Università di Pisa

Computer Architecture - Project: Performance analysis of a parallel K-Means algorithm

Antonio Patimo, Guillaume Quint

Academic Year: 2022/2023

Abstract

In this project we present the work done in developing a multi-threaded application for the K-Means algorithm, and its performance analysis over two different computer architectures: CPU and GPU. After each iteration, we performed a series of observations about the results obtained. This allowed us to present effective optimizations for both architectures, showing the improvement achieved. At the end we performed a brief comparison between the CPU's and GPU's version of the software.

Contents

1 Parallel K-Means algorithm

As a reference for comparing the two architectures, CPU and GPU, we chose a parallel implementation of the popular K-Means algorithm.

This is an unsupervised clustering algorithm which works over a dataset of points in a given dimension space. Given a parameter K, the algorithm finds K clusters among all points in the dataset.

The parallel version of this algorithm allows to distribute the dataset between multiple parallel executing units. In particular, the dataset is partitioned into n chunks, where n is the number of executing units (in our case, the number of threads). Each unit links each point of its partition with the closest centroid, then sums up the components of every point for each centroid and counts the number of points associated to each centroid. These partial results are then sent back to a master unit which aggregates them to compute a new approximation for the centroids. These new centroids are then sent to every computing unit iteratively until convergence of the centroids.

Listing 1 shows the pseudo-code for the parallel K-Means clustering algorithm.

Figure 1 shows a snapshot of 6 iterations in a bi-dimensional case with $K = 4$

```
pocedure K-Means-Parallel(dataset, K, n):
    centroids = generateRandomCentroids(K)
    partitions = generatePartitions(dataset, n)
    workers = generateWorkers(n, partitions)
    while (not converged):
        workers.send(centroids)
        partial_results = workers.execute()
        workers.synchronize()
        updateCentroids(partial_results)
```

Listing 1: Parallel K-Means pseudocode

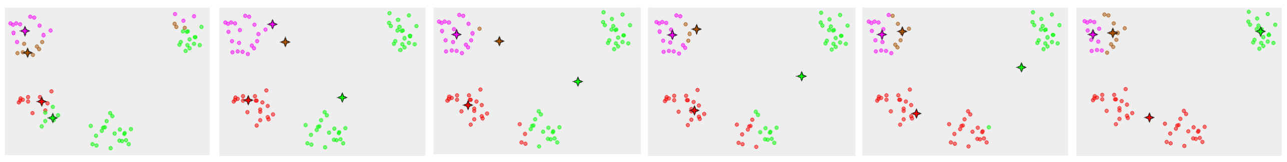


Figure 1: 6 iterations of the K-Means clustering algorithm

1 .1 Real use case application

We imagined a realistic scenario for which the parallel K-Means clustering might be useful.

An international Bank wants to identify client groups based on their monthly income and expenditure to propose personalized investment plans. We envisioned a typical volume of around 10 million users and 5 distinct groups. The product is therefore a customer clustering software for personalized advertisement.

2 CPU

In this section we're dealing with the CPU implementation of the software and its relative performance analysis. We then develop our observations about the obtained results in order to implement an optimization.

2 .1 Hardware Specification

All tests were conducted on an Intel i7-6700HQ of which we reported relevant hardware specifications in Table 1

CPU cores	4
Threads	8
L1 cache	4 x 32 KB 8-way set associative instruction caches 4 x 32 KB 8-way set associative data caches
L2 cache	4 x 256 KB 4-way set associative caches
L3 cache	6MB 12-way set associative shared cache
Clock speed	2.60GHz - 3.5GHz

Table 1: Hardware specification

2 .2 Implementation

Our software implementation accepts three arguments: a dataset of serialized points, the number of clusters and the number of threads. In particular the dataset of points must be preprocessed from a csv file using the CSVPreprocessor software that we developed to store data in a more convenient binary format. This allows us to perform a faster reading from file, so that we don't need to convert from a literal number format.

The dataset is then loaded in an array of classed points, which are represented as a tuple of coordinates and an index representing which cluster each point belongs to. Centroids are initialized by considering K points in the dataset chosen at random.

To build the partitions, points are distributed among all available threads.

Inside the performRounds() function, for each round we generate an array of threads executing the worker() function. We then wait on their termination and update our approximation for the centroids, until the maximum error goes below a certain threshold.

Each thread in the worker() function finds the closest centroid for every point in its partition. It then calls the aggregatePoints() function to sum up the coordinates of all points belonging to the same centroid.

In Listing 2 we can see the data structures used for the CPU implementation of the software.

```c
struct Point_s
{
    double* coords;
};
typedef struct Point_s Point;

struct ClassedPoint_s
{
    Point p;
    int k;
};
typedef struct ClassedPoint_s ClassedPoint;

struct Centroid_s
{
    Point p;
    Point* sum;
    int* partition_lengths;
};
typedef struct Centroid_s Centroid;

ClassedPoint* points;
Centroid* centroids;
```

Listing 2: cpu data structures

2 .3 Results

The executing times are shown with a varying number of threads and different dataset sizes in Figure 2. Results are obtained as a sample mean with population width of 30 samples. We also show the 95% confidence interval where values are appreciable. The execution times are obtained by dividing the total time elapsed to complete a program execution by the number of iterations required by the algorithm to converge. This is done to mitigate the fact that depending on the input, a different choices of initial centroids may influence the number of iterations, and therefore the executing time.

It is clear to see that an increase in the workload, i.e. a larger number of points to process, results in longer mean executing times.

The speedup graph in Figure 3 shows the relative speedup from the base case with 1 thread using a varying number of threads and dataset sizes. We can see that the maximum speedup achieved is at $3,57$ with 20 threads and dataset 100M. This indicates that we obtained good results even if we're not very near the theoretical maximum speedup of 8.

2 .4 Observations

From the data we extracted three main observations about the behaviour of the application's speedup in different executing conditions. We then tried to explain them using different profiling tools.

In particular, we exploited Intel's Vtune profiler, which provides deep insights into performance bottlenecks and inefficiencies in code, allowing to improve software's speed and efficiency. With features like hardware event-based sampling, multi-threading analysis, memory and I/O analysis, VTune profiler helped us identify hotspots, thread contention, memory access patterns, and more. It seamlessly integrates with popular development environments and supports various operating systems.

