K-means:
Two different versions using Java executors and CUDA

Niccolò Bellaccini
niccolo.bellaccini@stud.unifi.it

Tommaso Aldinucci
tommaso.aldinucci@stud.unifi.it

Abstract

Clustering problem involves many scientific areas and, due to its structure, it is suitable for a parallel development. We present a dual multithread version of k-means: a Java implementation and a CUDA variant. This work aims to evince the perfomance benefits which can be gained using a parallel paradigm.

Our model correctness has been evaluated in terms of speedup parameter. The experimental results demonstrate significant speedup values in both implementation, especially through the Graphics Processing Units (GPUs) usage.

Future Distribution Permission

1. Introduction

Clustering is the process of examining a collection of points, and grouping them into “clusters” according to some distance measures. The goal is that points in the same cluster have a small distance from one another, while points in different clusters have a larger one [5].

In literature there are many clustering algorithms, we deal with the k-means family. This kind of family assumes an Euclidean space and a previously known number of clusters, indicated with k, and splits the entire dataset in k clusters. Points belonging to the same cluster are associated with a single point called *centroid*, determined by an average of the positions of each point. In the rest of this article, we will use the convention of labeling every point by the cluster it belongs to.

Generally, k-means is widely used in unsupervised learning to get some information from big data *e.g.* handwritten digits classification, color quantization and many other.

1.1. Method overview

A typical layout of k-means algorithm is basically made up of two *for* statements: the label assignment and the centroids updating. Below we present a general form of the algorithm.

Algorithm 1: k-means algorithm

```
Input: Dataset $d$
Output: Labeled dataset
Choose k initial centroids
do
    foreach Point $p \in d$ do
        find the centroid $c$ nearest to $p$
        assign $p$ to the cluster represented by $c$
    end
    foreach Centroid $c$ do
        update the position of $c$
    end
while There's at least a label changing;
```

For our purpose we assumed $\mathbb{R}^2$ as the Euclidean space for the algorithm.

The k initial centroids may be chosen in various ways. The easiest approach is to select them randomly such that each resulting centroid differs from any other.

A more suitable strategy is to pick points far away from each other, hoping they will belong to different clusters at the end. This can be made choosing the first centroid randomly, and the other $k-1$ such that each one has the *max-min* distance from the previous ones calculated. Down below we show a pseudo-code of this procedure

```
Algorithm 2: Choose initial centroids
Input: Dataset $d$
Output: K initial centroids
Choose the first centroid $c_0$ randomly from $d$
for $i \in \{1 \dots (k-1)\}$ do
    $c_i \in \arg\max_{x \in d} \min_{j \in \{0 \dots (i-1)\}} dist(x, c_j)$
end
```

Let us focus for a moment on the complexity of the above algorithm. The first centroid can be chosen randomly in $O(1)$ then, supposing the cost to get the distance between two 2D-points approximately constant, we can select the second one in at most n iterations, the third in $2n$ iterations and so on. Therefore, the final cost can be calculated as:

$$\sum_{i=1}^{k-1} in = \frac{nk(k-1)}{2} \quad i.e.\ O(nk^2) \tag{1}$$

Since in most practical cases $k^2 \ll n$, $O(nk^2)$ is closer to $O(n)$ than $O(n^2)$.

In both methods proposed in this article we chosed this *max-min* strategy because, empirically, it has been shown more efficient compared to the *random* method, increasing the convergence speed.

2. Java approach

Java platform provides different mechanisms to make thread-safe and parallel code. Using not-sychronized data structure is, as much as possible, the best choice as it decreases system latency exploiting multithreading advantages.

Executors, Callables and *Futures* are tools provided since Java 5 that allow the realization of parallel models in a more user-friendly way, lightening the programmer by creating and managing threads.

2.1. Implementation specifics

From an implementation point of view, we made a class which wraps the entire dataset using an *ArrayList* structure containing the points. Each point is represented by an istance of a *Point* class whose state consists of two coordinates and a label.

The "KMeans" class, made only by static methods, encapsulates the core of the algorithm. In order to achieve label assignment and centroids updating phases in a parallel way, a class per-phase was developed inheriting from the *Callable* interface. An *ExecutorService* object is responsible for the creation and management of a group of tasks which will be assigned to a fixed number of threads, taking advantage from its *FixedThreadPool* creation.

During the label assignment is created a number of tasks equal to the number of threads in the pool. A different chunk of dataset (with size equal to $\lceil DatasetSize/threadPoolSize \rceil$) is assigned to each task filling all data. This strategy allows a multithread implementation through a not thread safe data structure, as *ArrayList*, without any kind of synchronization.

