K-Means Algorithm with CUDA

Analysis and solutions for dividing an algorithm's workload with the objective of reducing its execution time.

1st Vicente Moreira
Escola de Engenharia
Universidade do Minho
Braga, Portugal
pg50799@alunos.uminho.pt

2nd Joana Alves
Escola de Engenharia
Universidade do Minho
Braga, Portugal
pg50457@alunos.uminho.pt

***Abstract*—This is the final document in the evolution of the basic clustering algorithm K-means. Here, further possible optimizations, such as the use of a *MPI*, or *CUDA*, will be analyzed along with it's implementations and testing. The results will also be compared to the last algorithm iteration developed in the previous document to achieve a final evaluation of the overall project.**

***Index Terms*—K-Means, MPI, CUDA, Blocks, Threads**

I. Introduction

A. Context

In the last part of this project, our team developed a solution where the heavy workload of the k-means algorithm was split between threads among a CPU, reducing it's overall execution time. To further improve this program, our team aims to use more efficient parallelization techniques, either by increasing the amount of threads available, without reducing the overhead, or splitting the algorithm tasks between processes.

For the latter the use of a *MPI* (Message Passing Interface) is essential as it allows the communication and transfer of data between processes. With the use of many independent processes, where which one is assigned a specific task inside the algorithm, a *pipeline* can be achieved inside an application, reducing it's execution time.

As for the execution of many threads, it's well documented that GPU's are optimized and developed around the computing of large unsegmented blocks of data, having an extreme affinity to parallelization, where a larger number of threads is used to process these blocks of data at the same time with little to no overheads. To access these benefits, an API such as *CUDA* (Compute Unified Device Architecture) will help our team develop a better algorithm.

II. Tests and Metrics To Be Evaluated

In this part of the project, our team will evaluate the algorithm performance by two main metrics: execution time, and Level 1 Cache Misses, since these are well linked with the overall increase in the performance of an application.

To test the algorithm and its new versions, we will use a new number of Points and Clusters as a baseline between all tests, executing the algorithm using 1 Million Points and 320 Clusters. This change happened due to a restriction present on the use of the CUDA version, that will be explained in its corresponding chapter. We will also measure the application performance for up to 1024 Clusters, to evaluate the solution's scalability.

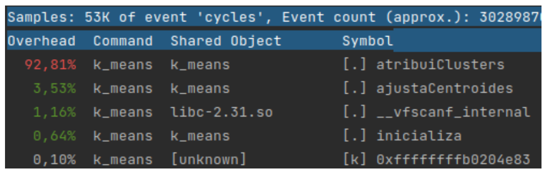
The tests will be performed using two machines: the SEARCH Cluster, provided by Uminho and used throughout the previous parts of this project and a personal Laptop with the following specs:

Laptop ASUS	
OS	Ubuntu LTS 20.04
CPU	Intel® Core™ i5-8300H CPU @ 2.30GHz × 8
RAM	8GB
GPU	Nvidia GeForce GTX 1050 Mobile

For each set of conditions, our team will execute 3 tests and calculate the average of the metrics, to further reduce variances inside the same machine.

III. Initial analysis

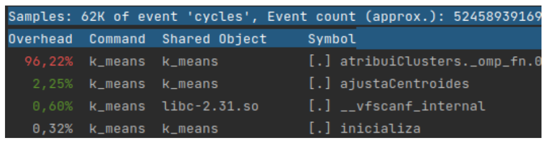
A. Computationally Heavy Functions



```
Samples: 53K of event 'cycles', Event count (approx.): 3028987
Overhead  Command  Shared Object      Symbol
  92,81%  k_means  k_means            [.] atribuiClusters
   3,53%  k_means  k_means            [.] ajustaCentroides
   1,16%  k_means  libc-2.31.so       [.] __vfscanf_internal
   0,64%  k_means  k_means            [.] inicializa
   0,10%  k_means  [unknown]          [k] 0xffffffffb0204e83
```

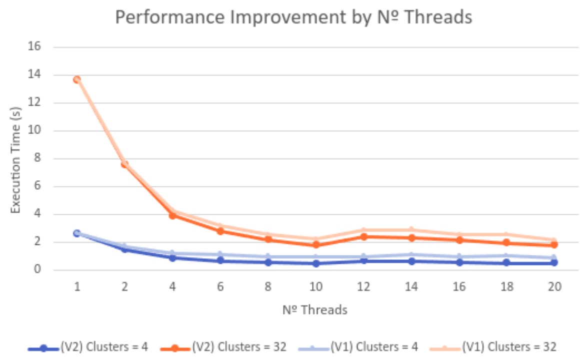
By analyzing the computationally heavy functions of our previous sequential version, we know that **atribuiClusters** is the main function of this algorithm since it always has a N*K calculations per iteration.

By paralelizing this function, using OMP (Open Multi-Processing) we successfully reduced the algorithm execution time but it still remained our applications heaviest function, implying that the only way to improve its execution time in a major way would come from a better parallelization method.



```
Samples: 62K of event 'cycles', Event count (approx.): 52458939169
Overhead  Command  Shared Object      Symbol
  96,22%  k_means  k_means            [.] atribuiClusters._omp_fn.0
   2,25%  k_means  k_means            [.] ajustaCentroides
   0,60%  k_means  libc-2.31.so       [.] __vfscanf_internal
   0,32%  k_means  k_means            [.] inicializa
```

This is further proven by our previous work, where we also implemented parallelization on the function **ajustaCentroides**, and achieved a smaller performance gain.



