Parallel k-means algorithm using CUDA

Ruben Teimas

Universidade de Évora, Évora, Portugal
February 9, 2021
m47753@alunos.uevora.pt

Abstract: K-means is a well known clustering algorithm which consists in finding the best cluster for each point of the data set. This algorithm is generally applied on huge amounts of data at once, therefore a lot of processing power is needed. In this study the algorithm was implemented sequentially in *C++* and in parallel making use of *CUDA*.

Keywords: Parallel Computing · GPGPU · CUDA · Clustering · k-means

1 Introduction and Motivation

This paper is about the advantages of parallel computing to solve certain problems. The problems to take more advantage from parallel computing are the ones that have the most independent parts between each functionality and involve a lot of computation.

Many Machine-Learning algorithms have this properties, thus, they are good and easy problems to parallelize. The one I'm using for the demonstration purpose is the *K-Means* algorithm.

2 State of the Art

With the increasing adoption of Artificial Intelligence and Machine Learning more data is being collected than ever.

Almost all interactions with our computers and smartphones serve as data collection, specially by big companies [1], as web/applications also tend to be very centralized for most common users (*ex*: Facebook, Google, Twitter).

These huge amounts of data are stored, pre-processed and used. However, to process such amount in a fairly small time an enormous processing power is needed.

In the last few years *CPU* single-core performance hasn't seen much improvement, which led to adding more cores (multi-core). To make use of these cores, software needs to be written with parallelism in mind.

There are several projects to make this process easier, such as *OpenMPI*, *OpenMP*, *UPC*, and others. While all of them help to write parallel software there are many differences between them. For instance, the *OpenMP* project is often used when we want to write parallel software while using high-level constructs. *OpenMPI* is used when we want do distribute the work across multiple computers connected to a network. These 2 projects are very commonly used. Another project is *UPC*, which is makes use of the Partitioned Global Address Space (PGAS) model.

With this idea of parallelism in mind other processing units started being explored in the beginning of the century, the Graphical Processing Units (GPU).

In 2006, *Nvidia* launched *CUDA*, the first *GPU* architecture designed for computing. *CUDA* is very good at making calculations and repetitive tasks due to its many-cores, ability to spawn many threads, memory hierarchy and other features related to its own architecture.

Devices that possess dozens, or sometimes hundreds, of cores go by the name of *many-core* devices, and should not be mistaken with *multi-core* devices which usually have between 2 and 10 cores, for instance, *CPUs*.

Some studies were conducted with focus on getting a better performance out of *K-Means* using the some of the technologies described above. Both for the *MPI* version [3] and *CUDA* [4] had the desirable result, with the last one having a *14x* speed-up when compared to the *CPU* version.

3 K-Means

K-Means is a clustering algorithm where we try to group similar data based on their underlying structure into clusters.

More specifically, it aims to partition a given a set of n observations into k clusters, where each observation belongs to the nearest mean (cluster centroid).

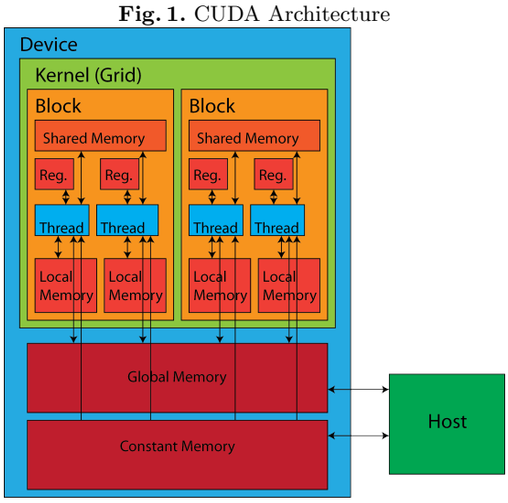
Algorithm 1 K-Means

```
Choose the number of clusters(K) and obtain the data points
Place the centroids c_1, c_2, ..., c_k randomly
for iterarions = 1, 2, ..., I do
    for each point ∈ Observations do
        cluster ← nearestCentroid(point)
        clusters[cluster] ← point
    end for
    for each cluster ∈ 1...k do
        new_centroids ← computeMeans(cluster)
    end for
end for
```

4 CUDA

This architecture is very used in highly parallel jobs as it takes advantage of the *GPU* architecture.

Fig. 1. CUDA Architecture



It has many core concepts that cannot be ignored when using it. The simplest one is the distinction between **Device** and **Host**, which translates to *GPU* and *CPU* respectively.

