Università di Pisa

Computer Architecture - Project: K-Means

Antonio Patimo, Guillaume Quint

Academic Year: 2022/2023

Abstract

In this project we present the work done in developing a multi-threaded application for the K-Means algorithm, and its performance analysis over two different computer architectures: CPU and GPU. After each iteration, we performed a series of observations about the results obtained. This allowed us to present effective optimizations for both architectures, showing the improvement achieved. At the end we performed a brief comparison between the CPU's and GPU's version of the software.

Contents

1 Parallel K-Means algorithm

As a reference for comparing the two architectures, CPU and GPU, we chose the Federated K-Means Algorithm, which is a parallel implementation of the popular K-Means algorithm.

This is an unsupervised clustering algorithm which works over a dataset of points in a given dimension space. Given a parameter K, the algorithm finds K clusters among all points in the dataset.

The Federated version of this algorithm allows to distribute the dataset between multiple parallel executing units. In particular, the dataset is partitioned into n chunks, where n is the number of executing units (in our case, the number of threads). Each unit links each point of its partition with the closest centroid, then sums up the components of every points for each centroid and counts the number of points associated to each centroid. These partial results are then sent back to a master unit which aggregates them to compute a new approximation for the centroids. These new centroids are then sent to every computing unit iteratively until convergence of the centroids.

Listing 1 shows the pseudo-code for the Federated K-Means clustering algorithm.

Figure 1 shows a snapshot of 6 iterations in a bi-dimensional case with $K = 4$

```
1 pocedure K-Means-Federated(dataset, K, n):
2     centroids = generateRandomCentroids(K)
3     partitions = generatePartitions(dataset, n)
4     workers = generateWorkers(n, partitions)
5     while (not converged):
6         workers.send(centroids)
7         partial_results = workers.execute()
8         workers.synchronize()
9         updateCentroids(partial_results)
```

Listing 1: Federated K-Means pseudocode

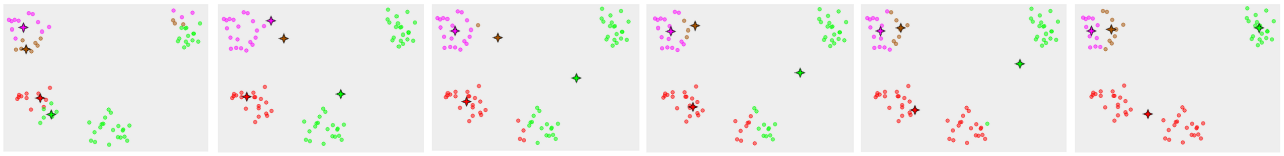

Figure 1: 6 iterations of the K-Means clustering algorithm

1 .1 Real use case application

We imagined a realistic scenario for which the Federated K-Means clustering might be useful.

An international Bank wants to identify client groups based on their monthly income and expenditure to propose personalized investment plans. We envisioned a typical volume of around 10 million users and 5 distinct groups. The product is therefore a customer clustering software for personalized advertisement.

2 CPU

In this section we're dealing with the CPU implementation of the software and relative performance analysis. We then develop our observations about the obtained results in order to implement an optimization.

2 .1 Hardware Specification

All tests were conducted on an Intel i7-6700HQ of which we reported relevant hardware specifications in Table 1

CPU cores	4
Threads	8
L1 cache	4 x 32 KB 8-way set associative instruction caches 4 x 32 KB 8-way set associative data caches
L2 cache	4 x 256 KB 4-way set associative caches
L3 cache	6MB 12-way set associative shared cache
Clock speed	2.60GHz - 3.5GHz

Table 1: Hardware specification

2 .2 Implementation

Our software implementation accepts three arguments: a dataset of serialized points, the number of clusters and the number of threads. In particular the dataset of points must be preprocessed from a csv file using the CSVPreprocessor software that we develop to store data in a more convenient binary format. This allows us to read faster from file, so that we don't need to convert from a literal number format.

The dataset is then loaded in an array of classed points, which are represented as a tuple of coordinates and an index representing which cluster each point belongs to. Centroids are initialized by considering K points in the dataset chosen at random.

To build the partitions, points are distributed among all available threads.

Inside the performRounds() function, for each round we generate an array of threads executing the worker() function. We then join on their termination and update our approximation for the centroids, until the maximum error goes below a certain threshold.

Each thread in the worker() function finds the closest centroid of every point in its partition. It then calls the aggregatePoints() function to sum up the coordinates of all points belonging to the same centroid.

In Listing 2 we can see the data structures used for the CPU implementation of the software

```
1  struct Point_s
2  {
3     double* coords;
4  };
5  typedef struct Point_s Point;
6
7  struct ClassedPoint_s
8  {
9     Point p;
10    int k;
11 };
12 typedef struct ClassedPoint_s ClassedPoint;
13
14 struct Centroid_s
15 {
16    Point p;
17    Point* sum;
18    int* partition_lengths;
19 };
20 typedef struct Centroid_s Centroid;
21
22 ClassedPoint* points;
23 Centroid* centroids;
```

Listing 2: cpu data structures

2 .3 Results

The executing times are shown with a varying number of threads and different dataset sizes in Figure **??**. Results are obtained as a sample mean with population width of 30 samples. We also show the 95% confidence interval where values are appreciable.