K-means Clustering Acceleration with CUDA (High Performance Computing Architectures)

G03: Afonso Alemão (96135), Rui Daniel (96317), Tiago Lourinho (96327)

***Abstract*—This paper presents a study on the acceleration of the k-means clustering algorithm using a heterogeneous system composed of a CPU and a GPU. The CUDA parallel computing platform and application programming interface model were utilized to achieve the speedup. The results show that for a small number of points, the GPU is not the optimal choice due to the fixed setup overhead, resulting in speedups of less than 1. However, for larger datasets, the GPU outperforms the CPU significantly by exploiting parallel processing, achieving up to 11.7x performance improvement for an input of 3 million points. Our findings demonstrate that the use of a GPU is only suitable for processing a considerably large number of points.**

***Index Terms*—Hardware acceleration, GPU, Parallel processing, C language, Student experiments**

I. Introduction & Architecture Overview

This assignment consists of the acceleration of an implementation of the k-means clustering algorithm, through a heterogeneous system composed of a CPU and a GPU.

For this purpose, we exploit the CUDA application programming interface and data-level parallelism (DLP) to offload computationally intensive (regular) tasks to a GPU. CUDA is a parallel computing platform and application programming interface model created by *Nvidia* to enable general-purpose computing on their own GPUs.

K-means clustering is a method of vector quantization that aims to partition n observations into k clusters in which each observation belongs to the cluster with the nearest mean (cluster centers).

CPUs (Central Processing Units) and GPUs (Graphics Processing Units) are two types of processors with different architectures and purposes. While CPUs are optimized for general-purpose computing, GPUs are designed to accelerate parallel processing tasks.

CPU has large caches, few threads per core, modest memory bandwidth, and relies mainly on caches and prefetching, while GPU has got small caches, many threads, large memory bandwidth, and relies heavily on multi-threading for performance.

CPUs aim to handle a wide range of tasks in a general-purpose computing environment. They have a few high-performance cores, optimized for running single-threaded applications.

On the other hand, GPUs are designed for high-performance parallel computing. They are optimized for executing the same instruction on multiple data elements in parallel: SIMD (Single Instruction, Multiple Data), so they take advantage of both massive DLP and thread-level parallelism.

In the GPU architecture, each core is called an SM (Simultaneous Multiprocessor) and a high-level engine schedules the execution of the grid of blocks of threads to SMs. Multiple blocks of threads may be allocated to the same SM, as long as it has enough hardware resources to support it. In the case there are more blocks than what the SMs can physically support, some blocks may have to await for the execution of another block to finish, before being assigned to an SM. Once a block is assigned to an SM, it remains there until completion. Blocks are dispatched in order: all blocks from one kernel have to be dispatched before starting dispatching blocks from the next kernel if they were launched in the same stream. The architecture focuses on increasing throughput performance rather than latency which is hidden behind groups of threads. On Nvidia GPUs, SIMD execution is performed in warps, i.e, in groups of 32 CUDA threads that share an instruction stream. Typical GPUs do not have access to the CPU's global memory, hence requiring explicit data transfers between CPU and GPU.

GPUs are not very efficient in irregular code structures (a large fraction of the code, but a reduced fraction of the execution time) where there is a large amount of instruction-level parallelism (ILP) but reduced or no DLP, and are very efficient in regular code structures (small number of lines, but representing the majority of the execution time) where there is a large amount of DLP.

So GPUs are efficient when dealing with regular and massively data-level parallel code, i.e, for tasks that require massive amounts of parallel processing which makes them much faster than CPUs in certain types of applications, such as scientific simulations, machine learning, and computer graphics.

II. Methodology

The main computational part of the algorithm implementation is in `kmeans_clustering.c` (more specifically, the `kmeans_clustering` function), therefore a version of the same file was created with a `.cu` extension to make use of the GPU capabilities. Finally, the `Makefile` was also modified to assure a correct compilation whichever the host/device target.

A. Solution Overview

*1) **Memory Management**:* To reduce the time spent on memory transfers there are only 3 moments of copies:

- Copy of symbols and features from host to device (1 time at the start).
- Copy of Δ from device to host to check for convergence (1 time per main loop iteration).
- Copy of final cluster centers and membership from device to host (1 time in the end).

