K-means algorithm with CUDA and JavaThreads

Pietro Zarri
pietro.zarri@stud.unifi.it

Abstract

The aim of this project is to verify how some parallelization techniques can improve execution time of K-means algorithm.
In particular two approaches are presented: CUDA and Java Threads.

1. Introduction

One of the most famous and spread clustering technique is the K-means algorithm. The main focus of this project is on how some code parallelization technique can improve the performance of this algorithm on a quite large dataset.
In particular in this experiment are presented two different approaches. The first one uses the CUDA architecture, the second one instead uses the Java threads support.
Moreover the results are analyzed by varying the number of clusters that K-means creates.

2. K-means

2.1. Clustering

Cluster analysis or **clustering** is the task of grouping a set of objects in such a way that objects in the same group (called a cluster) are more similar (in some sense) to each other than to those in other groups.
In theory, data points that are in the same group should have similar features, while data points in different groups should have highly dissimilar features. Clustering is a method of unsupervised learning and is a common technique for statistical data analysis used in many fields.

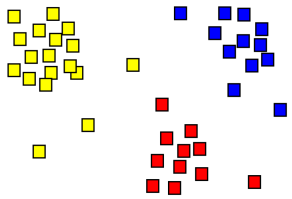

Figure 1. Cluster classification example

2.2. K-means algorithm

K-means[1] is one of the simplest unsupervised learning algorithms that solve the well known clustering problem.
The procedure follows a simple and easy way to classify a given data set through a certain number of clusters (assume k clusters) fixed a priori.
The main idea is to define k centroids, one for each cluster. The first step is to take each point belonging to a given dataset and associate it to the nearest centroid.
When no point is pending, the first step is completed and an early groupage is done. At this point we need to re-calculate k new centroids as barycenters of the clusters resulting from the previous step.
After we have these k new centroids, a new binding has to be done between the same dataset points and the nearest new centroid. A loop has been generated.
As a result of this loop we may notice that the k centroids change their location step by step until no more changes are done.

In other words centroids do not move any more. When this happens the algorithm ends.

Input: dataset
Output: labeled dataset
repeat
  foreach *point in the dataset* **do**
    find the nearest *centroid* to the point;
    label each point with its nearest *centroid*;
  end
  foreach *centroid* **do**
    update *centroid* coordinates;
  end
until *there's no label changed*;

Algorithm 1: K-means pseudocode

3. Dataset

For this experiment is used a set of 500000 points in the 2D space. The metric used to evaluate the distance between two point is the Euclidean distance.
The Euclidean distance between points **p** and **q** is the length of the line segment connecting them:

$$d(\mathbf{p}, \mathbf{q}) = \sqrt{\sum_{i=1}^{2}(p_i - q_i)^2}$$

4. CUDA approach

CUDA (Compute Unified Device Architecture)[2] is a parallel computing platform and application programming interface (API) model created by Nvidia. It allows software developers and software engineers to use a CUDA-enabled graphics processing unit (GPU) for general purpose processing.
The CUDA platform is a software layer that gives direct access to the GPU's virtual instruction set and parallel computational elements.
The CUDA platform is designed to work with programming languages such as C, C++, and Fortran.

4.1. Computer specification

All the tests with CUDA architecture are executed on an EC2 instance of Amazon Web Services (AWS). In particular was used a *p2.xlarge* instance with the following specification:

- Intel Xeon CPU E5-2686 v4 @2.30GHz
- 59GB RAM
- Nvidia Tesla K80 24GB GDDR5
- Xfce 4.12

4.2. Implementation

The two most expensive routine in term of execution time in the K-means algorithm are for sure the *for* loop that assigns to every point its nearest centroid and the following one that recalculate the position on every centroid.
Since the creation and handling of threads on GPU's is far less expensive than on CPU's one of the best strategy is remove the *for* loops and assign every iteration to a different thread[3]. This can be done thanks to CUDA *kernel* function, declared as `__global__`.
This keywords identifies a function callable only from the **host** (CPU and its memory) and executed on the **device** (GPU and its memory).
Threads rely on the following two unique coordinates to distinguish themselves from each other:

- *blockIdx* (block index within a grid)
- *threadIdx* (thread index within a block)

```
<<<ceil(p->size/BLOCK_SIZE),BLOCK_SIZE>>>
```

The number of blocks is determined by dividing the total number of points in the dataset by the dimension of every block, stored in the variable `BLOCK_SIZE`.
NVIDIA provides an *xls* file to evaluate which is the best value of block size with different configurations. Through tests with different number of centroids a good compromise was found with the value of 128 (threads/block).
To store each points centroids two *Struct of Array* (SoF) was used.

```
struct PointsSet{
    float x[SIZE];
    float y[SIZE];
    int labels[SIZE];
    int size;
}
```

```
struct CentroidsSet{
    float x[K];
    float y[K];
    int label[K];
    int size;
}
```

As mentioned before the key of CUDA parallelism is to move the execution of code from the CPU to the GPU, CUDA provides also specific API to allow memory transition between *host* and *device*. For this project in particular was used the following functions:

- *cudaMalloc*: Allocate space on the device.
- *cudaMemcpy*: Copy a value between host and device.
- *cudaFree*: Deallocate space on the device.

