Parallel K-means

Luigi Cennini
E-mail address
luigi.cennini@edu.unifi.it

Abstract

This project aims to implement and compare sequential and parallel versions of the K-means algorithm using both OpenMP and CUDA. The objective is to analyze performance and measure the speedup achieved by the parallel version compared to the sequential one.

1. System Specification

CPU: Intel i3-4005U (4 Threads) 1.600GHz
GPU: NVIDIA GeForce 920M

2. Introduction to K-means

K-means clustering is an unsupervised machine learning algorithm used for partitioning a dataset into a predetermined number of clusters. The objective of K-means is to group similar data points together and discover underlying patterns or structures within the data.

At its core, K-means operates by iteratively assigning each data point to the nearest cluster centroid and then recalculating the centroids based on the mean of the data points assigned to each cluster. This process continues until the centroids no longer change significantly, or a specified number of iterations is reached.

One of the key parameters in K-means is 'k', which represents the number of clusters the algorithm should identify.

One iteration of the algorithms works in this way

1. for every point in the dataset, the point is assigned to the nearest centroid
2. the centroid are updated and convergence is checked

The algorithm stops when

1. the maximum number of iteration is reached
2. convergence is reached. This happens when two consecutive iteration of the algorithm don't change the centroid position (given a certain tollerance).

3. Sequential implementation

I implemented a KMeans algorithm in C++ with a KMeans class. I used a array of structure approach (AoS) over a structure of arrays (SoA) because the SoA implementation will have drawbacks. In fact the K-means algorithm needs to access, for each point, all the coordinates of the point, for this reason the SoA approach will have too many cache misses.
In my implementation, the public method:

```
void fit(std::vector<Point>& dataPoints,int
    maxIteration);
```

takes a vector of data points and the maximum iteration number. For test purpose data points are generated by a python script ($create\_dataset.py$) which generates a csv file that is imported as a $Point$ vector in c++ using the function $loadCsv$ (defined in $utils.cpp$).

At each iteration of the algorithm the centroids vector attribute of the class is updated, and at the end it is possible to export the result of the fit

method using the $exportCsv$ function (defined in $utils.cpp$).

An example of the result of the algorithm can be seen in Figure 1.

3.1. centroids initialization

The function fit of the class K-means expect the vector $centroids$ to be already initialized. Centroids can be initialized in many ways, for example with an algorithm known as K-means++. For the sake of semplicity i decided to choose random K random points in the dataset and making them the initial centroids.

4. parallel implementation with openMP

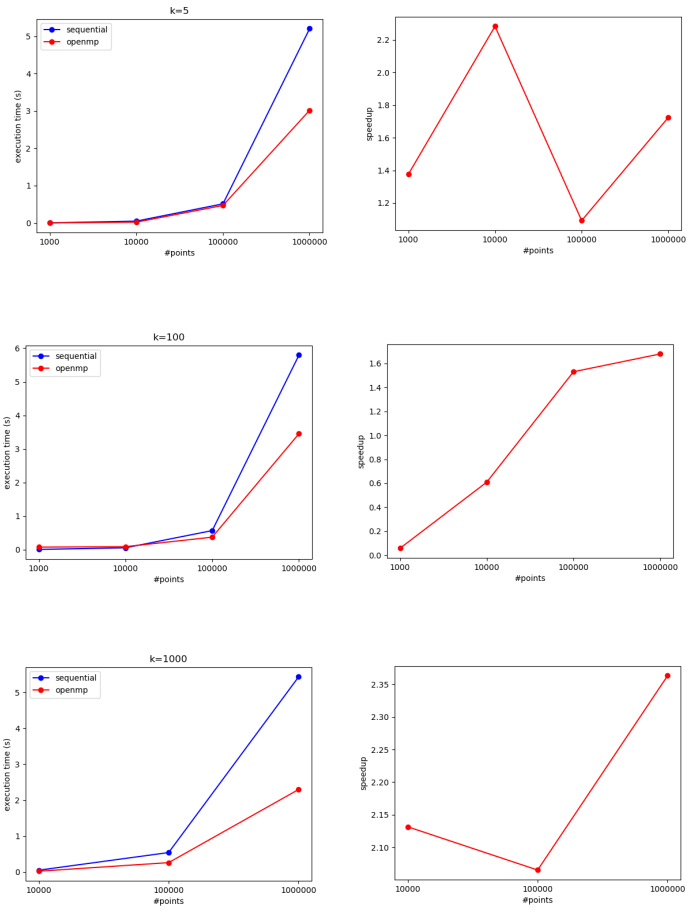
It is clear that the part which takes more time in every iteration of the algorithm is the first one (if $|dataPoints| >> k$), because it has to cycle through every point of the dataset. The embarrassingly parallel problem revolves around assigning each point to its respective cluster. Synchronization issues arise when updating the cluster with the mean of the points. To tackle this, I implemented a reduction. In my implementation, each thread possesses a private vector of centroids ($newCentroidsTmp$) on which operations are performed locally. Only at the end are these operations transcribed into global memory. To mitigate issues of false sharing, atomics are employed. Of course we can't directly use the $reduction$ directive of openmp because we need to do the reduction on a whole array of values, so we need to manually implement the reduction in this way:

```
#pragma omp parallel
{
    newCentroidsTmp = []
#pragma omp for schedule(static)
    for (point p in dataPoints){
        //assign cluster to p
        //update newCentroidsTmp
    }
    //perform global newCentroids update
}
```