B. *MPI Struggles*

With this analysis in mind, we decided not to develop any version of our algorithm that utilized MPI, since its main strong suit is dividing an algorithm tasks between processes and, by processing the data through ”blocks”, it allows the creation of an efficient pipeline. In this case, our pipeline would be severely botttlenecked by the function **atribuiCentroids**, resulting in a large amount of wasted time by other processes, and achieving a similar execution time to its sequential version.

Another consideration was utilizing MPI to divide this function workload, and although this is possible, our team concluded that this implementation of MPI would be no different to our parallelized OMP version and, by possibly using larger overheads between processes, our execution time would either match the OMP version or increase it.

C. *CUDA*

Since our analysis indicates that the only way to significantly increase our performance gain in our algorithm is by utilizing a more efficient parallelization strategy/method, we turned our focus on the use of CUDA, an API capaple of using the GPU’s highly optimized parallelization data processing.

One restriction that comes with the use of this API, is the use and definition of the number of blocks and number of threads per block that CUDA will use for the execution of a given function. These have a max value of 1024, which means we have a maximum number of 1.048.576 threads. This value later restricted our testing by only allowing our team to use up to 1M points in the algorithm.

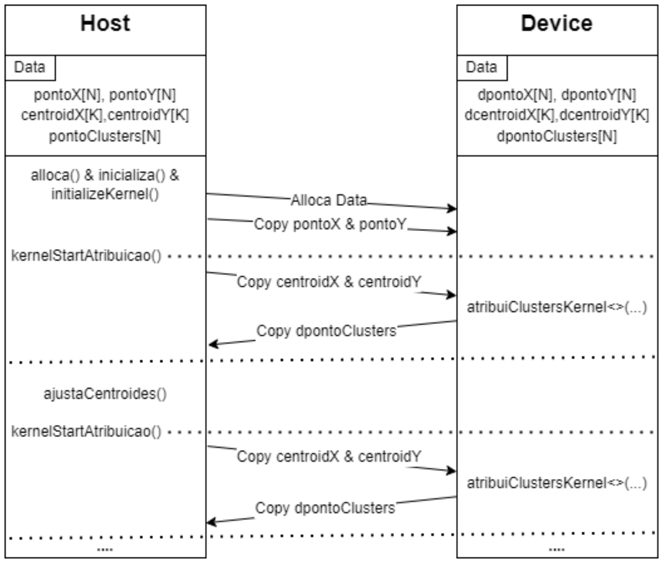
IV. First Version Development

A. *Strategy*

For our first version, we started by applying CUDA only on the **atribuiClusters** function. For this we needed to start by allocating all data arrays related to the points, centroids and cluster attribution on the device since they would all be needed for this phase. We also use this initialization process to copy the points data to the device, since this data is never changed in-between iterations.

After that, for each time the cluster attribution phase is called, we update the centroids data to the device, since the coordinates may have changed and, after executing the kernel function, we copy the results back to the host.

B. *Architecture*



C. *Results*

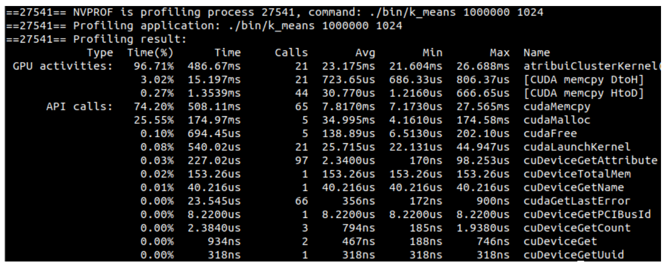
CudaV1		
Test	**TExec (s)**	**L1 Cache Misses**
N=1M, K=320, @SEARCH	1,57	1,19E+07
N=1M, K=1024, @SEARCH	1,78	2,29E+07
N=1M, K=1024, @Laptop	0,81	4,42E+07

(Further result analysis and comparison between versions will be discussed in the corresponding chapter)

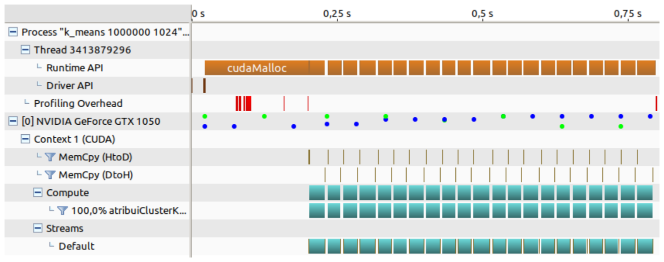
D. *Profiling*

With the use of **nvprof** (Nvidia Profiler) and **nvvp** (Nvidia Visual Profiler), we analysed our profiling metrics to see how the API calls to CUDA were handled and how much overhead might have been introduced by this method.