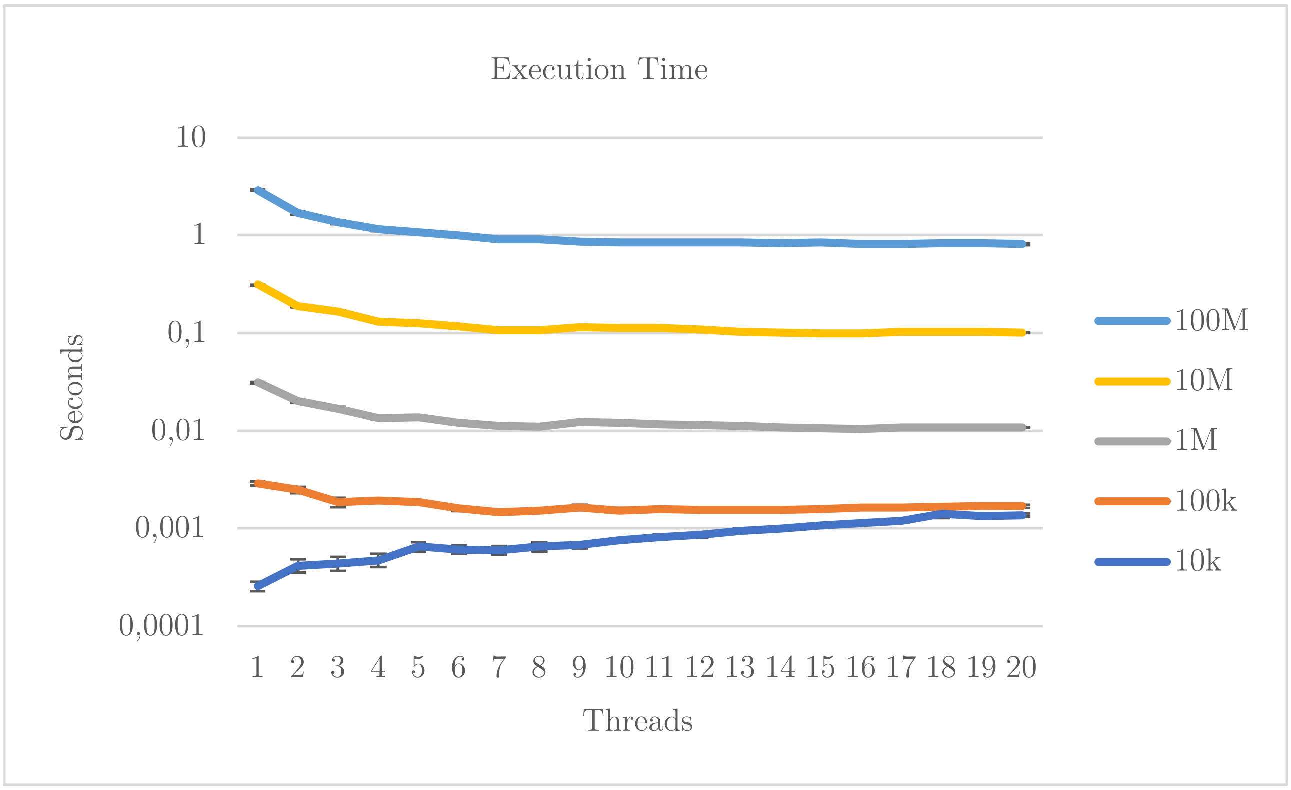


Figure 2: CPU execution time

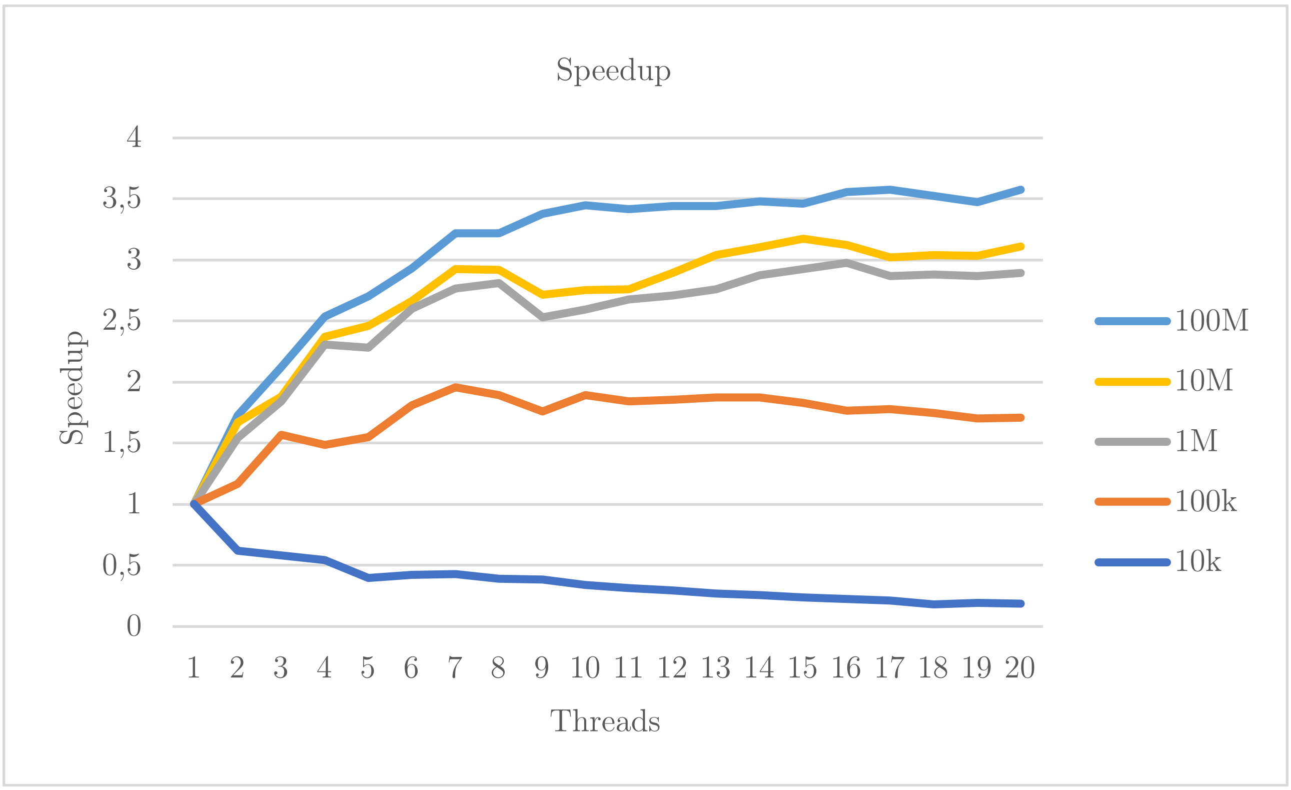


Figure 3: CPU speedup

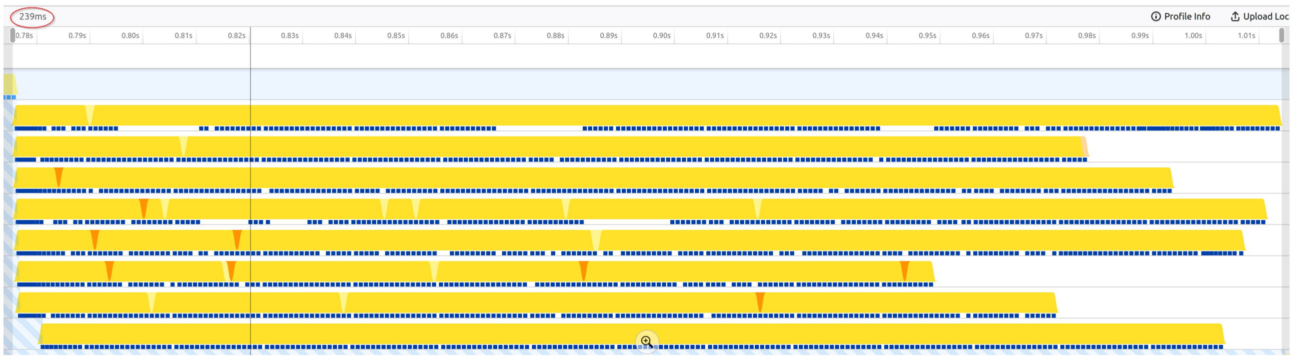


Figure 4: Total execution time with 8 threads is 239ms

Another tool that we used is Perf, which allowed us to collect performance data by monitoring hardware events, software events, and tracepoints. Hardware events include CPU cycles, cache misses, and branch instructions, while software events include function calls or specific instructions.

Finally, we also used Firefox profiler to obtain a graphical representation of the data collected.

2 .4.1 With a small dataset, increasing the number of threads reduces the speedup

From the speedup graph in Figure 3 we can observe that with small datasets, especially with 10k points, with a high number of threads we obtain progressively lower speedups. Using the VTune profiler, we observe that the CPU time spent waiting for thread termination goes from an average of 0,046s using 2 threads to 0,362s using 15 threads. Because the dataset is small, the total time spent by working threads in both cases is negligible. Therefore, with an increasing overhead we see an increase in execution time and so a reduction in speedup.

This confirms the intuition that the overhead due to handling multiple threads becomes more significant with a lighter work load.

2 .4.2 With larger datasets, increasing the number of threads yields an improved speedup with progressively diminishing returns until a threshold

With larger datasets, dividing the work among more threads is much more beneficial than the overhead introduced. For example, considering the case with the 10M points dataset, doubling the number of threads from 8 to 16 brings an average execution time from 212ms to 137ms. Meanwhile the overhead only increases from 26ms to 49ms. Therefore, the total iteration time is reduced from 239ms, as shown in Figure 4, to 186ms, as shown in Figure 5.

To observe a substantial reduction in performance due to overhead, we must increase a lot the number of threads or consider much smaller datasets. This is shown in Figure 6, where we considered the same statistics gathered up until a maximum of 20000 threads.

2 .4.3 With a larger dataset the speedup increases

We can observe that keeping constant the number of threads, a larger dataset is linked to a higher speedup. This can be explained by the fact that the sequential part of the application is almost constant with respect to the dataset size. Meanwhile, with a larger dataset, the work done in parallel increases.

Therefore, the ratio between CPU Time (i.e. the total CPU execution time considering all threads) and Elapsed Time (i.e. effective time elapsed to complete the execution) increases.

