GCA Project - K-means Clustering Report

Maxime Rombaut
Master of Electronics and ICT Engineering Technology

Abstract: In this report, we present our work on developing a CUDA-based application that implements a k-means clustering algorithm. The algorithm is implemented in a few different ways. One using only the CPU, the others use a combination of the CPU and GPU.
Keywords: CUDA, GPU, CPU, C++

1 Introduction

My master's thesis is about the anonymization of data. To anonymize the data a process called generalisation is applied. The generalisation process is based on hierarchies. The k-means algorithm is used to create such hierarchies. The algorithm is a clustering technique that clusters data into k distinct groups. This can take some time because the data that needs to be anonymzed can be large. In this project we will implement the k-means algorithm in a parallel manner to speed up this proces. The data we will use is location data or data points containing a latitude and longitude. To create the hierarchy the k-means algorithm will be ran a few times, each time with a k lower than the previous time. By doing this there are more points assigned to each cluster, increasing the anonymization level. To get representative timing measurements, every implementation will run with the same starting clusters. This also helps with the verification of the results.

2 CPU

The CPU implementation is for bench-marking purposes and isn't optimised. There will most likely be faster implementations of the k-means algorithm. However, the emphasis of this project is on using the GPU and not the CPU.
The implementation starts with reading the points data. Those will be put in a struct containing the latitude and longitude values, the cluster it is assigned to and the minimum distance to a cluster. After this, the clusters will be created. Each one will receive the same latitude and longitude value as one of the points. After this, the k-means algorithm can start.
The algorithm starts by assigning each point to a cluster. It does this by calculating the Euclidean distance between a point and all the clusters. It will be assigned to the cluster to which the distance is the smallest. When all the points are assigned, the locations of the clusters will be changed to the mean of all the points assigned to it. This process happens until the locations of the clusters don't change anymore or the maximum amount of iterations has been reached.

3 GPU

We have made a few different implementations of the GPU version. Each one trying to make the original GPU version a bit faster.

3.1 Original GPU version

In the first GPU implementation we do the assigning of the points to a cluster in parallel. All the points and clusters are copied to the global memory. To assign the points, we call a kernel with 1024 threads per block. The amount of blocks used is equal to $(locs + threads - 1)/threads$, with $locs$ the amount of points and $threads$ the amount of threads per block. So we use a thread per point. In the kernel, each thread calculates the distance of its point to all clusters and assigns the point to the cluster with the smallest distance. After every kernel is done, the locations are copied back to the device. The clusters are not because those haven't changed. After the copying, both the points and clusters on the device are freed.
Once this is done, we can recalculate the positions of the clusters. This happens in the same way as in previous implementation, via the CPU.

3.2 GPU with constant memory

The constant memory is optimised for read operations. In this implementation we wanted to make use of this. The first thing we do is initialise the constant memory so it can contain the required amount of clusters. For this implementation we used a different struct to represent the points because the previous struct used constructors to initialise the points but the constant memory didn't allow this.
In the previous implementation we copied the clusters and points to the global memory. In this implementation we only copy the points to the global memory. We use a $cudaMemcpyToSymbolAsync$ to copy the clusters to the constant memory. The kernel used in this implementation is the same as in the previous implementation, with the exception that we now get the clusters from the constant memory. When the kernel is finished, the points are copied back to the host. The points on the GPU are then freed. Next, the new cluster locations are calculated in the same way as in previous implementations.

3.3 GPU shared memory

The previous implementations calculate the new cluster locations on the CPU. In this implementation we try this on the GPU using shared memory. To do this, we first run the allocation kernel as in the original GPU implementation. When this is done, we synchronise the device and call another kernel. This kernel will calculate the new positions of the clusters. There will be a block per cluster and every block will have 1024 threads.
The first thing that happens in the kernel is the shared memory that is initialised on 0. When this didn't happen, there were some values that would stay in the shared memory causing

the calculations to be wrong. Every block uses its own shared memory so we can use a striding mechanism to deal with the race conditions. All the location values are floats so an atomic operation wasn't possible. Every thread will handle an amount of points equal to the amount of points divided by the amount of threads in a block. It will check if the point is assigned to the cluster of that thread. If it does, the thread will add the latitude and longitude to the shared memory. It will also count the amount of points that is assigned to its cluster and put that value in the shared memory. The problem with this implementation is that it results in thread divergence. Sometimes a thread won't have to do anything because the point it looks at isn't assigned to the cluster while this might not be the case for other threads. We could prevent this by sorting the points based on the cluster it is assigned to but this would probably take more time than this. We would be looping over all points, then we might as well do the calculations there.
The size of the shared memory is three times the amount of threads or 3072. This is because every thread needs to add the latitudes, longitudes and amount of occurrences in the shared memory. The amount of points isn't always a multiple of the amount of threads. This leads to points not being handled. To prevent this we calculate the remaining points. For each remaining point a separate thread will handle it. Once those are handled the threads will synchronise. At this point all the point data is in the shared memory. We use the reduction method to sum all those values. Every iteration the threads synchronise. At the end, all the data will be in the first 3 items of the shared memory. The first is the sum of the latitudes, the second the sum of longitudes and the last the amount of points assigned to this cluster. Thread 0 will divide the sum of latitudes and sum of longitudes by the amount of points assigned to this cluster and compare the values with the original value. If those are equal it will write a 1 to an array. That array will be used outside of the GPU to check if all the clusters have the same value as the previous round. When this is the case, algorithm can quit. This happens for every cluster.

