synthetic datasets were used. The synthetic datasets were created using a random generation process, and every dataset that was used for validating the algorithm had different types of records (the number of the elements of each record was between 2 and 4) and different ranges for the values in the dataset (1) [20].

The platform used to process the datasets using the k-means with centroid update algorithm is SciPy and scikit-learn, a free and open-source Python library used for scientific computing and technical computing. This platform allows running the k-means algorithm in a single thread or in a multithread manner.

The machine that was used to run the k-means algorithm has the following hardware components: CPU Intel Core i7 10510U 4 cores, 8 Logical Processors, 16 GB of DDR3 RAM memory at 1.80 GHz, 2304 Mhz, 512 SSD hard drive. The hardware components are not very important in this study because the number of cycles that were executed is of interest in this paper and not the execution time of the k-means algorithm.

The operating system that was used for validating the algorithm using the platform was Windows 10 and the build version was 10.0.18362. The

parameters 10 and were used for all datasets that were processed to validate the algorithm.

The parameters $\beta = 10$ and $\square = 0.0001$ were used for all datasets that were processed to validate the algorithm.

A. Synthetic Dataset with Records of Two Scalars

The first dataset consists of 10 million records, each record representing a vector with two numerical elements (d=2), $\{x_i = [x_{i1} \; x_{i2}]^T \in \mathbb{R}^2 \mid i = 1 \ldots 10000000\}$ The values of x_{i1} belong to [-500, 200] and the values of x_{i2} belong to [-100, 500] .

Four records were chosen as initial centroids, $c_1^{(1)} = [-400 \; -80]^T$, $c_2^{(1)} = [-200 \; 50]^T$, $c_3^{(1)} = [0 \; 300]^T$, $c_4^{(1)} = [50 \; 400]^T$, and the maximum number of iterations was set to $\mu_{max} = 200$.

The diagonal update matrix $\mathbf{M} = \text{diag}(a_1, a_2)$ was set such that $a_1 = a_2 >= 0$, and the parameter $a_1 = a_2$ is referred to as the correction factor. The classical k-means algorithm was applied first to this dataset, and this corresponds to the zero value of the diagonal update matrix, i.e. $\mathbf{M} = \text{diag}(0, 0)$ and the correction factor also set to zero, $a_1 = a_2 = 0$.

Implementation of K-Means Clustering Algorithm with CUDA

K-means clustering is a *hard clustering* algorithm which means that each datapoint is assigned to one cluster (rather than multiple clusters with different probabilities). The algorithm starts with random cluster assignments and iterates between two steps

1. Assigning data points to clusters based on the closest centroid (by some distance metric).
2. Updating the centroids based on the new cluster assignments from the previous step.

Eventually the cluster assignments converge giving the final result. In this CUDA implementation each of the two steps will be performed in parallel. The implementation is for the 1-dimensional case where each datapoint is a scalar but it could easily be extended to the multi-dimensional case.

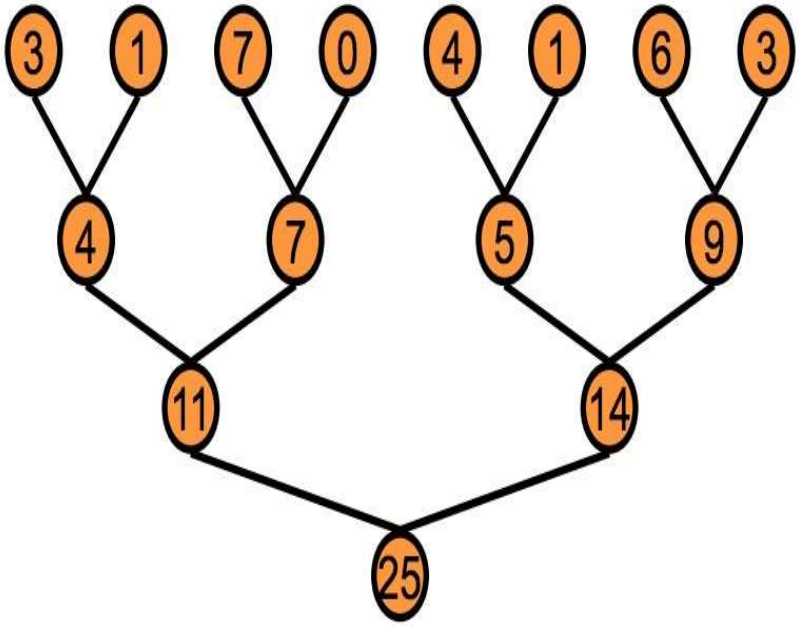
```
__global__ void kMeansClusterAssignment(float *d_datapoints, int *d_clust_assn, float *d_centroids)

{
```

```
    //get idx for this datapoint
    const int idx = blockIdx.x*blockDim.x + threadIdx.x;
    //bounds check
    if (idx >= N) return;
    //find the closest centroid to this datapoint
    float min_dist = INFINITY;
    int closest_centroid = 0;
    for(int c = 0; c<K;++c)
    {
        float dist = distance(d_datapoints[idx],d_centroids[c]);
        if(dist < min_dist)
        {
            min_dist = dist;
            closest_centroid=c;
        }
    }
    //assign closest cluster id for this datapoint/thread
    d_clust_assn[idx]=closest_centroid;
}
```