There is also the concept of a **kernel**, which is a function that runs on the device.

In the device we have **grids** which are groups of **blocks** which are groups of **threads**. Threads within the same block can use **shared-memory**, which is very useful when reusing data.

For a better and deeper understanding of the *CUDA* architecture the *CUDA C/C++ Programming Guide* [2] can be consulted.

5 Proposed Solutions

In order to understand the possible advantages of writing *K-Means* using *CUDA*, 4 different versions of the algorithm were written.

Sequential implementations are quite slow by itself, so to get the maximum out of this version, it was written in *C++* due to its fast execution speed and libraries. This version is not optimized as it just tries to recreate the pseudo-code for the algorithm.

When looking at the algorithm it is easy to understand that points are independent for each other, so the calculation of their distance from the cluster and respective assignment will also be independent, thus making it a good part for parallelization. However, when the adding the point's value to the cluster, we can have race conditions, as there will be many threads trying to access the same variable. If we can solve this problem the last part of the algorithm is quite simple to parallelize, as we just need to put each thread computing the new means of each cluster since they are independent from each other.

Fig. 2. Sequential Architecture

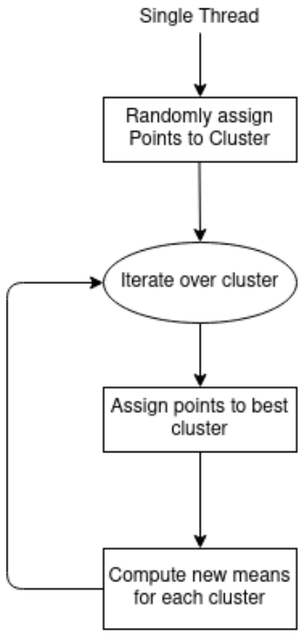
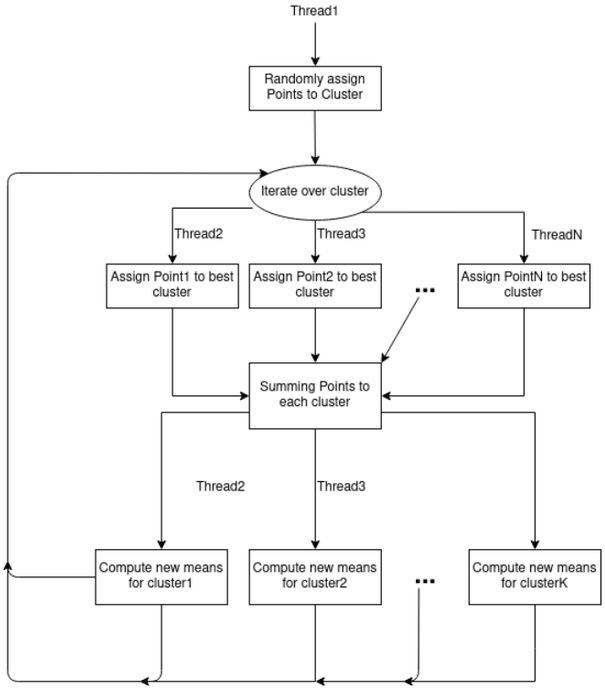


Fig. 3. Parallel Architecture



The first *CUDA* version was written in a naive way, using only basic concepts from the architecture. It explored the existence of threads and blocks and it used *atomic* operations (*atomicAdd*) to sum the point's value to the cluster's new means. These atomic operations prevent the race conditions we wanted to avoid, nonetheless they are tend to be slower than regular operations.

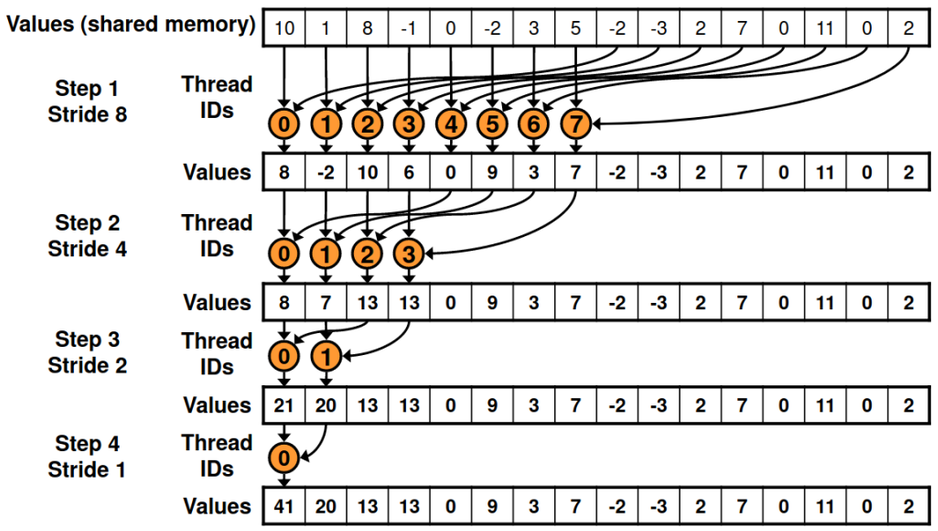
The second *CUDA* version tries to tweak the previous one by using *shared_memory* in one of the kernels. *Shared_memory* allows memory to be shared between threads and, because of that, the access to the memory is faster than to global memory. This type of memory is very useful when we are accessing the same variable multiple times, which we are doing (accessing means), so just by using the properties of the architecture we can get improvement.

