Saint Petersburg National Research University of Information Technologies, Mechanics and Optics (ITMO University)
Faculty of Informational Technologies and Programming

REPORT

about laboratory work № 4

«Parallel clustering methods»

Student

Miroshnikova A.D. **j4133c**

(Surname, initials) Group

Saint-Petersburg, 2020

1. GOAL OF LABORATORY WORK

The goal of the work is to apply parallel programming techniques to the unsupervised machine learning problem – clustering.

2. TASK DEFINITION

Two well-known clustering algorithms that were proposed are K-means and DBSCAN. The parallel approach to both of them should be described in terms of PCAM (partition – communicate – agglomerate – map). Both sequential and parallel K-means algorithms should be implemented and the speedup should be calculated.

3. BRIEF THEORY

K-means

The **K-means algorithm** is a popular clusterization algorithms which minimizes the following objective (within-cluster sum of squares):

$$argmin_S \sum_{i=1}^{k} \sum_{x \in S_i} \|x - \mu_i\|^2 = argmin_S \sum_{i=1}^{k} |S_i| var\, S_i = argmin_S \sum_{i=1}^{k} \frac{1}{2|S_i|} \sum_{x,y \in S_i} \|x - y\|^2$$

The classic K-means algorithm consists of two consequent steps:

1. Assignment stage. Assign each observation to the cluster with the nearest mean (centroid). Requires calculating the distance between all observed points and the existing centroids.

```
def _calculate_labels(self, X_train, centroids):
    if not self.parallel or self.metric != "euclidean":
        distances = cdist(X_train, centroids, metric=self.metric)
    else:
        distances = self._parallel_dist(X_train, centroids)

    labels = np.argmin(distances, axis=1)
    return labels
```

This step involves 3 hidden loops in the function *cdist* with size *num_objects*, *num_centroids*, and *dimensionality_of_the_space*.

2. Update stage. Recalculate centroids based on the points in the cluster.

This step involves 1 external loop with size *num_centroids*, and two internal hidden loops used during calculation of the mean value of the coordinates. The size of the first loop (which refers to the size of the dataset subset) is unknown, but summary length of all those loops equals to *num_objects*, and one more loop is of the size *dimensionality_of_the_space*.

```python
def _calculate_new_centroids(self, X_train, labels):
    centroids = []
    for label in range(self.num_clusters):
        ids = labels == label
        if ids.sum() > 0:
            cluster_centroid = X_train[ids].mean(axis=0)
            centroids.append(cluster_centroid)
        else:
            new_centroid = []
            for b_mean, b_std in zip(self.train_mean, self.train_std):
                new_centroid.append(np.random.normal(loc=b_mean, scale=b_std))
            centroids.append(np.asarray(new_centroid))
    return np.asarray(centroids)
```

Both of these steps can be parallelized using PCAM methodology. The deconstruction of the **assignment stage** involves the following steps:

1. Partition — divide the dataset to distinct points (point, dimension) and centroids to (centroid, dimension).
2. Communicate — we parallelize the computation over dimensions (each thread computes the distance over selected dimension), so that different combinations (point, dimension) and (centroid, dimension) are connected for each combination (point, centroid).
3. Agglomerate — distance calculations should be summarized over dimensions for each combination (point, centroid).
4. Map — each thread solves the task of computing the distance for specific combination (point, centroid).

The parallelization of the **update stage** can be done with the following steps:

1. Partition — divide the dataset with labels to distinct points (label, point, dimension).
2. Communicate — each thread should have access to the matrix with shape (labels, dimensions) and array with shape (labels) which stores the sum of point coordinates for this label and dimension and number of points for each label.
3. Agglomerate — the summarization of contributions of each point to the centroid coordinates is accumulated within one single matrix with shape (labels, dimensions).
4. Map — each thread adds the coordinate values for the point (label, point, dimension) to the matrix with shape (labels, dimensions) and counts the number of points into the array with shape (labels).

At the final stage each value in the matrix (labels, dimensions) is divided by counts in the array (labels).

Thus deconstructed K-means algorithm can benefit from parallelization using many libraries and approaches.

