PC-2023/24 Final-Term Project

Giacomo Magistrato
E-mail address
giacomo.magistrato@edu.unifi.

Abstract

The objective of this report is to analyze and evaluate the advantages of implementing parallelism through the GPUs utilization, in relation to the K-Means clustering algorithm. The report aims at highlighting how parallel programming can enhance the efficiency of the algorithm, and under which circumstances, with the eventually encountered limitations. More specifically, the benefits of parallelism have been outlined by implementing the algorithm in C++ and creating a parallel version of the program, in which part of the code is executed by a GPU. The source code will be released publicly.

1. Introduction

The K-Means clustering is an unsupervised machine learning algorithm used to process a given dataset and partition the contained data into K unique clusters. An algorithm is unsupervised if it works on unlabelled data. This means that it doesn't have any information on how the given records are related between themselves. The aim of these algorithms is to detect patterns and relationships within the input data, with no given prior direction. A cluster is a subset of the initial data in which all the contained elements share common features or characteristics. The K-Means algorithm relies on the position of the so called "centroids". A centroid represents the average center position of all the elements in a cluster. It actually synthesizes the characteristics of all the enclosed data points. The objective of the algorithm is to minimize the differences between the points inside each cluster while maximizing the ones in between all the clusters. In order to achieve the best consistency during the comparison of the sequential and parallel execution of the program, the same algorithm has been implemented in both versions. The two versions also share the same centroid initialization that is performed randomly, for each program execution. This ensures that the same clustering steps are performed in both cases. Since the objective of this report is to evaluate the differences in terms of performances, a dataset of 1 million records has been specifically generated for benchmarking purposes.

2. The K-Means Algorithm

The algorithm consists of a sequence of steps in which the data points are iteratively assigned to their nearest cluster, and the related centroids positions are consecutively updated. Since the given data could eventually be multi dimensional, it is usually necessary to change its domain by creating a space in which the distance between the provided points becomes meaningful and can actually be computed. This means that non-numerical data must be encoded first, so that it can be evaluated properly. The given dataset allows to work directly with no encoding necessity, and the preprocessing logic of the data has not been implemented. The implemented clustering logic comprises the following steps:

- **Initialization**: K centroids are randomly placed into the features' space starting from the given dataset. K must be selected by the user.

- **Assignment**: This phase consists of the computation of the distances in between each point and all the available centroids. Each data point is then assigned to the nearest clus- ter of which the centroid represents the center point, based onto a specific distance metric. In this case, the **Euclidean distance** between two n-dimensional points:

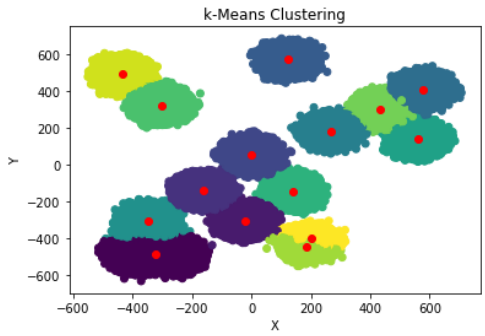
$$d(\mathbf{x}, \mathbf{y}) = \sqrt{\sum_{i=1}^{n} (x_i - y_i)^2}$$

$$\mathbf{x} = (\mathbf{x}_1, x_2, \ldots, x_n)$$
$$\mathbf{y} = (y_1, y_2, \ldots, y_n)$$

- **Update**: after assigning all the points in the batch to their cluster, each centroid's position is updated by calculating the average coordinates of the contained data points. During the next iteration, the new centroids' positions will be used to compute the distances again.

- **Stop criterion**: There are many ways to determine when to stop. In this case the stop criterion is a fixed number of iteration of the algorithm.

- **Result:** After the fixed iteration all the results are stored in external files. Using a Python script result can be plot as following:



2.1. GPU implementation

By applying the above algorithm in C++ there is a limitation in parallelizing the code execution. There are two synchronization points that are strictly required in order to get the desired out- come, as some of the involved steps must be exe- cuted consecutively. The algorithm consists of two main parts, for which two CUDA kernel functions have been ac- cordingly defined. The parallel version of the project then relies on the kernels, that are exe- cuted on the GPU, to achieve better performances.