Therefore, every other auxiliary data structure is only allocated inside the device. It should also be noted that every data structure in the device is allocated as a flattened array, to increase efficiency due to coalesced memory access. Also, the constant memory was used to access variables that didn't change during the execution.

2) Function: ***`kmeans_clustering`****:* The overview of the optimized function can be seen in Figure 1.

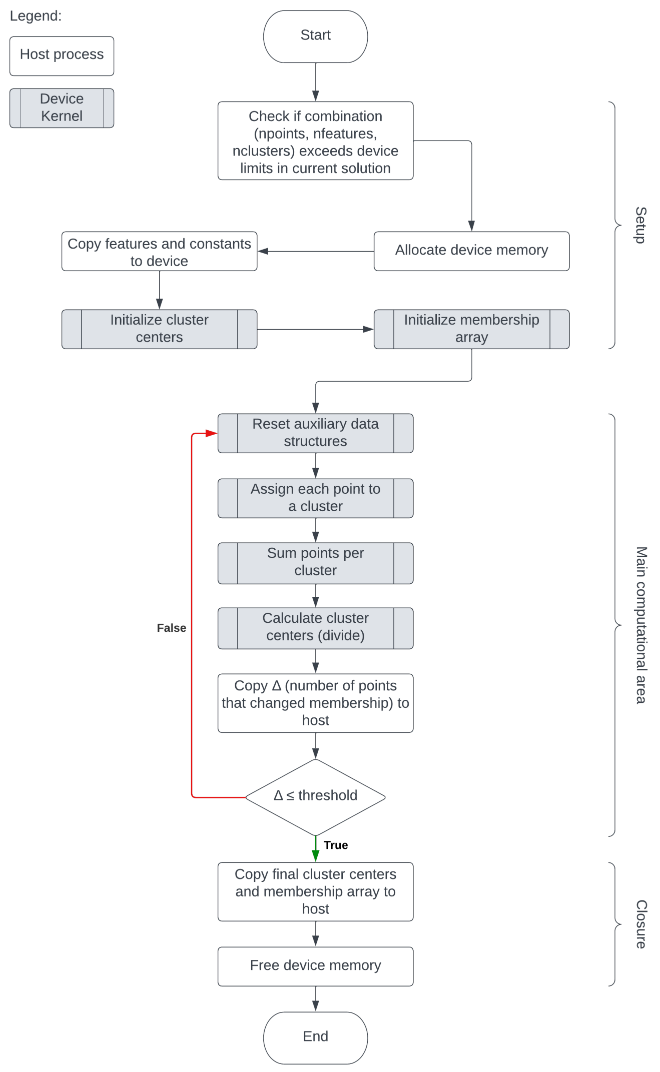
Legend:
Host process
Device Kernel
Start
Check if combination (npoints, nfeatures, nclusters) exceeds device limits in current solution
Allocate device memory
Copy features and constants to device
Initialize cluster centers
Initialize membership array
Setup
Reset auxiliary data structures
Assign each point to a cluster
Sum points per cluster
Calculate cluster centers (divide)
Copy Δ (number of points that changed membership) to host
Δ ≤ threshold
False
True
Main computational area
Copy final cluster centers and membership array to host
Free device memory
Closure
End

Fig. 1. Overview of the implemented solution.

The following sections will elaborate on and explain the kernels seen in the overview in more detail. It should also be noted that on $1D$ kernel launches the `THREADS_PER_BLOCK` found to be optimal in the experiments was 32 (the same as the warp dimension). From now on, for simplicity, `THREADS_PER_BLOCK` will be referred to as TPB, `TOTAL_THREAD_LIMIT` (per block, as defined in the device query output) as TTL, the number of clusters as `nclusters`, the number of features as `nfeatures` and `npoints` as the number of points in the input data.

3) Function: ***`config_check`****:* In order to verify if the input data is suitable for GPU processing we create a function `config_check` that verifies if the combination of the number of clusters, features and points leads to a grid/block distribution that fits in the GPU, according to our solution and implemented kernels.

4) Function: ***`updiv`****:* We use the `updiv` function in order to calculate the ceiling of the division between two integers.

B. Kernel: `init_cluster_centers`

The summary for this kernel is the following:

- Objective: Initialize cluster's centers.
- Launch parameters: Table I.
- Uses shared memory: No.
- Uses atomic operations: No.

