HW5

Delibas, Sabirenur
sabirenurd@gmail.com

Ulkutasir, Batuhan
ulkutasir.batuhan@gmail.com

June 2024

1 Introduction

In this report, we will examine the parallelization and performance enhancement of computational algorithms using GPU computing with CUDA, which stands for Compute Unified Device Architecture. Throughout this course, our main focus was parallelization and optimizing Lloyd's k-means clustering algorithm. To achieve this ultimate goal, we have used many different computation methods based on CPU programming. From OpenMP to MPI we have learned multithreading, parallel operations, and distributing tasks across different cores. In these operations, CPU processors handling all the communications and controlling the sequence. On the other hand, Nvidia CUDA provides a more explicit and flexible approach to parallelism, by using the massive parallel processing capabilities of GPUs. Unlike general purpose CPUs, GPUs are specially designed to handle thousands of threads concurrently, by their simpler yet effective chip design. Ultimately increasing performance of parallelized tasks. Therefore, this study will compare performance of CUDA-based parallel algorithm across various data sets and configurations, and demonstrate the benefits. At the end of the report, the advantages of CUDA will be demonstrated, by comparing execution times and overall performance from the algorithms that built in HW2 and HW3.

2 Implementation Details

Host and Device Operations: To explain GPU computing first thing to explain is the difference of host and device separation. In GPU computing basically both of the GPU and CPU handles the operations. The term "host" refers to CPU and its associated memory and its operations. The host is responsible for running the main application code controlling the execution order of "kernel functions" and launching them when needed. It basically operates as the previous coding techniques that examined before.

On the other hand, the "device" refers to the GP, which is specially designed for high-speed parallel computations. Therefore in CUDA, GPU handles the computationally intensive parts of the application. These intensive parts generally launched (or called) by the host and refereed as "Kernel Functions". Separated from CPU, GPU has its own memory space, allocated by "cudaMalloc" commands. Thus it requires explicit data transfers commands as "cudaMemcpy" between the host and device, synchronized when needed.

Kernel Functions: In CUDA, kernel functions are functions that run on the GPU. They are defined using the _global_ qualifier. Each kernel is executed by multiple threads in parallel.

`distanceKernel`: This one calculates the Euclidean distances from each point to every centroid.

`assignPointsKernel`: Here, we assign each point to the nearest centroid based on the distances we calculated earlier.

`updateCentroidsKernel`: This kernel function updates the positions of the centroids based on the points assigned to them.

`normalizeCentroidsKernel`: Lastly, we have the normalizeCentroidsKernel. It takes care of normalizing the centroids by averaging the points assigned to each one.

Thread Hierarchy: CUDA organizes threads into a hierarchy of grids and blocks. Each thread within a block has a unique thread ID. Threads within a block can cooperate by sharing data through shared memory and synchronizing their execution.

`blockIdx.x`, `blockDim.x`, and `threadIdx.x` are used to determine the unique thread ID within the grid. This ID is used to index data and perform parallel computations. We use a 1D grid of 1D blocks for simplicity, where the number of blocks is calculated as

$$\left(\frac{Np + \text{THREADS\_PER\_BLOCK} - 1}{\text{THREADS\_PER\_BLOCK}}\right)$$

Memory Management: CUDA provides various types of memory for different purposes. Global memory is allocated using cudaMalloc and freed using cudaFree. Memory allocation and management are critical for efficient CUDA programming, ensuring that data is available to GPU threads during kernel execution.

Memory Transfer: Data must be transferred between the host (CPU) and device (GPU) before and after kernel execution. cudaMemcpy is used for these transfers as explained before. Although these are essential execution codes over usage might cause delays and non-optimised results.

Synchronization: Synchronization ensures that all threads have completed their execution before the program proceeds. cudaDeviceSynchronize is used to synchronize the host with the device, ensuring that all preceding kernel executions and memory operations are completed.

Execution Time Measurement: We measure the execution time with CUDA events. The start and end time of the execution of the kmeans algorithm is recorded using the `cudaEventRecord` function. The elapsed time is then calculated in by using `cudaEventElapsedTime` in order to be transformed in second to report.

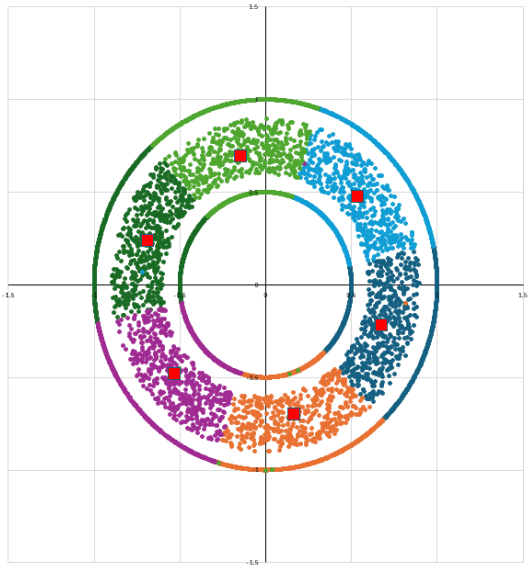
3 Results & Discussion

Elapsed Execution Times: This is the first section we would like to discuss in our code. We have compared the given datasets and their execution times in Table 1. Naturally, an increasing time pattern by increasing the number of data points is evident. A very significant output is that the three-dimensional dataset converged immensely fast. This implies the significance of parallel operations powered by GPU computing. However, the elapsed time for the "Birch" dataset, containing 100,000 data points, was unexpectedly high at 25.6 seconds. This anomaly can be might be caused by our method of parallelization for the distance calculations, where the device computes distances between each pair of data points. At the level of this huge data cluster calculating distances was tedious process. This design selection ensured accurate centroid calculations by trading of from time