Updating Centroids

The next step is to recompute the centroids given the cluster assignments computed in the previous step. This is much more tricky to parallelize since the centroid computations depend on all of the other data points in its cluster. However, operations that rely on distributed datasets to compute a single output value can still be parallelized and are called *reductions*. A sum reduction is shown in the diagram below.



The general procedure is to partition the input array and perform the sum on each partition in parallel then merge the partitions and repeat the process until all partitions have been merged and you are left with the final sum value. This parallelization allows for logarithmic complexity rather than linear as with the serial case.

The centroid recomputation can also be thought of as a reduction operation and the code is shown below.

```
__global__ void kMeansCentroidUpdate(float *d_datapoints, int *d_clust_assn,
float *d_centroids, int *d_clust_sizes)

{


      //get idx of thread at grid level
```

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


	//bounds check
	if (idx >= N) return;


	//get idx of thread at the block level
	const int s_idx = threadIdx.x;


	//put the datapoints and corresponding cluster assignments in shared memory so that they can be summed by thread 0 later
	__shared__ float s_datapoints[TPB];
	s_datapoints[s_idx]= d_datapoints[idx];


	__shared__ int s_clust_assn[TPB];
	s_clust_assn[s_idx] = d_clust_assn[idx];


	__syncthreads();


	//it is the thread with idx 0 (in each block) that sums up all the values within the shared array for the block it is in
```

```
	if(s_idx==0)
	{
		float b_clust_datapoint_sums[K]={0};
		int b_clust_sizes[K]={0};

		for(int j=0; j< blockDim.x; ++j)
		{
			int clust_id = s_clust_assn[j];
			b_clust_datapoint_sums[clust_id]+=s_datapoints[j];
			b_clust_sizes[clust_id]+=1;
		}

		//Now we add the sums to the global centroids and add the counts to the global counts.
		for(int z=0; z < K; ++z)
		{
			atomicAdd(&d_centroids[z],b_clust_datapoint_sums[z]);
			atomicAdd(&d_clust_sizes[z],b_clust_sizes[z]);
		}
	}
```

```
        __syncthreads();


        //currently centroids are just sums, so divide by size to get actual
centroids
        if(idx < K){
                d_centroids[idx] = d_centroids[idx]/d_clust_sizes[idx];
        }


}
```

In the above code, `s_idx` refers to the thread at the block-level (i.e. multiple threads will have the same `s_idx` assuming more than one block is being used). The next bit of code

```
__shared__ float s_datapoints[TPB];
s_datapoints[s_idx]= d_datapoints[idx];


__shared__ int s_clust_assn[TPB];
s_clust_assn[s_idx] = d_clust_assn[idx];
```

is the partition step of the reduction where the global datapoint (`d_datapoints`) and cluster assignment (`d_clust_assn`) variables are brought into shared memory for the block that thread `idx` is in.

The next bit of code is only executed within the first thread of each block (i.e. `if(s_idx==0)...`). This thread creates two variables local to that thread - `b_clust_datapoint_sums` which is an array of length `K` that holds the sums of the datapoints assigned to each cluster and `b_clust_sizes` which is also an array of length `K` that holds the sizes of each cluster. The thread then traverses the datapoints and cluster assignments in the shared memory for its block and accumulates the results in these local variables.

The next step is to combine these intermediate results into the global variables `d_centroids` and `d_clust_sizes` to get the final update result. This is done with the `atomicAdd` function which ensures that race conditions will not occur when multiple threads are writing to the same place. However, we don't want our centroids to be sums so the last step is to divide the sums by the cluster sizes which can be done in parallel at the global thread level.

References

- K-means clustering
- CUDA reductions