- The first kernel manages the data points assignments

- The second includes the centroids' updates

2.2. Assignment

As stated previously, the first kernel is responsi- ble for computing all the data points assignments to their nearest clusters. After copying the data points contained in the cur- rent batch to the device, the kernel is launched with the required number of blocks and threads, in order to handle all points simultaneously.
The "global " keyword tells the compiler that the kernel is launched by the host, but executed on the device. All the pointers to the device allocated memory are passed here as references in order to allow each thread to work with them. The kernel is typically launched with a call in which the total number of blocks and threads per block is specified just before the invocation. Below is presented the code for this section:

```
__global__ void clusterAssignment(const float *d_xval, const float *d_yval, int *d_clusterval,
const float *d_centroidX, const float *d_centroidY, int K, bool *d_done,
                                  float *d_clusterSumX, float *d_clusterSumY,
                                  int *d_clusterSize){

    //indice del thread a livello grid
    const int idx = blockIdx.x*blockDim.x + threadIdx.x;
    *d_done = true;
    if (idx >= N) return;
    float min_dist = INFINITY;
    int closest_centroid = 0;

    float dist;
    for(int i = 0; i < K; i++)
    {
        float sum = 0.0;
        sum += pow(d_centroidX[i] - d_xval[idx], 2.0);
        sum += pow(d_centroidY[i] - d_yval[idx], 2.0);
        dist = sqrt(sum);
        if(dist < min_dist){
            min_dist = dist;
            closest_centroid = i;
        }
    }

    //assegno id-cluster al thread corrente
    if( d_clusterval[idx] != closest_centroid) {
        d_clusterval[idx] = closest_centroid;
        *d_done = false;
    }
```

The kernel consists of two sections: in the first one is computed the distance beetween every points and the centroids, in order to find the nearest one. The second one assign for every points the nearest cluster.

2.3. Assignment

The kernel responsible for the update of each centroid is launched again with a ”global” function. In this case after setting 2 variables corrisponding to the clusters sum of the coordinates to 0, the kernel function perform an atomic add foreach points for each cluster. Then these sums are divided for the size of each cluster in order to perform the average point, the new centroid. The code for this section is presented below:

```
__global__ void clusterPointsSum(float* d_xval, float* d_yval, int* d_clusterVal,
float* d_clusterSumX, float* d_clusterSumY, int* d_clusterSize){

    //indice del thread a livello grid
    const int idx = blockIdx.x * blockDim.x + threadIdx.x;

    if (idx >= N) return;

    int clusterId = d_clusterVal[idx];
    //sommo tutti i punti appartenenti ai clusters
    atomicAdd(&(d_clusterSumX[clusterId]), d_xval[idx]);
    atomicAdd(&(d_clusterSumY[clusterId]), d_yval[idx]);
    atomicAdd(&(d_clusterSize[clusterId]), 1);
```

3. Performance Analysis

The testing phase has been carried out by executing both the sequential and the parallel algorithm implementations multiple times. Both versions share the same centroids initializations and solution steps thanks to the generated random seed. The performed tests consist of multiple execu-tions with different combinations of clusters, data points numbers and different numbers of centroids.

3.1. Speedup

In order to evaluate the benefits in terms of performances between the two versions of the program, the speedup metric has been calculated. The speedup is the Sequential Execution Time divided by the Parallel Execution Time

3.2. Considerations

By fixing the number of data points and looking at the elapsed times of the two versions, it is possible to notice that the time required by the CPU grows much faster than the time required by the GPU, but in both cases the increment is strictly dependent on the number of clusters. So the number of clusters has a big impact on the computation compared to the number of data points. It's worth noticing that the bigger the input data size is, the more is beneficial to achieve parallelism thanks on the GPU.

3.3. Results

In order to evaluate the overall performances, each one of the following combinations of clusters and data points sizes has been tested 10 times. The tests have been structured with all the combinations in between the following sets:
Clusters = 5, 15
Datapoints = 1000, 100000, 750000, 1000000
TPB = 8,16, 32, 128
Speedups are show below:

K-Means Speedups with K = 5

	8	16	32	128
1000	0.88	0.71	0.70	0.55
100000	5.04	7.20	8.6	7.11
750000	5.45	7.86	10.02	10.5
1 M	5.56	7.92	10.05	11.16

K-Means Speedups with K = 15

	8	16	32	128
1000	1.10	0.90	0.88	0.62
100000	6.4	7.91	10.65	13.21
750000	7.5	11.92	17.70	18.30
1 M	7.60	12.20	18.01	18.51

4. Conclusion

As can be seen from the results, running the algorithm in parallel with small datasets leads to terrible speedups, regardless of the number of centroids. As expected as the size of the dataset increases, the speedup also increases and we can see a big difference in terms of speedup by changing the number of centroids. With K = 15 the number of operation grows and parallelism works very well in this cases.

Due to the observations made above we can see that the worst speedup is 0.55 and occurs in the case where K = 5 and the dataset size is the smallest while the best speedup is 18.51 and occurs in the case where K = 15 and the dataset size is the greatest.

The source code of the project and its related re- porting are publicly available on GitHub: https://github.com/Mayo98/KMeansParallel