Due to the missing GUI in the SEARCH Cluster, needed for the use of **nvvp**, we will be showing the profilings from the Laptop test:



```
==27541== NVPROF is profiling process 27541, command: ./bin/k_means 1000000 1024
==27541== Profiling application: ./bin/k_means 1000000 1024
==27541== Profiling result:
            Type  Time(%)      Time     Calls       Avg       Min       Max  Name
 GPU activities:   96.71%  486.67ms        21  23.175ms  21.604ms  26.688ms  atribuiClusterKernel
                    3.02%  15.197ms        21  723.65us  686.33us  806.37us  [CUDA memcpy DtoH]
                    0.27%  1.3539ms        44  30.770us  1.2160us  666.65us  [CUDA memcpy HtoD]
      API calls:   74.20%  508.11ms        65  7.8170ms  7.1730us  27.565ms  cudaMemcpy
                   25.55%  174.97ms         5  34.995ms  4.1610us  174.58ms  cudaMalloc
                    0.10%  694.45us         5  138.89us  6.5130us  202.10us  cudaFree
                    0.08%  540.02us        21  25.715us  22.131us  44.947us  cudaLaunchKernel
                    0.03%  227.02us        97  2.3400us     170ns  98.253us  cuDeviceGetAttribute
                    0.02%  153.26us         1  153.26us  153.26us  153.26us  cuDeviceTotalMem
                    0.01%  40.216us         1  40.216us  40.216us  40.216us  cuDeviceGetName
                    0.00%  23.545us        66     356ns     172ns     900ns  cudaGetLastError
                    0.00%  8.2200us         1  8.2200us  8.2200us  8.2200us  cuDeviceGetPCIBusId
                    0.00%  2.3840us         3     794ns     185ns  1.9380us  cuDeviceGetCount
                    0.00%     934ns         2     467ns     188ns     746ns  cuDeviceGet
                    0.00%     318ns         1     318ns     318ns     318ns  cuDeviceGetUuid
```



(Due to the way the profilers work, both profilings correspond to different executions so their data may not match)

From this analysis we can spot a few relevant points:

- The **cudaMalloc** API call has a small but noticeable overhead, taking around 177ms to process.
- We can see there is 21 calls to the function **atribuiClustersKernel**, which matches the first execution plus the 20 pre-defined iterations. These in total take around 500ms to execute, averaging 24ms per call of this function.
- The two data copies combined take around 16ms to execute. 750us per iteration.
- By using the RuntimeAPI time in **nvvp** (800ms) we can subtract the **cudaMalloc** and **atribuiClustersKernel** execution time and obtain the overhead/downtime that our application has due to its **ajustaCentroids** function executing on the Host.
  800ms - 500ms - 177ms = **123ms of CUDA downtime.**
  123ms - 16ms = **107ms of Host processing Time.**
- We can confirm (via **nvvp**) that before the kernel execution, there is a 4096kB data copy (HtoD) that occurs twice, corresponding to the centroid data.
  (K * sizeof(float): 1024*4 = 4096kB)
- We can confirm (via **nvvp**) that after the kernel execution, there is 4MB data copy (DtoH), corresponding to the cluster attribution data.
  (N * sizeof(int): 1.000.000*4 = 4MB)

V. Second Version Development

A. Strategy

For the next iteration, we aim to reduce the need to transfer data between host and device in every iteration. This data transfer is needed since the host needs the last cluster attribution to execute the **ajustaCentroids** function and, after adjusting the centroids, this data needs to be transfered to the device again to execute the next attribution.

To avoid this, and to increase the applications parallelization, we want to execute the **ajustaCentroids** function utilizing the CUDA capabilities. In this function, we use three auxiliary arrays with K size, two to measure the sum of the X and Y coordinates and 1 to keep track of how many points the cluster has. To adapt this function into it's CUDA implementation we will need these 3 arrays on memory, increasing the data usage on the Device. (12 * K bytes)

One of the challenges introduced in the CUDA implementation is the number of Threads to use in this section. If we use K number of Threads, all of them we will have to iterate over every point, which is slower and undesired. If we use N number of Threads, we need to utilize a synchronizing method, so we can "reset" all the auxiliary arrays, before starting the sums and, before calculating the new centroid coordinate, we have to wait for all the threads to finish the sum process. This implementation will also need some data race control process, since some threads might write to the same array location.

B. Synchronization

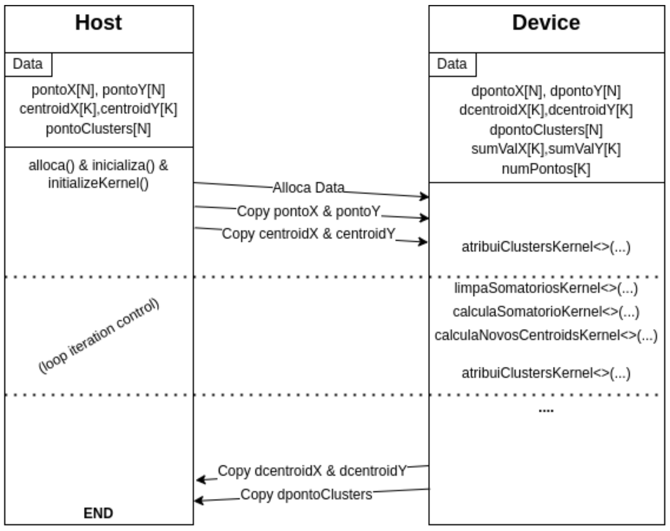
Our team planned to use the CUDA call '__syncthreads()' to handle the syncronization process but after some research on the documentation provided, we found out that this function only synchronizes threads within a block, which doesn't work for what we intend, since we need every thread to wait for the "resetting" of the auxiliary arrays (this problem also happens with the final centroid coordinate calculation phase).