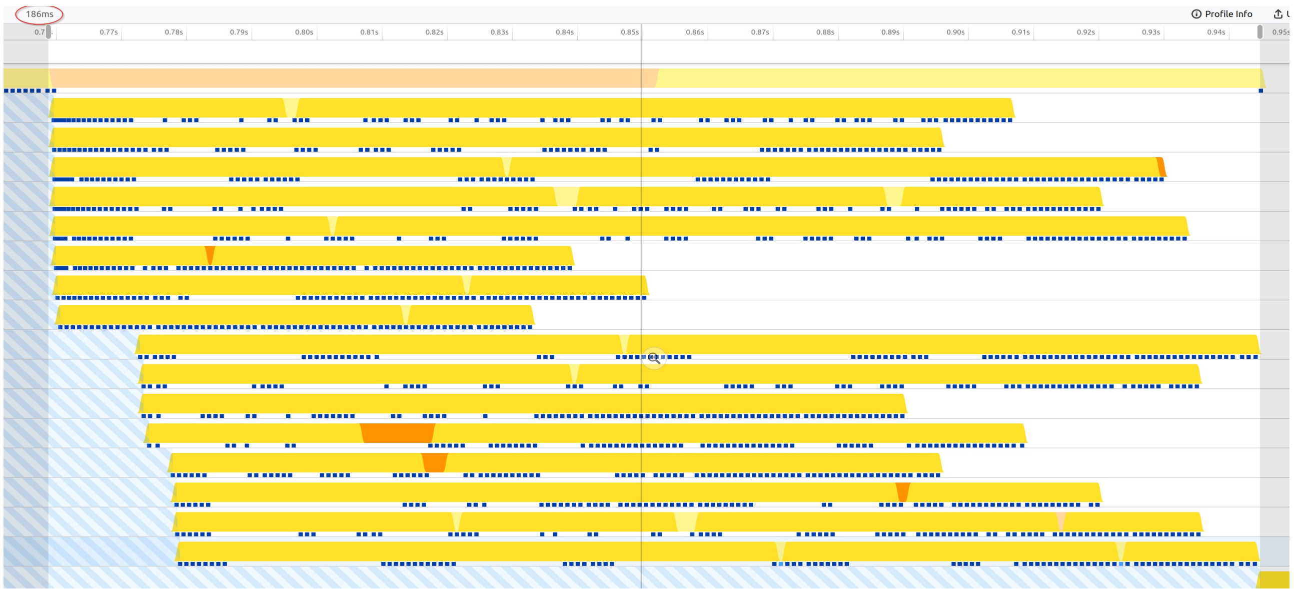


Figure 5: Total execution time with 16 threads is 186ms

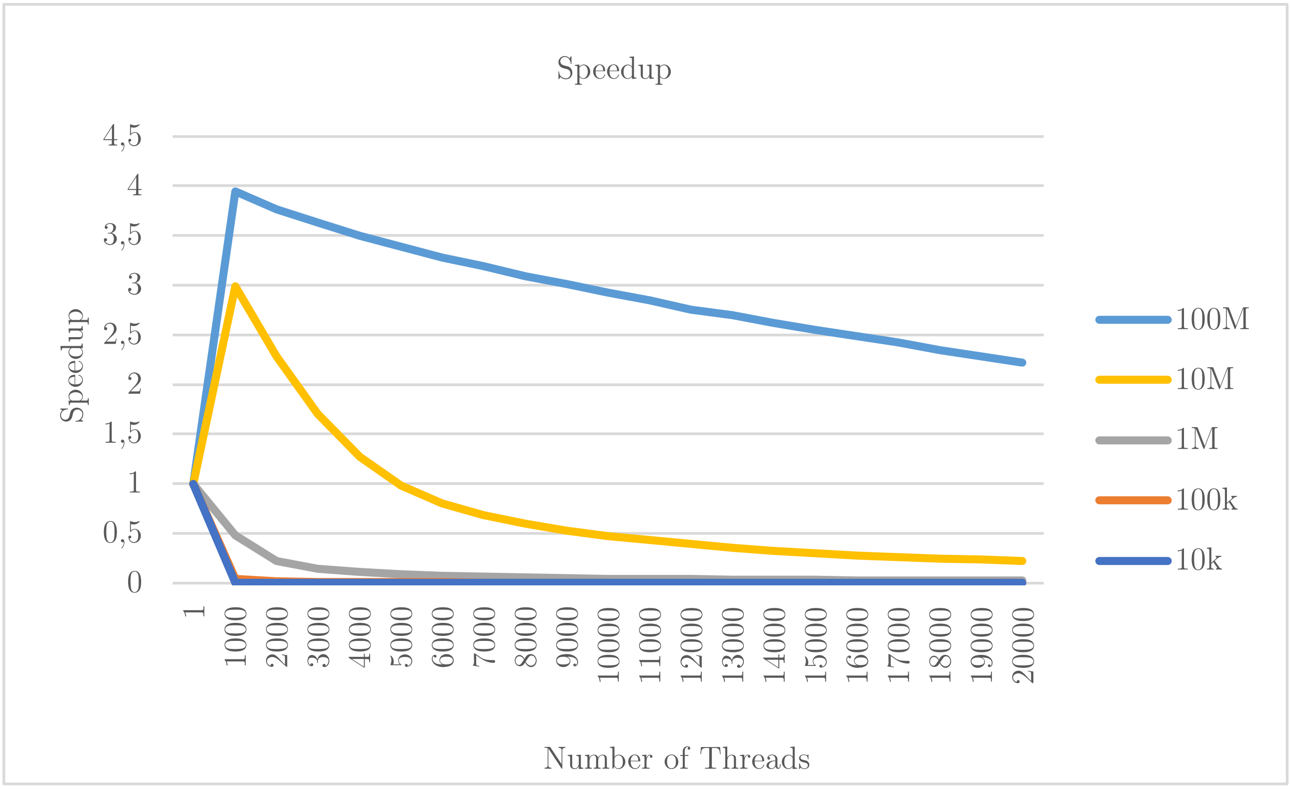


Figure 6: CPU speedup with a large number of threads

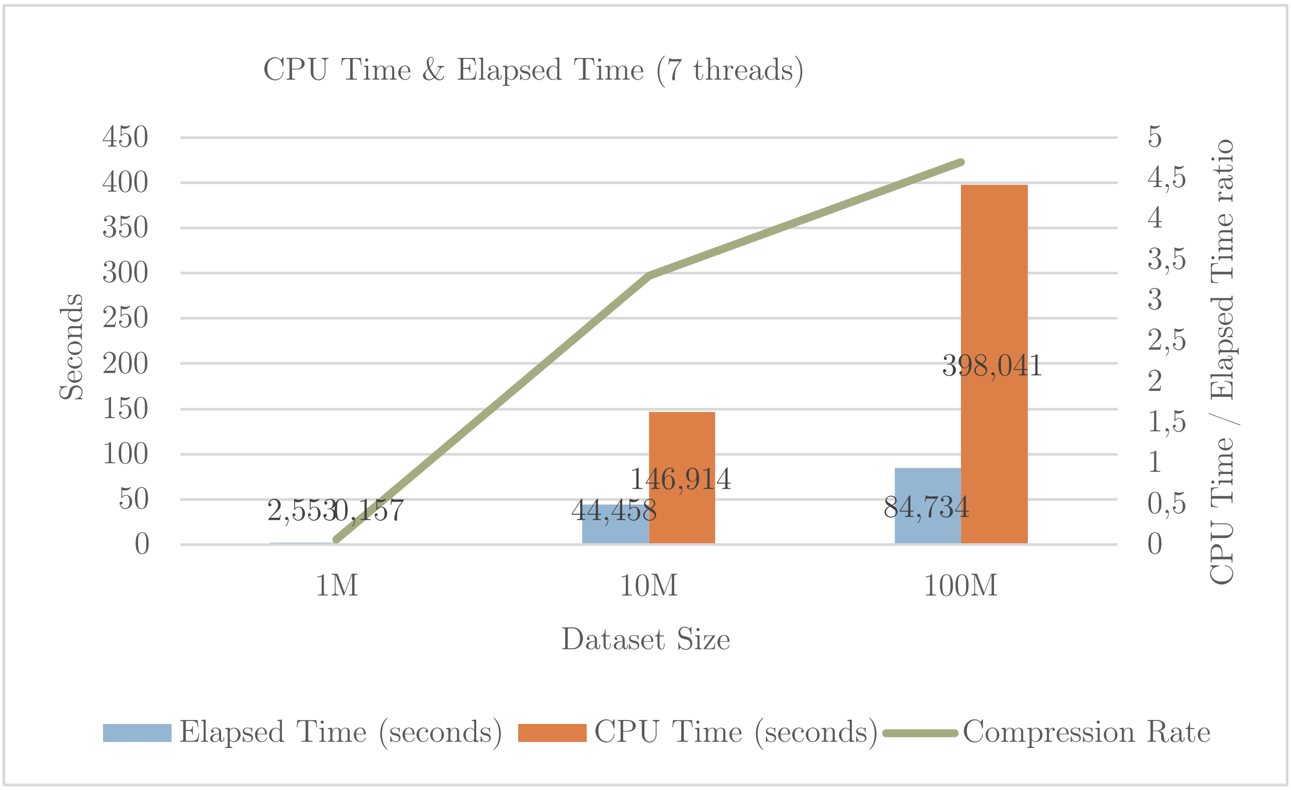


Figure 7: CPU time and elapsed time for different datasets

In Figure 7 we show both CPU time and elapsed time for different datasets. We can observe that for larger datasets, the ratio between those quantities (represented by the green line) increases.

2 .5 False sharing optimization

Using the data extracted from the profilers, especially from the Firefox profiler, we observed that we spent a large portion of the worker execution inside the aggregatePoints() subfunction, as shown in Figure 8. This led us to investigate on what caused those slow executing times.

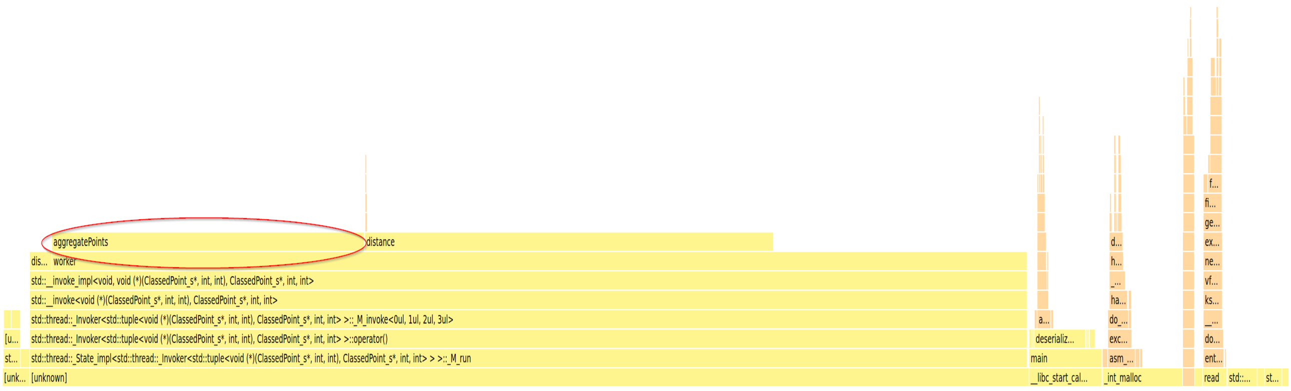
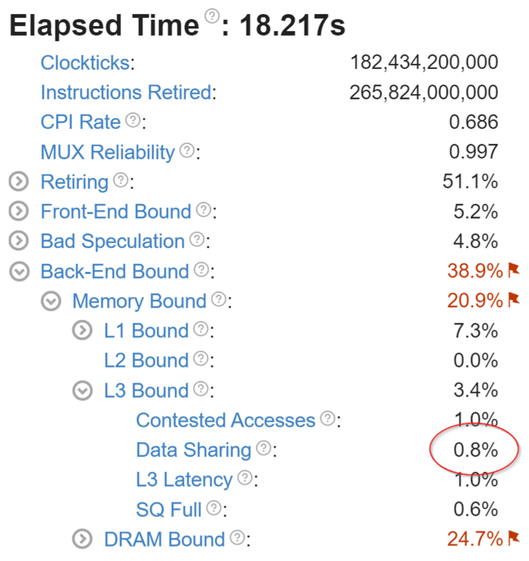

aggregatePoints
distance
dis... worker
std::_invoke_impl<void, void (*)(ClassedPoint_s*, int, int), ClassedPoint_s*, int, int>
std::_invoke<void (*)(ClassedPoint_s*, int, int), ClassedPoint_s*, int, int>
std::thread::_Invoker<std::tuple<void (*)(ClassedPoint_s*, int, int), ClassedPoint_s*, int, int> >::_M_invoke<0ul, 1ul, 2ul, 3ul>
std::thread::_Invoker<std::tuple<void (*)(ClassedPoint_s*, int, int), ClassedPoint_s*, int, int> >::operator()
std::thread::_State_impl<std::thread::_Invoker<std::tuple<void (*)(ClassedPoint_s*, int, int), ClassedPoint_s*, int, int> > >::_M_run
[unk... [unknown]
deserializ...
main
__libc_start_cal...
_int_malloc
read
std::...
st...

