K-Means clustering

OpenMP and CUDA

Marco Di Rienzo

Final-term assignment

Introduction

Abstract

The goal of this project is to parallelize the K-Means Clustering algorithm and analyze the performance gain with respect to the sequential version.

One parallel versions has been created using the **OpenMP** framework and another using **CUDA** to run on the GPU.

The performance gain is measured in terms of

$$\text{speed-up} = \frac{\text{sequential time}}{\text{parallel time}}$$

Pseudocode

Algorithm 1: K-Means Clustering

Data: k, $max\_iterations$, $observations$
Result: $centroids$, $labels$
1. Select k random $observations$ as starting $centroids$;
while *not* $max\_iterations$ **do**
  foreach $observation$ **do**
    2. Compute the distance between each $centroid$ and $observation$;
    3. Assign $observation$ to closest $centroid$;
  end
  4. Compute new $centroids$ (mean of $observations$ in a cluster);
end

The algorithm can be divided in 4 steps.

The **embarassingly parallel** portion of the algorithm is the **FOR** loop (steps 2. and 3.), every iterarion is independent from one another.

Observations can be distributed among any number of threads.

Implementation

Sequential

The sequential implementation is basically a C++ translation of the pseudocode, with some nuances:

1. o*bservations* and *centroids* are organized as **Structure of Arrays** to improve the alignment with cache lines and benefit of the memory bursts when accessing the various coordinates;

2. step 4. is partially executed in the for loop, in particular the sum required to compute the mean is updated as soon as a point is assigned to a centroid. This avoids having to loop again on every observation.

OpenMP

The directive **#pragma omp parallel for** allows splitting observation indices among threads, so that each one will be assigned a different subset of the dataset.

N = **coresNumber** threads

Thread 1
for (int i=0;
i<datasetSize/N; i++) {
- assign point[i] to centroid
- **update local partial sum**

}

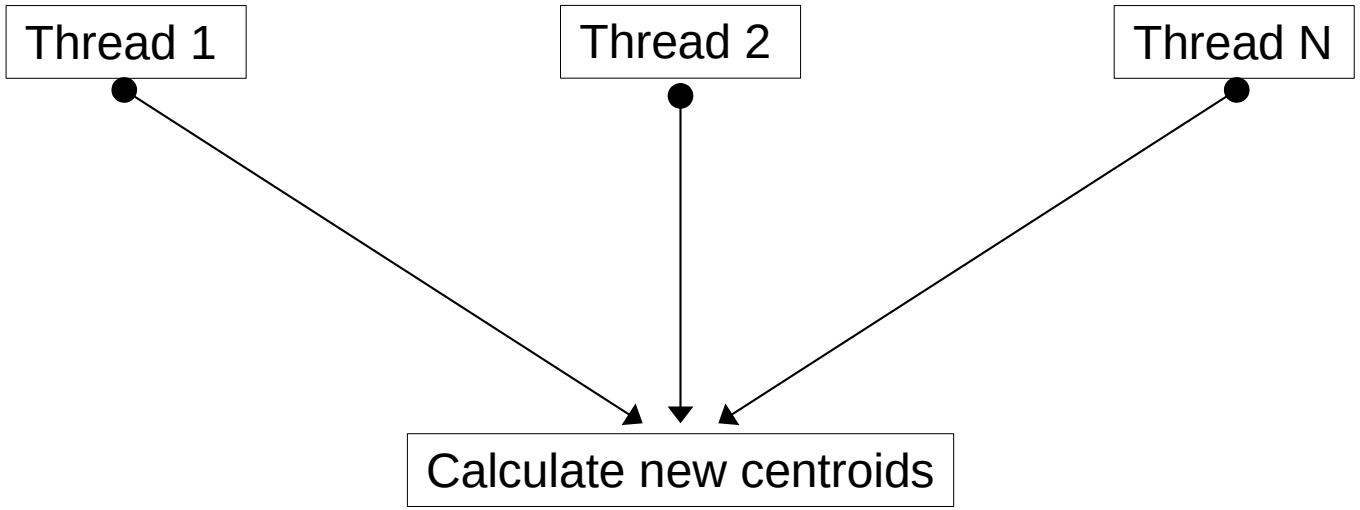
Thread 2
for (int i=datasetSize/N;
i<datasetSize*2/N; i++) {
- assign point[i] to centroid
- **update local partial sum**

}

Thread N
for (int i=datasetSize*(N-1)/N;
i<datasetSize; i++) {
- assign point[i] to centroid
- **update local partial sum**

}

Since a partial sum of points in a cluster is maintained in each thread, a final **reduction** is needed to extract the total sum and be able to compute the mean; this is accomplished with the directive **#pragma omp declare reduction(...)** which allows to specify a reduction function to conglomerate the data of each thread.