The solution we found is commonly named "CPU synchronization", where we divide the **ajustaCentroid** function in many smaller Kernel functions. (**limpaSomatoriosKernel**, **calculaSomatorioKernel**, **calculaNovosCentroidsKernel**). Since the Host implicitly waits for every thread to finish before executing the next Kernel function, this provides the synchronization method we need. It also provides a way to utilize less Threads in the smaller functions, since the resetting and the calculation of the new centroid coordinate phase only require K Threads to run at its full speed.

C. Data Race

On the **calculaSomatorioKernel**, since we have a N number of threads summing/incrementing to a K array, it's inevitable that there will be some Data Races. To control this, we utilized the native CUDA function **atomicAdd**(addr,val), which ensures that no value can get "lost" amid the sum process, even if it costs some processing time.

D. Architecture


Host
Data
pontoX[N], pontoY[N]
centroidX[K],centroidY[K]
pontoClusters[N]
alloca() & inicializa() & initializeKernel()
(loop iteration control)
END
Device
Data
dpontoX[N], dpontoY[N]
dcentroidX[K],dcentroidY[K]
dpontoClusters[N]
sumValX[K],sumValY[K]
numPontos[K]
Alloca Data
Copy pontoX & pontoY
Copy centroidX & centroidY
atribuiClustersKernel<>(...)
limpaSomatoriosKernel<>(...)
calculaSomatorioKernel<>(...)
calculaNovosCentroidsKernel<>(...)
atribuiClustersKernel<>(...)
....
Copy dcentroidX & dcentroidY
Copy dpontoClusters

E. Results

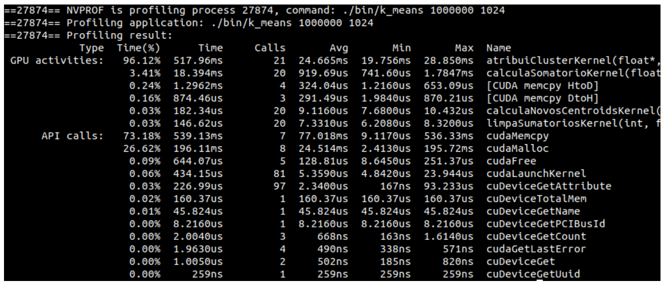
CudaV2		
Test	**TExec (s)**	**L1 Cache M.**
N=1M, K=320, @SEARCH	1,45	6,51E+6
N=1M, K=1024, @SEARCH	1,71	1,92E+7
N=1M, K=1024, @Laptop	0,73	3,60E+07

We can see we had time reductions in every test performed:

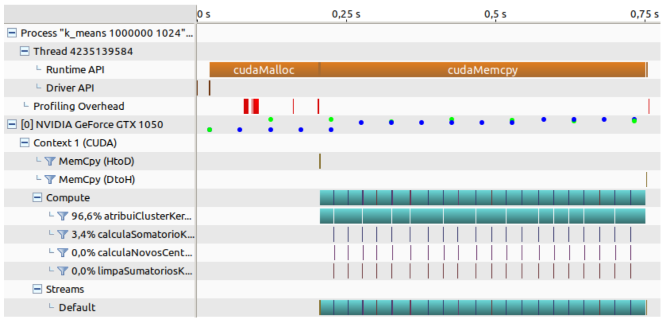
- **N=1M, K=320, @SEARCH:** 1,57s → 1,45s (-0,12s)
- **N=1M, K=1024, @SEARCH:** 1,78s → 1,71s (-0,07s)
- **N=1M, K=1024, @Laptop:** 0,81s → 0,73s (-0,08s)

