UNIVERSITÀ
DEGLI STUDI
FIRENZE

# Parallel computing

## Parallel implementation of K-means algorithm

Relatore: Leonardo Di Iorio

# Introduction

The following presentation will show and compare three different implementations of the K-means clustering algorithm:

- **Sequential**
- **OpenMP**
- **CUDA**

UNIVERSITÀ DEGLI STUDI FIRENZE

# K-means algorithm

K-means is one of the most widely used clustering algorithms. It's based on centroids and has an iterative structure. It's articulated in three steps:

- **Inizialization**;
- **Point assignment**;
- **Centroids update**.

# Inizialization

Given a datasets of size **n**:

1. The number **K** of clusters is chosen;
2. **K** points in the dataset are choosen to represent the initial centroids;
3. Initial centroids are chosen randomly within the points in the datasets;
4. Initial centroids must not be coincident.

# Point assignment

At this stage the algorithm associates each point in the dataset with the nearest centroid:

↓

- It calculates the Euclidean distance between each point and the centroids, then assigns the point to the minimum distance centroid.

# Centroids update

As a consequence of the point assignment step, it is likely that the composition of the clusters has changed. In this step we then calculate the new centroid of each cluster:

↓

- The algorithm computes the average of all points belonging to the cluster under consideration.

# Sequential

The sequential version of the code is implemented in C++. Despite this, the code is not developed with an object-oriented programming approach:

- To avoid slowdowns in execution;
- To achieve code that is as similar as possible between the sequential and parallel versions;

# Sequential

Then a single class was implemented, called **Kmeans**, characterized by only three methods:

- The constructor;
- **The *kmeansIteration()* method:** which is responsible for the implementation of the operations related to a single iteration of the algorithm ;
- **The *computeKmeans()* method:** which is responsible for orchestrating the iterative execution of the algorithm;

UNIVERSITÀ DEGLI STUDI FIRENZE

# Sequential

- The implementation related to the point assignment step:

```
22      float* newCentroids= new float[numClusters*dim]{0};
23      for(int n=0;n<numPoints;n++){
24         int nearestCluster=0;
25         int minDistance=INT_MAX;
26         for(int c=0;c<numClusters;c++){
27          float euclideanDistance=0;
28          for(int d=0;d<dim;d++){
29             euclideanDistance+=pow(*(points+n*dim+d)-*(centroids+c*dim+d),2);
30          }
31          if(euclideanDistance<minDistance){
32          minDistance=euclideanDistance;
33          nearestCluster=c;
34          }
35         }
36         for(int j=0;j<dim;j++){
37           *(newCentroids+nearestCluster*dim+j)+=*(points+n*dim+j);
38         }
39         membersCounter[nearestCluster]++;
40      }
```

# Sequential

- The implementation of the step related to the update of centroids:

```
43      for(int i=0;i<numClusters;i++){
44          for(int j=0;j<dim;j++){
45              *(newCentroids+i*dim+j)=(*(newCentroids+i*dim+j))/membersCounter[i];
46          }
47      }
48
49      for(int i=0;i<numClusters;i++){
50          for(int j=0;j<dim;j++){
51              *(centroids+i*dim+j)=*(newCentroids+i*dim+j);
52          }
53      }
```

# OpenMP

The parts of the implemented code that need to be parallelized in order to achieve performance improvement are the external for loop:

- Both those related to the point assignment step and those related to the udpate step of the centroids;

To parallelize the loops for the following directive has been used:

- **#pragma omp parallel for**

# OpenMP

As for the management of critical sections, in order to avoid race condition it is necessary to introduce a specific OpenMP directive before the expressions to the lines 37 and 39 of the code in the previous slide:

- **#pragma omp atomic**

# CUDA

- Parallelization is performed with respect to the **n** points in the dataset with a single **kernel** function;
- Compared to the parallel version obtained with OpenMP, the external loops **for** were replaced by **if** structures;
- These structures determine whether or not the centroid is updated, based on the following thread index:

```
int t = blockIdx.x * blockDim.x + threadIdx.x;
```

- The call to the kernel function:

```
kmeansIterationKernel << <ceil(numPoints / (float)blDim), blDim >> >
```

# CUDA

The main functions used in the code:

- **cudaMalloc** and **cudaFree** were used to allocate and release memory on the GPU;
- The **cudaMemcpy** function was used to pass the arrays from host to device;