Besides that there is no real tweak as we still have essentially locks by using the *atomic* operations.

The last *CUDA* version tries to tweak the previous version even further by getting rid of *atomic* operations. Instead, we approached the *tree reduction* technique which is represented in the *figure 4*.

Shared memory was used in both *kernels*, since we are performing 2 reductions. In the first kernel we reduce each *block* and in the last, with the reduction from each *kernel*, we reduce all the blocks.

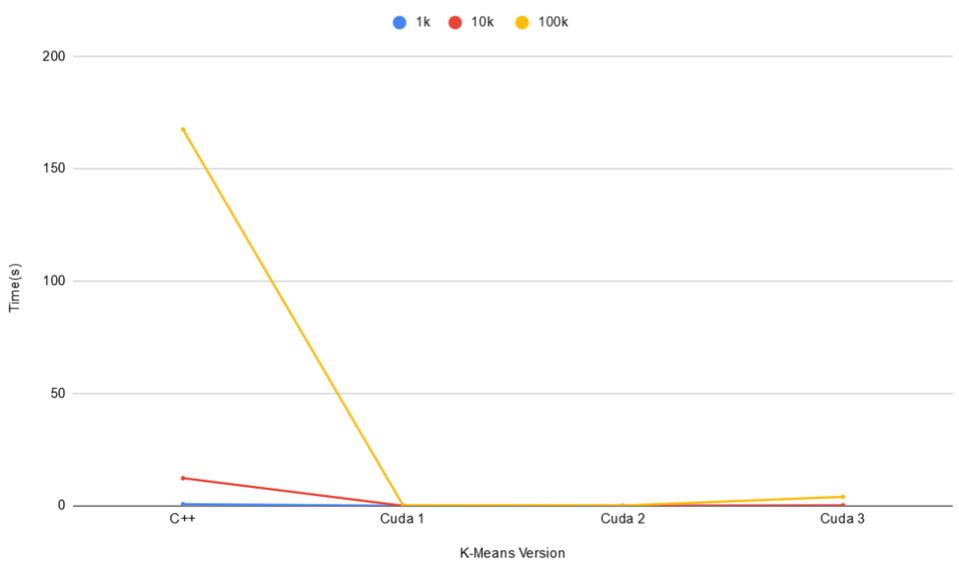
Fig. 4. Tree Reduction



6 Results

Using *g++* to compile the sequential version and *nvcc* to compile the *CUDA* versions the executables were obtained. Said executables were used for the benchmarks bellow. The benchmark was performed in a machine with a *Intel i7-7700HQ* with 8 cores of *3.800GHz*, a *NVIDIA GeForce GTX 1050 Mobile* with 4*GBs* of *VRAM* and 16*GBs* of *RAM*.

Fig. 5. Sequential version and CUDA versions using 1024 threads



The implementations were tested with 3 different sets of data and number of clusters, the first set had **1000 points** and **300 clusters**, the second had **10000 points** and **500 clusters** and the last one had **100000 points** and **600 clusters**.

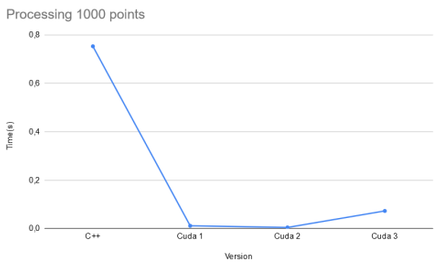


Fig. 6. 1024 threads processing 1000 points.