F. Profiling

We proceeded to analyze this version profiling, confirming that this version runs both the algorithm phases:



```
==27874== NVPROF is profiling process 27874, command: ./bin/k_means 1000000 1024
==27874== Profiling application: ./bin/k_means 1000000 1024
==27874== Profiling result:
            Type  Time(%)      Time     Calls       Avg       Min       Max  Name
 GPU activities:   96.12%  517.96ms        21  24.665ms  19.756ms  28.850ms  atribuiClusterKernel(float*,
                    3.41%  18.394ms        20  919.69us  741.60us  1.7847ms  calculaSomatorioKernel(float
                    0.24%  1.2962ms         4  324.04us  1.2160us  653.09us  [CUDA memcpy HtoD]
                    0.16%  874.46us         3  291.49us  1.9840us  870.21us  [CUDA memcpy DtoH]
                    0.03%  182.34us        20  9.1160us  7.6800us  10.432us  calculaNovosCentroidsKernel(
                    0.03%  146.62us        20  7.3310us  6.2080us  8.3200us  limpaSumatoriosKernel(int, f
      API calls:   73.18%  539.13ms         7  77.018ms  9.1170us  536.33ms  cudaMemcpy
                   26.62%  196.11ms         8  24.514ms  2.4130us  195.72ms  cudaMalloc
                    0.09%  644.07us         5  128.81us  8.6450us  251.37us  cudaFree
                    0.06%  434.15us        81  5.3590us  4.8420us  23.944us  cudaLaunchKernel
                    0.03%  226.99us        97  2.3400us     167ns  93.233us  cuDeviceGetAttribute
                    0.02%  160.37us         1  160.37us  160.37us  160.37us  cuDeviceTotalMem
                    0.01%  45.824us         1  45.824us  45.824us  45.824us  cuDeviceGetName
                    0.00%  8.2160us         1  8.2160us  8.2160us  8.2160us  cuDeviceGetPCIBusId
                    0.00%  2.0040us         3     668ns     163ns  1.6140us  cuDeviceGetCount
                    0.00%  1.9630us         4     490ns     338ns     571ns  cudaGetLastError
                    0.00%  1.0050us         2     502ns     185ns     820ns  cuDeviceGet
                    0.00%     259ns         1     259ns     259ns     259ns  cuDeviceGetUuid
```



- The **cudaMalloc** API call had a small execution time increase of around 12ms, due to the increased allocatted data in the device for the auxiliary arrays used in the **ajustaCentroids** function.
- The three Kernel functions of **ajustaCentroids**, take up to a combined 19ms, which is around a **100ms reduction** compared to the **123ms CUDA** Downtime present in the last version due to the host running the **ajustaCentroids** function. This reduction is also observed generally in the test result above.
- The number of **cudaMemcpy** API calls is reduced to 7 calls compared to the last version 65 calls. These 7 calls are divided by 4 at the beginning and 3 at the end, only taking up to 2ms of execution time, a **14ms time reduction**.

VI. Third Version Development

A. Strategy

With the majority of the algorithm execution running exclusively on the CUDA API, our team decided to look for techniques inside CUDA to raise it's efficiency and further reduce execution time.

For this, we decided to implement the use of the shared memory present within the CUDA blocks. This memory has a faster read/write access time and is useful for data that is repeatedly accessed/shared between threads.

B. Shared Memory in atribuiClustersKernel

Being the heaviest function even in its CUDA implementation, we started by analyzing how could the use of shared memory improve this function. Since the Point Data and the Attribution Data is generally large in size and its data is accessed in unique positions by every thread, it's an undesirable candidate for this technique.

Meanwhile the centroid data, usually smaller in size, is accessed by all threads, being iterated in its entirely, which makes it a perfect candidate for the use of shared memory. For this, at the beginning of the kernel function, we allocate the shared memory within the block, along with the attribution of the centroid data in this memory. We also have to add a synchronization technique, to ensure the data in the shared memory is all loaded before the rest of the function execution can happen by using the **__syncthreads** CUDA function.

C. Results

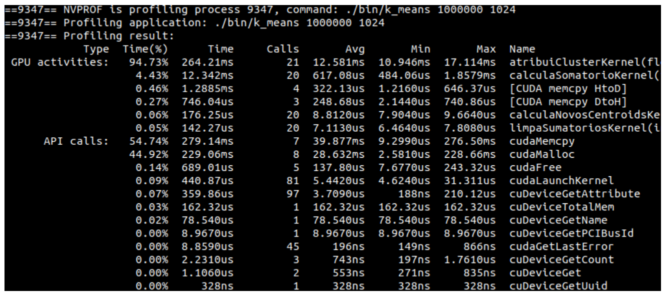
CudaV3		
Test	**TExec (s)**	**L1 Cache M.**
N=1M, K=320, @SEARCH	1,38	3,92E+6
N=1M, K=1024, @SEARCH	1,50	9,04E+6
N=1M, K=1024, @Laptop	0,48	2,23E+07

In this version, we had a larger performance gain on the tests with a larger number of Clusters. These findings will be discussed in the next chapter.

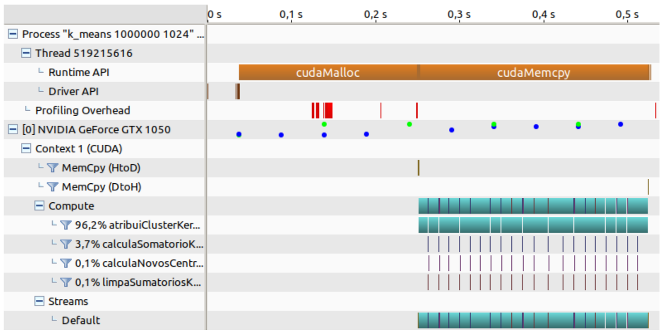
- **N=1M, K=320, @SEARCH:** 1,45s → 1,38s (-0,07s)
- **N=1M, K=1024, @SEARCH:** 1,71s → 1,50s (-0,21s)
- **N=1M, K=1024, @Laptop:** 0,73s → 0,48s (-0,25s)