# CUDA

- In order to avoid problems during execution, the atomic expression **atomicAdd** was used to increment the counter of points belonging to a cluster in the assignment phase;
- In order to ensure the correct output with respect to updating centroids, the **__threadfence** function was inserted before the section of code related to centroids updating;

# Experimental setup

- CPU Intel i7-10750H;
- GPU NVDIA GeForce RTX 2070 Super;
- The stop criterion for all versions of the algorithms is a maximum number of iterations;
- The points constituting the datasets are randomly generated by a uniform distribution and are characterized by three dimensions;
- For the time measurements we have used **system_clock** from the **chrono** header.

UNIVERSITÀ DEGLI STUDI FIRENZE

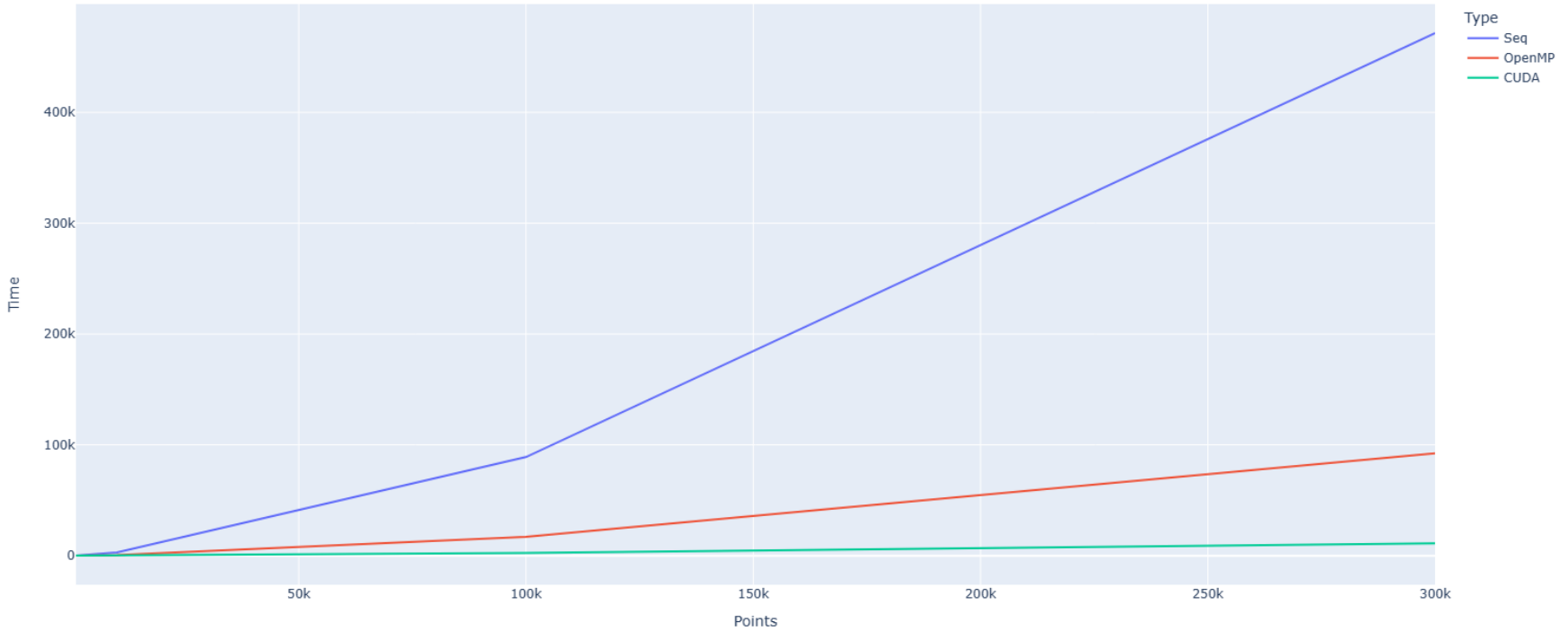
# Results of the comparison

Results were visualized by means of two main types of experiments:

- In the first case, the proportion of clusters, **K** , was kept stable and the number of points **n** was increased;
- In the second case, the number of points was fixed and we varied the number of clusters.