To summarize, all the resources are copied from the host to the device, then the *kernel* function `assignLabelKernel` launches one thread for every point and calculates the nearest centroid. Then `updateCentroidsKernel` function updates the position of every centroid. Again is used one thread for every point. All the sums needed to calculate the two coordinates of the centroids are preformed through `atomicAdd` function to guarantee mutual exclusion.

```
__global__ void assingLabels(...){
  int index=threadIdx.x + blockIdx.x * blockDim.x;
  if (index < points->size) {
    float minDistance=distance(p->x[index],c->x[0],p->y[index],
                                                    c->y[0]);
    int minIndex = 0;
    int j;
    float tmpDistance;
    for (j=1; j<centroids->size; j++) {
      tmpDistance=distance(p->x[index],c->x[j],p->y[index],
                                                    c->y[j]);
      if (tmpDistance < minDistance) {
        minDistance = tmpDistance;
        minIndex = j;
      }
    }
    if (points->labels[index] != minIndex) {
        atomicAdd(u, 1);
    }
    points->labels[index] = minIndex;
  }
}
```

Listing 1. assignLabel kernel function

```
__global__ void updateCentroidsKernel(...) {
 int index=threadIdx.x + blockDim.x * blockIdx.x;
  if(index < p->size) {
   atomicAdd(&(c->x[points->labels[index]]),
                                        p->x[index]);
   atomicAdd(&(c->y[points->labels[index]]),
                                        p->y[index]);
   atomicAdd(&(size[p->labels[index]]), 1);
  }
}
```

Listing 2. Update Centroid Kernel

4.3. Results

In the following table are presented the execution times on a 500000 points dataset. The sequential execution and the CUDA parallel execution were tested more times with a different number of clusters.

N Clusters	Sequential	Parallel	Speedup
5	3.6935 s	0.865781 s	4.29
10	8.0494 s	1.165572 s	6.93
15	15.7597 s	1.811773 s	8.07

Table 1. K-means CUDA with 500000 points

The speedup Sp is defined as $Sp = ts/tp$ where, p is the number of processors, ts is the completion time of the sequential algorithm and tp is the completion time of the parallel algorithm.

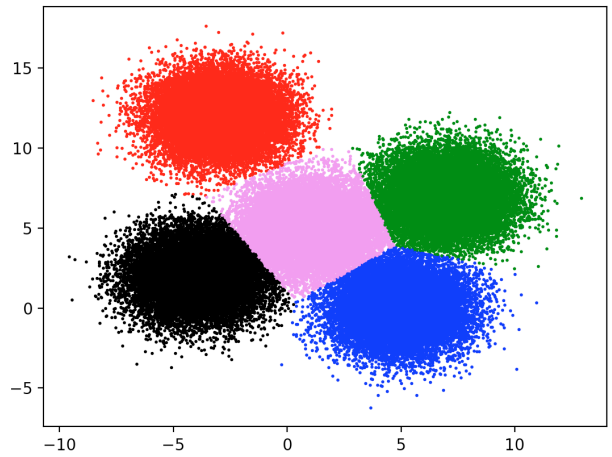

Figure 2. Classification with CUDA parallelization on 5 clusters dataset

5. JAVA Threads approach

Unlike many other computer languages, Java provides built-in support for multithreading[4]. Multithreading in Java contains two or more parts that can run concurrently. A Java thread is actually a lightweight process.
Threads can be created by using two mechanism:

- Extending Thread class
- Implementing the Runnable/Callable interface

From Java 5 lots of other feature were introduced to allow users an easy and fast management of Java threads.

5.1. Computer specification

All the tests with JAVA Threads technique are executed on a MacBook Pro with the following specifications:

- 2,7 GHz Intel Core i5 dual-core
- 8GB RAM
- Intel Iris Graphics 6100 1536 MB
- macOS Catalina 10.15.5

5.2. Implementation

In order to implement the K-means algorithm with Java were built different classes:

- Point: contains point's coordinates and its label.
- Centroid: contains centroid's coordinates, an id and the number of points that are assigned to it.
- PointSet: contains an *Arraylist* of all Points and has the task to initialize all points.
- CentroidSet: contains an *ArrayList* of Centroids and has the task to initialize all centroids.
- Kmeans: contains the sequential and the parallel implementation of the algorithm.