To know when all the tasks will be terminated, we created a barrier mechanism using the *InvokeAll* method (of the *ExecutorService* class) and the *Future* interface to find out if some points changed their cluster during the assignment step.

Similarly, centroids-updating phase uses the mentioned classes to split the workload into k tasks. Centroids are also stored in an *ArrayList* structure where each thread deals with the computation of different centroids writing on indipendent positions of the array; during the task i, only a thread of the pool determines the centroid of the cluster associated with i label, avoiding the use of synchronized data structures and race conditions.

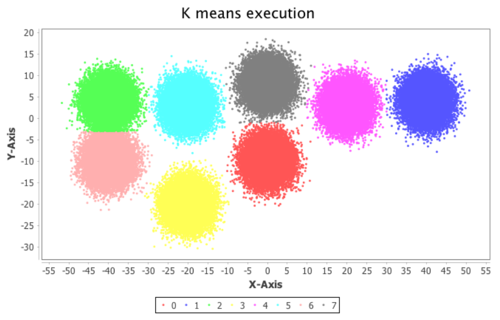

Figure 1. Java execution with $k = 8$

3. CUDA approach

GPUs usage has been increased thanks to technology improvements which have been allowing massively parallel programming. From 2007 NVIDIA has produced a new hardware architecture for parallel elaboration called CUDA (Compute Unified Device Architecture) leading to a general-purpose development, using NVIDIA APIs [3].

Generally, a fine-granularity idiom may be used to fully exploit the hardware potential in order to solve naturally parallelizable problems. For these reasons the k-means algorithm was redesigned in a CUDA implementation view. Further details will be provided in the next subsection.

3.1. Implementation specifics

Since threads creation and management cost less in GPUs than CPUs, CUDA programming is often useful to remove *for* statements assigning just one iteration to each thread. Below, a pseudo-code of CUDA k-means implementation is shown.

Algorithm 3: k-means CUDA algorithm

```
Input: Dataset $d$
Output: Labeled dataset
Choose k initial centroids
do
    Let $p$ the point related to the current thread
    Phase 1:
        find the centroid $c$ nearest to $p$
        assign $p$ to the cluster represented by $c$
    end
    Wait for all threads to terminate phase 1
    Phase 2:
        concur to the computation of the related
        centroid
    end
    Wait for all threads to terminate phase 2
while There's at least a label changing;
```

A C-style *struct* was used to store the entire dataset organizing all fields as arrays, obtaining a *Struct of Array* (SoA). A SoA is a struct where the components are stored in separated arrays, improving memory efficiency thanks to *coalesced* memory access pattern.

CUDA framework permits device computation through *kernel functions* (entry point of GPU code), using data structures located in GPU's RAM. For this reason, CUDA provides specific APIs to handle the *host/device* memory transition and vice versa. The functions *cudaMalloc()* and *cudaMemcpy()* respectively allocate device memory and copy data beetween host and device in both directions [4].

In 2013, CUDA 6 introduced *Unified Memory* which makes possible to access both CPU and GPU memories with the use of a single pointer. This can be made with *cudaMallocManaged()* function that is responsible for managing all your data in this virtually "shared memory". However, for practical purposes, tipically this solution reveals inefficient so, after we experienced it, we opted for a "classical" implementation.

The assignment phase was achieved defining a *kernel* function where, for each point, a different thread determines to which cluster the point belongs to. The *kernel* grid has a simple unidimensional organization, made by a number of blocks equal to $\lceil DatasetSize/BlockDim \rceil$. The setup of the blocks dimension is a crucial operation that significantly affects application perfomance. NVIDIA [1] provides (inside CUDA SDK toolkit) a useful *xls* file that calculates threads device occupancy given the block size and other parameters. This file suggested an incomplete utilization of our GPU whether the block-size is less then 64, leading us to choose a block-size of 256 as a right trade-off after we tested different configurations.

The main idea behind the updating phase is that each thread represents a single point and concurs to the update of only one centroid, introducing possible race conditions due to modification of a shared data in centroids array. For this reason, each thread atomically adds the coordinates of the represented point to the relative centroid, checking the point label.

A CUDA program can perform an *atomic add* operation on a memory location through the function call:

```
int atomicAdd(int* address, int val)
```

This function avoids any kind of software-level synchronization between threads inside the kernel function, allowing an efficient *mutually exclusive* add operation.

It is clearly that, at host side, there is a CUDA statement forcing the application to wait for kernel completation; a host function then summarizes every sum making the average.

Assignment and updating phases are repeated till there is at least a point which changed its label. In order to do this, in the assignment phase, each thread increases a shared variable counting the number of label changes. Because the correct value of it is not strictly necessary (we only want to know if its value is greater than 0), this is executed in a not-synchronized context, saving an atomic operation.