CUDA

Each thread is assigned a **single** *observation*, this way, the FOR loop can be removed and so the maximum parallelization possible is achieved.

N = **datasetSize** threads

Thread idx=1 • assign point[idx] to centroid • **atomically update global partial sum**	Thread idx=2 • assign point[idx] to centroid • **atomically update global partial sum**	Thread idx=N • assign point[idx] to centroid • **atomically update global partial sum**

Step 1. is done on the host.
Two kernels are used to complete step 2., 3. and 4.

```
Kernel 1:
   assign points to centroids
   update partial sums

cudaDeviceSynchronize();

Kernel 2:
   use final sums to calculate new centroids
```

Two kernels are necessary because the whole *grid* needs to be synchronized to get final sums.

The memory management is as follows:

• *observations* and *centroids* are copied from host to device global memory

At each kernel execution:
• *centroids* are pulled by threads into shared memory
• threads atomically update sums into global memory

• When final *centroids* are found, structures are copied back to host.

Tests

Hardware:

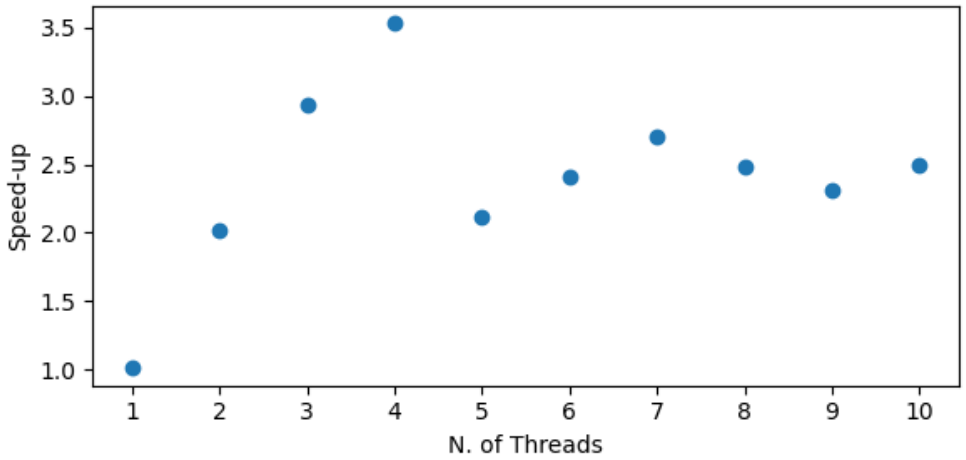
- CPU Intel Core i5-4670 (4-core)
- GPU Nvidia GeForce GTX 770 (compute capability 3.0)

Dataset:

- Number of elements: 100,000
- Points dimension: 2
- Number of centroids: 10 (if not stated otherwise)

OpenMP

The parallel program was run with varying number of threads with the purpose to find the optimum value.