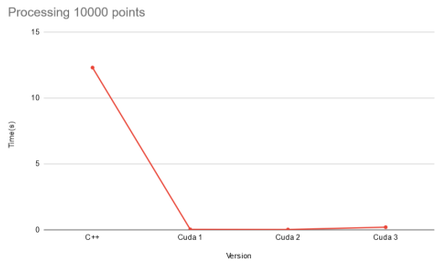


Fig. 7. 1024 threads processing 10000 points.

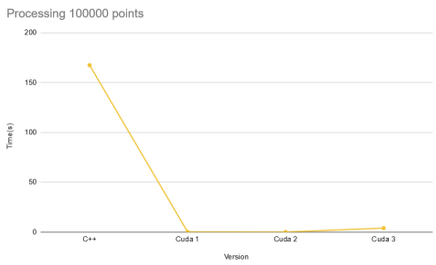
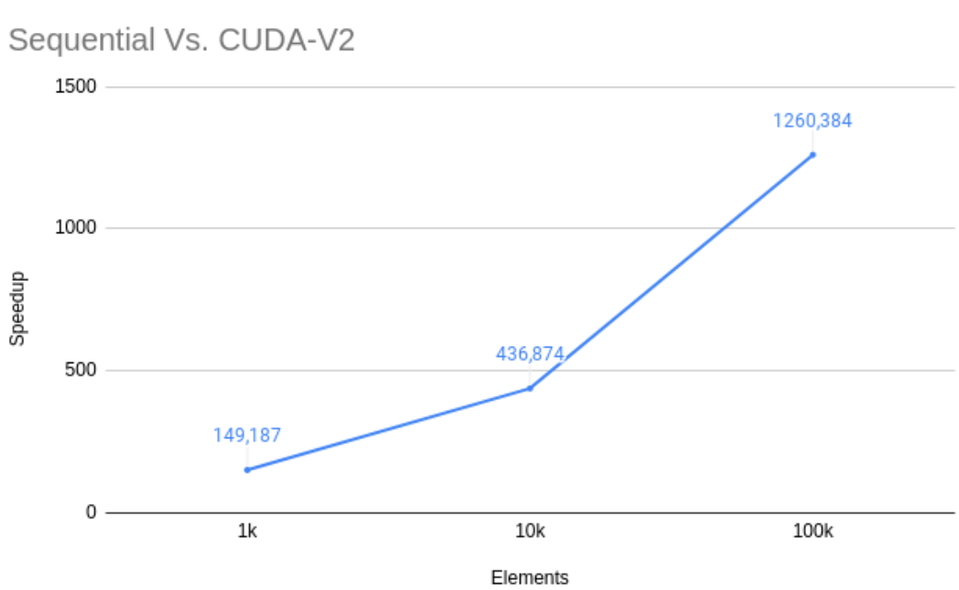


Fig. 8. 1024 threads processing 100000 points.

As it can be seen in the graphics, after benchmarking the results were not fully as expected.

All the *CUDA* versions outperformed the sequential one by a lot, however, the version that was expected to perform better (version 3) was outperformed by the others, with **version 2 having the best result**.

Fig. 9. Speedup ratio for Sequential version Vs. *CUDA*-V2



As the amount of data increases the speedup ratio increases exponentially, with the smallest speedup being 149,187 and the bigger one being 1260,384.

7 Conclusions and future work

The *CUDA version 3* was theoretically the best version given the *CUDA* architecture, however, that was not confirmed by this study.

This unexpected event might be because of a defective implementation but there is no guarantee about it.

In order to improve this study different implementations can be done using deeper concepts of the *CUDA* architecture, including redoing the *version 3*.

Benchmarking each *kernel* in separate in order to detect bottlenecks would also be a great continuation of this project.

Even with this drawback, major speedups were achieved when compared to sequential versions.

References

1. The Data Big Tech Companies Have on You `https://www.security.org/resources/data-tech-companies-have` (Visited on 03/01/2021)
2. CUDA C/C++ Programming Guide `https://docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf` (Visited on 22/12/2020)
3. Zhang, Jing Wu, Gongqing Xuegang, Hu Li, Shiying Hao, Shuilong. (2013). A Parallel Clustering Algorithm with MPI – MKmeans. Journal of Computers. 8. 10.4304/jcp.8.1.10-17.
4. Zechner, Mario Granitzer, Michael. (2009). Accelerating K-Means on the Graphics Processor via CUDA. Proceedings of the 1st International Conference on Intensive Applications and Services, INTENSIVE 2009. 7-15. 10.1109/INTENSIVE.2009.19.