Figure 8: The worker spends most of its time inside the aggregatePoints function

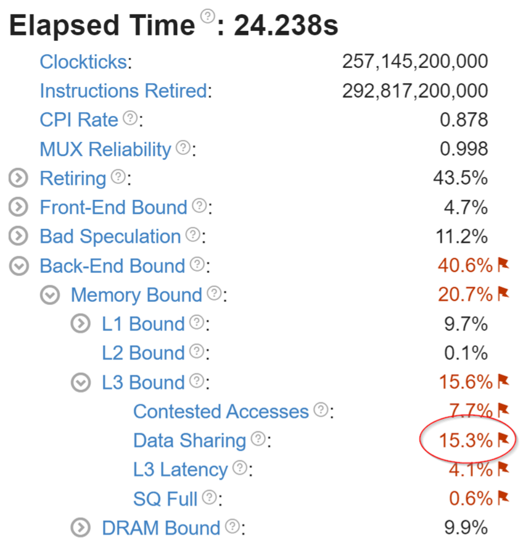
2 .5.1 Causes

By executing the microarchitecture analysis in Vtune, we discovered that we were bound by frequent DRAM transfers, as shown in Figure 9. This was due to the fact that each thread accessed its partition two times in two different loops: once to find the closest centroid, and once to update the centroid with the coordinates of each point.



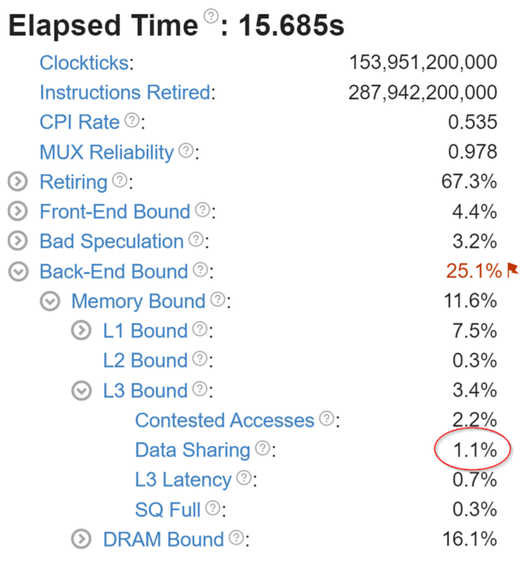
```
Elapsed Time: 18.217s
  Clockticks:                      182,434,200,000
  Instructions Retired:            265,824,000,000
  CPI Rate:                                  0.686
  MUX Reliability:                           0.997
  Retiring:                                  51.1%
  Front-End Bound:                            5.2%
  Bad Speculation:                            4.8%
  Back-End Bound:                            38.9%
    Memory Bound:                            20.9%
      L1 Bound:                               7.3%
      L2 Bound:                               0.0%
      L3 Bound:                               3.4%
        Contested Accesses:                   1.0%
        Data Sharing:                         0.8%
        L3 Latency:                           1.0%
        SQ Full:                              0.6%
      DRAM Bound:                            24.7%
```

Figure 9: Microarchitecture analysis without optimization



```
Elapsed Time: 24.238s
  Clockticks:                      257,145,200,000
  Instructions Retired:            292,817,200,000
  CPI Rate:                                  0.878
  MUX Reliability:                           0.998
  Retiring:                                  43.5%
  Front-End Bound:                            4.7%
  Bad Speculation:                           11.2%
  Back-End Bound:                            40.6%
    Memory Bound:                            20.7%
      L1 Bound:                               9.7%
      L2 Bound:                               0.1%
      L3 Bound:                              15.6%
        Contested Accesses:                   7.7%
        Data Sharing:                        15.3%
        L3 Latency:                           4.1%
        SQ Full:                              0.6%
      DRAM Bound:                             9.9%
```

Figure 10: Microarchitecture analysis with the first optimization



```
Elapsed Time: 15.685s
  Clockticks:                      153,951,200,000
  Instructions Retired:            287,942,200,000
  CPI Rate:                                  0.535
  MUX Reliability:                           0.978
  Retiring:                                  67.3%
  Front-End Bound:                            4.4%
  Bad Speculation:                            3.2%
  Back-End Bound:                            25.1%
    Memory Bound:                            11.6%
      L1 Bound:                               7.5%
      L2 Bound:                               0.3%
      L3 Bound:                               3.4%
        Contested Accesses:                   2.2%
        Data Sharing:                         1.1%
        L3 Latency:                           0.7%
        SQ Full:                              0.3%
      DRAM Bound:                            16.1%
```

Figure 11: Microarchitecture analysis with the second optimization

To lower memory accesses, we tried factoring the second loop inside the first: by immediately updating the correct centroid once found.

Unexpectedly, we obtained slower performances. Further profiling the code, we observed an actual decrease in memory accesses, as expected, but we discovered a large increase in data sharing among threads, as shown in Figure 10.

This is explained by the fact that, even if every thread accesses a private element of the sum array inside the centroid structure, these locations are contiguously stored in memory for each

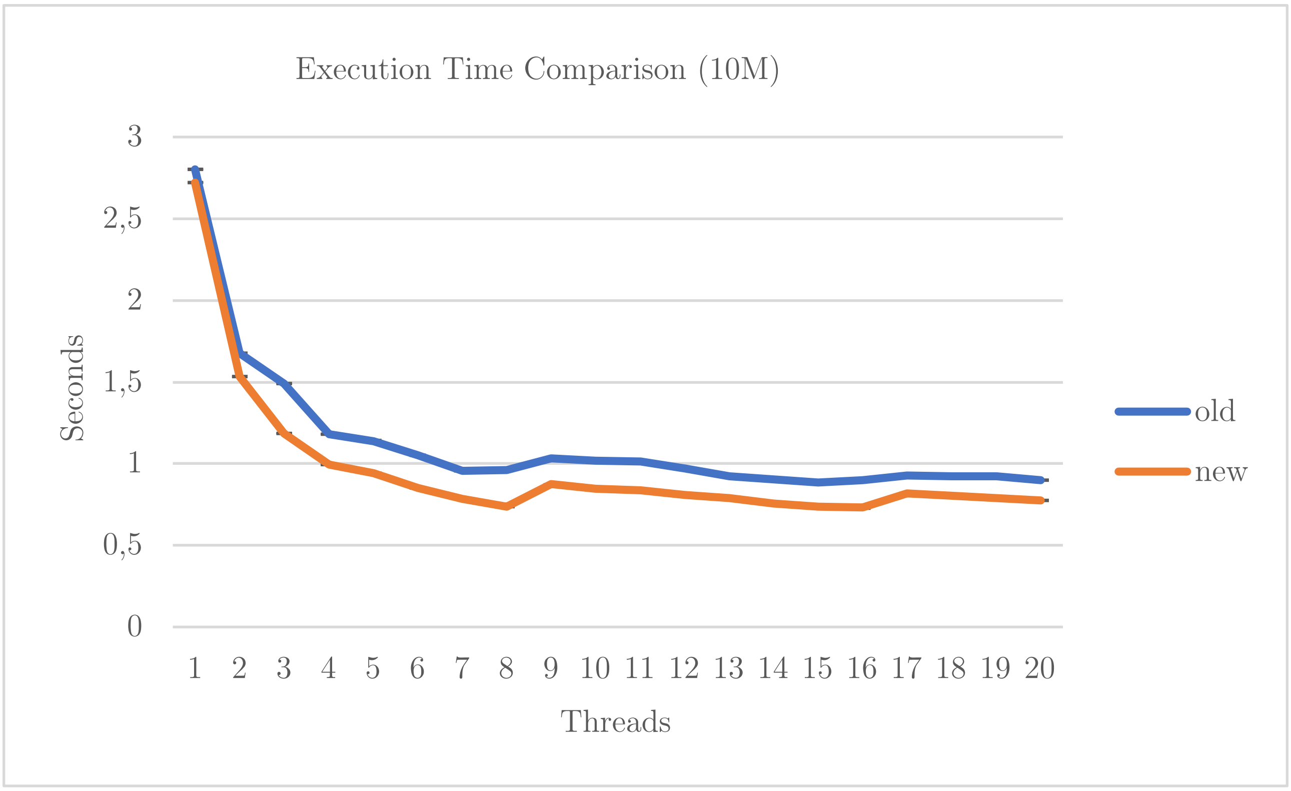


Figure 12: CPU execution times comparison after the optimization with 10M

thread. Therefore, these continuous write accesses led to an increase in false sharing, greatly degrading performances.

2 .5.2 Solution

In order to process the partition elements only in one loop and still avoiding the false sharing problem described, we employed a solution consisting in using private local variables for each thread to store the computed aggregation. Finally, those results are written back in the correct global data structure.

In this way, we restored low levels of false sharing, as it was in the original implementation and successfully reduced the number of memory accesses as shown by the memory bound metric, as we can see in Figure 11

2 .5.3 Results

The final results obtained by implementing the above optimization are shown in Figure 12 and the final relative speedup in Figure 13, considering the case with 10M dataset.

We can clearly observe lower execution times for every number of thread, and a greater computing parallelization, as shown by the speedup graph.