D. Profiling

We proceeded to our final profiling, to see how much the kernel functions benefitted from the use of shared memory:



```
==9347== NVPROF is profiling process 9347, command: ./bin/k_means 1000000 1024
==9347== Profiling application: ./bin/k_means 1000000 1024
==9347== Profiling result:
            Type  Time(%)      Time     Calls       Avg       Min       Max  Name
 GPU activities:   94.73%  264.21ms        21  12.581ms  10.946ms  17.114ms  atribuiClusterKernel(fl
                    4.43%  12.342ms        20  617.08us  484.06us  1.8579ms  calculaSomatorioKernel(
                    0.46%  1.2885ms         4  322.13us  1.2160us  646.37us  [CUDA memcpy HtoD]
                    0.27%  746.04us         3  248.68us  2.1440us  740.86us  [CUDA memcpy DtoH]
                    0.06%  176.25us        20  8.8120us  7.9040us  9.6640us  calculaNovosCentroidsKe
                    0.05%  142.27us        20  7.1130us  6.4640us  7.8080us  limpaSumatoriosKernel(i
      API calls:   54.74%  279.14ms         7  39.877ms  9.2990us  276.50ms  cudaMemcpy
                   44.92%  229.06ms         8  28.632ms  2.5810us  228.66ms  cudaMalloc
                    0.14%  689.01us         5  137.80us  7.6770us  243.32us  cudaFree
                    0.09%  440.87us        81  5.4420us  4.6240us  31.311us  cudaLaunchKernel
                    0.07%  359.86us        97  3.7090us     188ns  210.12us  cuDeviceGetAttribute
                    0.03%  162.32us         1  162.32us  162.32us  162.32us  cuDeviceTotalMem
                    0.02%  78.540us         1  78.540us  78.540us  78.540us  cuDeviceGetName
                    0.00%  8.9670us         1  8.9670us  8.9670us  8.9670us  cuDeviceGetPCIBusId
                    0.00%  8.8590us        45     196ns     149ns     866ns  cudaGetLastError
                    0.00%  2.2310us         3     743ns     197ns  1.7610us  cuDeviceGetCount
                    0.00%  1.1060us         2     553ns     271ns     835ns  cuDeviceGet
                    0.00%     328ns         1     328ns     328ns     328ns  cuDeviceGetUuid
```

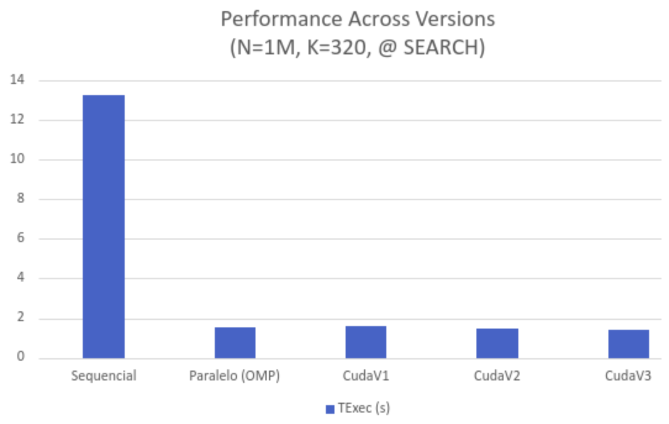


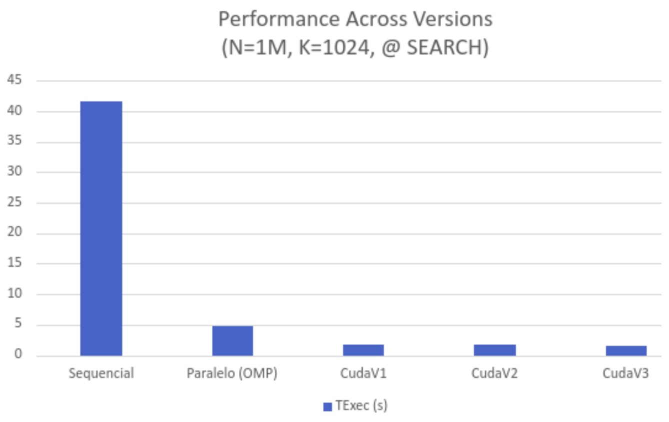
- The **atribuiClusterKernel** function had a 236ms time reduction, representing a close to 50% performance gain on this function.

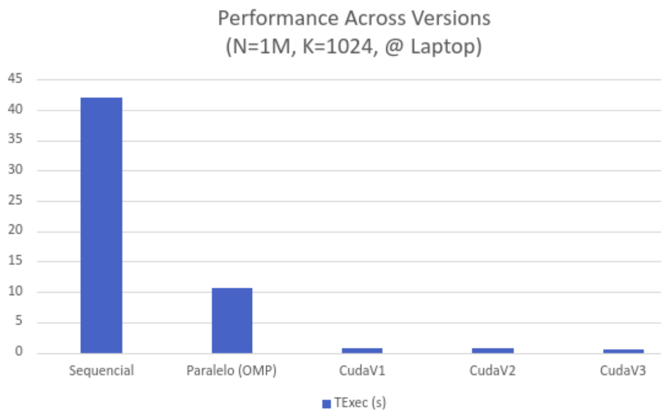
VII. Results comparison and analysis

A. *General Comparison*

We will start by showing a general comparison of the result on all three tests, including the perfomance of the sequential and parallelized version developed in previous parts of this project