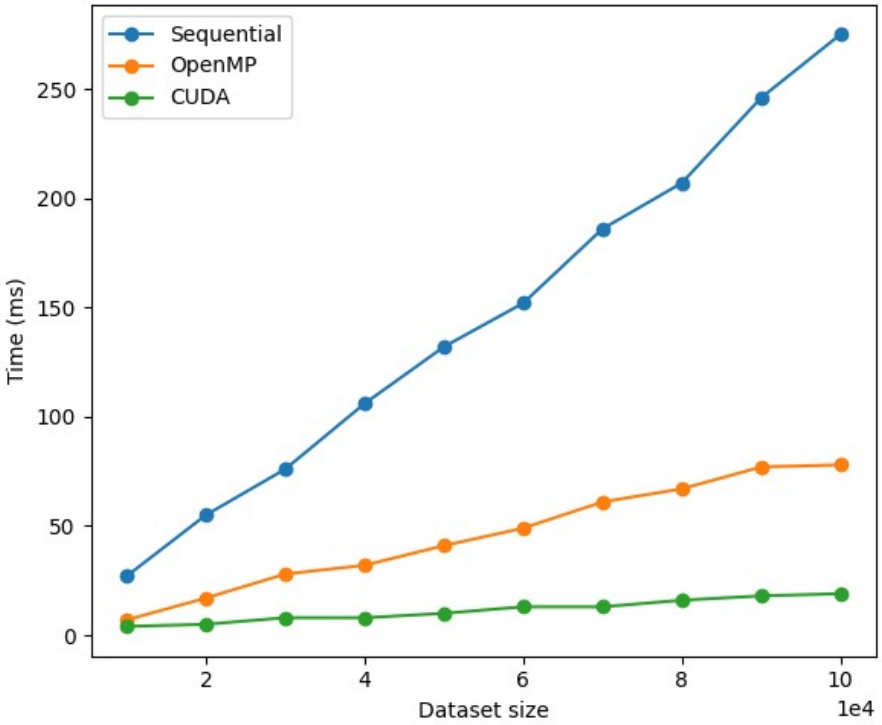


Max speed-up: **3.53** with **4** threads

Around **2.4** with more threads

CUDA

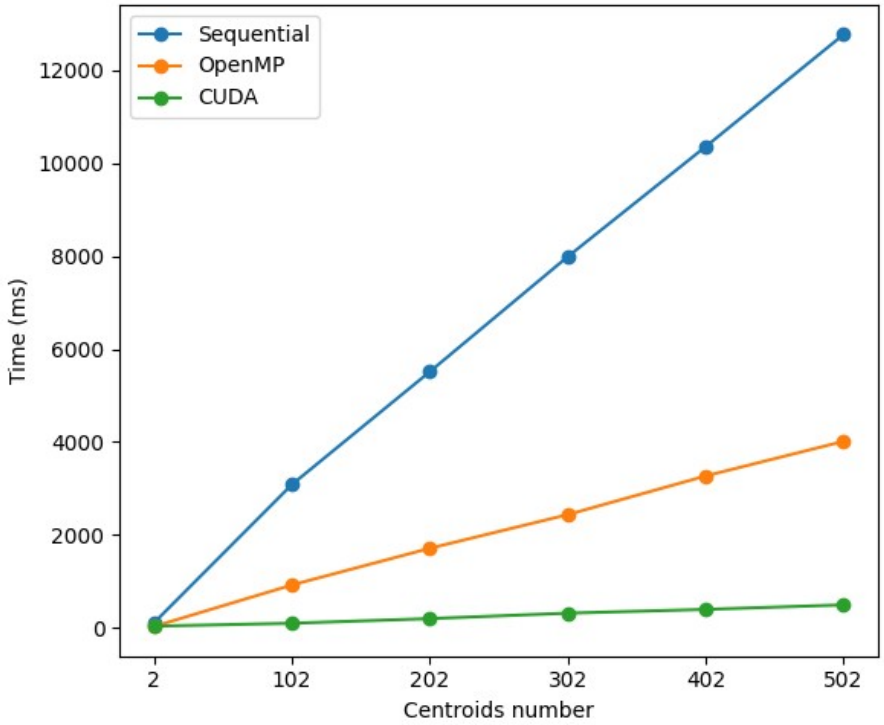
The first test was done increasing the dataset size to appreciate the scaling of the various parallelization methods.



Max speed-up: **14.47**
with 100k elements

Scaling to dataset size:
- Sequential: **linear**
- OpenMP: **linear**
- CUDA: **sub-linear**

Another test was done by increasing the number of centroids.
Since the loop on the centroids is executed inside the thread, the execution time grows linearly for every implementation, but the greater the parallelizzation, the greater the speed-up.



OpenMP Max speed-up: **3.32**

CUDA Max speed-up: **29.13**

Scaling to centroids number:
- Sequential: **linear**
- OpenMP: **linear**
- CUDA: **linear**