TABLE I
GRID AND BLOCK DISTRIBUTION FOR `INIT_CLUSTER_CENTERS`, `RESET_AUX_DATA` AND `DIVIDE_CLUSTERS`

Distribution	X	Y	Z
Grid	1	1	1
Block	`nfeatures`	`nclusters`	1

We implemented a CUDA kernel function, `init_cluster_centers`, that initializes each cluster center to one of the first `nclusters` points, respectively. In order to optimize memory access, we made sure that the threads in a warp are loading different features for the same point. This was done to ensure that the memory accesses are coalesced.

Finally, an if statement was added to ensure that only threads within the specified cluster and feature bounds would execute the code inside the kernel. By launching only 1 block it is assumed that $\texttt{nclusters} \cdot \texttt{nfeatures} \leq TTL$ (1024), as it was sufficient for the tests made (in which $\texttt{nclusters} \cdot \texttt{nfeatures} = 170$). Therefore a further improvement that could be made to support higher dimensions would be to launch the number of blocks dynamically and adjust the kernel implementations accordingly.

C. Kernel: `init_membership`

The summary for this kernel is the following:

- Objective: Initialize the membership array.
- Launch parameters: Table II.
- Uses shared memory: No.
- Uses atomic operations: No.

TABLE II
GRID AND BLOCK DISTRIBUTION FOR `INIT_MEMBERSHIP` AND `ASSIGN_MEMBERSHIP`

Distribution	X
Grid	$\left\lceil \frac{\texttt{npoints}}{TPB} \right\rceil$
Block	TPB

This kernel aims to initialize the membership array used to keep track of the assigned cluster for each data point in the dataset. We launch the number of blocks necessary to execute the operations over all the points. Each thread assigns a value of -1 to an element of the membership array (for a total of `npoints` elements). This operation is necessary for the initialization of the clustering algorithm.

D. *Kernel:* ***`reset_aux_data`***

The summary for this kernel is the following:

- Objective: Reset the auxiliary data needed for the next loop iteration.
- Launch parameters: Table I.
- Uses shared memory: No.
- Uses atomic operations: No.

In order to reset the auxiliary data for the next `do...while` loop iteration, we launch a grid of CUDA thread blocks to perform a kernel function named `reset_aux_data` that runs on GPU. The operation of resetting `d_delta` is executed only by 1 CUDA thread. The `d_new_centers_len` array has `nclusters` elements, so the operation of resetting it is executed by `nclusters` CUDA threads, i.e., by 1 CUDA thread per cluster. The `d_new_centers` array has `nclusters` $\cdot$ `nfeatures` elements, so the operation of resetting it is executed by all launched CUDA threads.

The optimization performed consisted in joining the reset of multiple data structures into a single kernel.

E. *Kernel:* ***`assign_membership`***

The summary for this kernel is the following:

- Objective: Find the index of the nearest cluster center and assign the membership of that point.
- Launch parameters: Table II.
- Uses shared memory: Yes.
- Uses atomic operations: Yes.

The pseudocode for this kernel can be seen in Algorithm 1.

Algorithm 1 Assign Membership

```
Load clusters centers to shared memory
Initialize $\Delta$ in shared memory
Synchronize threads in the same block

for nclusters iterations do
    Compute Euclidean distance squared to center
    Update minimum distance and current closest center
end for
if membership changes then
    Increase $\Delta$ by 1 (atomic add in shared memory)
end if
Assign the membership to the point

Synchronize threads in the same block
Update $\Delta$ in main memory (atomic add in device's global memory of the
$\Delta$ in shared memory, done by only 1 thread per block)
```

For each point, in each iteration, the algorithm needs to find the index of the nearest cluster center (increasing Δ by 1 if membership changes) and assign the membership of that point.

In order to perform that computation, we launch a grid of CUDA thread blocks to perform a kernel function named `assign_membership` that runs on GPU. We launch the number of blocks necessary to execute the operations over all the points.

The kernel is executed by all launched CUDA threads, each one representing a point. We did not parallelize the nested inner loops.

There is a variable Δ in the device's global memory which is rewritten by different threads, so in order to avoid race conditions we need to use atomic additions that allow thread cooperation via shared or global memory. That way concurrent threads do not interfere with each other while accessing the same memory location.

