PC-2020/21 Parallel implementations of k-means clustering in Java and CUDA

Cosimo Cinquilli
E-mail address
cosimo.cinquilli@stud.unifi.it

Gabriele Giannini
E-mail address
gabriele.giannini3@stud.unifi.it

Abstract

Our project aims at analyze two different approaches in parallelizing the k-means clustering algorithm, implemented in two different languages, Java and C++. We implemented the parallel version in C++ to get access to the GPU hardware through the usage of the CUDA API. To better evaluate the impact of our decisions in the parallelizing process and also to have more meaningful data, we also developed two sequential versions, both in Java and C++. This paper goes through some of the implementations peculiarities and contains the metrics and some considerations about them.

1. Introduction

The k-means clustering algorithm it's a well established and used algorithm that operates on vectors. We referred to Wikipedia to get the details of the algorithm and ultimately implemented a "naive" version.

1.1. The algoritnm

As we implemented it, the k-means clustering consist of just two main phases: computing the centroids of the clusters and update the clusters.

To compute the centroids we simply take the mean of every coordinate of the vectors in a cluster as the corresponding coordinate of the centroid of that cluster.

Whilst updating the clusters is achieved by computing the Euclidean norm of the difference between any vector and every centroid and putting the vector in the cluster corresponding to the centroid for which the obtained norm was the minimum one. This operations are repeated until convergence is achieed (that is, two consecutive iterations output the same clusters, with no modfications).

To obtain the final clustering, however, more invocations of the complete algorithm are computed (10). Every invocation start on a randomly initialized set of centroids, randomly choosen among the vectors to be clusterized. For each of them a sum of the norms between each vector and its centroid of reference is computed. The clustering that is chosen as output is the one that has the lowest value for this sum.

2. Two different approaches

The differences in working with the Java platform and the CUDA API are many, but the two implementations are comparable and somewhat similar. They both parallelize the computation of the norm, the assignment of vectors to clusters and the updating of the centroids. They are both briefly explained below.

2.1. The Java implementation

We have subdivided the code in 4 classes: Common, Kmeans, MainWithThreads and KmeansPar. In the Kmeans class resides the code for the sequential implementation in Java, fr comparison purposes. The KmeanPar and MainWithThreads classes implement the actual parallel algorithm executed on tha Java lower level Thread API, also using some atomic types and barriers (java.util.concurrent.CyclicBarrier).

The Common class contains some utility functions and useful constant as EXECUTIONS_COUNT (default to 10) and THREAD_COUNT (default to 16, as the numer of hardware threads in the CPU on which metrics has been collected).

2.2. The CUDA implementation

Two functions has been declared as __global__ : normA2 and clustering.

The normA2 function uses the GPU to compute the Euclidean norm parallelized for every difference vector between any vector and each centroid. In other words, if K is the number of requested clusters and N is the number of vectors to clusterize, there will be K*N threads to run on the GPU to compute all the norms.

The clustering function uses the norms calculated before to assign each vectors to a cluster and in the meantime builds the new centroids coordinates.

For implementing a sequential version we adapted the code from the CUDA version into a normal C++ program (still using simple arrays instead of vectors from the STL) so the two versions should be as closely related as possible.

3. Compilation and execution

Cmake has been used to manage the building process of the CUDA and C++ sequential implementations. Using the CmakeLists.txt file included in the project, on a systema with the CUDA tools installed, should produce the two executable files: kmeans-cuda and kmeanSeq.

The programs both accept some command line options:

- -c <integer> to pass the number of requested clusters
- -tr <integer> to pass the total run count, meaning the number of k-means invocations to run
- -f <path-to-file> to pass the input file path
- -p to switch on printing the final clustering on the terminal

For the Java version no particular build tool configuration is provided, but opening the project in IntelliJ IDEA should automatically import the run configurations to build and run the provided classes (they are included in the repository in kmean/.idea/runConfigurations).

4. Performance metrics

Here we present and compare metrics obtained from the various programs we wrote. All metrics shown have been collected on the same system, consisting of a octacore (16 threads) CPU and a nVidia GeForce 1070ti (8GB of dedicated memory).
We consider that the parallelism degree of the execution on the GPU is 1920, that is the number of Cuda cores, since they act like execution limit for the 12 milion of threads we launch every kmean execution.

Programs has been ran with an total run count of 10.

4.1. Java implementation

4.1.1 Sequential program

- **Completion time**: 1,031,985,158µs
- **Throughput**: $9.690 \cdot 10^{-04}$ operations executed in 1µs. 1/Service time.
- **Service time**: 103.1985µs mean time on any complete execution of a k-means clustering.

4.1.2 Low level thread implementation

- **Completion time**: 670,914,565µs
- **Throughput**: $1.490 \cdot 10^{-2}$ operations executed in 1µs. 1/Service time.
- **Service time**: 67,091.5µs mean time on any complete execution of a k-means clustering.
- **Scalability**:
  - throughput par/throughput seq = 1.538
- **Speedup**: ts/tp = 1,538
- **Efficiency**: speedup/numthreads = 0.096

4.2. CUDA implementation

4.2.1 Sequential program

- **Completion time**: 154,594,431µs
- **Throughput**: 0.7762 operations executed in 1µs. 1/Service time.
- **Service time**: 1.2882µs mean time on any complete execution of a k-means clustering.

4.2.2 Parallel program

- **Completion time**: 23,388,788µs
- **Throughput**: 5.13 operations executed in 1µs. 1/Service time.
- **Service time**: 0.1949µs mean time between the start of two consecutive threads.
- **Scalability**:
  - throughput par/throughput seq = 6.6091
- **Speedup**: ts/tp = 6.6097
- **Efficiency**: speedup/numthreads = 0.0034

5. Conclusions

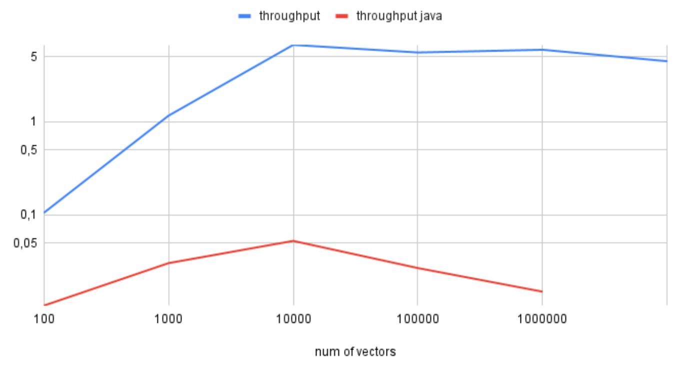

Figure 1: comparioson between the throughput trend of Cuda and Java implementations with respect to a growing number of vectors to analyze. (logarithmic scale on both axes)

In conclusion we observed that both implementations work better than their sequential implementations.

From the figure above we see that the throughput of both implementations grows with the number of vectors until the number of vector is less than 10,000.

From the metrics calculated above is possible to see that efficiency is very low for the Cuda implementation but probably this metric is less important due to the big number of Cuda cores used for the computations. In this case we consider the throughput as the most significant metric and it grows linearly until it stabilizes with more than 10,000 vectors analyzed.