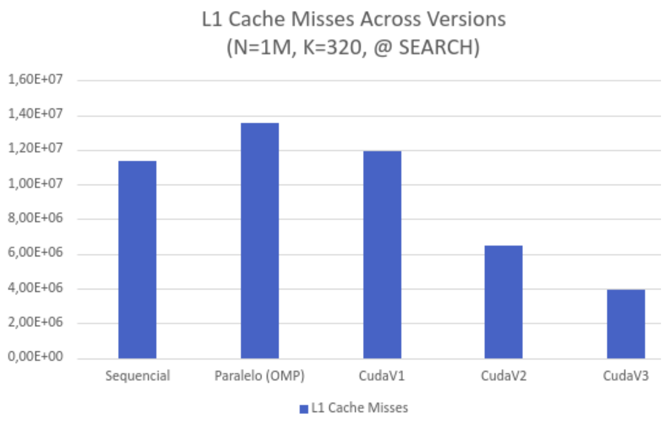


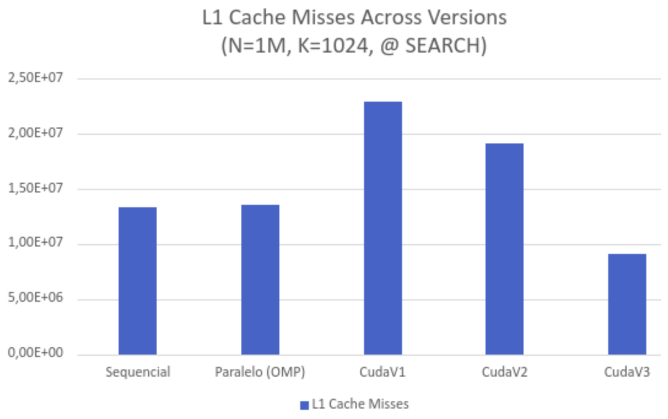


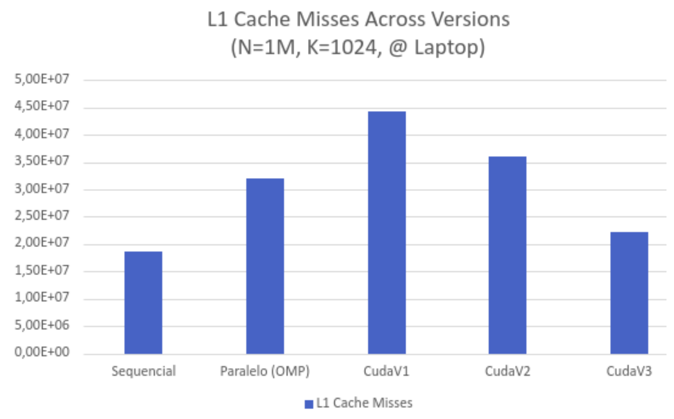
B. *L1 Cache Misses*

We also analysed the L1 Cache Misses between tests and versions. We can see that within the three CUDA versions developed we can always see a steady decrease in this metric, which reaffirms our observed time reduction.

We can also confirm that the sequential version, although has a lesser level of L1 Cache Misses, has higher execution times since the workload is executed on a single thread, which decreases the likelyhood of Cache Misses.



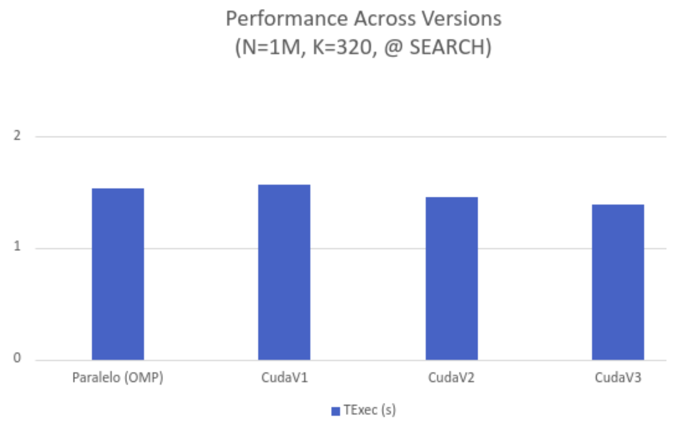


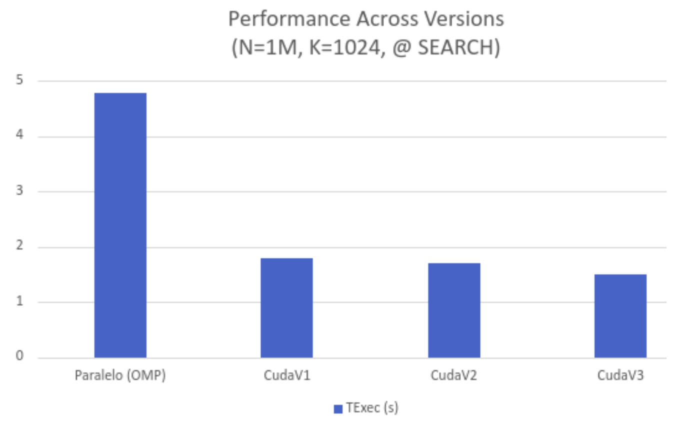


C. Scalability

To test scalibility, we used the two Cluster tests on the SEARCH machine with a K value of 320 and 1024, to analyse how well our CUDA versions scale:

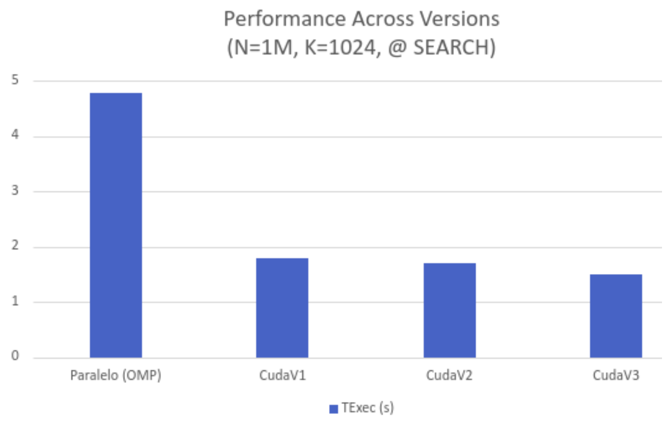
As we can see, compared to the OMP version, which sees a execution time increase of 3s, a 3x increase when the K value changes from 320 to 1024, which is also roughly a 3x increase on the K value, our CUDA version only increase a maximum of 200ms, only a 1,13x execution time increase, which shows the potencial scalibility of this solution.

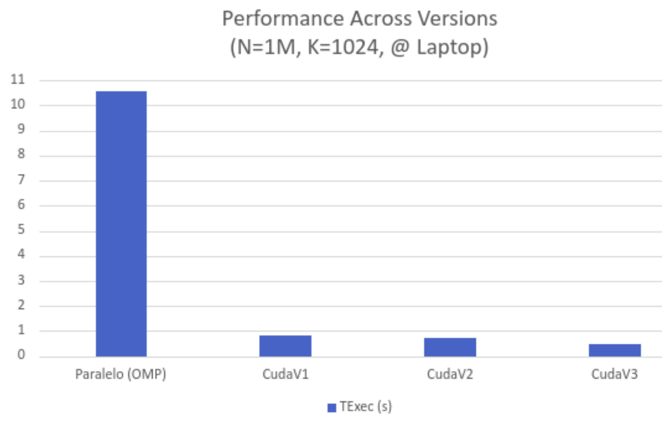




D. SEARCH VS Laptop

In this result set, we can see that the Server machine, with its high core CPU, beats the Laptop on the OMP version, but the Laptop, with its more advanced GPU, has a better performance time than the server on the CUDA versions.

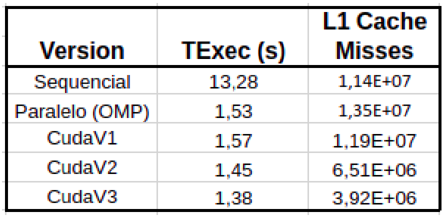




VIII. Conclusion

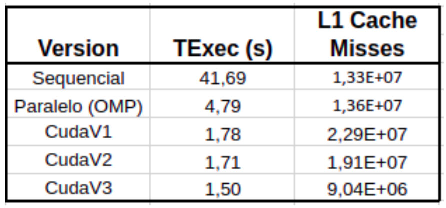
Our team is very satisfied with the results achieved along this project, by improving the algorithm's sequential version, and then improving execution time by using parallelization techniques, including solutions on CPU or GPU, we increased the algorithm performance, observing up to 110x time gain in some of our tests.

IX. Attachments



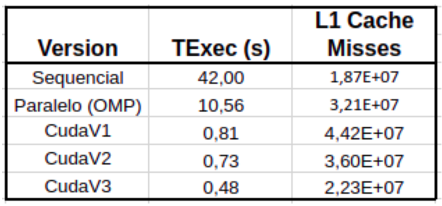
Version	TExec (s)	L1 Cache Misses
Sequencial	13,28	1,14E+07
Paralelo (OMP)	1,53	1,35E+07
CudaV1	1,57	1,19E+07
CudaV2	1,45	6,51E+06
CudaV3	1,38	3,92E+06

$N = 1M, K = 320, @SEARCH$



Version	TExec (s)	L1 Cache Misses
Sequencial	41,69	1,33E+07
Paralelo (OMP)	4,79	1,36E+07
CudaV1	1,78	2,29E+07
CudaV2	1,71	1,91E+07
CudaV3	1,50	9,04E+06

$N = 1M, K = 1024, @SEARCH$



Version	TExec (s)	L1 Cache Misses
Sequencial	42,00	1,87E+07
Paralelo (OMP)	10,56	3,21E+07
CudaV1	0,81	4,42E+07
CudaV2	0,73	3,60E+07
CudaV3	0,48	2,23E+07

$N = 1M, K = 1024, @Laptop$