DBSCAN

The **DBSCAN algorithm** is another popular algorithms based on density of the points in a given neighboring area. Classically it operates with the following **concepts**:

- *Eps* neighboring area: let p be the center of a sphere in the dataset D. For data within the radius Eps of the object's area, a collection of points contained in the sphere is $N_{Eps}(p)=\{q\in D|dist(p,q)\leq Eps\}$.
- *Density*: at the position of data point p in the dataset D, the number of points, Num, contained in the neighborhood *Eps* is its density.
- Core point: at the poisition of data point p in the dataset D, if the density (Num) in the neighborhood Eps satisfies $Num\geqslant MinPts$, it is called a core point.
- Border point: at the position of data point p in the dataset D, if the density in the neighborhodd Eps satisfies $Num\leqslant MinPts$, it s called a border point.
- Noise point: all the objects other than the core and border points in D.
- Direct density-reachable: given objects , w $p,q\in D$ here there is a core point and this is inside the Eps neighborhood of q, it is said that from p to q is direct density-reachable, i.e. $q\in N_{Eps}(p)$, $|N_{Eps}(p)|\geqslant MinPts$.
- Density-reachable: given objects $p_1,p_2,p_3,p_4,\ldots,p_n\in D$, where $p_1=q,p_n=q$, if p_{i+1} is directdensity-reachable from p_i , thep p is density-reachable from q.
- Density-connected: given objects $p,q\in D$, if there is a point $o\in D$ that is density-reachable from p and q, then p and q are density-connected.

The **sequential algorithms** consists of the following steps:

1. All the data objects in dataset D are marked as unchecked. Starting from any unchecked data point p, mark it as «checked», then check its Eps neighborhood and calculate the number of objects in the neighborhood m. If m satisfies $m\geqslant MinPts$, then create a new cluster C1, and add p to C1, then add all the points in the neighborhood to the collection of objects N1.
2. For the collection of objects N1, if object q therein has not been checked, then mark q as «checked», and then check its Eps neighborhood and calculate the number of objects in the neighborhood n. If n satisfies $n\geqslant MinPts$, then these objects are added to the object collection. If q does not belong to any cluster, then add q to C1.
3. Repeat step (2) until N1 is empty.
4. Repeat steps (1) to (3) until all the data objects are marked as «checked».

This algorithm can be parallelized using **PCAM methodology** in the following way:

1. Partition — divide the dataset into computing nodes by grids of selected size. The size to be selected depends on Eps.
2. Communicate — each of the computing nodes is processing its own grid of dataset and creating its own clusters.
3. Agglomerate — after each computing node finishes its calculations, the results are passed to the master node. The master node solves the problems of the border cases —

sometimes the noise points in one grid are actually members of the cluster from the other grid. Also it fuses the clusters from different grids.

4. Merge — at this stage the resulting clusters are renamed, that is, each cluster will hae only one global cluster number.

   This algorithm can be implemented on the servers with MapReduce approach.

4. ALGORITHM (METHOD) OF IMPLEMENTATION

The code for computations was implemented using Python 3.6.9 and *numba* library (version 0.52.0). The GPU used during measurements was GeForce GTX 1060 MaxQ, with CUDA version 10.2. Complete source code and data gathered are available on github https://github.com/Miffka/parallel_algo_itmo/tree/master/task4.

The selected data set was “Image data House” from http://cs.joensuu.fi/sipu/datasets/. It contains 34112 points in RGB space (number of dimensions is 3).

House
(256x256)

N=34112, D=3

RGB-values, quantized to 5 bits per color ts txt
RGB-values, 8 bits per color ts txt

The sequential K-means algorithm was implemented with use of SciPy function *cdist* which computes the distances between points in train dataset and centroids, and with use of NumPy built-in *.mean()* method which computed the new centroids in cycle over number of centroids.

Three CUDA kernels with global memory have been implemented — for assignment stage, update stage, and the function which calculates the distance between old centroids and new centroids.