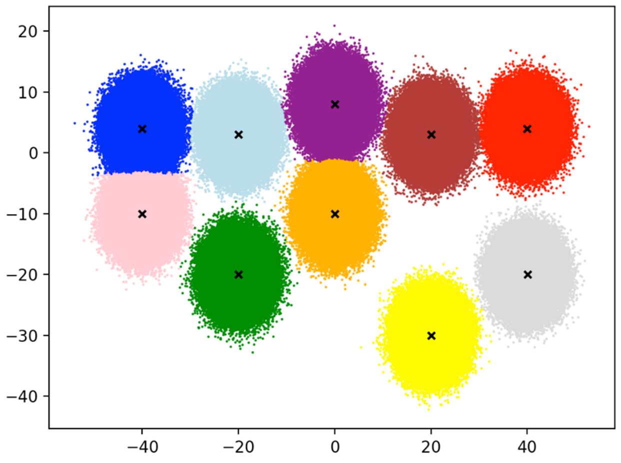

Figure 2. Cuda execution

4. Results

Proposed results highlight benefits obtained by the multithread approach, in terms of speedup parameter. The speedup S_p is defined as $S_p = t_s/t_p$ where, P is the number of processors, t_s is the completion time of sequential algorithm and t_p is the completion time of parallel algorithm [2].

4.1. Java scores

The following scores are related to the computational time of *do-while* statement referring to algorithm 1. Each reported result was achieved on a machine with a quad-core Intel(R) Core(R) i7 processor @2.5 GHz, 16 GB DDR3 RAM @1600 MHz and MacOS Sierra 10.12 operating system.

Some tests on different datasets were made using a *python* script to generate blobs for clustering. The two following tables represent the results obtained by varying dataset size (3 million and 200 thousand) and the number of clusters. The last one reports the speedups referred to the previous tables.

	K=5	**K=8**	**K=10**
3M	0.767	10.030	85.143
200K	0.110	0.314	0.628

Table 1. Java timings (s): 8 threads

	K=5	**K=8**	**K=10**
3M	3.682	49.362	433.509
200K	0.235	1.132	2.323

Table 2. Java timings (s): 1 thread

	K=5	**K=8**	**K=10**
3M	4.8	4.9	5.1
200K	2.1	3.6	3.7

Table 3. Java speedups 1/8

Speedups table reveal that, especially with large datasets, we can significantly improve the performance of the algorithm adopting a multithread method. A further python script was used to obtain all reported timings, averaging over 50 different results.

4.2. CUDA scores

The same kind of tests were achieved on CUDA implementation using the previously adopted datasets. The used server mounts an Intel(R) Xeon(R) CPU E5-2620 v3 @ 2.40Ghz and one NVIDIA GeForce GTX TITAN X. Down below we report the results obtained.

	K=5	K=8	K=10
3M	0.18	0.40	0.63
200K	0.07	0.14	0.18

Table 4. CUDA (sec)

	K=5	K=8	K=10
3M	3.21	11.61	19.78
200K	0.31	0.71	0.92

Table 5. C++ single thread(sec)

	K=5	K=8	K=10
3M	17.83	29.03	31.40
200K	4.43	5.07	5.11

Table 6. CUDA speedup

To get the average times, a simple *python* script was written here too. Timings (and so speedups) related to CUDA implementation can significantly differ with regard to the specifical GPU hardware. Indeed, the same implementation was tested on a NVIDIA GeForce GT 750M (laptop GPU) getting a speedup almost equal to 1.

Table 4 shows considerable improvements in particular for 3 million of points, justifying a CUDA usage especially for large datasets. Comparing speedups for smaller datasets, a Java (eventually C++) implementation would be more suitable.

5. Conclusions

We described a Java and CUDA based methods to exploit parallelism in k-means algorithm. The first implementation uses Java threads through *java.util.concurrent* package, providing a modern-style code and taking advantages of this multithread high level framework. The CUDA version is a little bit more "raw" than Java variant, aiming particularly for a massively parallel execution and exploiting the low-cost of GPU threads. We underline that, in both implementation, no synchronized data structure was used. The proposed solutions proved to be computationally efficient showing considerable speedups.

6. References

References

[1] CUDA Occupancy Calculator. Web search using keywords "CUDA Occupancy Calculator".

[2] L. S. Calvin Lin. *Principles of parallel programming*. Pearson.

[3] W.-m. W. H. David B. Kirk. *Programming Massively Parallel Processors*.

[4] T. M. John Cheng, Max Grossman. Professional cuda c programming.

[5] J. U. Leskovec A. Rajaraman. *Mining Massive Datasets*.