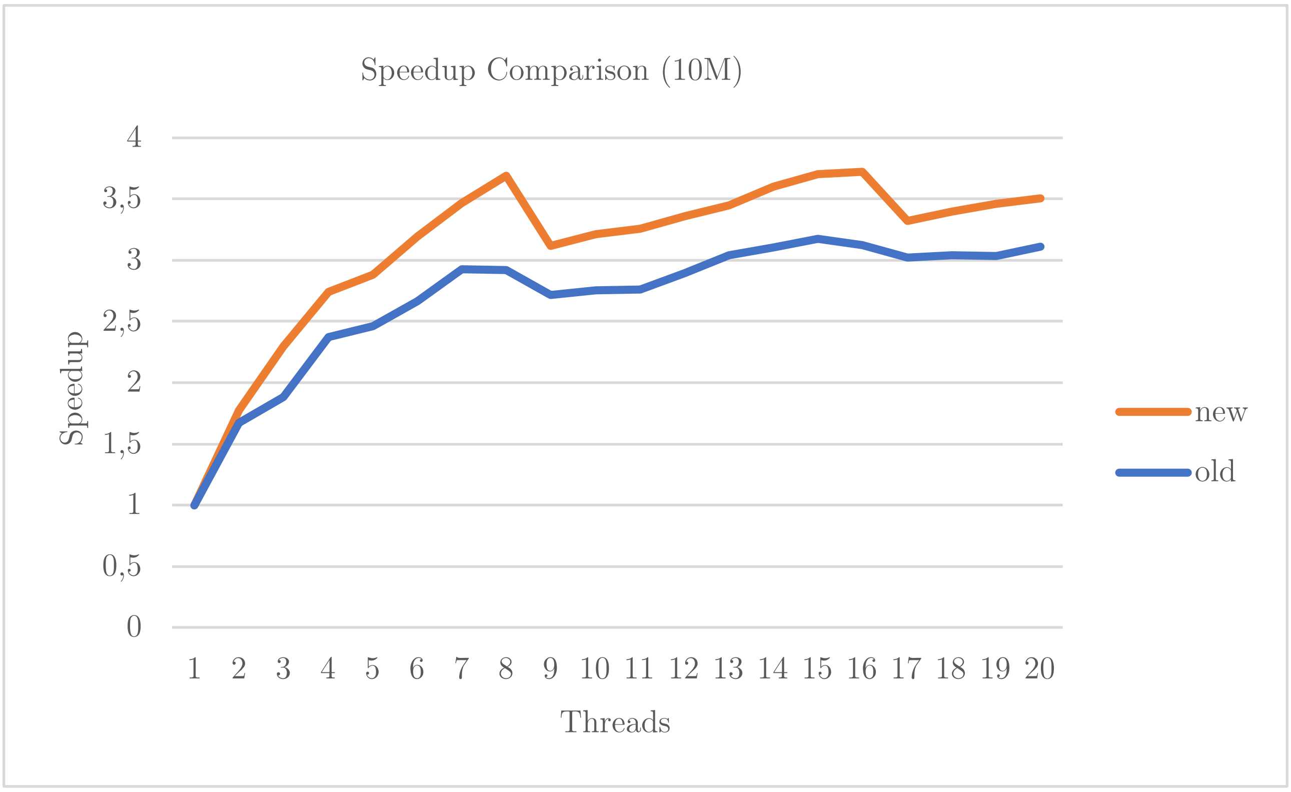


Figure 13: CPU speedup comparison after the optimization with 10M

3 GPU

The next section deals with porting the parallel K-means algorithm to a GPU architecture. We firstly analyzed the performances of a naive implementation and then we proposed an optimized version which takes in consideration the specific properties of this architecture.

3 .1 Hardware Specification

To perform our tests we employed a Nvidia GTX 1050 with the following characteristics

Architecture	Pascal
Cuda Cores	640
Multiprocessors	5
Clock rate	1,493 GHz
Global Memory Size	3,946 GiB
Single Precision FLOP/s	1,911 TeraFLOP/s
Double Precision FLOP/s	59,72 GigaFLOP/s
Constant Memory Size	64 KiB
L2 Cache Size	512 KiB
Global Memory Bandwidth	112,128 GB/s

Table 2: Hardware specification

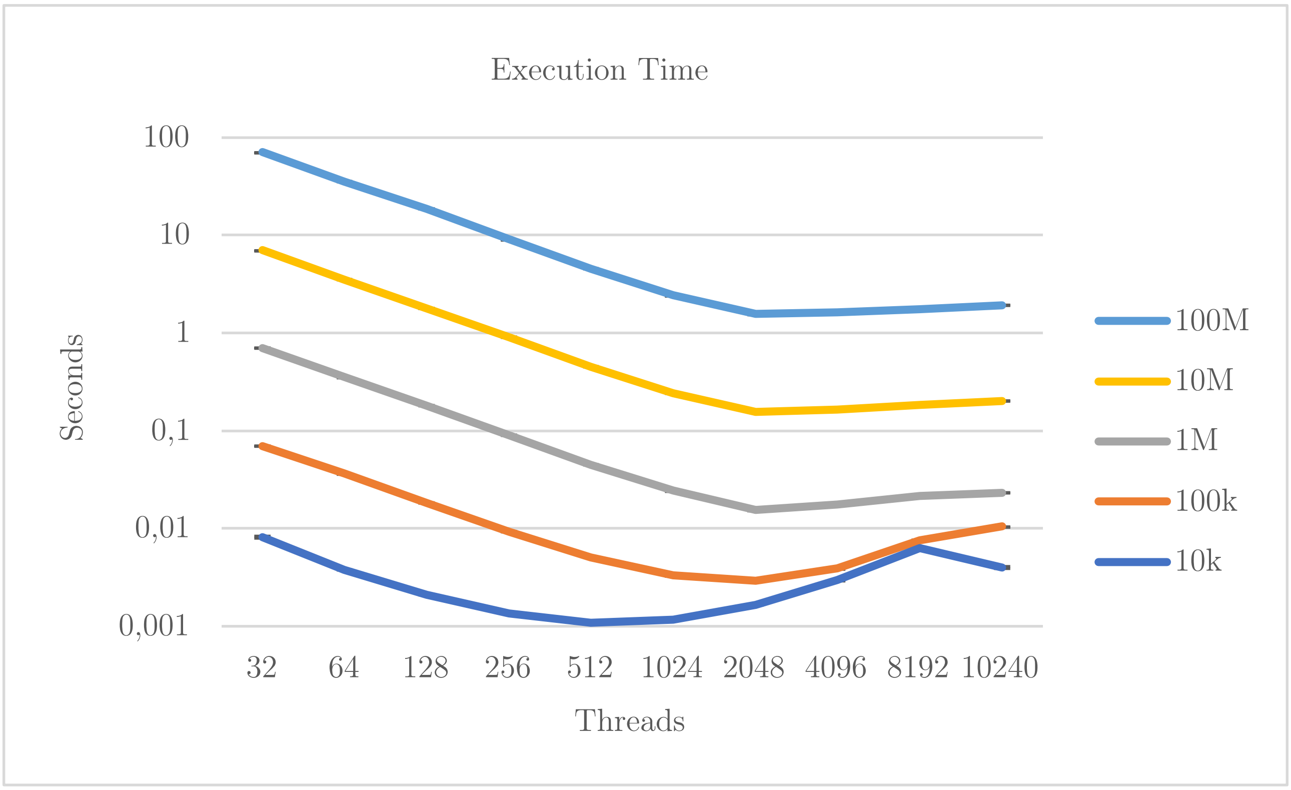


Figure 14: GPU execution times

3 .2 Implementation

The worker() function executed by threads in the cpu implementation became a gpu kernel run in parallel by the gpu threads. This required to allocate and copy all necessary data from the host to the external device, i.e. all dataset points and the current centroids for current iteration.

The dataset is initially loaded once into the global memory of the device. At each iteration of the algorithm, the kernel receives the current positions of all centroids and computes the partial results that are then sent back to the host. After synchronizing the termination of all gpu threads, the cpu aggregates the partial results obtained and updates the centroids, which are sent back to the device for a new iteration until convergence.

Because the algorithm doesn't require any communication between threads, all data is therefore stored in global memory.

3 .3 Results

In Figure 14 we can see the executing time of different dataset sizes using a varying number of threads. We choose to use a constant value of 128 threads per thread-block in order to obtain a large number of warps that can be scheduled by the warp scheduler in order to hide possible memory latency.

In Figure 15 we show the relative speedup in the same cases. We can notice that we reach a peak of almost 45 increase in performance with 100M, 10M and 1M using 2048 threads.

Note that each label in the threads axis doubles the number of executing threads, therefore the interval between each step is not constant.

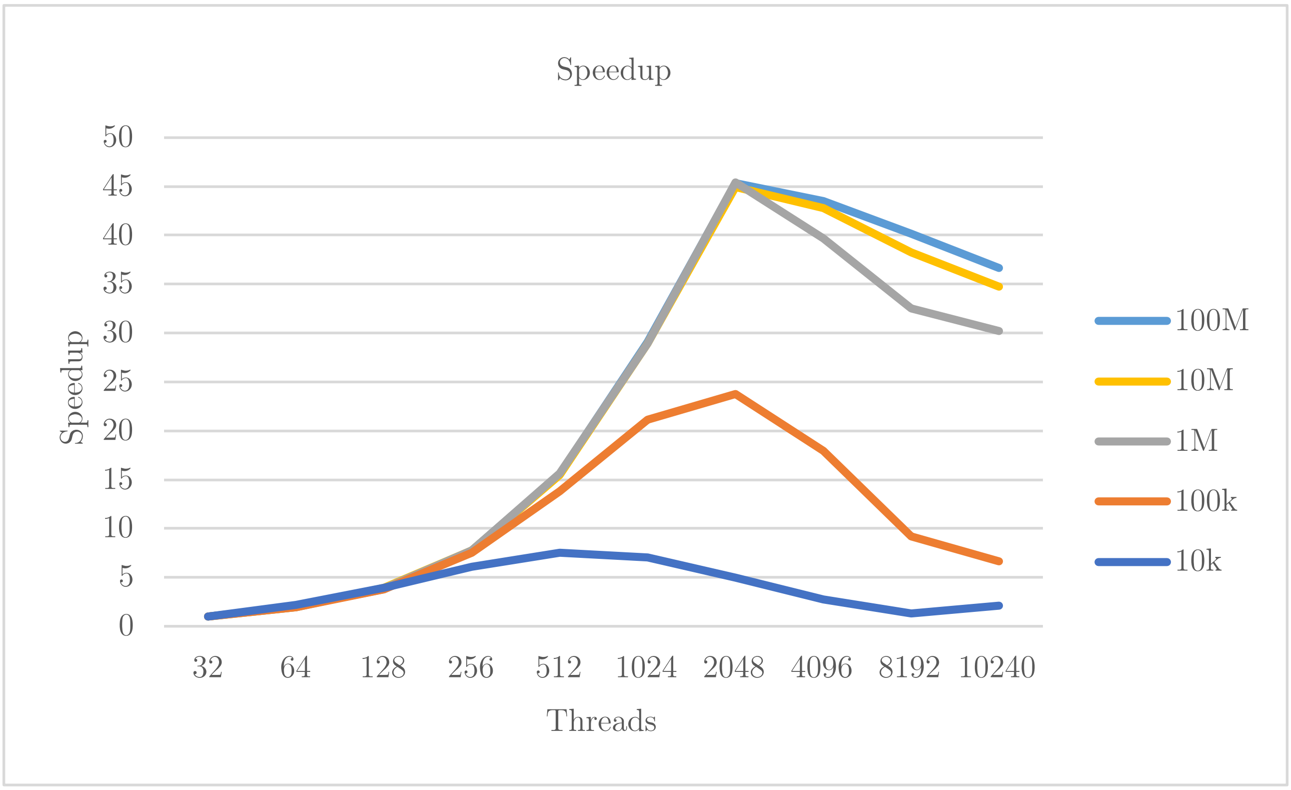
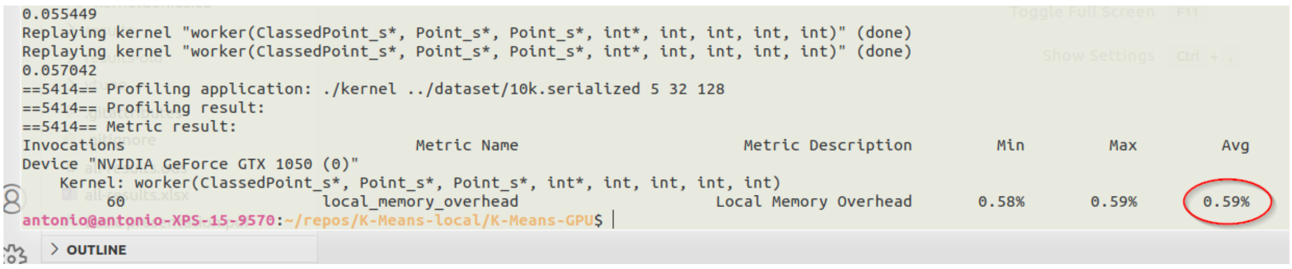


