Exploring the capabilities of CUDA on a C-based k-means algorithm

João Silva
Escola de Engenharia
Universidade do Minho
Braga, Portugal
pg50495@alunos.uminho.pt

João Torres
Escola de Engenharia
Universidade do Minho
Braga, Portugal
pg50499@alunos.uminho.pt

***Abstract*—In this report, we investigate the use of CUDA, a parallel computing platform and programming model developed by NVIDIA, to improve the performance of the Lloyd's K-Means algorithm, a popular clustering technique, implemented in C. Our implementation utilizes the GPU's ability to perform large numbers of simple mathematical operations simultaneously, resulting in a significant performance improvement compared to the CPU-based version of the algorithm. We also discuss the trade-offs and considerations involved in using CUDA for this application, as well as potential future work. Overall, our results demonstrate that CUDA can be a powerful tool for accelerating the performance of the Lloyd's K-Means algorithm and similar computationally intensive tasks.**

***Index Terms*—C, K-means, Lloyd, Parallelism, CUDA**

I. Introduction

Clustering is a widely used technique in machine learning and data analysis, with many different algorithms available for grouping similar data points together. One popular algorithm is the Lloyd's K-Means algorithm, which iteratively refines clusters based on the mean of the data points assigned to each cluster. However, as the amount of data increases, the computational complexity of the algorithm can become a bottleneck, making it difficult to process large datasets in a reasonable amount of time.

To address this issue, researchers have turned to using parallel computing architectures, such as Graphics Processing Units (GPUs), to improve the performance of the Lloyd's K-Means algorithm. CUDA, a parallel computing platform and programming model developed by NVIDIA, allows developers to write code that can be executed on a GPU, making it a popular choice for accelerating computationally intensive tasks.

In this report, we explore the use of CUDA to improve the performance of the Lloyd's K-Means algorithm implemented in C. We will discuss the design and implementation of our CUDA-based version of the algorithm, and compare its performance to the CPU-based version.

II. Changes made to the previous algorithm

The overall structure of the program is similar to the previous assignment, with two main sections: Initialization and the k-means algorithm. During initialization, all necessary components are set up, such as allocating space for arrays and filling them with random numbers.

The k-means function is where the majority of the changes were made. Although the basic logic remains the same, the computationally intensive portion of the program was transferred to the GPU using CUDA, allowing for parallel processing.

III. Implementation

A. Structures

The data storage structures used in this assignment are similar to those used in the previous one. The points are still stored in arrays and the cluster information for each point is in a separate array. This means that "$cluster[i]$" represents the cluster to which the point "$(points\_x[i], points\_y[i])$" belongs.

Copies of these arrays are also created for use in the GPU, and are denoted by the prefix "$d\_$" (e.g. "d_points_x" is a copy of the "$points\_x$" array).

B. Initialization

The first step in the initialization process is to allocate the necessary memory for the arrays that will store the information related to the points and clusters. This is a crucial step as it ensures that there is enough space to store all the data that will be used in the algorithm.

To do this, we use the C function $malloc()$ for allocating memory in the host and $cudaMalloc()$ for allocating memory in the GPU.

The second step is to fill the $points$ arrays with random values and select one of these points as the centroid for each cluster. This sets the starting point for the k-means algorithm and allows it to classify the points into different clusters based on their initial positions.

Finally, the resulting arrays are copied to the GPU memory using $cudaMemcpy()$, allowing the GPU to access and use these values for its calculations.

C. K-Means

As in the previous assignment, the K-means algorithm remains divided into three parts:

1) **Resetting the clusters**: Due to the fact that the centroids will change in each iteration, the points associated with the cluster will also change, resulting in the cluster's size not remaining constant. As a result, the cluster's $size$ variable is reset to zero at the beginning of each new iteration. Additionally, the $new\_centroid$ variable is also reset to zero, as it represents the sum total of all points within the cluster and should always begin at zero.

```
for (int i = 0; i < K; i++){
  size[i] = 0;
  newCentroid_x[i] = 0.0;
  newCentroid_y[i] = 0.0;
}
```

2) **Assigning points to each cluster**: This step is the most computationally demanding task. For every point, the distance between each cluster and the point will be calculated and compared to determine the closest cluster to the point. Once this comparison is completed, the point is assigned to the closest cluster. To improve the algorithm's performance, this task is sent to the GPU to be processed in parallel.

```
int blocks= (N + NUM_THREADS_PER_BLOCK
- 1) / NUM_THREADS_PER_BLOCK;

kernel_computeDistances<<<blocks,
NUM_THREADS_PER_BLOCK>>>
(d_centroid_x, d_centroid_y,
d_points_x, d_points_y, d_cluster);
```