For test purpose i choose to not implement the stop condition (in both sequential and parallel version) since i noticed that many times the number of iteration is less than the thread available, so speedup ended up being not noticeable.

5. speedup analysis

For the speedup analysis the same centroids are used for both the sequential and parallel run. Dataset are generated with increasing number of points and the times of the sequential and parallel program are compared. Here we can see the time and speedup for $K = 5, 100, 1000$ clusters.



We can see that the maximum speedup we get is about 2.3.

6. CUDA

CUDA version of kmeans can be implemented as the sequential one, implementing a reduction

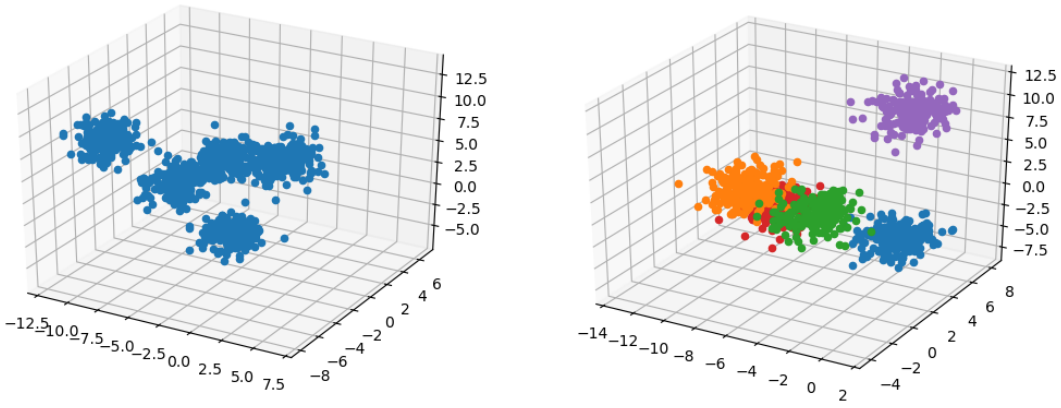


Figure 1. on the left a 1000 points dataset with 3 dimension, on the right the result of K-means with K=5

in order to reduce the number of atomics operation. The reduction is implemented in the kernel $centroidUpdate$. The other kernel i implemented is $centroidAssign$, it's purpose is to assign a cluster to each point. This operation is an embarassingly parallel one and doesn't require any coordination between threads.

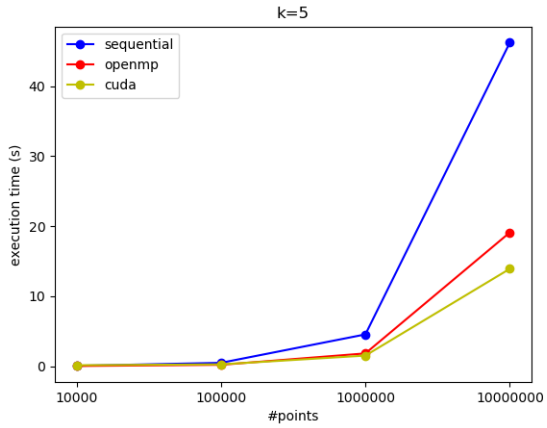
For data storage i decided to follow the sequential implementation and go for 3 dimension point dataset. Every point is stored with each coordinate next to each other and than there is another component that rapresent the cluster label which the point is assigned to during the algorithm. Same thing happens for centroids, but every 4th cell the centroid cardinality is stored.