The first kernel calculating the distances between the points in train dataset and centroids, was implemented as 2-dimensional grid with unequal number of threads for each dimension. The index over the first dimension was assigned to the index of point in dataset, the index of second dimension — to the index of centroid.

```
@cuda.jit()
def kernel_dist(X_train, centroids, dist_matrix):
    """
    Calculates euclidean distance between row and column in numba global.
    """
    row, col = cuda.grid(2) # from 0 to num_objects - 1, from 0 to NUM_CENTROIDS

    if row < dist_matrix.shape[0]:
        current_sum = 0
        for j in range(X_train.shape[1]):
            current_sum += (X_train[row, j] - centroids[col, j]) ** 2
        dist_matrix[row, col] = current_sum
```

```
def _parallel_dist(self, X_train, centroids):
    num_objects = X_train.shape[0]
    num_centroids = centroids.shape[0]
    X_train_cuda = cuda.to_device(X_train)
    centroids_cuda = cuda.to_device(centroids)
    distance_matrix = cuda.device_array((num_objects, num_centroids))

    kernel_dist[(BLOCKS_PER_GRID, NUM_CENTROIDS // 2), (NTHREADS, 2)](
        X_train_cuda, centroids_cuda, distance_matrix
    )
    dist = np.sqrt(distance_matrix.copy_to_host())
    return dist
```

The second kernel calculating the new centroids was implemented with use of atomic arrays — values for each dimension were accumulated in the distinct arrays as well as the number of points with this assigned cluter label. The parallelization used 1-dimensional grid with the only index assigned to the number of point in the dataset.

```
@cuda.jit()
def kernel_centroid(X_train, labels, sum_channel_1, sum_channel_2, sum_channel_3, count_array):
    row = cuda.grid(1)
    global_idx = row
    if row < X_train.shape[0]:
        cuda.atomic.add(sum_channel_1, global_idx, X_train[global_idx, 0])
        cuda.atomic.add(sum_channel_2, global_idx, X_train[global_idx, 1])
        cuda.atomic.add(sum_channel_3, global_idx, X_train[global_idx, 2])
        cuda.atomic.add(count_array, int32(labels[global_idx]), 1.0)
```

```
def _calculate_parallel_centroids(self, X_train, labels):
    num_dimensions = X_train.shape[1]
    X_train_cuda = cuda.to_device(X_train.astype(np.float32))
    labels_cuda = cuda.to_device(labels.astype(np.int32))
    sum_channel_1 = cuda.device_array((self.num_clusters,), dtype=np.float64)
    sum_channel_2 = cuda.device_array((self.num_clusters,), dtype=np.float64)
    sum_channel_3 = cuda.device_array((self.num_clusters,), dtype=np.float64)
    count_array = cuda.device_array((self.num_clusters,), dtype=np.float64)

    kernel_centroid[(BLOCKS_PER_GRID), (NTHREADS)](
        X_train_cuda, labels_cuda, sum_channel_1, sum_channel_2, sum_channel_3, count_array
    )
    sum_matrix_host = np.vstack([
        sum_channel_1.copy_to_host(),
        sum_channel_2.copy_to_host(),
        sum_channel_3.copy_to_host(),
        ]).T
    count_array_host = count_array.copy_to_host().reshape(-1, 1)
    centroids = sum_matrix_host / count_array_host
    return centroids
```

The third kernel calculating the distance between old and new centroids was implemented with parallelization over one dimension — the number of centroids.

```
@cuda.jit()
def kernel_pointwise_dist(centr_1, centr_2, distance_array):
    row = cuda.grid(1)
    if row < distance_array.shape[0]:
        current_sum = 0
        for j in range(centr_1.shape[1]):
            current_sum += (centr_1[row, j] - centr_2[row, j]) ** 2
        distance_array[row] = current_sum
```

```
def _calculate_parallel_centroid_dist(self, centr_1, centr_2):
    num_objects = centr_1.shape[0]
    centr_1_cuda = cuda.to_device(centr_1)
    centr_2_cuda = cuda.to_device(centr_2)
    distance_array = cuda.device_array((num_objects,))

    kernel_pointwise_dist[(NUM_CENTROIDS // 2), (2)](
        centr_1_cuda, centr_2_cuda, distance_array
    )
    dist = np.sqrt(distance_array.copy_to_host())
    return dist
```

The experiments that were scheduled included:

- variation of the number of the clusters — {2, 4, 8, 16, 32, 64, 128, 256};
- number of threads — {8, 16, 32, 64}.