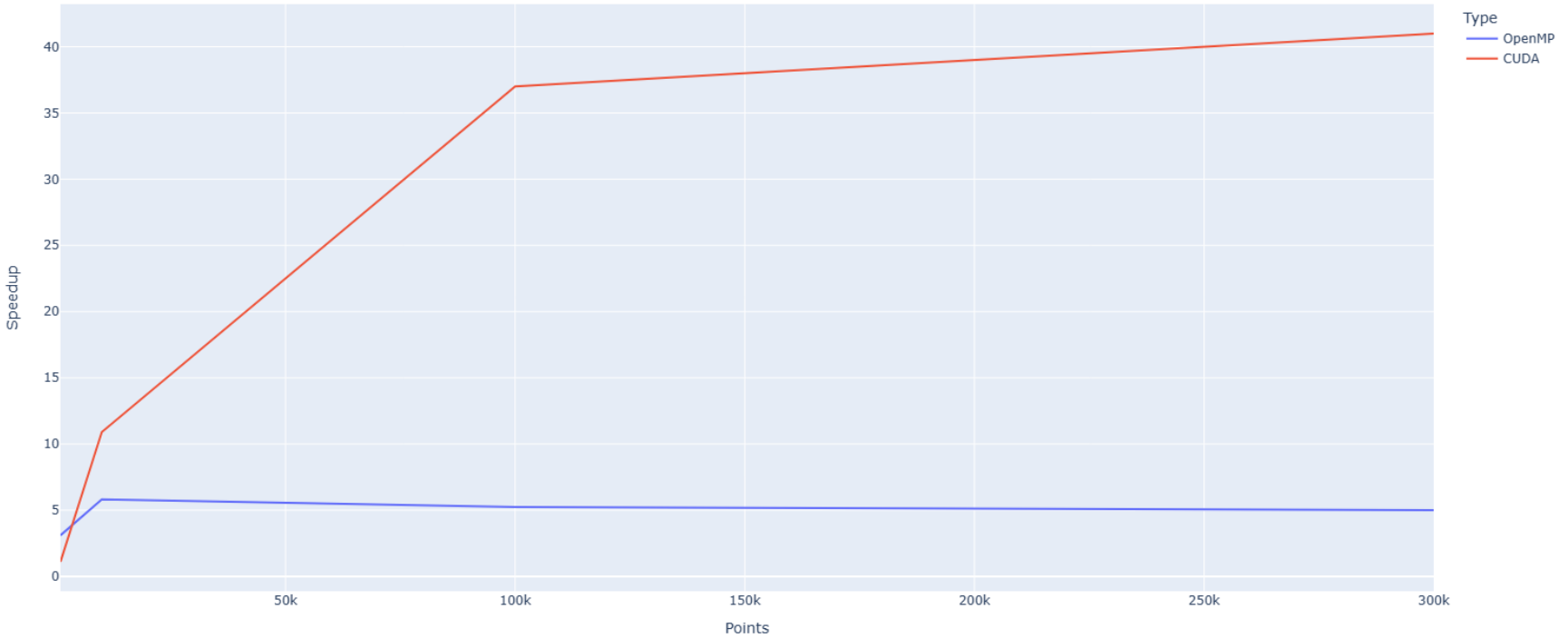
UNIVERSITÀ DEGLI STUDI FIRENZE

# Results of the comparison



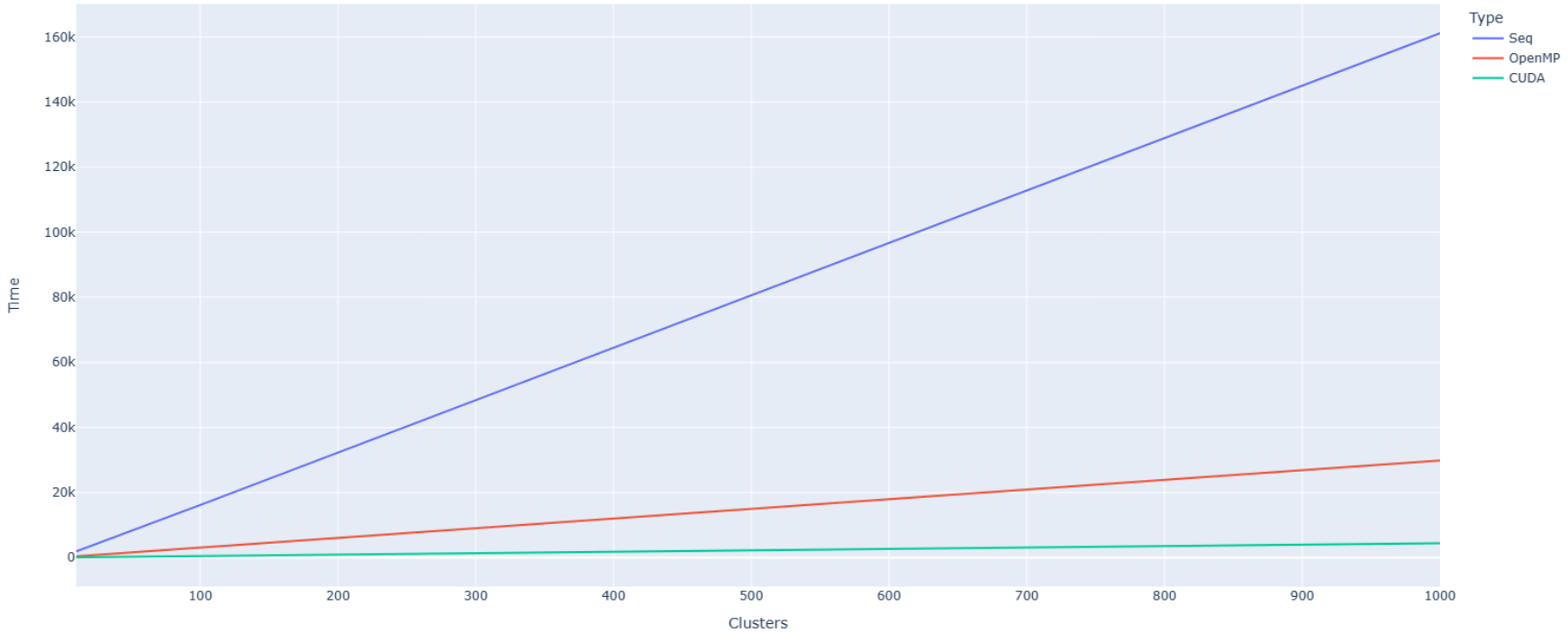
UNIVERSITÀ DEGLI STUDI FIRENZE

# Results of the comparison



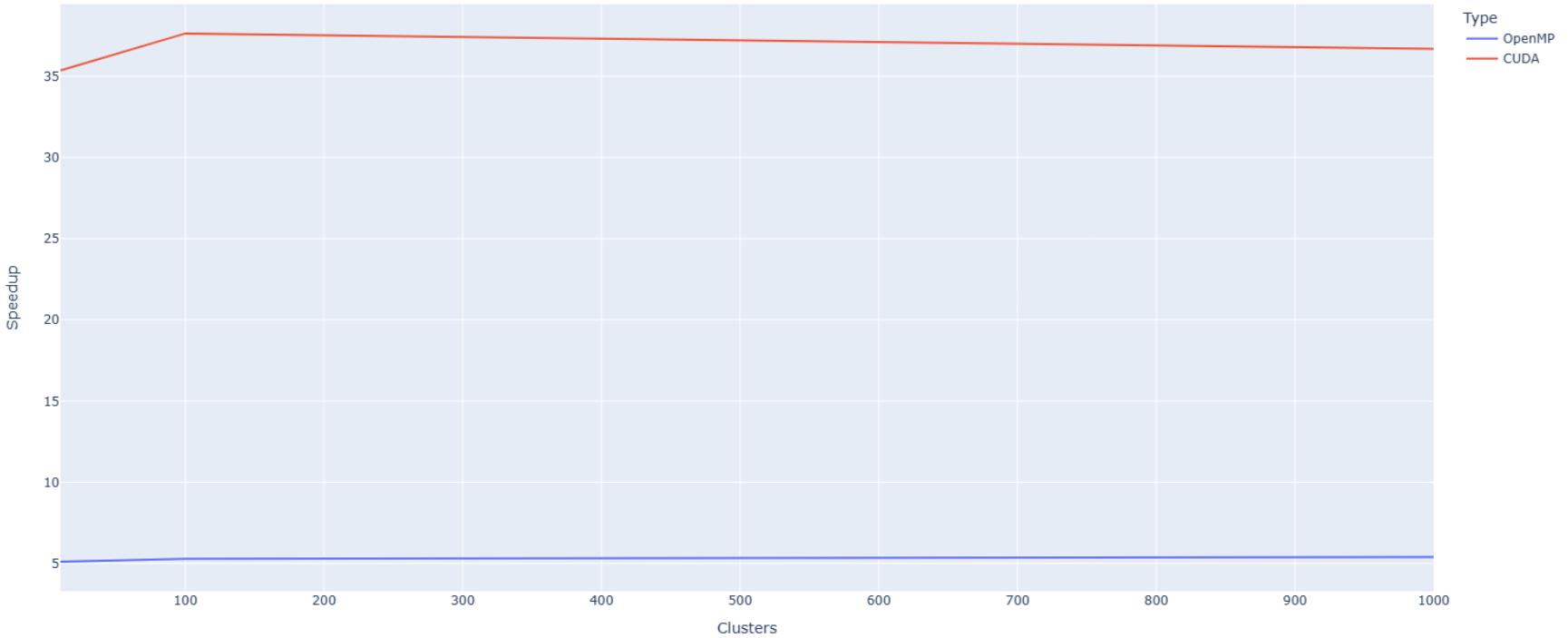