Figure 15: GPU speedup



```
0.055449
Replaying kernel "worker(ClassedPoint_s*, Point_s*, Point_s*, int*, int, int, int, int)" (done)
Replaying kernel "worker(ClassedPoint_s*, Point_s*, Point_s*, int*, int, int, int, int)" (done)
0.057042
==5414== Profiling application: ./kernel ../dataset/10k.serialized 5 32 128
==5414== Profiling result:
==5414== Metric result:
Invocations                               Metric Name                        Metric Description         Min         Max         Avg
Device "NVIDIA GeForce GTX 1050 (0)"
    Kernel: worker(ClassedPoint_s*, Point_s*, Point_s*, int*, int, int, int, int)
         60                 local_memory_overhead                     Local Memory Overhead       0.58%       0.59%       0.59%
antonio@antonio-XPS-15-9570:~/repos/K-Means-local/K-Means-GPU$ |
```

Figure 16: Local memory overhead with 32 threads

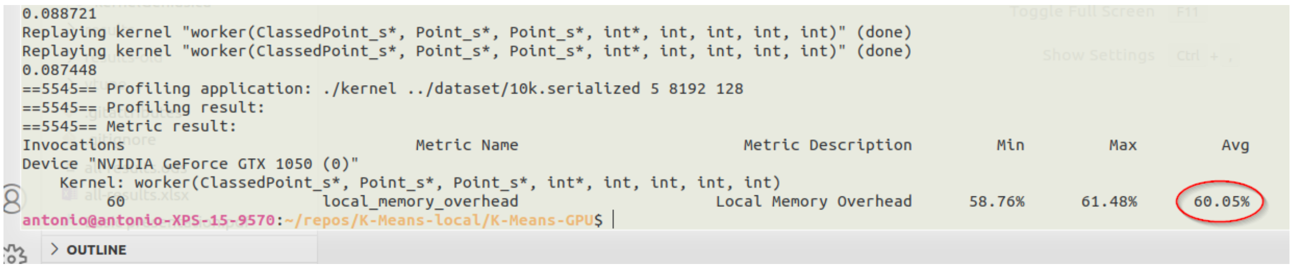
3 .4 Observations

From the data we extracted three main observations about the behaviour of the application's speedup in different executing conditions. We then tried to explain them using different profiling tools, in particular, we used Nvidia nvprof and its GUI version Visual Profiler.

3 .4.1 With a small dataset, increasing the number of threads reduces the speedup

Due to kernel's local memory allocations, necessary to store the partial aggregations done by each thread, the overhead introduced on a small dataset becomes relevant on the overall execution time. With many threads, this overhead effect increases, resulting in longer runs, and consequently a reduced speed-up. Specifically, we observe an increase in local memory overhead from $0,59\%$ with 32 threads to $60,05\%$ with 8192 threads. These results are shown in Figure 16 and Figure 17.

On the other hand, with larger datasets, this effect becomes negligible, therefore the local memory overhead percentage is much lower.



```
0.088721
Replaying kernel "worker(ClassedPoint_s*, Point_s*, Point_s*, int*, int, int, int, int)" (done)
Replaying kernel "worker(ClassedPoint_s*, Point_s*, Point_s*, int*, int, int, int, int)" (done)
0.087448
==5545== Profiling application: ./kernel ../dataset/10k.serialized 5 8192 128
==5545== Profiling result:
==5545== Metric result:
Invocations                               Metric Name                        Metric Description         Min         Max         Avg
Device "NVIDIA GeForce GTX 1050 (0)"
    Kernel: worker(ClassedPoint_s*, Point_s*, Point_s*, int*, int, int, int, int)
         60                       local_memory_overhead                     Local Memory Overhead      58.76%      61.48%      60.05%
antonio@antonio-XPS-15-9570:~/repos/K-Means-local/K-Means-GPU$ |
```

Figure 17: Local memory overhead with 8192 threads

3 .4.2 With larger datasets, increasing the number of threads yields an improved speedup

Taking into consideration the 10M dataset, increasing the number of threads reduces kernel's executing time. This correlates with a higher GPU utilisation, shown by the achieved occupancy in Table 3

Threads	Average Time	Occupancy
32	7,212s	1,6%
2048	165,44ms	20%

Table 3: GPU occupancy

3 .4.3 With a very high number of threads speedup diminishes

Considering the difference between the cases with 2048 threads and 8192 threads, we obtain a higher overhead, which reduces warp efficiency and therefore increases execution time, as shown by Table 4

Threads	Average Time	Occupancy	Warp efficiency
2048	165,44ms	20%	99,5%
8192	189,26ms	45.2%	95.5%

Table 4: GPU occupancy and warp execution efficienty

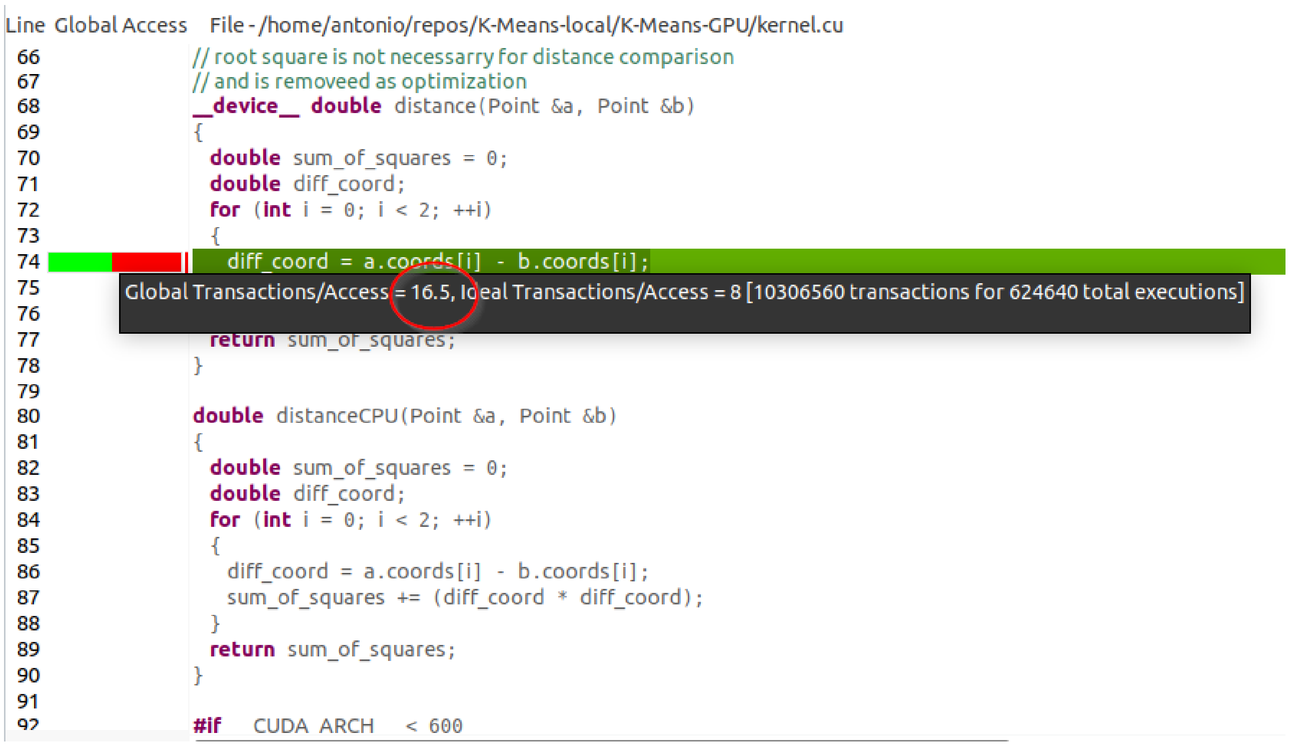
3 .5 Global memory access pattern optimization

Using Nvidia's visual profiler, we observed the amount of time spent by each instruction executed by the kernel, focusing on the slower ones first.

We observed that while calculating the distance of each point from each centroid, we obtained a ratio of memory accesses over transactions much higher than what was recommended, as shown in Figure 18. This pointed us into investigating whether we could improve the spatial locality of relevant data (in this case, the point coordinates) to improve performances.