3.4 Other implementations

In this part we will mention some of the implementations we tried but didn't work. These won't be included in the comparison.

Points in constant memory When calculating the new cluster locations in the kernel, all the points have to be read at least K times with K the amount of clusters. To make this faster we wanted to put the points in the constant memory because we can read faster from that than from the global memory. The problem with this is that it can only hold up to 64KB. We can look at how much bytes one point needs: the struct we want to put in the constant memory contains 3 floats and an integer. Each float is 4 bytes and the integer is 4 bytes too. This makes 16 bytes per point. The total size of the constant memory is 65536 bytes. 65536 / 16 = 4096. We can store a maximum of 4096 points in the constant memory. This is too small for the points, but it can work for the clusters. 4096 clusters is a realistic value but in some cases where there are many points this might not be enough. This project is more about the comparison between the CPU and GPU and less about the feasibility of the created hierarchies, so we will just keep it to a maximum of 4096 points.

Shared memory and constant memory We have tried to make a combination of both the constant memory and shared memory. We put the clusters in the constant memory when assigning the points to the clusters. After this, the clusters are copied to the global memory. The calculation of the new locations for the clusters is the same as in 3.3. This implementation didn't give the correct results or kept running so it won't be a part of the comparison in the next section.

Asynchronous The idea in this part was that we could calculate the new cluster positions while the points where assigned to the clusters. This is infeasible because we need the assignment data to calculate the new positions.

4 Comparison

We ran the implementations in 2 different ways. In the first, every implementation runs in sequence for every amount of clusters. This means that for every implementation the hierarchy for the current amount of clusters is calculated. In the next iteration the same will happen but for less clusters and so on. In the second way, every implementation ran for the required amount of clusters and then the next implementation ran. This didn't change the results that much so we won't include those results. We tested the implementations for 50 000 and 100 000 points using different amount of clusters.

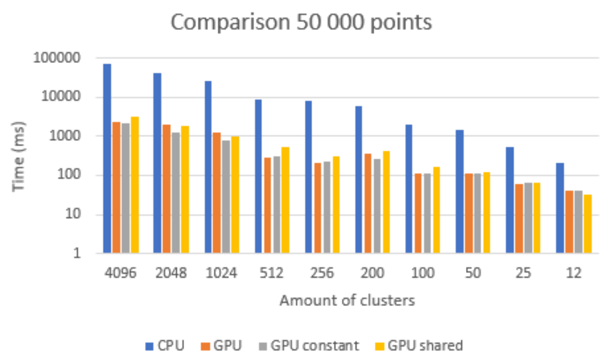

Figure 1: *Comparison 50 000 points*

On figure 1 we can see that the use of the GPU improved the timing results. For the larger amount of clusters, the GPU implementations were about a 100 times faster than the CPU implementation. For the smaller amounts of clusters about 10 times. Between the different GPU implementations there are some differences. The GPU and GPU constant implementations are roughly the same. The implementation with the shared memory is always slower than the other 2 except when the amount of clusters is very small. This could be because of the thread divergence in the kernel implementation that calculates the new cluster locations and because the overall complexity of this kernel is larger than the the complexity of the CPU function that calculates the new

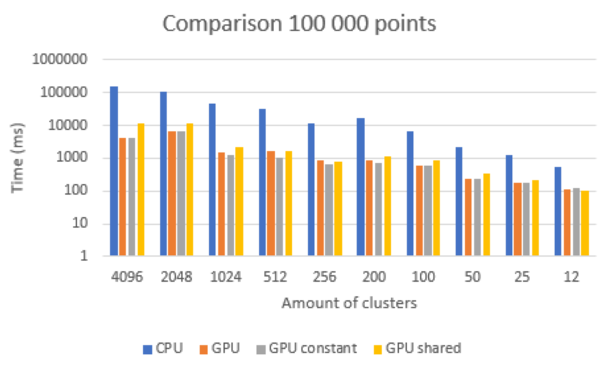

Figure 2: *Comparison 100 000 points*

locations. The results of the shared memory implementation were sometimes not what was expected. There is probably a bug in the code but we haven't found it yet. However, we think that this doesn't affect the timing results so we kept this implementation for the comparison. On figure 2 we can see the timing results when 100 000 points were used. We can draw the same conclusion as before. The results of this test took a bit longer to complete, as was expected.

5 Conclusion

Overall the GPU implementations are faster than the CPU implementation. The allocation of points to clusters in the GPU results in the largest speed gain. In our implementation the parallelisation of the calculating the new cluster locations wasn't faster than the CPU implementation.

6 Github link

`https://github.com/MaximeRombaut1412/GCA_labs.git`