UNIVERSITÀ DEGLI STUDI FIRENZE

# Results of the comparison



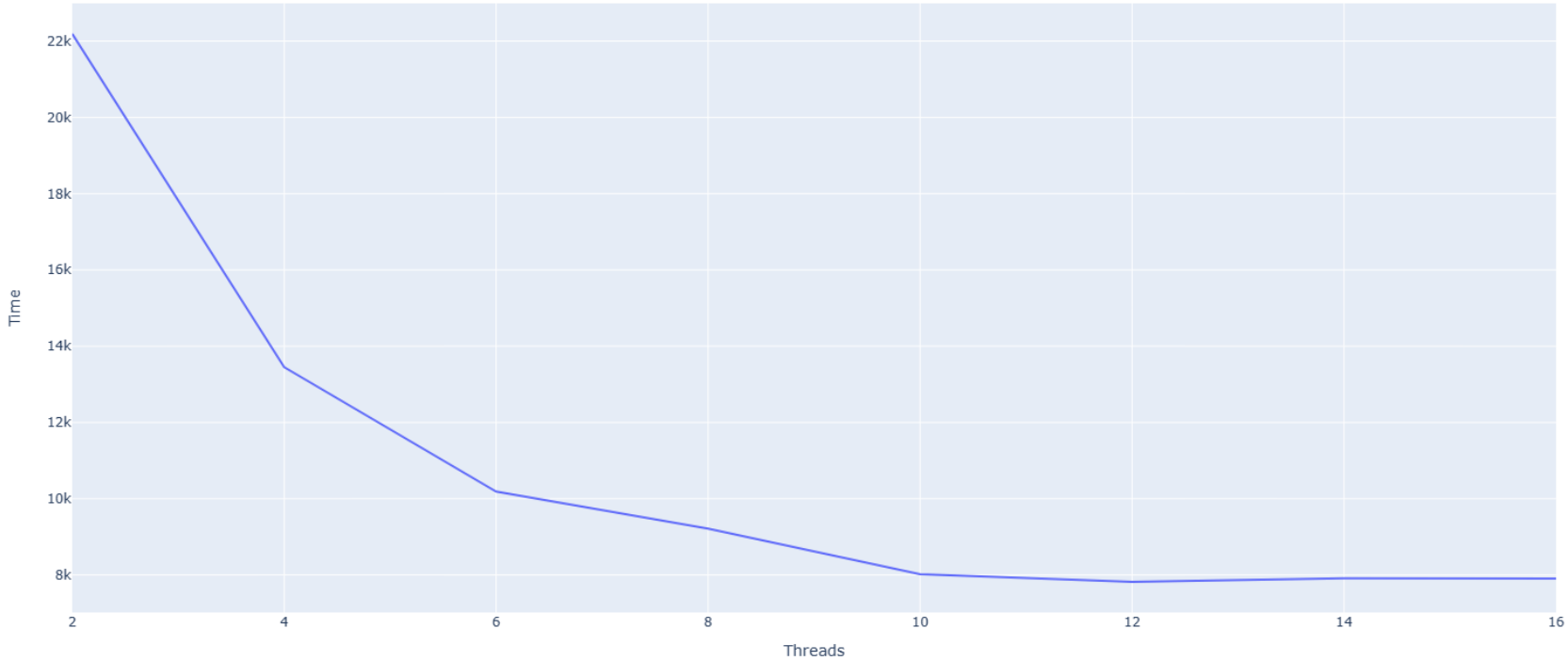
UNIVERSITÀ DEGLI STUDI FIRENZE

# Results of the comparison



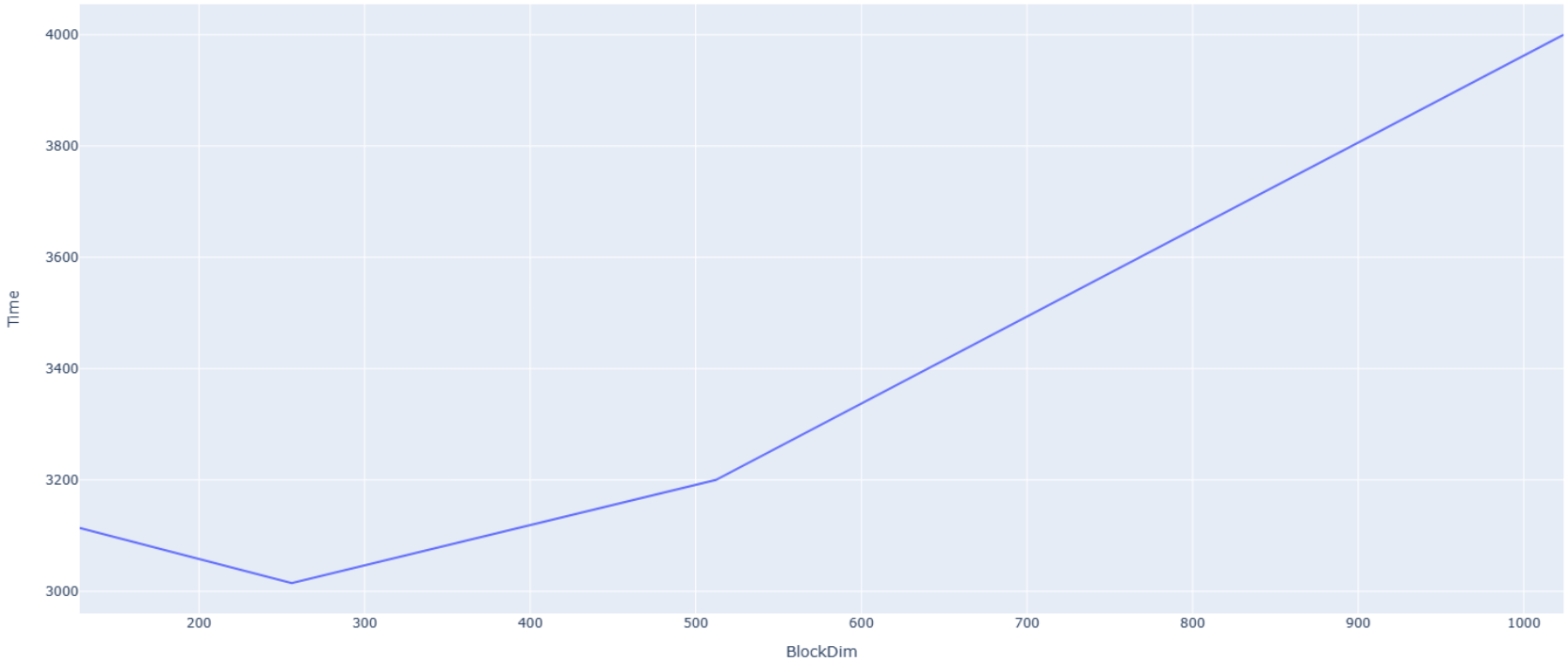
UNIVERSITÀ DEGLI STUDI FIRENZE

# Number of threads evaluation



UNIVERSITÀ DEGLI STUDI FIRENZE

# BlockDim evaluation



# Conclusion

- The configuration that offers better performance turns out to be the one in CUDA;
- Very often such an implementation requires a higher workload than a similar OpenMP implementation;
- In many contexts it may be convient to consider the OpenMP approach.


UNIVERSITÀ
DEGLI STUDI
FIRENZE

# Grazie per l’attenzione!