In order to achieve better performance, we optimize the Δ manipulation. We declare a Δ shared variable initialized to 0. Then, in the first stage, each point updates the Δ in the shared memory, using an atomic addition. Finally, 1 CUDA thread per block updates the device's global memory Δ using an atomic addition. With this approach, the number of atomic operations in the device's global memory is reduced, however, we still need to perform atomic operations in shared memory which is faster than in global memory since it avoids the higher latency of accessing global memory.

Furthermore, the updates of Δ both in device's shared and global memory are only performed if the value of Δ is not already above or equal to the threshold. That way, the number of atomic operations is highly reduced.

Another optimization is to cooperatively load `d_clusters` from the device's global memory into a shared array in the shared memory of each block. If the number of threads in each block is enough to load all data, each CUDA thread loads one element, otherwise, each cluster loads `nfeatures` elements.

This way, we prevent each thread from having to load all the `d_clusters` elements from the device's global memory. After a cooperative load of that memory, each thread can now access all the `s_clusters` elements from the shared memory which is faster.

Reducing the total number of threads and making each thread process more than 1 point was also tested in order to make better use of the effort of loading data to shared memory. However, as it didn't show performance improvements, it was discarded.

F. *Kernel:* ***`sum_clusters`***

The summary for this kernel is the following:

- Objective: Sum and count the points located in the same cluster.
- Launch parameters: Table III.
- Uses shared memory: Yes.
- Uses atomic operations: Yes.

The pseudocode for this kernel can be seen in Algorithm 2.

TABLE III
GRID AND BLOCK DISTRIBUTION FOR `SUM_CLUSTERS`

Distribution	X	Y	Z
Grid	$\left\lceil \frac{\texttt{npoints}}{\left(\frac{TTL}{\texttt{nfeatures}}\right)} \right\rceil$	1	1
Block	$\frac{TTL}{\texttt{nfeatures}}$	`nfeatures`	1

Algorithm 2 Sum Clusters

```
Initialize new centers and new centers length in shared memory
Synchronize threads in the same block

Get index of cluster the point belongs to
Update cluster length (atomic add in shared memory, done only by 1 thread per point)
Sum point's features to assigned cluster (atomic add in shared memory)

Synchronize threads in the same block
Store new centers length (atomic add in device's global memory of the values in shared memory, done by only 1 thread per block)
Store new centers (atomic add in device's global memory of the values in shared memory)
```

The purpose of this kernel is to sum the features of each data point belonging to the same cluster and store the result in `d_new_centers`. Additionally, it also counts the number of points belonging to each cluster and stores the result in `d_new_centers_len` (collecting the information necessary to later calculate the center by averaging).

The kernel uses shared memory to optimize memory access and reduce memory latency. The use of shared memory allows threads, in the same block, to share data without going through global memory, which can result in significant performance improvement. In this case, two arrays are allocated in shared memory, `s_new_centers` and `s_new_centers_len`.

`s_new_centers` is a $1D$ float array with size `nclusters` $\cdot$ `nfeatures`. It is used to store the partial sums of the feature values for each cluster center. Each cluster center has `nfeatures` elements, and the partial sums for a particular cluster center are stored in a contiguous block of memory.

`s_new_centers_len` is a $1D$ integer array with size `nclusters`. It is used to store the number of points that are assigned to each cluster center.

Both `s_new_centers` and `s_new_centers_len` are initialized to 0 in shared memory at the beginning of the kernel function. Then, each CUDA thread associated with a point updates them, using atomic addition. Finally, the final sums are then written back to the device's global memory using atomic operations.

That being said, atomic operations are used to ensure that memory access is serialized and to avoid race conditions. In this kernel, atomic operations are used to update the `d_new_centers` and `d_new_centers_len` arrays, ensuring that the results are accurate and consistent. Specifically, the `atomicAdd` function is used to update the shared memory arrays, which are later accumulated into the global memory arrays. This improved program performance because atomic operations are faster in shared memory than in the device's global memory.

Another possible approach in order to avoid the use of atomic operations would be to allocate in the device's global memory a $3D$ array to store the points associated with each cluster and perform several parallel reductions to update `d_new_centers` and `d_new_centers_len`. That being said, this approach was not tested due to time constraints.