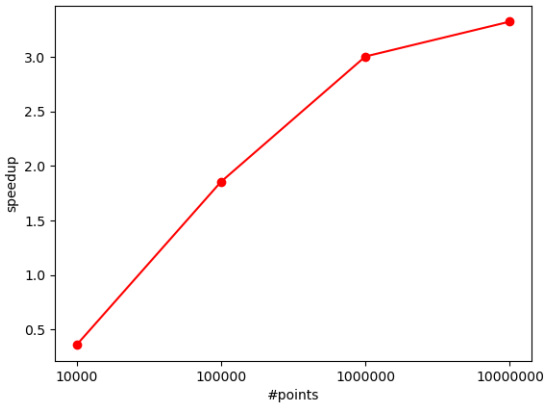
So after the execution of the kernel $centroidAssign$, every $4th * i$ $(i = 1, .., N)$ cell of the data points array is labeled with the corresponding closest centroid label. Similarly, after the execution of $centroidUpdate$ kernel, every $4th * i$ $(i = 1, .., K)$ cell of the centroids points is labeled with the corresponding cardinality and the points component are updated with the sum of associated points.

The kernel $centroidUpdate$ uses shared memory and performs a reduction in order to reduce the number of atomicAdd performed. Every first thread of a block calculates the sum of the dimension of the points associated to a cluster and then performs the atomicAdd.

7. Speedup analysis

Here all 3 versions are compared, with datasets up to 10^7 points.





We can see that the speedup for the CUDA ver-

sion is more than 3. Blocks of 512 threads were chosen for the parallel version because analyzing performance with $nvprof$ with higher thread counts leads to performance degradation. For example here we can see first the profiled output with thread blocks of dimension 1024 than 512.

```
==78696== Profiling application: ./cuda 1000000_5.csv 1000000_5_centroid.csv
==78696== Profiling result:
            Type  Time(%)      Time     Calls       Avg       Min       Max  Name
 GPU activities:   93.07%  4.10219s       100  41.022ms  40.693ms  44.085ms  centroidUpdate(float*, float*, int)
                    6.47%  285.31ms       100  2.8531ms  2.7960ms  2.9012ms  centroidAssign(float*, float*, int)
                    0.23%  10.351ms       102  101.48us  1.0880us  10.233ms  [CUDA memcpy HtoD]
                    0.22%  9.7747ms       102  95.838us  1.5360us  9.5821ms  [CUDA memcpy DtoH]
```

```
==78797== Profiling application: ./cuda 1000000_5.csv 1000000_5_centroid.csv
==78797== Profiling result:
            Type  Time(%)      Time     Calls       Avg       Min       Max  Name
 GPU activities:   78.76%  1.07809s       100  10.781ms  10.622ms  14.555ms  centroidUpdate(float*, float*, int)
                   19.77%  270.67ms       100  2.7067ms  2.6916ms  3.7813ms  centroidAssign(float*, float*, int)
                    0.76%  10.351ms       102  101.48us  1.1200us  10.235ms  [CUDA memcpy HtoD]
                    0.71%  9.7634ms       102  95.719us  1.5360us  9.5823ms  [CUDA memcpy DtoH]
```

We can notice how the bottleneck is the $centroidUpdate$ kernel as we expected and that with smaller block sizes the performance is better.

Of course if we choose block sized too small, perfomance might be worse too.

```
==83508== Profiling application: ./cuda 1000000_5.csv 1000000_5_centroid.csv
==83508== Profiling result:
            Type  Time(%)      Time     Calls       Avg       Min       Max  Name
 GPU activities:   79.25%  1.10666s       100  11.067ms  10.950ms  13.974ms  centroidUpdate(float*, float*, int)
                   19.31%  269.70ms       100  2.6970ms  2.6795ms  4.0611ms  centroidAssign(float*, float*, int)
                    0.74%  10.357ms       102  101.54us  1.0880us  10.241ms  [CUDA memcpy HtoD]
                    0.70%  9.7473ms       102  95.561us  1.5360us  9.5794ms  [CUDA memcpy DtoH]
```

Here block sizes of 256 are tested.