3) **Updating the size and centroids**: Finally, the $size$ variable of each cluster is updated based on the results obtained in the second step. The value of each point within a cluster is added to the $newCentroid$ variable, which is then divided by the cluster's size to calculate the new centroid. The cluster's *centroid* variable is then updated with the value obtained from this division. Since the execution is set to always stop after 20 iterations, there is no need to check if the solution has converged.

```
for (int i = 0; i < N; i++) {
    size[cluster[i]]++;
    newCentroid_x[cluster[i]]
        += points_x[i];
    newCentroid_y[cluster[i]]
        += points_y[i];
}
-------------------------------------
for (int i = 0; i < K; i++){
    newCentroid_x[i] /= size[i];
    newCentroid_y[i] /= size[i];
    centroid_x[i] = newCentroid_x[i];
    centroid_y[i] = newCentroid_y[i];
}
```

IV. CUDA PROGRAMMING

We took advantage of the techniques used in the previous assignment that made the code ready for parallelism. Rather than utilizing pragma directives, we moved the most computationally intensive section of the algorithm to the GPU.

The for loop in which the distances to the centroids were calculated was replaced by a global function that takes the copies of the data arrays as parameters.

Once the GPU kernel is finished executing, the data from the device arrays are copied back to the host arrays, allowing for the next step to begin.

It's important to note that during initialization, two variables are passed to the kernel: *NUM_THREADS_PER_BLOCK* and $block$. The value of $block$ is calculated as:

$$block = \frac{(N + NUM\_THREADS\_PER\_BLOCK - 1)}{NUM\_THREADS\_PER\_BLOCK}$$

, while *NUM_THREADS_PER_BLOCK* is assigned a static value of 256.

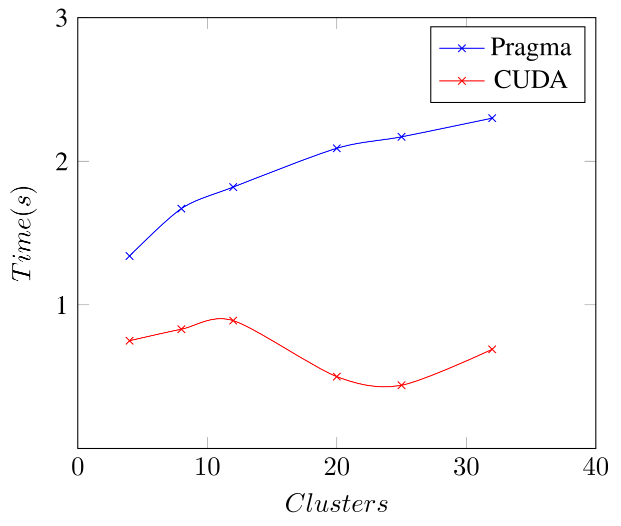
V. RESULTS

TABLE I
COMPARING EXECUTION TIMES BETWEEN PRAGMA VS CUDA

	4 clusters		**32 clusters**	
Pragma vs CUDA	Pragma	CUDA	Pragma	CUDA
Time (s)	0.80	0.75	1.49	0.69

Examining the table, it is evident that while the performance difference is not substantial when utilizing only 4 clusters, there is a notable difference when testing with 32 clusters. This implies that the version of the code that utilizes CUDA has a superior scalability compared to using pragma.

To gain insight into how parallelism with CUDA scales in comparison to pragma, a series of tests were conducted with varying numbers of clusters. The results of these tests can be seen in the following graph:



It is apparent from the data presented in the graph that the CUDA version of the K-Means algorithm exhibits superior scalability when compared to the version using pragma. As the number of clusters increases, the performance of the CUDA version consistently remains better, regardless of the number of clusters used. This suggests that the use of CUDA for parallel processing has a positive impact on the overall performance of the algorithm, making it a more efficient solution even when dealing with large datasets.

It was also crucial to evaluate the impact of a varying number of points on performance. To do this, tests were conducted to measure the execution time when using 100 thousand, 1 million, 10 million, and 100 million points.

TABLE II
EXECUTION TIMES FOR VARIOUS NUMBER OF POINTS

Number of points	100K	1M	10M	100M
Time (s)	0.381521	0.382162	0.383269	0.402916

As can be observed from the table, the CUDA version handles an increase in points effectively. The performance remains relatively consistent, even when the number of points increases from 100 thousand to 100 million. Based on this, it is safe to conclude that the use of CUDA is a preferable option, particularly when applied to very large datasets.

VI. CONCLUSION

In conclusion, this assignment demonstrated the potential of using CUDA to explore parallelism in the popular K-Means algorithm. By utilizing the parallel computing capabilities of the GPU, we were able to significantly improve the performance of the algorithm compared to the standard, CPU-based version. Our implementation also highlighted the trade-offs and considerations involved in using CUDA, such as the need for data transfer between the CPU and GPU and the limitations of memory on the GPU.

Overall, this assignment highlighted the benefits of using CUDA for data-intensive tasks like K-Means and similar algorithms. It also showed that with proper implementation and understanding of the CUDA programming model, we can achieve significant speedups and improved performance. As a future work, we can explore the scalability of the algorithm with the increasing dataset size and compare the performance of our CUDA implementation with other parallelization frameworks like MPI.