G. Kernel: ***`divide_clusters`***

The summary for this kernel is the following:

- Objective: Update cluster centers, by averaging.
- Launch parameters: Table I.
- Uses shared memory: Yes.
- Uses atomic operations: No.

In order to replace old cluster centers with `new_centers` by averaging the position of the points located within a cluster, we launch a grid of CUDA thread blocks to perform a kernel function named `divide_clusters` that runs on GPU. The same operation is executed by all launched CUDA threads: in case there are elements in the associated cluster, it computes the quotient between the sum of the position of the points located within that cluster and the number of these points.

In order to achieve better performance, one thread per cluster cooperatively loads `d_new_centers_len` from the device's global memory into a shared array in the shared memory of each block, as it is accessed multiple times.

This way, only `nclusters` CUDA threads need to load `d_new_centers_len` elements from the global memory, and after syncing with the other threads, all the threads can perform that load through the shared memory.

III. Experimental Results

A. *Obtained speedups*

The solution was tested in the cuda2 machine using the `test_all.py` script that performs multiple runs for each test and system, averaging the time results. In these tests, there are 34 features. We used the default value of finding 5 clusters. The obtained results, with the average of 5 runs and no other groups in the machine at the time, can be seen in Table IV. They include for each test the host time when compiled with flag `-O3`, the GPU time, total error (Manhattan distance between the centers of the clusters calculated by the CPU and GPU), and the speedup.

TABLE IV
OBTAINED RESULTS (IN CUDA2 AND WITH K=5)

Test	Host -O3 Time [ms]	GPU Time [ms]	Total Error (Manhattan distance)	Speedup
kdd_cup	2130.896	1166.944	7.706	1.826
100_34.txt	0.097	259.607	0.000	0.000
100_34f.txt	0.089	255.252	0.000	0.000
300_34.txt	0.524	253.351	0.000	0.002
300_34f.txt	0.438	260.039	0.000	0.002
1000_34.txt	5.890	258.215	0.000	0.023
1000_34f.txt	3.143	256.843	0.000	0.012
3000_34.txt	27.691	264.989	0.000	0.104
3000_34f.txt	21.956	270.285	1.170	0.081
10000_34.txt	330.732	332.773	567.338	0.994
10000_34f.txt	171.431	307.230	0.667	0.558
30000_34.txt	595.480	428.883	0.000	1.388
30000_34f.txt	565.391	452.186	0.797	1.250
100000_34.txt	9545.885	1578.016	149.956	6.049
100000_34f.txt	2539.948	1037.005	0.116	2.449
300000_34.txt	5999.751	1576.364	3.253	3.806
300000_34f.txt	5484.995	1529.647	0.013	3.586
1000000_34.txt	21166.803	2810.187	0.517	7.532
1000000_34f.txt	12367.643	2196.839	0.003	5.630
3000000_34.txt	43618.817	3743.554	0.508	11.652

A statistical evaluation of the results can be seen in Table V. It is possible to observe that a significant speedup was achieved (maximum of 11.7). However, even though the majority of the tests have negligible error (close to 0), some tests have a significant error, explained for example by the floating point operations that by being done in a different order cause different approximations. These approximations can then lead the algorithm to converge to a different center, causing the observed error.

TABLE V
SUMMARY OF OBTAINED ERRORS AND SPEEDUPS

	Min	Max	Average	Standard Deviation
Speedup	0.000	11.652	2.347	3.180
Total Error (Manhattan distance)	0.000	567.338	36.602	129.297

We also report the speedups obtained in function of the number of points, for both integer and floating point input values, in Fig. 2, where the horizontal axis is on the logarithmic scale due to the difference in scale in the number of points of the different input data. As expected, the speedup increases with the number of points. There is a visible outlier in the integer part with 10^5 points as the approximations led to another convergence center, thus executing fewer iterations of the algorithm.

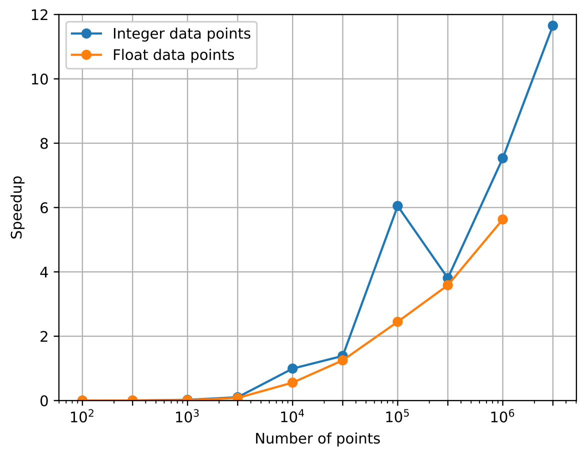

Fig. 2. Obtained speedups as a function of the number of points.