To parallelize the updating operations of points and centroids were built two class that implement the *Callable* interface:

- AssignLabelThread
- UpdateCentroidThread

The Callable interface compared to *Runnable* interface allow to return a value. This is import because we can keep track on how many updates are done on every iteration of the algorithm.

```
@Override
public Integer call() {
int updates = 0;
for (int i=start; i<end; i++) {
 Point p = points.getPoint(i);
 float minDistance = computeDistance(...);
 int minIndex = 0;
 for(int j=1; j<NumCentroids; j++) {
  float distance = computeDistance(...);
  if (distance < minDistance) {
   minDistance = distance;
   minIndex = j;
  }
 }
 if (p.getLabel() != minIndex) {
  p.setLabel(minIndex);
  updates++;
 }
}
return updates;
}
```

Listing 3. AssignLabelThread

Implementors define a single method with no arguments called `call`.
In order to catch all the results return by all threads executed in one iteration the *Future* interface is used.
The `get()` method gets the results. If the result isn't yet available, it'll block until it is.
The number of threads used for the first phase is set equal to the number of core, so all the dataset's points are split into 4 different chunk and every chunk is assigned to one thread. For the updating centroid phase is used one thread for every centroid.
In order to handle all these threads is used the *ExecutorService*[5] interface. The Java ExecutorService interface is responsible for the creation and the management of a pool of threads.

```
@Override
public Boolean call() {
 float x = 0;
 float y = 0;
 int count = 0;
 for (int i=0; i<points.getSize(); i++) {
  if(points.getPoint(i).getLabel()==
                              centr.getId()){
   count ++;
   x += points.getPoint(i).getX();
   y += points.getPoint(i).getY();
  }
 }
 x /= count;
 y /= count;
 centr.setX(x);
 centr.setY(y);
 centr.addPoints(count);
 return Boolean.TRUE;
}
```

Listing 4. UpdateCentroidThread

5.3. Results

The following table reports this Java Threads results: the first column shows the number of clusters generated by the algorithm. The second and the third one show the time of execution respectively of the sequential and the parallel implementation. All the time reported are the result of the average made of 20 executions.
The speedups in the fourth column show the effectiveness of this methods and reduce significantly the execution time of this algorithm, with better results with a higher number of clusters.

N Clusters	Sequential	Parallel	Speedup
5	3.0293 s	1.8163 s	1.67
10	6.7653 s	2.9163 s	2.32
15	8.0296 s	3.2324 s	2.48

Table 2. K-means Java Threads with 500000 points

6. Conclusion

In this project were implemented two different strategy to parallelize the execution of the clustering algorithm K-means: CUDA and Java Threads. Both have succeeded pretty good in improving performances but have some differences.

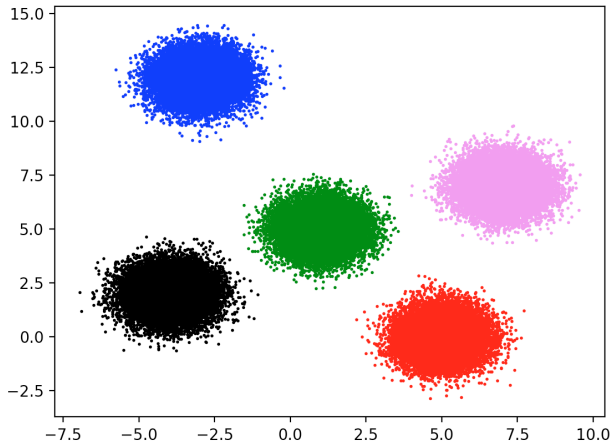

Figure 3. Classification with Java Threads on 5 clusters dataset

The Java Threads strategy provides a very easy and powerful to exploit multi-threading.
CUDA on the other hand is a more low-level approach, available only on NVIDIA GPUs, that represents the fine granularity principle at its highest level. This is possible thanks to the very high number of threads that a GPU can handle.

Bibliografia

[1] ”Wikipedia”. K-means. https://it.wikipedia.org/wiki/K-means.

[2] ”NVIDIA”. Cuda. https://developer.nvidia.com/cuda-downloads.

[3] ”Norm Matloff”. Programming on parallel machines.

[4] ”Oracle”. Java thread. https://docs.oracle.com/javase/tutorial/essential/concurrency/runthread.html.

[5] ”Oracle”. Executor service. https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/ExecutorService.html.