3 .5.1 Causes

The data structures used by the application are organized as arrays of structures, as shown in Listing 2, which shows that to access the coordinates of multiple points in parallel by multiple



Line Global Access File - /home/antonio/repos/K-Means-local/K-Means-GPU/kernel.cu

```
// root square is not necessarry for distance comparison
// and is removeed as optimization
__device__ double distance(Point &a, Point &b)
{
  double sum_of_squares = 0;
  double diff_coord;
  for (int i = 0; i < 2; ++i)
  {
    diff_coord = a.coords[i] - b.coords[i];
```

Global Transactions/Access = 16.5, Ideal Transactions/Access = 8 [10306560 transactions for 624640 total executions]

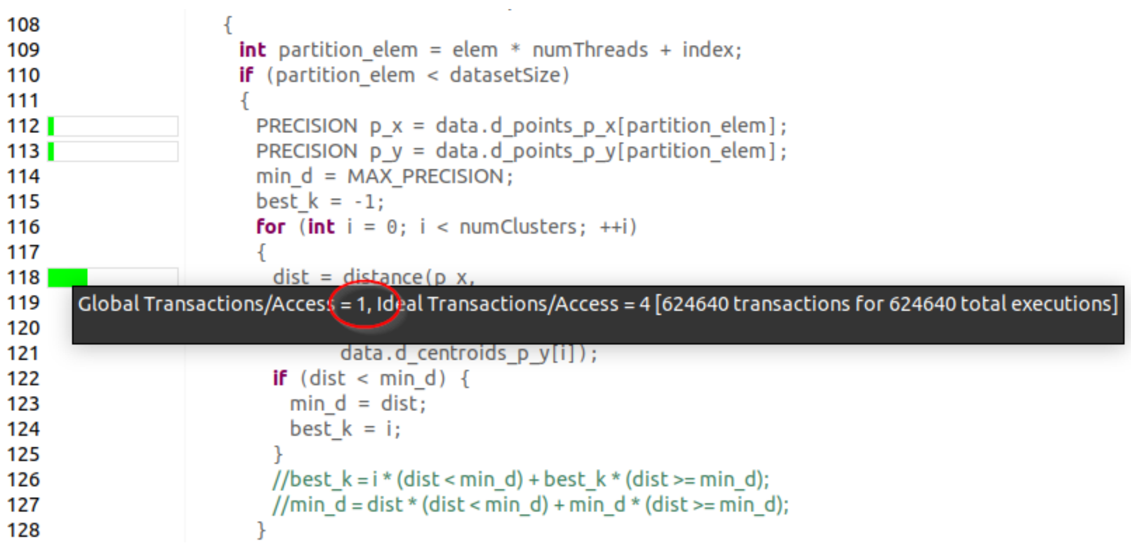
```
  return sum_of_squares;
}

double distanceCPU(Point &a, Point &b)
{
  double sum_of_squares = 0;
  double diff_coord;
  for (int i = 0; i < 2; ++i)
  {
    diff_coord = a.coords[i] - b.coords[i];
    sum_of_squares += (diff_coord * diff_coord);
  }
  return sum_of_squares;
}

#if   CUDA ARCH   < 600
```

Figure 18: Transactions per access without optimization

threads, we must skip to different address locations that are far apart in memory. This leads to a poor utilization of the memory transfer bus.

3 .5.2 Solution

We changed those data structures (and all relevant code handling them) in favor of an approach based on a single structure of arrays instead, as shown in Listing 3



```
        {
          int partition_elem = elem * numThreads + index;
          if (partition_elem < datasetSize)
          {
            PRECISION p_x = data.d_points_p_x[partition_elem];
            PRECISION p_y = data.d_points_p_y[partition_elem];
            min_d = MAX_PRECISION;
            best_k = -1;
            for (int i = 0; i < numClusters; ++i)
            {
              dist = distance(p_x,
Global Transactions/Access = 1, Ideal Transactions/Access = 4 [624640 transactions for 624640 total executions]
                  data.d_centroids_p_y[i]);
              if (dist < min_d) {
                min_d = dist;
                best_k = i;
              }
              //best_k = i * (dist < min_d) + best_k * (dist >= min_d);
              //min_d = dist * (dist < min_d) + min_d * (dist >= min_d);
            }
```

Figure 19: Transactions per access with optimization

```
#define PRECISION float
struct KMeansData_s
{
    // clustered_Point
    PRECISION *points_p_x;
    PRECISION *points_p_y;
    int *points_k;

    // Centroid
    PRECISION *centroids_p_x;
    PRECISION *centroids_p_y;
    PRECISION *centroids_sum_x;
    PRECISION *centroids_sum_y;
    int *centroids_partition_lengths;

    // clustered_Point
    PRECISION *d_points_p_x;
    PRECISION *d_points_p_y;
    int *d_points_k;

    // Centroid
    PRECISION *d_centroids_p_x;
    PRECISION *d_centroids_p_y;
    PRECISION *d_centroids_sums_x;
    PRECISION *d_centroids_sums_y;
    int *d_centroids_partition_lengths;

};
typedef struct KMeansData_s KMeansData;
```

Listing 3: gpu data structures

With this data access pattern, all threads inside a warp access contiguous memory locations to perform operations on their points. This also holds for every operation done over the centroids. In this way we maximize the utilization of the data transfer bus, as shown by Figure 19

3 .5.3 Results

In Figure 20 and Figure 21 we show the execution times and speedups regarding the optimized version of the code for each dataset. We can notice an improvement in speedup for the majority of the datasets taken into cosideration.

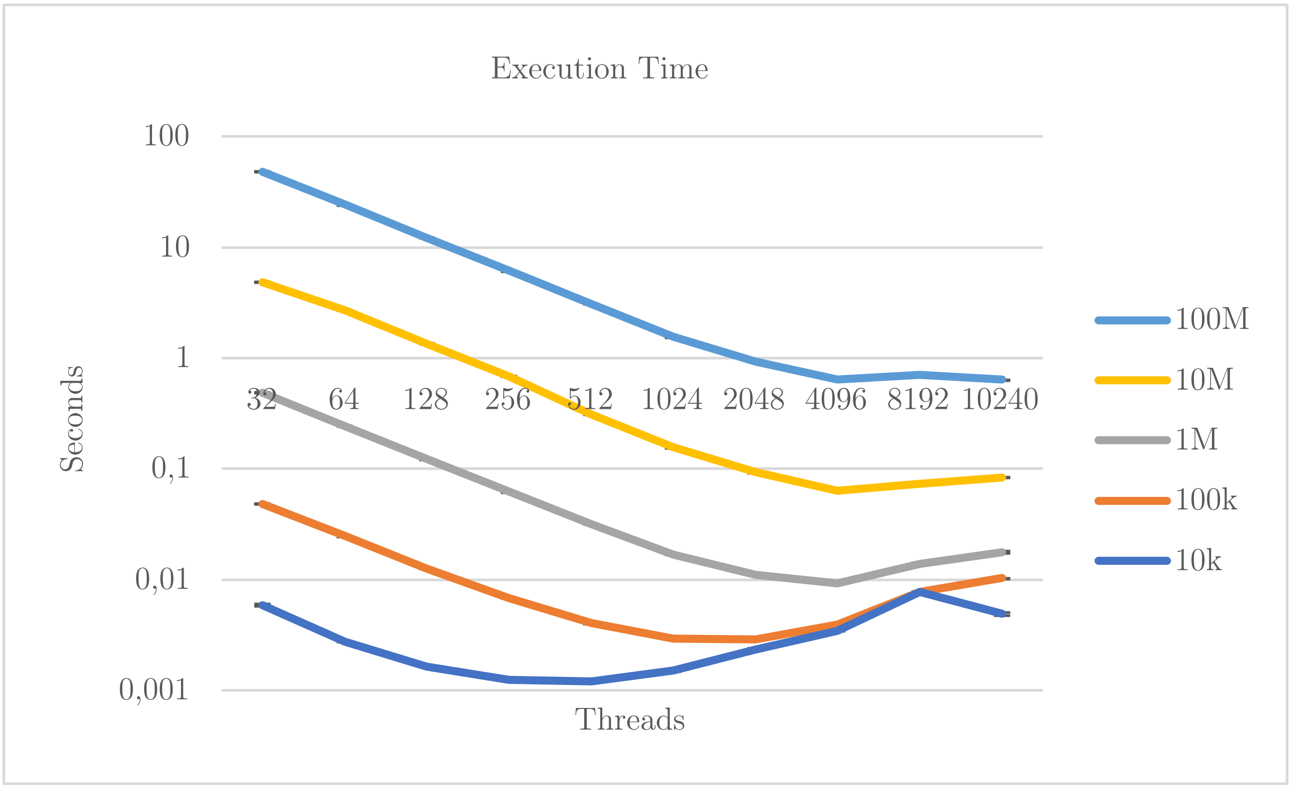


Figure 20: GPU execution times with optimization

In Figure 22 we show the results obtained in terms of execution time taking into consideration the 10M dataset, comparing the unoptimized and optimized versions of the code. In particular, we can observe that we obtain a reduction of almost 50% in execution times.

In Figure 23 we see the comparison between the relative speedup of the two solutions.

We can note that we obtain for 4096 threads a speedup of around 77, which is a relative speedup of 1.8 times with respect to the unoptimized case.