The speedup increases with the number of points as expected, because with the increase of the number of parallel tasks the GPUs take advantage of both massive DLP and thread-level parallelism.

For a small number of points, the speedups are less than 1, and even close to 0. For these cases, it is not worth it to use the GPU to parallelize the program. As seen in Table IV, the time in the GPU to process small amounts of data is always around 250ms, so it can be interpreted by the time to set up the application to run on the GPU. In this case, given the small amount of data, this overhead isn't compensated later.

When we run the program using the GPU, we don't obtain the maximum possible performance achievable ideally. One of the reasons for that is due to Amdahl's Law which states that dependencies limit maximum speedup due to parallelism, i.e, the serial fraction of a program limits the maximum speedup achievable in parallelization.

The maximum achievable speedup by this GPU and our solution depends on several factors, including the number of parallelizable tasks in the code, load balancing, thread divergence, and the balance between computation and communication which includes synchronization and explicit data transfers between CPU and GPU.

Another overhead factor is the efficiency of memory access. We use the web to track down the memory bandwidth of an Nvidia GTX 1070 GPU and we obtained 256 GB/sec (fast). However, the real bandwidth with CUDA is much smaller than this value.

This happens because it takes time for the application (via graphics driver) to provide GPU a single kernel program binary and tell GPU to run the kernel in an SPMD fashion (run N CUDA threads), and also to map thread blocks to cores using a dynamic scheduling policy that respects resource requirements with thread block scheduler. Furthermore, in the event of a certain amount of thread blocks fit on a core, the next block may not fit due to insufficient shared storage and will have to wait for one of the currently executing thread blocks to finish. In the first iteration of CUDA implementation, the time taken to run the kernel is slower because the GPU was powered down to conserve energy and it takes time to power up and stabilize. In the subsequent iterations, this doesn't happen because the GPU is already powered up and stabilized from the previous iterations.

Furthermore, accessing memory can be slow due to memory latency. If the code has many memory accesses or if the memory accesses do not coalesce, the GPU may spend a significant amount of time waiting for data, reducing performance.

There are examples of factors that we took into consideration while designing and developing our solution for the proposed problem.

B. *Nvidia Profiler*

Even though the results shown were obtained in the cuda2 machine, the majority of the development and testing was done in another machine, as this made the testing easier and allowed the use of the Nvidia profiler to identify bottlenecks (in the cudas machine it isn't possible due to permissions).

The system used in the development has the following specifications:

- **CPU:** Intel Core i5-7600K 3.8 GHz Quad-Core Processor;
- **GPU:** MSI GeForce GTX 1060 6GB 6 GB Video Card [4];
- **Memory:** G.Skill Ripjaws V 16 GB (2 x 8 GB) DDR4-2400 CL15 Memory;
- **Motherboard**: MSI Z270 PC MATE ATX LGA1151 Motherboard;
- **Storage:** Samsung 850 Evo 250 GB 2.5" Solid State Drive;
- **Storage:** Samsung 870 Evo 500 GB 2.5" Solid State Drive;
- **Power Supply:** SeaSonic M12II 520 W 80+ Bronze Certified ATX Power Supply;
- **CPU Cooler:** Nox Hummer H-312;

- **Case:** NOX Hummer ZX ATX Mid Tower Case.

An example of the profiler output can be seen in Table VI, where it is possible to observe that most of the time is spent running the heavy computational parts as intended (64.59% and 34.03% for `assign_membership` and `sum_clusters` kernels, respectively). However, in earlier versions of the implementation, this was not the case, as the program spent the majority of the time on memory copies, therefore the profiler was helpful in identifying and correcting this bottleneck.

Overall, the profiling results suggest that the program is primarily compute-bound (due to the need for atomic operations) and that the time spent on memory transfers is relatively small.