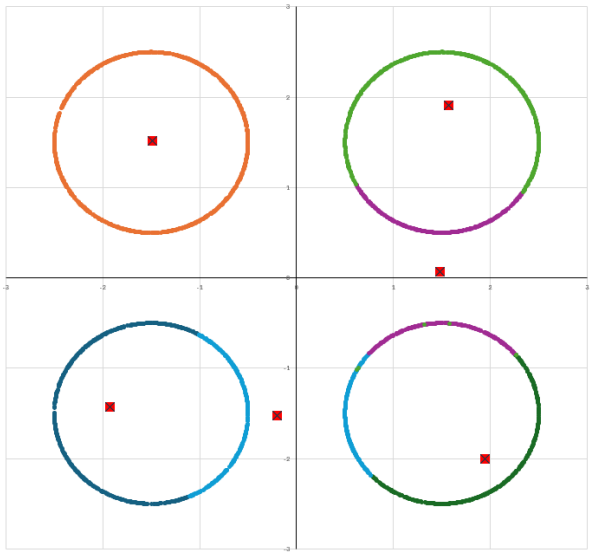
Data Set Name	**Dimension**	**Number of Data Points**	**Elapsed Time (sec)**
Hepta	3D	212	0.003
Smile	2D	1000	0.012
Circle	2D	4000	0.041
Isolation	2D	9000	0.057
Brich	2D	1000000	25.626

Table 1: Elapsed Time of Different Data Sets Compared in Seconds

Overall Logic and Structure Comparison: Secondly, we have explored various parallelization methods for C programming so far. We must admit that implementing GPU computation for Lloyd's K-means algorithm was a challenging task. Additionally, using Google Colab was also another challenge due to limited GPU times. However, when compared to MPI and its general operational syntax like rankings, shared threads, and buffered communications GPU programming becomes easier to comprehend. We believe that GPU employs a similar strategy as creating sub-functions. Thinking the host as the main function and the device operations as its sub-functions simplified our understanding since we found kernel functions easier to manage than shared multi-ranked threading operations. These approaches are also reflected in the number of lines written: the GPU parallelized code consists of 322 lines, while the MPI version has 260 lines, demonstrating that similar levels of parallelization could be achieved with fewer commands in different approaches.



(a) Resulting centroid points of Isolation data set obtained by GPU computation



(b) Resulting centroid points of Circles data set obtained by GPU computation

Figure 1: Accuracy of the centroid locations for data sets named Isolation and Circles

Computation Time Differences with Serial and OpenMP Computation: Finally, this is consistently most important part of this report as we always examined importance of the parallelization in this course. As discussed in the elapsed time discussion we believe we have successfully implemented the GPU computation by delivering accurate centroid results in short time in almost of the data sets. As we did in HW 2 we wanted to demonstrate centroid locations for each clusters. They are given for two set of data in the Figure 1. It is also evident that our code has room for improvements as it operates poorly in the Brich dataset. Therefore it is not included in the computation time comparison as it would be tedious to demonstrate that much difference. In the Figure 2 computation times for different number of data clusters compared in between three computing methods where Serial, OpenMP and GPU.

4 Conclusion

In conclusion, this report demonstrates an algorithm application with GPU computing method which is named NVIDIA CUDA. Unlike previous methods involving CPU-based parallelization with OpenMP and MPI, CUDA utilizes the massive parallel processing power of GPUs to handle computationally intensive

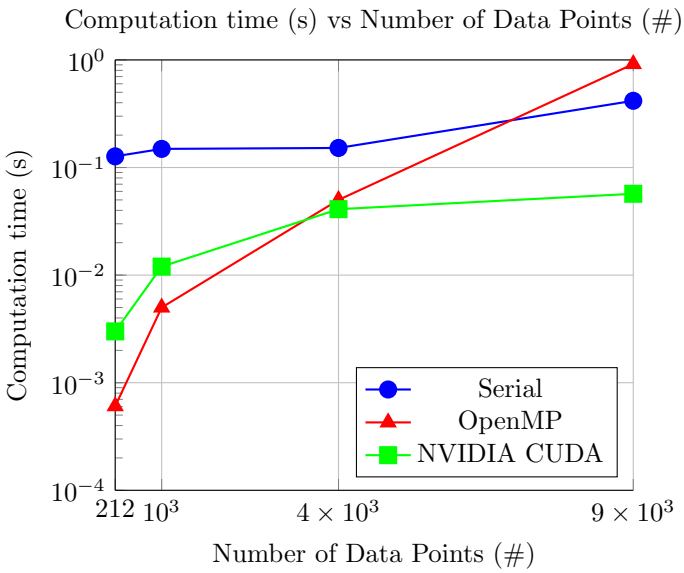


Figure 2: Comparison of computation times (in logarithmic scale) in different data sets with three different computation methods.

tasks more efficiently. The separation between the host (CPU) and device (GPU) ensures a flexible and powerful approach to parallel computation. Although we could not fully optimize parallel operations since our code operates poorly in Brich data set we believe we have fully implemented Lloyd's k-means clustering algorithm using CUDA. If we focus on the Isolation data set (Np=9000) we can observe both the OpenMP and Serial computations were not able to match the speed of the optimized CUDA implementation, especially when it came to dealing with the larger problems. The overall performance and execution time benefits of using NVIDIA CUDA for GPU computing are evident and promising. We believe that we have shown the potential of GPUs in speeding up parallel computation by this homework.