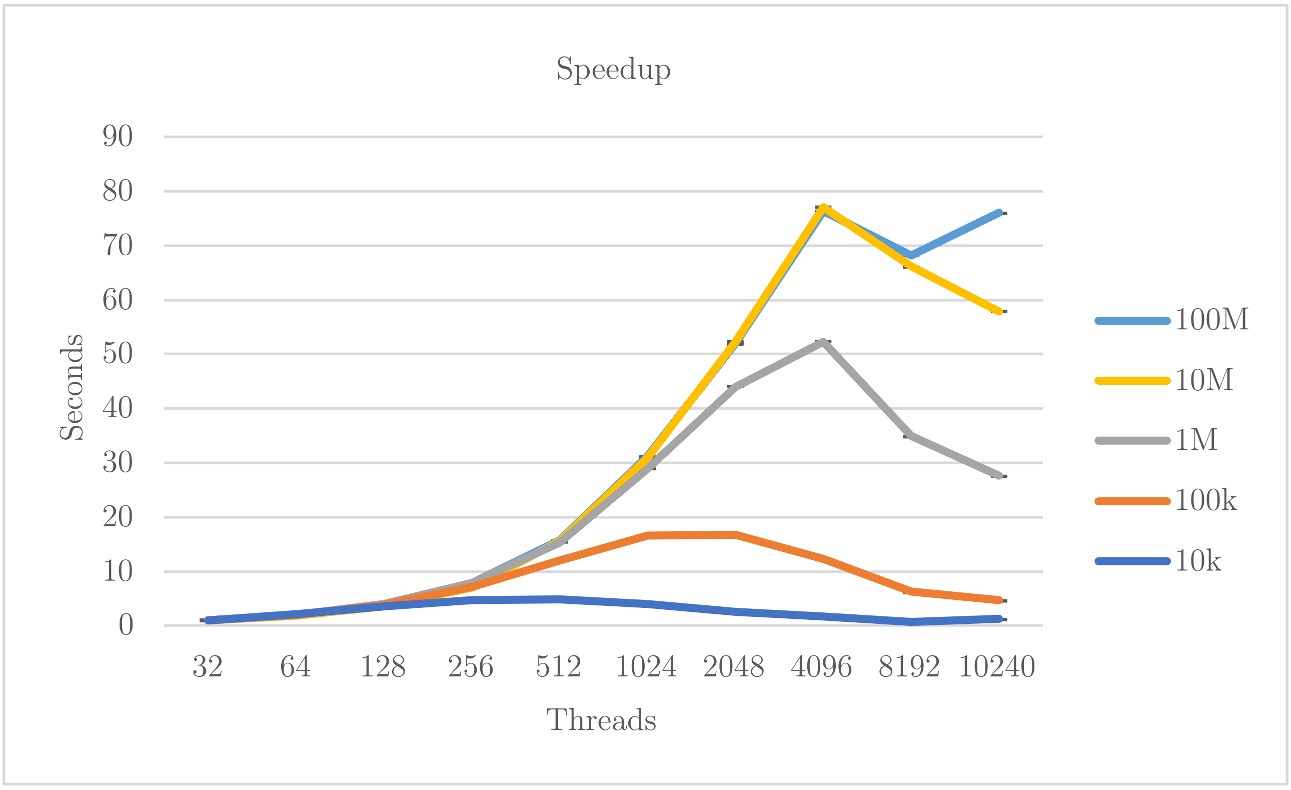


Figure 21: GPU speedup with optimization

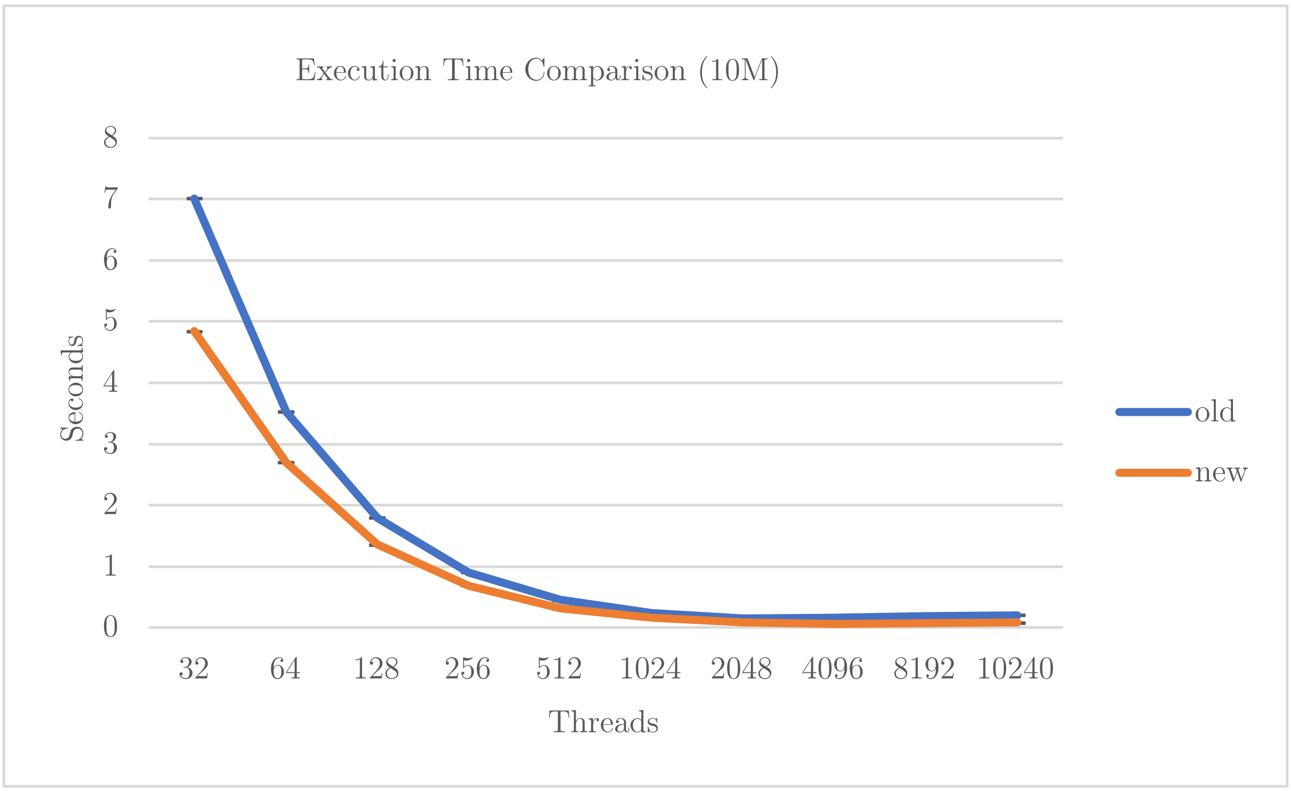


Figure 22: GPU execution times comparison with optimization

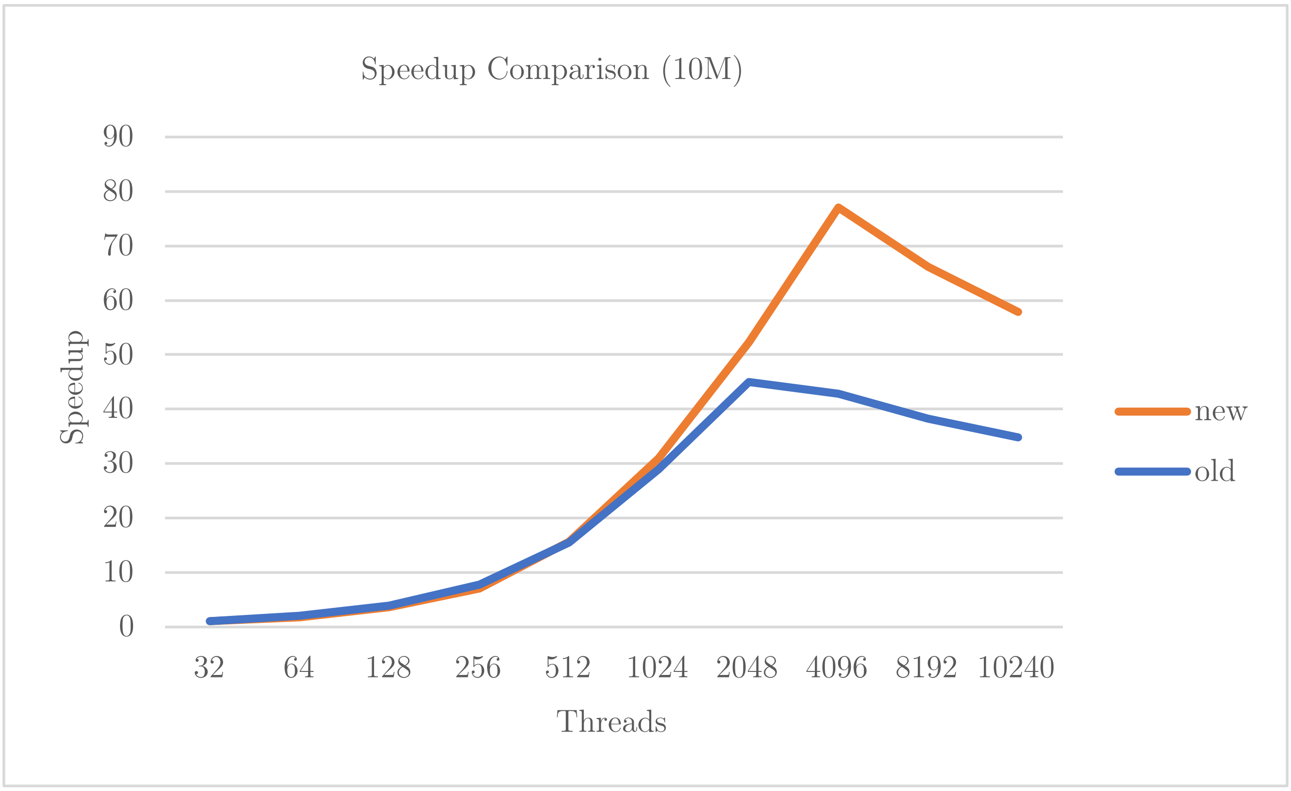


Figure 23: GPU speedup comparison with optimization

4 CPU vs GPU comparison

In Table 5 we can see a comparison between the best performing configurations for both the CPU and GPU versions that we analyzed, showing for every dataset the minimum execution time obtained with the optimized version of the software, both on CPU and GPU.

	CPU		GPU	
Dataset	Min exec time	Threads	Min exec time	Threads
10k	0,00028	1	0,00121	512
100k	0,00127	8	0,00288	2048
1M	0,00881	16	0,00930	4096
10M	0,08127	16	0,06279	4096
100M	0,76043	18	0,63529	4096

Table 5: CPU vs GPU comparison

We can observe that for 10k, 100k and 1M dataset, the CPU is performing slightly better, meanwhile for all larger datasets, the GPU version is faster. Those results are in line with what we expect because it becomes much more efficient to distribute among multiple executing units a larger workload.

5 Conclusions

In this project we analyzed a parallel K-Means implementation for both a CPU and GPU, highlighting the pros and cons of each architecture. In particular, we obtained lower execution times only for larger datasets and much higher speedups, which is expected with this type of devices. Further analysis with even larger datasets may result in even better performances.