TABLE VI
PROFILER RESULTS FOR INPUT DATA `3000000_34.TXT`

Type	Time(%)	Time	Calls	Avg	Name
GPU activities	64.59%	3.30239s	108	30.578ms	assign_membership
	34.03%	1.73987s	108	16.110ms	sum_clusters
	1.27%	64.681ms	5	12.936ms	[CUDA memcpy HtoD]
	0.10%	5.0201ms	114	44.036μs	[CUDA memcpy DtoH]
	0.01%	276.83μs	1	276.83μs	init_membership
	0.00%	254.26μs	108	2.3540μs	divide_clusters
	0.00%	178.87μs	108	1.6560μs	reset_aux_data
	0.00%	2.9130μs	1	2.9130μs	init_cluster_centers
API calls	96.03%	5.04328s	108	46.697ms	cudaMemcpy
	2.52%	132.51ms	6	22.085ms	cudaMalloc
	1.23%	64.684ms	1	64.684ms	cudaMemcpy2DAsync
	0.11%	5.6089ms	6	934.81μs	cudaMemcpyAsync
	0.07%	3.9220ms	6	653.67μs	cudaFree
	0.02%	1.2683ms	434	2.9220μs	cudaLaunchKernel
	0.01%	291.15μs	1	291.15μs	cuDeviceGetPCIBusId
	0.00%	120.85μs	2	60.423μs	cudaDeviceSynchronize
	0.00%	102.65μs	101	1.0160μs	cuDeviceGetAttribute
	0.00%	50.099μs	434	115ns	cudaGetLastError
	0.00%	17.338μs	1	17.338μs	cuDeviceGetName
	0.00%	12.445μs	4	3.1110μs	cudaMemcpyToSymbolAsync
	0.00%	1.0240μs	3	341ns	cuDeviceGetCount
	0.00%	578ns	2	289ns	cuDeviceGet
	0.00%	303ns	1	303ns	cuDeviceTotalMem
	0.00%	248ns	1	248ns	cuDeviceGetUuid

C. Tests in other systems

We also tested our solution in the system described in section III-B, and ran successfully all the tests, even though the speedups were slightly different, as expected due to the change of specifications.

However, in the cuda1 machine, all tests run successfully except `3000000_34.txt`, as it gives a device error related to the launch of the kernel `init_membership`. The strange part was that the invocation was with 93750 blocks each one with 32 CUDA threads, which is well below the defined limits of this machine:

- Maximum number of threads per block: 1024;
- Max dimension size of a thread block (x, y, z): (1024, 1024, 64);
- Max dimension size of a grid size (x, y, z): (2147483647, 65535, 65535).

Given that it was only 1 test that didn't run successfully in cuda1 and given that all the tests ran successfully in cuda2 (a machine with the same specifications as cuda1) and in the system described in section III-B, this error was not further investigated due to time constraints.

IV. CONCLUSIONS

We successfully accelerated the initial implementation of the k-means clustering algorithm, by making use of a heterogeneous system composed of a CPU and a GPU, exploiting the CUDA parallel computing platform and application programming interface model.

The main conclusion taken was that for a small number of points, the speedups are less than 1, and even close to 0 due to the fixed setup overhead of using the GPU, so the best choice would be to use only the CPU. On the other hand, with the increase in the number of points, the GPU greatly outperforms the CPU by exploiting the parallel processing tasks, achieving for example 11.7x performance for the `3000000_34.txt` input data. Therefore, and as expected, the use of a GPU is only suitable for a considerably large number of points to process.

REFERENCES

[1] *An easy introduction to CUDA C and C++*. Aug. 2022. URL: https : / / developer . nvidia . com / blog / easy - introduction-cuda-c-and-c/.

[2] Professor Aleksandar Ilic. *Parallel and Heterogeneous Computing Systems Slides - Slides of Theoretical Classes 2022/2023, 1st Semester (MEEC)*.

[3] Inc. IT Creations. URL: https://www.itcreations.com/nvidia-gpu/nvidia-geforce-gtx-1070-gpu.

[4] *MSI GeForce GTX 1060 Gaming X 6G Graphics Card*. URL: https : / / www . bhphotovideo . com / c / product / 1264797 - REG / msi_geforce_gtx_1060_gaming . html / specs.

[5] Professor Pedro Tomás. *High Performance Computing Architectures Slides - Slides of Theoretical Classes 2022/2023, 2nd Semester (MEEC)*.