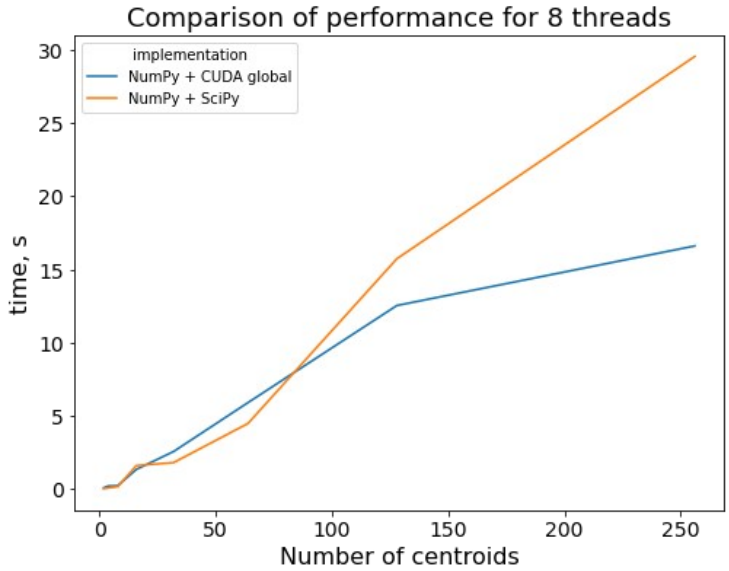
5. RESULT AND EXPERIMENTS

The results were analyzed in Python 3.6.9.

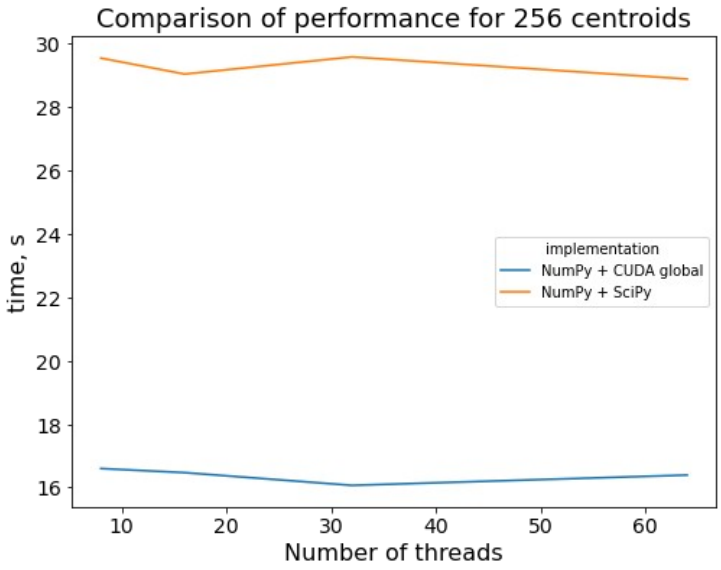
As in the previous work with CUDA, the first call of the clustering was thrown away since JIT (just-in-time) compilation happened during the first call of the function and took long time.

The influence of the number of the clusters and number of threads was analyzed.

The benefit of CUDA parallelization is observable only when number of centroids becomes large enough – 128 centroids or more. For the lower number of centroids the time needed for clustering is compatible with the sequential implementation.



As in previous work, the number of threads did not influence the performance of the parallelized algorithm.



6. CONCLUSION

Two popular clusterization algorithms, K-means and DBSCAN, were analyzed and the parallelization methods with PCAM methodology were proposed. Both sequential and parallel variants of K-means algorithm were implemented. This algorithm benefits from the parallel approach when the number of clusters becomes significant – 128 or more. That is, parallel approach to the machine learning problems should be considered when the investigator deal with comparatively big data.